UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

SISTEMA DE MONITOREO Y ANALISIS DE DATOS (SMAD), UTILIZANDO LA TECNOLOGIA MULTIHILOS Y EL DISEÑO ORIENTADO A OBJETOS

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACION

P R E S E N T A N :

MIGUEL ANGEL PADILLA CASTAÑEDA

DAVID PEREZ SANCHEZ

DIRECTOR DE LA TESIS: DR. JESUS SAVAGE CARMONA

CIUDAD UNIVERSITARIA 2000

David

Dedicatoria

El siguiente trabajo se lo dedico a mi familia que siempre me ha apoyado con mis estudios, especialmente a mi madre que nos permitió tener acceso a una educación superior. A mis hermanos Joaquín y Alberto que de alguna forma influenciaron en mi para seguir estudiando. A mis hermanas Cecilia y Gladis Graciela que siempre fueron muy complacientes conmigo. A la memoria de mi Padre y de mi hermano Noé qué estoy seguro que hubieran sentido orgullosos. Finalmente una dedicatoria a mi profesor de primaria de 5to y 6to año al cuál le hice la promesa de titularme algún día.

Agradecimientos

Agradezco de manera muy especial a todas y cada una de las personas que de alguna forma me ofrecieron su amistad y que gracias a su presencia la vida se va llenado de anécdotas que hacen que valga la pena estar aquí y seguir adelante. Espero acordarme de cada uno de ellos y de ellas y si por alguna razón omito algún nombre no es por otra cosa que por falta de memoria. Gracias a mis amigos de los primeros años de la primaria. Rafael, Daniel, Elizabeth, Juan Carlos, Montoya, Toño, Ruben alias el Pollo, a Margarita y a Yolanda. Gracias a mi maestra Rosa de 1er año de la cual siempre estuve enamorado. Un agradecimiento especial al maestro Toño por ser de los que más confío en mi. Gracias también a la maestra Amparo que nos hizo sufrir a mas de uno, a la maestra Berna y a la maestra Hortensia. A mis Amigos de la secundaria Milton, Montealegre, el Tony, el Simi a Alejandro, a el güero, a Eduardo, a mi tocayo David Alberto que siempre fue un buen adversario en el estudio, a Ivonne de "2do E", a Elizabeth que siempre dijo estar orgullosa de mi, en realidad el que siempre estuvo orgulloso de ella fui yo. Gracias a las locas de Noemi, Adriana y Maribel. A mis profesores de secundaria: Anselmo de biología, Zoria de Física, Gregorio de Matemáticas, Monsalve de Química, Jorge de Civismo, Martha de Geografía y Ruben de Historia.

A mis amigos del CCH, Javier Portilla, Roberto, David, Ramón, Israel Arancivia, Luis Belmont, Edmundo, Ingrid, Andrea, Yuriria, Evelia, a Gabriela Villanueva que cuando se caso le rompío el corazón a mas de uno, a Noemi a Laura, de manera muy

especial a Adba y a mis amiguisimos Edgar Pantoja y Gerardo Palencia que por cierto todavía no encontramos a la cuarta tortuga. Algunos de mis profesores que aunque no se me grabaron sus nombres si recuerdo que materias nos dieron en primer lugar el que nos dio Matemáticas III y IV, el que nos dio Historia ya que nos divertimos mucho haciendo las radionovelas, el viejo Loco que nos dio Física II y III, el profesor Francisco de Biología y Metodología experimental, el profesor de Economía que nos abrió los ojos a mi y a muchos mas. A la profesora de Geografía por sacarnos seguido de viaje.

A mis amigos de la Universidad, Alberto Guillen, Angel Monroy, el RamnonCinco, a Dario de manera especial a Erick Canales y los amigos de Guerrero Eduardo Cabrera y Carlos Zamitiz. A todos los compañeros del PTC en especial a Edwin Navarro, Alejandro Pérez, Mario, Sergio, Gustavo y Arturo Gardida que por cierto en su mayoría todos eran muy bueno jugadores de DOOM. A Maite, Catalina y Sandra por ser las únicas amigas de computación a Violeta y de manera muy especial a Alejandra Karina de la cual espero que sigamos siendo amigos por muchos años más. A los profesores: Francisco Patiño, Raúl Luna, Lauro, Tan Li Yi, Salva Calleja y Laura Sandoval. Finalmente a mi compañero de Tesis y amigo desde que entre a la Universidad Miguel Angel Padilla, que la verdad si no hubiera sido por él, nunca hubiera terminado este trabajo.

Miguel

Dedicatorias

A mis padres, María de la Luz y Miguel Angel, por darme la vida, por su amor, por los valores que me han inculcado, por enseñarme a luchar, luchar en libertad.

A mi hermana, Luz del Carmen, por su alegría y vitalidad, por su incondicional solidaridad.

A Tair, por su sinceridad, empuje y por todos los momentos que hemos vividos juntos.

A mis primos, Gilda Carolina, Juan Carlos, Luis Fernando y Rubén Fernando, por su fraternidad, por los disgustos y alegrías que siempre estamos dispuestos a buscar.

A mis abuelos, Enedina (qepd), Rubén, Jesús (qepd) y Carmen, porque donde se encuentren siempre están conmigo, apoyándome aún en mis errores.

A mis tíos Jesús, Gilda, Roberto, por su paciencia, enseñanzas, consejos y porque creen en nosotros, los jóvenes.

A todos mis amigos, en especial a Arturo, Carlos, David, José Alberto, por compartir conmigo mucho más que su amistad.

A mis profesores, por su afán desinteresado de ver en los estudiantes, la esperanza de un mundo mejor.

A mis amigos del Programa de Tecnología en Computación, porque gracias a al factor $PTC^{nerd} * P_2 T(e_2)$, todos los días aprendía algo nuevo de cada uno.

A mis compañeros del Foto Club Ingenieros que me hicieron comprender que la Universidad va más allá de los libros.

A la UNAM por haberme dado la oportunidad de estudiar y servir a mi país, educación es libertad.

Introducción

La tendencia actual de distribuir la información de los sistemas de cómputo, ha provocado la creciente necesidad de almacenar y manipular grandes cantidades de datos de manera cada vez más sofisticada y eficiente. Por otra parte, actualmente los usuarios requieren tener la mayor información posible de las actividades productivas en las que participan y tener así ventajas competitivas reales.

Las dos anteriores observaciones acentúan la importancia cada vez mayor del desarrollo de sistemas de información abiertos y avanzados que utilicen diversos recursos, sin límites impuestos por las plataformas y herramientas de desarrollo elegidas. Esto obliga a usar metodologías, técnicas y herramientas de diseño que permitan obtener aplicaciones que integren diversos componentes conectados en red, así como el uso de tecnologías como la *programación orientada a objetos*, la arquitectura *cliente-servidor* y la tecnología *multihilos*; expandiendo así, las capacidades de procesamiento y almacenamiento de datos, procurando tiempos de respuesta adecuados.

Ante estas necesidades, nos hemos propuesto tres objetivos que ayudarán a solucionar las mismas; el primero, es la creación de una *metodología de diseño orientado a objetos* para el diseño de sistemas de información distribuidos. El segundo objetivo es proponer la *tecnología multihilos* como alternativa en el desarrollo de *software* distribuido y aplicaciones concurrentes. Por último, diseñar y desarrollar un *sistema de monitoreo y análisis de datos* que permita a los usuarios supervisar procesos productivos y conocer eventos, esto con el fin de ayudar a la toma de decisiones en base a las reglas que se hayan establecido.

El sistema permitirá tener el registro de los datos que se generen de manera constante en sus diversos procesos y operaciones; a su vez, tendrá mayor control sobre los mismos, lo cual será de gran importancia para la toma de decisiones.

Este sistema recibirá los datos involucrados en las operaciones del proceso en cuestión y los analizará para monitorear su comportamiento. El análisis se basará en el uso de estos datos para el cálculo de fórmulas que el usuario defina, las cuales establecerán las reglas que deben cumplir las operaciones.

Los componentes del sistema están divididos en dos partes fundamentales, una se encuentra del lado del cliente e integran las interfaces de monitoreo; la otra se encuentra del lado del servidor y la forman los encargados del análisis de datos.

Para una mejor comprensión, el presente trabajo se ha dividido en dos partes; en la primera, se presenta el estudio de los elementos del proyecto la cual involucra a los primeros dos capítulos. En la segunda parte, se hace una descripción completa del análisis, diseño y desarrollo del proyecto, para lo cual usamos varios conceptos explicados en la primera parte. Esta segunda parte está formada por los capítulos del tres al sexto.

En el primer capítulo se explica una metodología de diseño orientada a objetos que engloba varias técnicas de diseño como son diagramas de flujo de datos, diagramas de entidad relación y modelos de objetos para la utilización en el modelado de datos de sistemas distribuidos.

En el capítulo dos, se hace una síntesis de algunos de los conceptos involucrados en la tecnología *multihilos*, algunos de los cuales fueron utilizados en la implantación del sistema que se propone en este trabajo y que han sido estandarizados como un paradigma de programación cada vez más utilizados no sólo en las universidades y centros de investigación, sino también en las industrias.

En el siguiente capítulo, se analizan factores que intervienen en el sistema, se identifica cada uno de los actores principales y se definen sus funciones. Posteriormente en el capítulo cuatro, se propone la arquitectura del sistema que mejor involucra a todos los factores y elementos identificados, la arquitectura describe al proyecto como un sistema distribuido con componentes abiertos, y está basada en el modelo *cliente-servidor*.

Luego en el capítulo cinco, se explica detalladamente el diseño del componente responsable de llevar a cabo el análisis de los datos llamado *Servidor Monitor*. Éste, entre otras cosas, debe detectar las transacciones en el proceso productivo, tomar los datos involucrados, llevar a cabo el análisis de estos en base a las fórmulas definidas por los usuarios para tal proceso, y finalmente, enviar mensajes a los usuarios que indiquen algún suceso que rompa las reglas dictadas. Para el diseño y desarrollo del *Servidor Monitor* se utiliza la tecnología *multihilos* expuesta en el primer capítulo.

En el último capítulo, se explica el diseño de los componentes que se encuentran del lado del cliente y las interfaces, cuyas funciones son las de definir las fórmulas o reglas para el proceso, y realizar las tareas de monitoreo de las señales enviadas por el *Servidor Monitor* para indicar los eventos cuyos datos no cumplen con las fórmulas definidas. Para el diseño y desarrollo de las interfaces se utilizaron los conceptos de diseño orientado a objetos vistos en el capítulo dos.

Después de los seis capítulos que componen esta investigación, se exponen las conclusiones a las que se llegaron después de ver los resultados de la propuesta.

Finalmente y como complemento para la mejor comprensión de esta tesis, se presenta un conjunto de apéndices que incluye documentación esencial de los componentes principales del sistema, así como un glosario de términos manejados en este trabajo, diferentes diagramas y modelos, así como sus respectivos diccionarios de datos y el pseudocódigo de algunas de las rutinas más importantes.

Parte 1

Capítulo 1 Metodología orientada a objetos para el diseño de aplicaciones distribuidas

En la actualidad los sistemas de cómputo se dice que tienden a ser distribuidos, esto implica que estos sistemas estén repartidos entre dos o más elementos. Particularmente en sistemas de cómputo lo que se está repartiendo es parte del código y/o los datos de algún programa, aplicación o tarea entre dos o más computadoras.

Este tipo de esquemas se conocen como sistemas de memoria distribuida por tener varias unidades de procesamiento, cada una con memoria independiente, que de alguna manera, a través de una red, se comunican con las demás con la finalidad de realizar en forma cooperativa alguna tarea o compartir algún recurso. "Un *sistema distribuido* se define como una colección de computadoras autónomas ligadas por una red, con *software* diseñado para sistemas distribuidos."[1] El *software* para sistemas distribuidos, es aquel que permite a las computadoras coordinar sus actividades y compartir los recursos del sistema.

Por ejemplo, un esquema típico de un sistema distribuido es una red de área local donde se tiene conectadas varias computadoras de diversos tipos, las cuales pueden brindar o accesar los diferentes servicios en red que son proporcionados por algunas de esas computadoras conectadas a la red (ver fig. 1.1).

Actualmente la computación en red depende mas de modelos de programación que de implantaciones de *hardware* (como se verá en el siguiente apartado), esto nos lleva a preguntarnos ¿cuál podría ser la tecnología que permita el desarrollo del cómputo distribuido?. Por diversas razones se propone a la orientación a objetos como esa tecnología.

En este capítulo se presentarán de forma general algunos de los conceptos básicos de la teoría orientada a objetos y algunas de las ideas de las metodologías que actualmente se basan en el análisis y diseño orientados a objetos, para finalmente obtener puntos de convergencia entre esas metodologías que sirvan para el análisis y diseño de los sistemas en general.

---

[1] Colouris G. *Distributes Systems. p. 2*

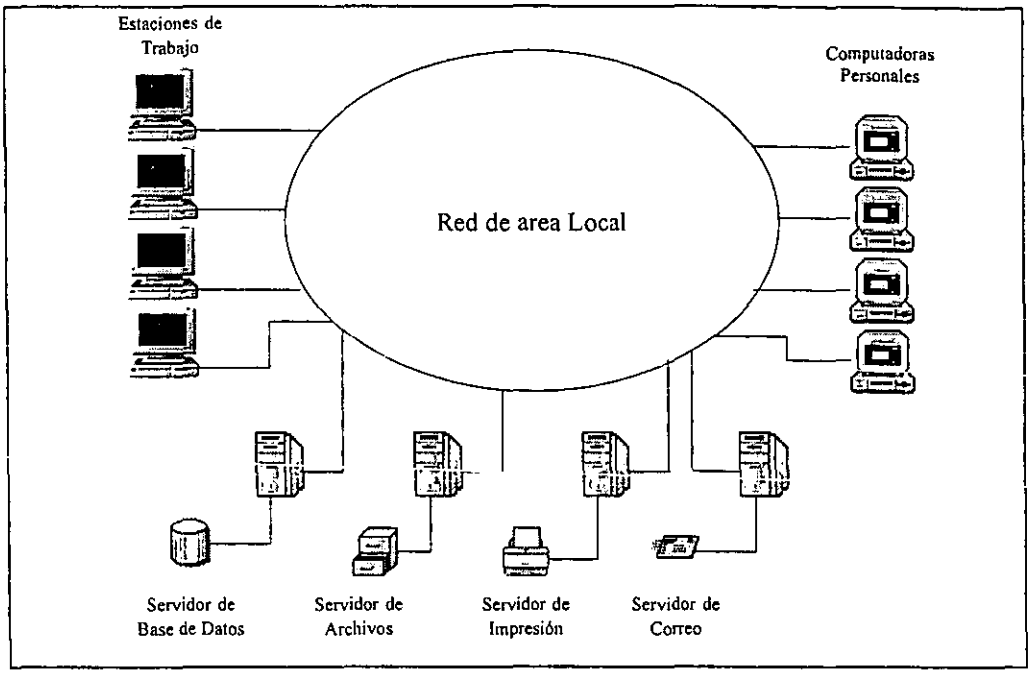


Figura 1.1 Sistema Distribuido

1.1 Arquitectura de sistemas de cómputo en red

1.1.1 Sistemas Centralizados

Los primeros sistemas de cómputo en red son los llamados *sistemas centralizados* (*Host Based Computing*). En este tipo de arquitecturas todo el procesamiento ocurre en una máquina central. Las aplicaciones y los datos se encuentran en una máquina con gran capacidad de procesamiento (*mainframe* o *minicomputadora*) a la que se le conectan varias terminales, las cuales sirven para enviar o desplegar datos (ver fig. 1.2). Este tipo de ambientes fueron muy usados por varios años. Las ventajas que se tenían eran que las máquinas centralizadas ofrecían un procesamiento confiable, así como una sólida seguridad y control de los datos. Sin embargo, el costo de los equipos, así como el mantenimiento de *hardware* y *software* eran altísimos.

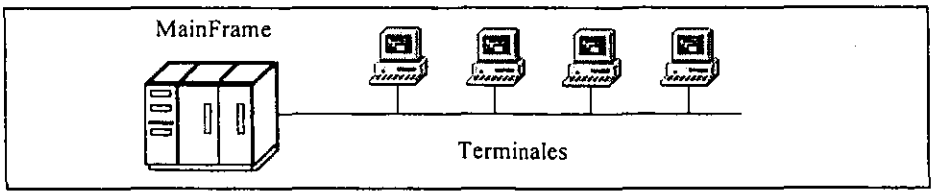


Figura 1.2 Sistemas Centralizados

1.1.2 Sistemas basados en redes *LAN*

A mediados de los años ochenta con el auge de las computadoras personales, una nueva arquitectura de sistemas de cómputo en red surgió, los *sistemas basados en redes LAN* (*PC/LAN Based Computing*). En estos ambientes el costo era significativamente menor que en un sistema centralizado. La idea principal de este esquema era todo lo contrario al de un ambiente centralizado en lo que se refiere al procesamiento.

Existía una máquina que servía como almacén de archivos la cual podía ser accesada por las otras máquinas de la red *LAN*, de manera que estas máquinas podían ejecutar las aplicaciones especificadas en esos archivos, es decir, el procesamiento ahora se realizaba en cada una de las máquinas de la red *LAN* (ver fig. 1.3).

Figura 1.3 Sistemas basados en redes LAN

Este tipo de ambientes se comportan bien con aplicaciones pequeñas donde no se necesita un gran poder de procesamiento de datos. Sin embargo, cuando los requerimientos crecen, se tienen problemas de rendimiento, de concurrencia o de seguridad. No hay punto de comparación entre el rendimiento que ofrece un sistema centralizado y uno basado en redes *LAN*.

Haciendo una analogía para comprender mejor esto, supóngase que existieran dos formas de poder localizar un número de teléfono; una de las formas, la que correspondería al sistema centralizado, sería llamar a una central donde hay varias operadoras especializadas que atienden los llamados, éstas de forma muy rápida localizan el número de teléfono solicitado; la otra forma, que correspondería a la de redes *LAN*, es acudir a un establecimiento donde se tienen directorios telefónicos que no tienen ordenado su contenido y pedir uno de éstos, de manera que uno tendría que buscar hoja por hoja el número de teléfono a localizar.

1.1.3 Sistemas Cliente/Servidor

A finales de los años ochenta se comenzó a popularizar una nueva estrategia para cómputo en red, que dependía más del modelo de programación que de las implantaciones de *hardware*. Este modelo es conocido como *cliente/servidor* y desde cierto punto de vista se basa tomando parte de los dos esquemas anteriores. *Cliente/servidor* es un modelo de programación independiente de la plataforma de *hardware*, donde se tiene un proceso llamado *servidor* que se dedica sólo a atender peticiones y un proceso llamado *cliente* que se dedica sólo a hacer las peticiones al proceso *servidor*. El envío y la respuesta de las peticiones se hace a través de una red de comunicaciones (ver fig. 1.4).

Figura 1.4 Modelo Cliente/Servidor

Aunque ambos procesos pueden correr en la misma máquina, el verdadero potencial es que se tengan en máquinas distintas de forma que se aprovechen el procesamiento, el almacenamiento y la memoria de cada una de las máquinas en las tareas en que cada una de ellas pueda rendir mejor. Por ejemplo, el servidor puede ser una máquina con capacidad de procesamiento alto, con un manejo de *entrada/salida* aceptable que pueda tener control de seguridad y un manejo de concurrencia bueno; por otra parte, el cliente puede ser una máquina donde nos importe más tener un ambiente gráfico aceptable, y donde se tenga la capacidad de ejecutar tareas de escritorio. *Cliente/Servidor* permite ejecutar cada función de una aplicación en el procesador más apropiado para ella y seleccionar tanto el *hardware* como el *software* que mejor se adecuen a las necesidades que se tengan.

1.2 La "orientación a objetos" como tecnología de integración

Un sistema distribuido se puede visualizar como un conjunto de componentes que están en interacción entre sí para lograr un fin común. Lo primero que se tiene que hacer

es identificar el tipo de componente que se necesita, posteriormente definir cómo va a estar integrado y cuál va a ser su función.

Respecto al modelo de objetos se menciona que "el análisis y diseño orientado a objetos modela al mundo como objetos que tienen propiedades, comportamiento y eventos que activan operaciones las cuales modifican el estado de los objetos. Los objetos interactúan de manera formal con otros objetos"[2] Si comparamos el esquema que se tiene de los sistemas distribuidos y el esquema de *análisis/diseño* orientado a objetos, nos pareciera que este último se adecua perfectamente para ser una representación de los sistemas distribuidos (ver fig. 1.5).

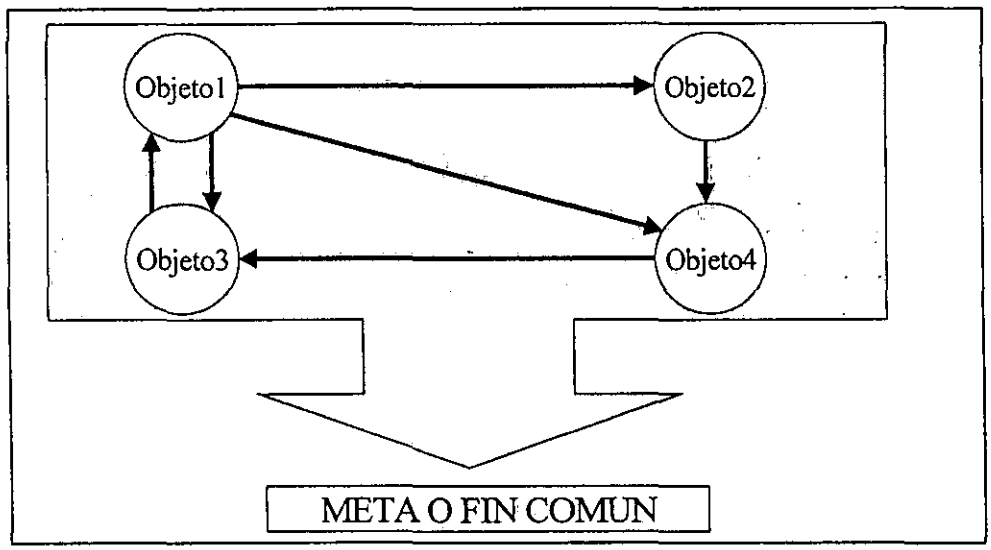


Figura 1.5 Interacción entre objetos

Lo mejor para una buena implantación física de un sistema, es utilizar una metodología de *análisis/diseño* que sea compatible con el paradigma del lenguaje o ambiente de desarrollo a utilizar. Por ejemplo, si se utiliza *análisis/diseño* orientado a objetos, lo más apropiado sería el uso de un lenguaje o ambiente de desarrollo que soporte la programación orientada a objetos, si se utiliza *análisis/diseño* estructurado, lo indicado sería utilizar un lenguaje que sea estructurado. De esta forma el pasar del diseño al desarrollo es casi automático.

Sin embargo, en un sistema distribuido, lo que se presenta con frecuencia es la integración de varios módulos, cada uno de los cuales se puede implantar con herramientas que soportan diferentes paradigmas. Posiblemente algunos necesariamente se

[2] James, M. y James J. Análisis y diseño... p. 10.

tengan que realizar con lenguajes orientados a objetos, algunos otros con lenguajes estructurados, etc. Entonces lo que se necesita es una metodología la cual pueda integrar varios paradigmas.

La orientación a objetos podría ser la tecnología para sistemas de múltiples paradigmas. La principal razón, es por que muchas de las demás tecnologías están integrando el soporte orientado a objetos. Por ejemplo se menciona que "las herramientas de inferencia de la comunidad de inteligencia artificial han evolucionado de estar basadas en reglas ha estar basadas en *marcos de referencia*[3]; un marco es esencialmente, un tipo de objeto y los métodos utilizados por el tipo de objeto son los del procesamiento de reglas"[4]

Otro ejemplo son las bases de datos relacionales actuales, casi todos los gestores de estas bases de datos permiten el uso de tipos de datos binarios, lo que permite almacenar en la base de datos, objetos de cualquier tipo arbitrario. Otra característica de estos sistemas es el uso de *procedimientos almacenados*[5] y *triggers*[6], que se pueden ver como métodos asociados a las entidades.

Los métodos de algún objeto se pueden implementar con técnicas de programación que no necesariamente sean orientadas a objetos, en especial con lenguajes por procedimientos como en el caso de *C* o *PASCAL*. De hecho, se podría afirmar que los métodos de algún objeto se pueden crear con cualquier técnica adecuada. Debido a que la tendencia de todos los paradigmas es incluir la tecnología orientada a objetos, el enfoque orientado a objetos tarde o temprano incluirá a las demás técnicas de programación.

Casos como el de *C++* que es un lenguaje que soporta la programación orientada a objetos y la programación por procedimientos se están repitiendo en nuevas versiones de lenguajes que utilizan otro tipo de paradigmas, como los lenguajes funcionales *LISP* o los de inteligencia artificial como *PROLOG*, todos se pueden incluir dentro de este grupo.

Las herramientas *CASE* también están siendo influenciadas por la orientación a objetos. La mayoría de las herramientas *CASE* se basan ahora en el análisis y diseño de

---

[3] Mas adelante (ver apartado 1.5.1.5) se menciona que un *marco de referencia* es un concepto que sirve para agrupar a elementos generales de un dominio dado, esta definición no es precisamente a lo que menciona el autor en esta cita.

[4] Op. cit. p. 11.

[5] Un procedimiento almacenado es una colección de sentencias *SQL* precompiladas almacenadas en la base de datos.

[6] Un *trigger* es un procedimiento almacenado que se invoca de manera automática cada vez que se realiza una operación de modificación de datos (insertar, actualizar o borrar) en una tabla.

orientación a objetos. Por ejemplo, las herramientas *CASE* para *análisis/diseño* de bases de datos relacionales, se pueden utilizar para representar la estructura estática y las relaciones entre objetos sin ningún problema.

La orientación a objetos por todas estas razones parece ser hasta el momento la respuesta de análisis y diseño para sistemas distribuidos donde precisamente se requiere que haya una integración de varias tecnologías (fig. 1.6).

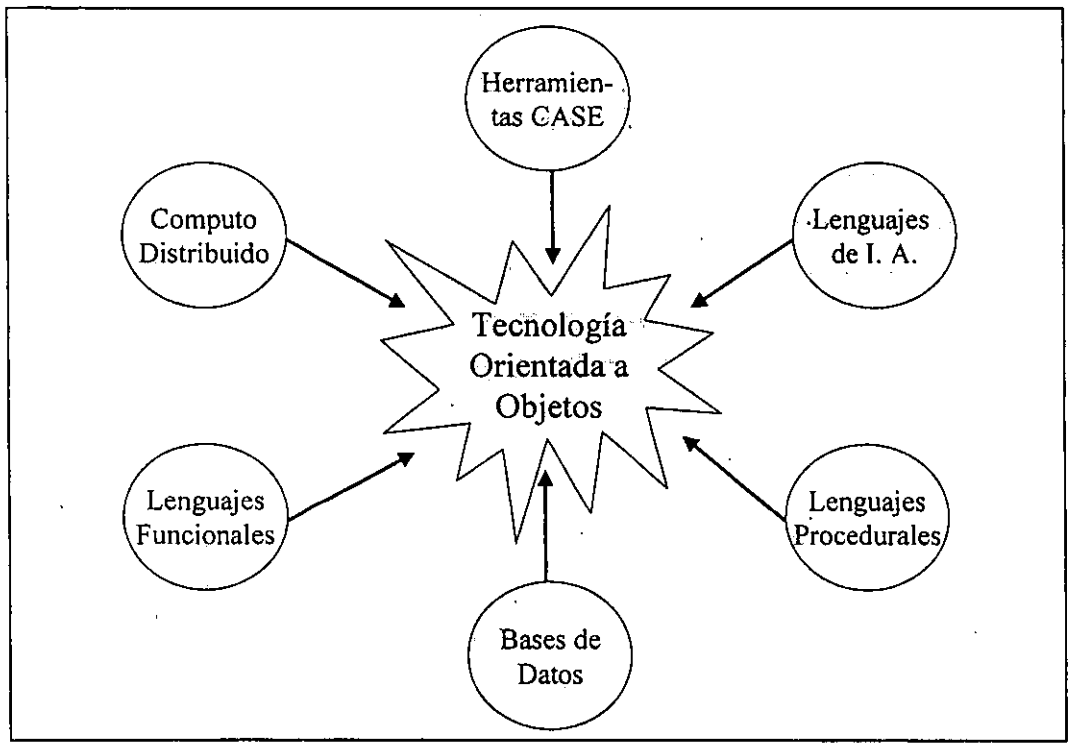


Figura 1.6 Orientación a objetos usando tecnología de integración

1.3 Conceptos básicos de orientación a objetos

Con el análisis y diseño orientados a objetos, lo que se pretende es tener un modelo de objetos en el cual se tengan representados objetos, que de cierta manera mantengan relaciones entre sí que permitan conseguir un objetivo o meta común y resuelvan un conjunto de requerimientos. A continuación se verán con más detalle algunos de los conceptos básicos de este modelo.

1.3.1 Noción de objeto y tipo de objeto

De manera intuitiva se define a un *objeto* como: una idea o noción compartida de algo que puede ser tangible (como por ejemplo: una casa, un árbol, una persona, un lápiz, un

perro, un automóvil, etc.) o intangible (como por ejemplo; una venta, una compra, una reunión, una reservación, una inscripción, etc.). Al respecto también se define a un objeto como "un concepto, abstracción o cosa con límites bien definidos y con un significado a efectos del problema que se tenga entre manos"[7] El que un objeto tenga un significado implica que tiene una definición la cual nos permite identificarlo o reconocerlo.

Por ejemplo, supongamos que se está analizando el proceso de inscripciones de una escuela, se tendría que tomar en cuenta a los alumnos como parte fundamental del sistema y definir que un alumno es una persona a la cual se le proporcionan clases dentro de la escuela, la cual debe proporcionar nombre, edad, domicilio y teléfono para inscribirse en la escuela, eso es definir el *tipo de objeto* "alumno".

Si en determinado momento la persona llamada Pedro López proporciona todas las características antes mencionadas y se le asigna el número que lo identifica dentro de la escuela, entonces él es un objeto del *tipo de objeto* "alumno". *Tipo de objeto* se define entonces como la idea que tenemos del objeto y el objeto es en sí uno de los elementos que pertenecen a ese tipo de objeto (ver fig. 1.7).

Figura 1.7 Objeto y tipo de objeto

1.3.2 Definición de clase, objeto y conceptos relacionados

Los tipos de objeto se podrían definir como categorías de objetos y los objetos como instancias de un tipo de objeto determinado. De hecho los términos objeto e *instancia* en orientación a objetos son términos intercambiables.

Existe otro término para definir a los tipos de objeto que es más conocido en el contexto de orientación a objetos, este termino es el de *clase*. Aunque algunos autores

[7] Rambaugh, J. *Modelado y diseño...* p. 45

sugieren que existen diferencias entre lo que se considera una *clase* y un *tipo de objeto*[8], nosotros los consideraremos también como términos intercambiables. "Una *clase* es lo que representa a un conjunto de objetos que comparten una estructura común y un comportamiento común"[9].

Dos aspectos importantes que deben tomarse en cuenta de esta definición, son saber a qué se están refiriendo las palabras estructura y comportamiento. *Estructura* se refiere precisamente a las "estructuras de datos" que van a formar el objeto; el comportamiento se refiere a las operaciones que puede llevar acabo ese objeto sobre su estructura, la manera de llevar acabo estas operaciones se especifica en lo que se conocen como los métodos del objeto. Un *método* es la definición de una operación que controla la estructura de cierto objeto.

Se puede pensar entonces que los objetos son estructuras de datos y métodos; los valores que tengan las estructuras de datos nos darán el *estado* de los objetos. El *estado* de un objeto se puede definir como las condiciones en que se encuentra la estructura del objeto en determinado tiempo. Entonces el *comportamiento* de un objeto es cómo responderá a los cambios de estado que sufre. Una definición de objeto que involucra estas ideas es que un *objeto* es algo que tiene un estado, comportamiento e identidad.

La "*identidad* significa que un objeto se distingue por su existencia inherente y no por las propiedades descriptivas que puedan tener"[10], es decir, la identidad es una característica que tienen los objetos para garantizar que cada objeto es diferente de otro no importando si es de la misma o de diferente clase.

De alguna forma los objetos necesitan de una solicitud para que hagan algo, la forma en que realicen ese algo, es que se produzca una operación, es decir, se necesita invocar uno de sus métodos. A la solicitud para que se lleve acabo una operación en un objeto se le conoce como *mensaje*. La fig. 1.8 muestra una representación de objetos y una clase plasmando estas ideas.

---

[8] Martin sugiere que el termino *clase* se utiliza en la implantación del sistema. Booch por ejemplo aunque afirma que es indistinto el uso de *clase* o *tipo* sí aclara lo siguiente: "Aunque los conceptos de clase y tipo son similares, se incluyen los tipos como elemento separado del modelo de objetos porque el concepto de tipo pone énfasis en el significado de la abstracción en un sentido muy distinto"[6]

[9] Booch, G. *Análisis y diseño...* p.120.

[10] Booch, G. Op. Cit.

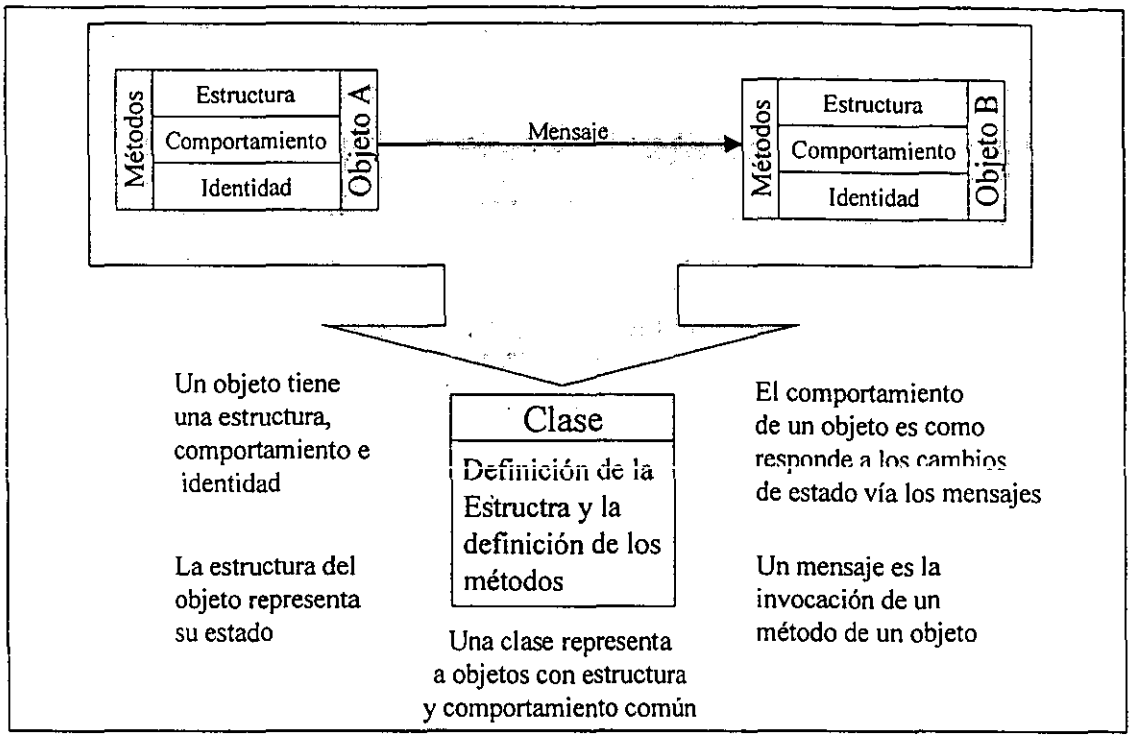


Figura 1.8 Representación de clase y objetos

1.3.3 Encapsulamiento y polimorfismo

Por la forma en que se comunican los objetos, se está protegiendo la estructura interna de cada uno de ellos (datos y métodos), ¿por qué se puede afirmar esto?, la respuesta es precisamente porque cuando queremos que un objeto realice cierta operación, lo tenemos que hacer mediante la invocación de alguno de sus métodos, al cual no se tiene acceso directamente, como tampoco a los datos que pudiera estar cambiando y que son los datos de la estructura de ese objeto. Esta característica recibe el nombre de *ocultamiento* o *encapsulamiento* de información.

Todos los detalles específicos de los datos del objeto y la codificación de sus operaciones (implantación de los métodos) están fuera del alcance de los demás, incluyendo a otros objetos o cualquier otro ente externo. El encapsulamiento es una característica muy importante de los objetos ya que permite la modificación de la implementación de un objeto sin que se tenga que modificar otras partes de las aplicaciones donde se este haciendo uso de ese objeto.

Una de las propiedades esencial para que algo sea considerado como "orientado a objetos" es el concepto de polimorfismo. Polimorfo sugiere tener varias formas para algo

común. El polimorfismo que generalmente se considera tener en orientación a objetos, es el que se puede tener a través de los métodos el cual consiste en tener una operación que adopta varias formas de implantación, es decir, tener distintas definiciones de un método que va a realizar el mismo propósito operativo.

Generalmente el polimorfismo se va a dar cuando se tengan diferentes clases, las cuales tiene un mismo método en común pero que precisamente por ser diferentes cada una de ellas, va a realizar esa operación de forma distinta, por lo tanto, cada una va a tener diferente código en la definición de ese método que realiza la misma operación. De forma más concreta, el *polimorfismo* es responder de manera diferente a un mismo mensaje por clases que son distintas unas de otras. En la figura 1.9 se representan estas dos características que tienen los objetos por vía de la definición y forma de invocación de sus métodos.

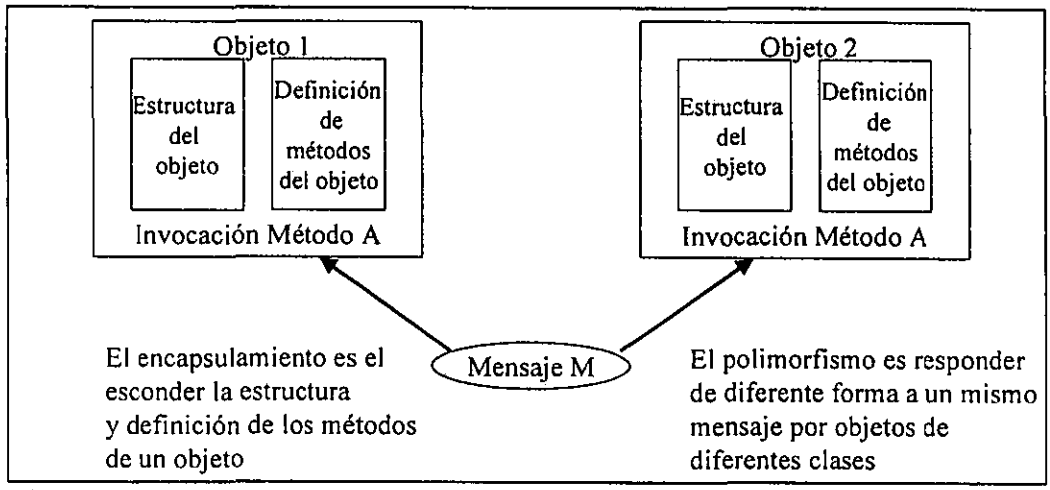


Figura 1.9 Encapsulamiento y Polimorfismo

1.3.4 Herencia

La herencia es la propiedad más importante que se tiene en orientación a objetos después del concepto de clase. Básicamente la *herencia* es una relación entre clases que permite que una clase pueda heredar parte de la estructura y parte del comportamiento de otra clase. Cuando la clase hereda los datos y métodos de una sola clase padre, se le conoce como *herencia simple*; cuando la clase hereda de más de una clase padre, se le conoce como *herencia múltiple* porque se está heredando la estructura y el comportamiento de múltiples clases. Lo importante de la herencia es que las clases que

heredan pueden tener sus propios métodos y estructura de datos además de los que heredan de su(s) clase(s) padre(s).

Permitir la herencia entre clases tiene repercusiones positivas para todo lo que se refiere a reutilización de código, ya que las estructuras de datos y comportamiento que se tengan en común en diferentes clases se escribe sólo una vez y se crea una clase que lo contenga de tal forma que las demás clases sólo lo heredan sin necesidad de volverlo a escribir o a definir. De igual manera, el hacer bibliotecas de clases implica la construcción de un número muy grande de clases, sin embargo, si se desarrollan utilizando el concepto de herencia, gran parte del código solo se escribirá una vez aunque se tengan muchas y de diferentes clases. La fig. 1.10 representa la herencia entre clases.

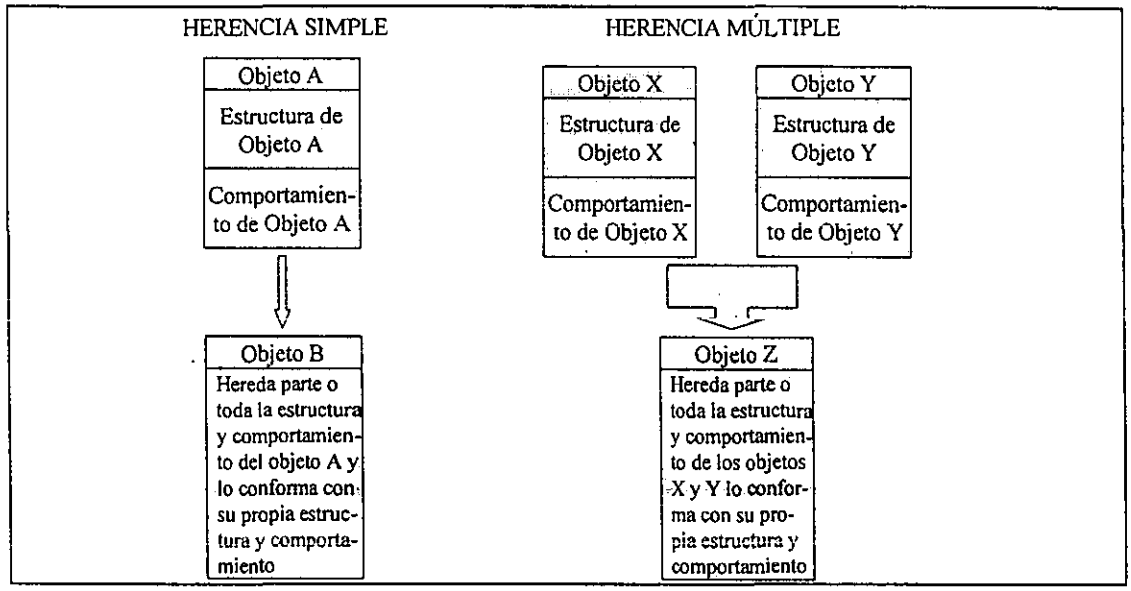


Figura 1.10 Herencia entre clases

Los conceptos de herencia y polimorfismo hacen que se presenten aspectos inesperados y contradictorios. Con la herencia se observa que se puede heredar el comportamiento, sin embargo, por medio del polimorfismo se puede redefinir ese comportamiento, de hecho, esto nos puede llevar a que se tengan clases que no tengan definido ninguno de sus métodos, ya que la implantación de éstos sólo se hará en las "clases hijas" por medio del polimorfismo, estas clases por lo tanto nunca podrán generar objetos y se les conoce como *clases abstractas*. El concepto de clase abstracta es útil, pues

nos permite tener un nivel de abstracción muy alto, pero puede ser muy arriesgado si se usa demasiado, puesto que no estamos reutilizando código en lo absoluto.

1.4 Fundamentos de la teoría de objetos

De alguna forma lo que pretende con el análisis y diseño orientado a objetos es realizar una clasificación de objetos, lo que nos llevará como consecuencia a contar con un modelo correcto que cumpla con los requerimientos. De acuerdo a lo que se mencionó (ver apartado 1.3.1) un tipo de objeto se podía considerar como la idea que se tiene de un objeto, es decir, el concepto que se tiene del objeto. El realizar una clasificación es una manera de ordenar los conceptos conforme a ciertas relaciones que existen entre ellos[11]. A continuación veremos algunos conceptos que nos ayudan a realizar una clasificación de objetos.

1.4.1 Abstracción

La *abstracción* es un mecanismo de razonamiento que permite fijar nuestra atención en aquellos aspectos que son comunes y que nos interesan de ciertos objetos, desechando los que no lo son. Esto parece algo complicado pero como muchos autores afirman, en realidad la abstracción es una herramienta muy eficaz que nos permite enfrentar la complejidad de las cosas.

Por ejemplo, cuando vamos a una biblioteca nos encontramos en primera instancia con un gran número de libros que son diferentes entre sí, sin embargo, gracias a la abstracción, pronto nos podemos percatar de que los libros están agrupados y ordenados por una característica que tiene en común que es el tema del que tratan; así encontramos libros de historia, geografía, computación, etc. La abstracción permite que sea más fácil encontrar el libro que estamos buscando. En la teoría de objetos, la abstracción es muy útil ya que nos permite identificar tipos de objetos, detectando aquellas características comunes de un conjunto de objetos que se está analizando (fig. 1.11).

[11] Al respecto Booch menciona lo siguiente: "La clasificación es relevante para todos los aspectos del diseño orientado a objetos. La clasificación ayuda a identificar jerarquías de generalización, especialización y agregación entre clases. Reconociendo los patrones comunes de interacción entre objetos, se llegan a idear mecanismos que sirven como alma de la implantación. La clasificación también proporciona una guía para tomar decisiones sobre modularización."[9]

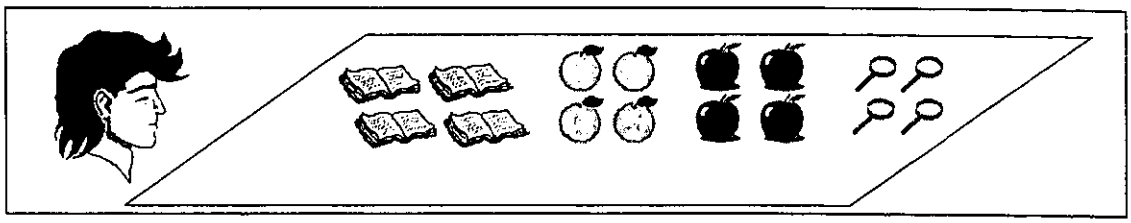


Figura 1.11 Abstracción

1.4.2 Composición

La *composición* o *agregación* es un concepto muy útil que permite formar un concepto a partir de varios más sencillos; desde una perspectiva de objetos, sería poder visualizar a una o más tipos de objetos (que se pueden considerar hasta cierto punto sencillos) como partes de un tipo de objeto más complejo. La composición reduce la complejidad al tratar muchas cosas como una sola, por ejemplo, podemos tratar al cuerpo humano como una unidad, aunque en realidad se trate de una configuración de varios componentes.

Por otra parte, si se ve desde el punto de vista opuesto, es decir, realizando una descomposición, también se presenta como forma de enfrentar la complejidad. Por ejemplo, si tenemos que estudiar a la célula, es más factible que lo hagamos con mayor éxito si estudiamos cada uno de las partes que la forman por separado.

Usar la composición, establece un cierto tipo de relación entre la clase que va a ser formada y las clases que se van a usar para ello. Estas relaciones son las que se conocen como relaciones del tipo "*parte de*". Por ejemplo las siguientes clases: llantas, carrocería, chasis, puertas motor y sistema eléctrico, son *parte de* la clase automóvil (ver fig. 1.12).

Por lo general este tipo de relaciones implican que la clase compleja que se está construyendo va a ser dependiente (ver dependencia funcional apartado 1.4.4) de las clases utilizadas para realizar este propósito, sin embargo, esto no es una regla.

La composición por lo tanto, va a ser una herramienta que nos ayuda a definir jerarquías de clases, donde la relación que se tenga entre un nivel y otro es de la forma: "una clase es *parte de* otra clase".

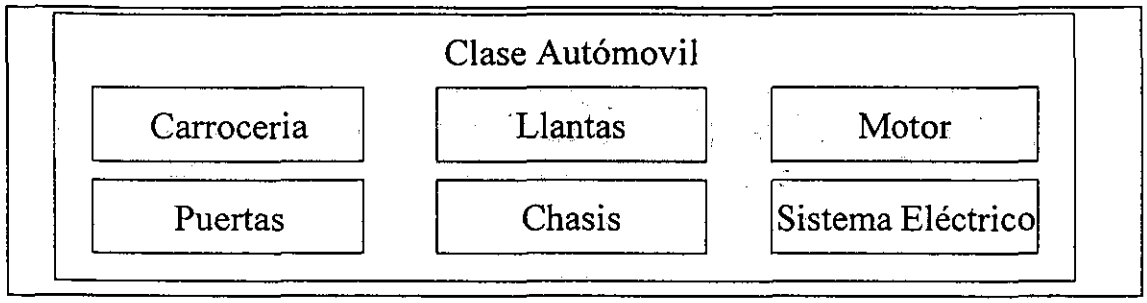


Figura 1.12 Composición

1.4.3 Generalización/especialización

La *generalización* es lo que nos permite definir clases, éstas contienen lo común y esencial de otras clases, de forma que las comprenden a todas, es decir, formar un concepto más general a partir de otros más específicos, tomando de éstos las cosas que tienen en común.

Por ejemplo, hombre y mujer, se podrían catalogar como clases específicas, las cuales quedarían comprendidas dentro de una clase más general que pudiera ser la clase de los seres humanos, a su vez, esta clase también podría quedar dentro de otra clase más general, por ejemplo la clase de los seres vivos.

La *especialización* por el contrario, es lo que nos permite definir clases, las cuales tienen ciertas características particulares que las hacen diferentes de las demás clases con las que comparte ciertos aspectos en común, es decir formar un concepto más específico a partir de uno más general.

Tomando el ejemplo anterior, pero con este enfoque, tendríamos que dentro de la clase seres vivos, hay una clase más específica que es la clase de los seres humanos; a su vez, dentro de ésta existen otras clases más específicas, las cuales son la clase de los hombres y la de las mujeres. Tanto la generalización como la especificación, van a denotar un tipo especial de relación entre el tipo de objeto más específico y el tipo de objeto más general dentro de la jerarquía que se halla formado. Estas relaciones son las que se denotan como relaciones del tipo "*es un*" y no son otra cosa mas que la herencia entre clases. Por ejemplo un hombre *es un* tipo de ser humano y un ser humano *es un* tipo de ser vivo. A las clases más globales se les llama *superclases* y a las clases más específicas o "clases hijas" se les conoce como *subclases*[12] (ver fig. 1.13).

[12] Una característica importante de las instancias de una subclase es que una instancia de una subclase es

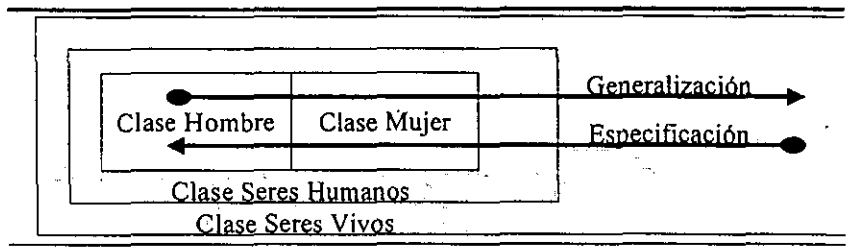


Figura 1.13 Generalización/Especificación

La generalización y la especialización nos permiten tener jerarquías de objetos donde la relación que se tiene entre un nivel y otro es la herencia, lo que implica que la subclase hereda parte o toda la estructura y el comportamiento de la superclase. Sin embargo, la subclase puede redefinir su estructura y comportamiento (ver sección 1.3.4).

1.4.4 Relaciones

Las *relaciones* son los vínculos que se pueden presentar entre dos o más clases y pueden ser de tres tipos diferentes, dos de los cuales (relaciones "*parte de*" y la herencia "*es un*") como ya se constató, definen una jerarquía de clases y niveles de especialización entre las clases (ver apartados 1.4.2 y 1.4.3).

La otra forma de relación entre clases se presenta cuando es posible realizar una *conexión lógica* entre dos clases que aparentemente son independientes entre sí. Por ejemplo, si se tiene la clase "alumno" y la clase "asignatura", aparentemente no tiene nada en común, no se pueden englobar en una clase mas general ni tampoco forman parte de una clase más compleja, sin embargo, sí podemos afirmar que los alumnos se inscriben a una o más asignaturas.

Esta pequeña oración es la que nos indica que efectivamente existe una relación entre las clases. De hecho este tipo de relaciones son las más comunes y dependen de cómo esté definido el problema[13].

simultáneamente una instancia de todas sus clases antecesoras, es decir, que al momento de crearse un objeto de una subclase también se crean objetos de las clases padres que pudiera tener ese objeto.

[13] Booch llama a este tipo de relación una asociación y al respecto menciona; "las asociaciones son el tipo más general de relación entre clases pero también el de mayor debilidad semántica. A medida que se continúa el diseño y la implementación, se refina a menudo estas asociaciones débiles orientándolas hacia una de las otras relaciones de clase más concretas."[10]

Para las relaciones entre clases, siempre hay que denotar ciertas cuestiones que son importantes que nos indican y describen de manera precisa el vínculo que existe entre las clases, en primer lugar está la *cardinalidad* o *multiplicidad*, que define el número de instancias que puede corresponderse entre las clases que tiene la relación.

Por ejemplo, cuando se dijo que un alumno se inscribe en una o más asignaturas, significa que una instancia de la clase "alumno" puede corresponder con una o más instancias de la clase "asignatura". Supóngase que se tiene las siguientes instancias de asignaturas: matemáticas, español, biología, geografía y civismo; y sea Juan López una instancia de la clase "alumno", entonces Juan López se puede inscribir a la asignatura de matemáticas, a la de español, a la de geografía etc.; de manera análoga, para una asignatura varios alumnos pueden estar inscritos, es decir, en matemáticas pueden estar inscritos Juan López, Daniel Rodríguez, Alejandra Sánchez, etc. En esta relación se dice que hay una cardinalidad muchos a muchos, puesto que cada instancia de las clases puede corresponder con una o más instancias de la otra clase. Los casos más generales de cardinalidad que se pueden dar entre dos clases son: *uno a uno*, *uno a muchos* y *muchos a muchos*.

Por ejemplo la relación entre marido y esposa sería de uno a uno, la relación entre venta y producto sería de uno a muchos porque un producto sólo se puede registrar en una venta, y una venta sí puede constar de muchos productos. La cardinalidad dependen de suposiciones y de la forma en que este definido el problema.

Dentro de las relaciones también existen condiciones de *dependencia funcional*, esto es cuando alguna instancia de una clase necesita forzosamente de la existencia de otra instancia de la clase con la que tiene la relación para poder existir. Para ilustrar esto volvamos el ejemplo de producto y la venta, una venta forzosamente tiene que existir si existe algún producto para venderse, una instancia de producto, por otro lado, no necesita de ninguna existencia de alguna instancia de venta para que exista. La fig. 1.14 muestra la representación de una relación entre clases.

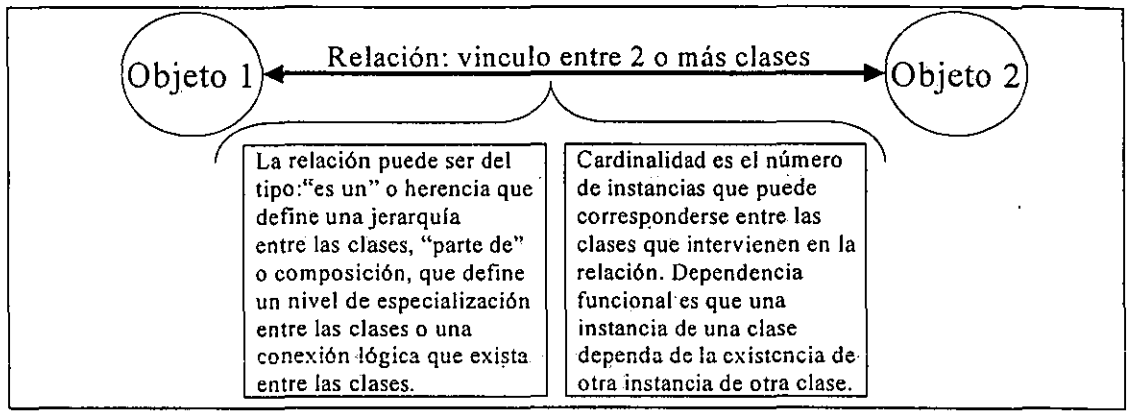


Figura 1.14 Relación entre clases

1.5 Método de análisis/diseño *orientado a objetos*

Existen actualmente un sin número de metodologías orientadas a objetos, de las cuales no se puede afirmar que alguna de ellas sea la mejor o la peor, sin embargo, sí se puede observar puntos de concordancia entre ellas, o puntos que las hacen sobresalir, lo cual nos indica que estos conceptos están siendo aceptados por una mayoría.

En este apartado trataremos de explicar algunas de estas ideas de forma que al final podamos concretizar que efectivamente este tipo de análisis se puede aplicar de hecho a cualquier tipo de sistemas incluyendo a las aplicaciones distribuidas. También se podrá constatar que se utilizan varias estrategias que son utilizadas en otros tipos de análisis y diseños orientados a otros paradigmas; comprobando lo dicho anteriormente de que la orientación a objetos es una tecnología de integración. Muchas de estas ideas son esencialmente de las metodologías de Booch, Rumbaugh y Jacobson, aunque también se tratan algunos conceptos de las metodologías de James Martin y de la de Jordon/Codd. Se utiliza la notación conocida como *UML* (*Unified Metodology Lenguague*) propuesta por los tres primeros autores.

1.5.1 Modelo de requerimientos

Varios autores están de acuerdo en que para realizar el análisis, se necesita definir claramente el problema que se supone se trata de resolver, por ejemplo Rambaugh (*OMT: Object Oriented Technology*) afirma que "el análisis comienza con la definición de un problema generado por clientes y posiblemente por los desarrolladores"[14]. De hecho, se

[14] Booch, G. Op. Cit. p. 180.

podría decir que el modelo de requerimientos en la mayoría de los casos sirve para definir las tareas del sistema y especialmente para la delimitación del problema. Sin embargo, existen otras alternativas dentro del análisis orientado a objetos, que sirven para formalizar más la obtención de requerimientos del sistema, entre las cuales destacan el *análisis de casos de uso* y el *análisis del dominio*. Estas dos técnicas son importantes por que son presentadas de manera más formal y como parte esencial del modelo de requerimientos del sistema en la metodología de Jacobson (*Objectoty*).

El *análisis del dominio* de un sistema consiste en identificar los componentes más comunes y generales dentro de un dominio dado. Jacobson quizás es el que más a fondo remarca la idea de obtener un modelo de requerimientos donde en primer lugar se tenga un esquema completo de los requerimientos que nos delimite el problema y que después nos lleve ha obtener el dominio del problema.

El obtener el dominio del problema, dicta distintas políticas a seguir, la primera sería el proponer que el sistema se empiece de cero; la segunda, proponer que se utilice partes o todo lo que ya existe de un sistema anterior, o por último, una estrategia que consiste en basarnos en el funcionamiento de sistemas parecidos tomando lo que más se adecue al funcionamiento de sistema que se está analizando. Jacobson propone una herramienta que precisamente nos permite esquematizar el conjunto de requerimientos y es lo que se conoce como los *casos de uso*.

1.5.1.1 Casos de uso

Un *caso de uso* es "un camino específico de uso del sistema utilizando parte de la funcionalidad de éste[15]. Cada *caso de uso* se puede ver como una serie de eventos iniciados por un usuario que está adoptando un determinado rol en el sistema. El rol que adopta el usuario se define con el término de *actor*. Se puede decir que un *caso de uso* es una secuencia de transacciones llevadas acabo entre un actor y el sistema. Supongamos que queremos modelar utilizando *casos de uso*, por ejemplo, el funcionamiento de un teléfono celular; primero podríamos preguntarnos ¿quiénes utilizan los teléfonos celulares? para identificar a los actores; después, determinar para cada uno de éstos actores cuáles son las tareas que realizan, esto nos llevaría a identificar *casos de uso*, por ejemplo, podríamos determinar el siguiente: "hacer llamada". Todo esto se puede

representar mediante *diagrama de casos de uso*, donde se tiene actores *casos de uso* y la interacción entre éstos dos mediante flechas (ver fig.1.15).

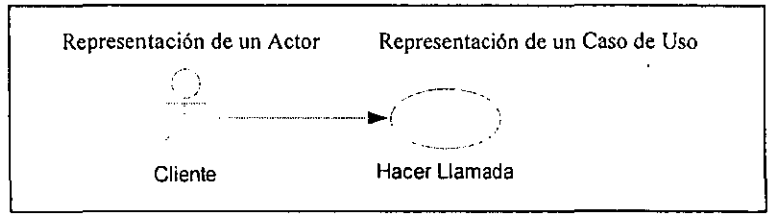


Figura 1.15 Diagrama de "caso de uso"

De forma similar se podrían encontrar otros *casos de uso* para el sistema de teléfonos celulares. Lo importante es que una vez que se tenga identificados para cada uno hay que desarrollar una especificación (lo que nos determina la descripción del conjunto de requerimientos), la cual podría incluir la siguiente información:

1) todos los detalles importantes acerca del *caso de uso*,
2) los pasos críticos entre la interacción del actor y el sistema,
3) información acerca del actor,
4) una lista de precondiciones que deben de conocerse para proveer la funcionalidad del sistema,
5) una división de responsabilidades entre el actor y el sistema, y
6) mostrar algunos requerimientos técnicos que se necesitan si se trata de modelar algún *servicio.*

1.5.1.2 Modelo de requerimientos usando "casos de uso"

El modelo de requerimientos que propone Jacobson está integrado por tres partes, el modelo de *casos de uso*, la descripción de las interfaces de usuario y el modelo de objetos de dominio del problema. El modelo de *casos de uso* especifica la funcionalidad del sistema desde una perspectiva de usuario. Este modelo utiliza a los actores para representar los roles que los usuarios del sistema pueden desempeñar y los *casos de uso* para representar que es lo que los usuarios deben ser capaces de hacer con el sistema.

Lo primero que se debe realizar es identificar a los actores del sistema, una forma sencilla de hacer esto, es ver a los actores como todo aquello que es externo al sistema y que necesita intercambiar información con él. Los actores pueden modelar a usuarios

[15] Jacobson, Ivar; et. al. *Objet-oriented software engineering.* p. 162.

humanos, pero también a otros sistemas que necesiten comunicarse con el sistema. Una vez definidos los actores, hay que definir de qué manera van a interactuar con el sistema, en otras palabras, definir los *casos de uso* que cada actor se supone va a realizar. Para identificar los *casos de uso*, Jacobson propone hacer preguntas como las siguientes:

1) ¿Cuáles son las principales tareas de cada actor?,
2) ¿El actor va a tener que leer/escribir/cambiar alguna información del sistema?,
3) ¿El actor va a tener que informar al sistema si hubo cambios en el exterior? y
4) ¿El actor desea ser informado acerca de los cambios inesperados?.

Por lo general este tipo de preguntas va a definir un *caso de uso* si se pueden contestar de manera afirmativa. Sin embargo, el determinar los *casos de uso*, puede en muchas ocasiones convertirse en un proceso interactivo que nos lleve a definir nuevos *casos de uso* a partir de un *caso de uso* que se esté analizando en detalle. Jacobson menciona que "en un *caso de uso* se tiene un propósito o curso básico a seguir o desarrollar, pero también se tiene variantes y errores en ese curso básico que se pueden describir como cursos alternativos a seguir o desarrollar"[16].

Para poder modelar este tipo de comportamiento, se introduce el concepto de asociación extendida, la cual muestra cómo la descripción de un *caso de uso*, puede estar definida, compuesta o extendida por otro *caso de uso* (fig. 1.16).

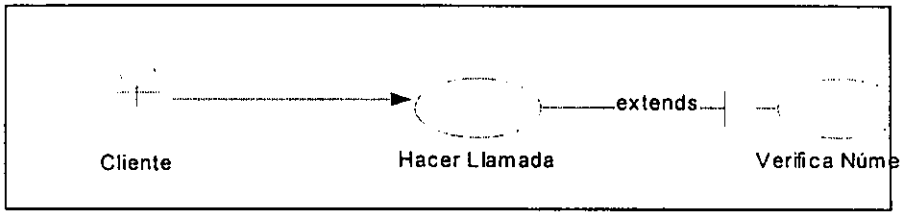


Figura 1.16 Diagrama de "caso de uso" con asociación extendida

Los *casos de uso* definidos por la asociación extendida se pueden presentar en casos como los siguientes:

1) Para modelar partes opcionales de un *caso de uso*.
2) Para modelar algún curso alternativo y complejo que puede seguir un *caso de uso* y el cual ocurre rara vez.
3) Para modelar diferentes casos de uso que pueden estar dentro de otro *caso de uso*.

[16] Coad, P. Y Yourdon, E. *Object- Oriented...* p. 162

1.5.1.3 Diagramas de Iteración

Cada *caso de uso* tiene un nombre que lo identifica y una descripción o explicación de lo que trata de modelar. Esta descripción puede ser breve o constar de varias etapas o pasos a seguir. Jacobson propone que en el modelo de requerimientos la descripción sólo debe hacerse con palabras. Sin embargo, existe un diagrama para plasmar las actividades de los *casos de uso*, es decir un diagrama que indique cómo se llevan acabo las tareas en un *caso de uso* específico. Estos diagramas se conocen como *diagramas de iteración* u otros autores como Rambauugh los han llamado *diagramas de estado*[17]. Rambauugh menciona que sirven para representar un *escenario*. Un *escenario* es una secuencia de sucesos entre un agente externo y un componente del sistema, por lo tanto, los diagramas de iteración sirven para modelar los sucesos de los *casos de uso*.

Los diagramas de iteración contienen actores, objetos y mensajes. Los actores se representan de la misma forma que en los *diagramas de casos de uso*, los objetos se representan con rectángulos con un nombre para identificarlos, de los cuales, se dibujan unas líneas verticales que son las líneas de seguimiento de sucesos y los mensajes son flechas verticales que unen a las líneas de seguimiento de sucesos que pueden ir del actor a los objetos o viceversa, o de un objeto a otro objeto y representan la comunicación entre actores y objetos, y entre dos objetos dentro de un *caso de uso* (fig. 1.17).

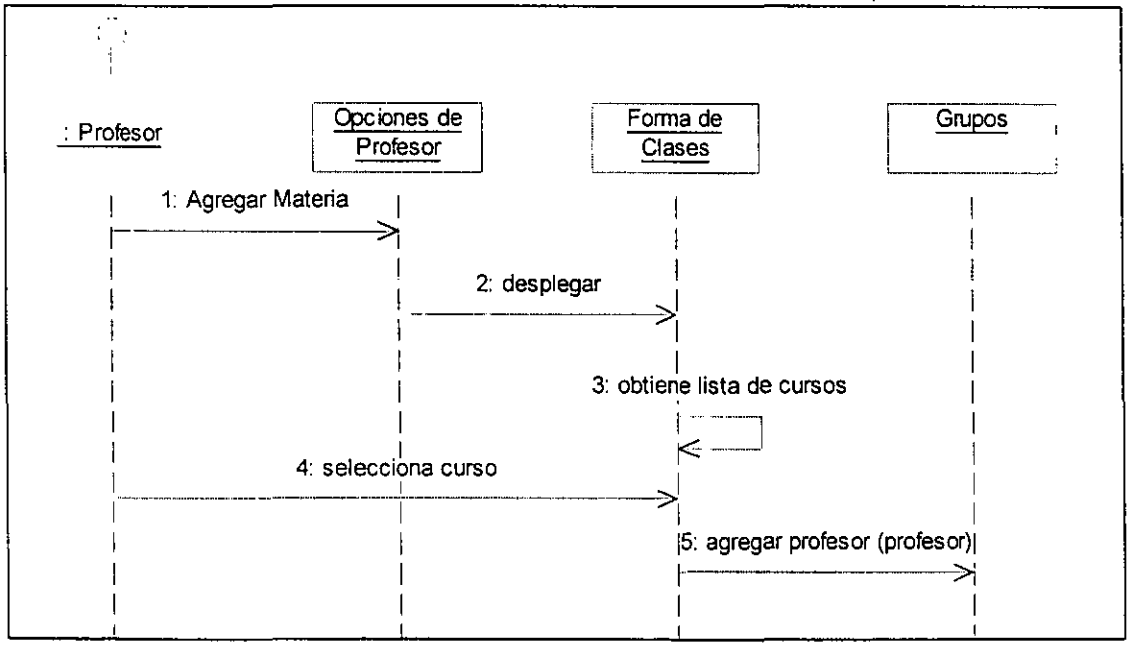


Figura 1.17 Diagrama de Iteración

[17] En el apartado 1.5.2.6 se definen otro tipo de diagramas conocidos como *diagramas de transición de estados*, que son diferentes a los diagramas que Rambauugh propone y también son conocidos como *diagramas de estados*.

1.5.1.4 Análisis del dominio del problema

Finalmente, tomando como base toda la información recabada, se define el *modelo de objetos del dominio*, el cual, su principal propósito es proporcionar un modelo lógico formado por una serie de objetos generales (objetos del dominio), los cuales tiene una relación directa en el ambiente de la aplicación donde el sistema necesita manejar información acerca de ellos. Este modelo captura todos los conceptos principales y fundamentales que se necesitan en la aplicación. Un resultado importante de realizar este modelo es que nos ayuda a identificar de manera general el tipo de sistema que se quiere realizar. "Un análisis de dominios puede conducir al desarrollador a una comprensión de las abstracciones y mecanismos principales"[18]. Un concepto que se puede aplicar a este tipo de modelos encaminados a establecer el tipo de sistema que se está analizando, es lo que se conoce como los marcos de referencia (*frames*) los cuales se explican en el siguiente apartado.

1.5.1.5 Marcos de referencia (*Frames*)

Un *marco de referencia* (*frame*) nos permite representar componentes generales para un dominio particular dentro de un sistema. Ejemplos de lo que se puede representar en un marco de referencia son: las principales tareas a realizar en un sistema, las entidades más representativas del sistema, los módulos que integran el sistema, etc. (ver fig. 1.18).

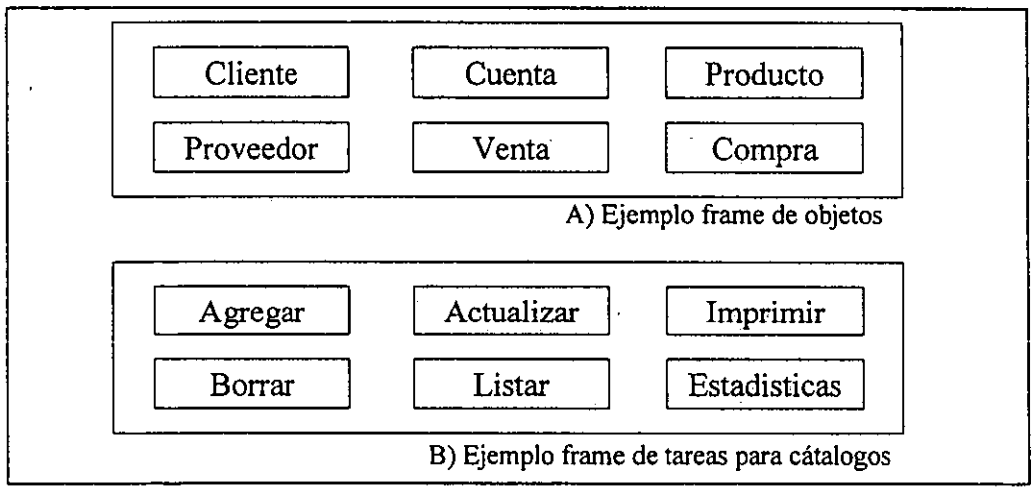


Figura 1.18 Ejemplos de marcos de refernecia (Frames)

[18] Cooch, G. Op. Cit. p. 180

Por lo tanto, con los marcos de referencia se puede representar cualquier elemento general de un sistema, lo cual es muy útil si se quiere conocer el dominio del sistema. Identificando estos elementos se puede tener una comprensión de los mecanismos importantes que van a servir para todo el desarrollo del sistema. Andleigh y Gretzinger[19] proponen utilizar esta técnica para prácticamente representar todo tipo de abstracciónes; localizaciones, actividades, objetos, vistas, relaciones entre objetos, herencia entre objetos, etc. Aunque es cierto que puede ayudar mucho a identificar los conceptos principales, un marco de referencia no debe ser usado para representar alguna idea específica, ya que como se mencionó, está enfocado a conceptos generales, no a particulares.

Con los *casos de uso* se pueden obtener una descripción de requerimientos de un sistema, después se define el modelo de objetos de dominio, el cual queda descrito a través de marcos de referencia, lo que finalmente nos llevaría a tener un modelo para comenzar con el análisis formal del sistema. De hecho, se podría considerar al modelo de requerimientos como parte del análisis de un sistema, sin embargo, al análisis lo trataremos como un modelo aparte, esto porque el modelo de requerimientos sólo es considerado por Jacobson, mientras que otros autores suponen que ya se tiene una descripción inicial del sistema y a partir de ahí comienzan a desarrollar su metodología.

1.5.2 Análisis (modelo lógico del sistema)

El análisis del sistema consiste en determinar un modelo lógico que describa lo que realiza el sistema. Un modelo lógico, es una representación donde no importa la implementación de cómo se van a llevar acabo las tareas que se describen, ni tampoco detalles técnicos de cómo van a estar definidas. En el análisis lo que se busca es determinar de forma clara, concisa y completa, qué va a hacer el sistema no importado y cómo le va a hacer. Por lo tanto, el resultado del análisis orientado a objetos, debe de ser un modelo en el cual se tengan identificados cada uno de los objetos que intervienen en el sistema, así como las relaciones entre ellos y las tareas que estos objetos van a realizar, de esta manera el análisis cumpliría con su objetivo.

Existen varios puntos de convergencia entre las metodologías en cuanto al análisis una vez que se tiene definido el conjunto de requerimientos del sistema. Rumbaugh propone que una vez que se tiene la definición del problema se pasa a la definición del

[19] Gretzinger, M. *Distribted*... pp. 134-143.

modelo de objetos, el cual es un modelo que muestra la jerarquía de clases encontrada y las relaciones que tienen estas clases. Por su parte, Booch propone que una vez terminado el análisis de dominio del problema, se continúe con un proceso de desarrollo repetitivo que consiste en primera instancia, en identificar clases y objetos. Jacobson propone que una vez que se definió el modelo de requerimientos, hay que mapear directamente de los *casos de uso*, a los objetos que intervienen en el sistema, determinando las relaciones que existen entre ellos. Martin propone dentro de lo que él llama el análisis de la estructura de un objeto, definir los tipos de objetos y las relaciones que tienen. Coad y Yourdon proponen que se tiene que identificar los objetos pero al mismo tiempo sus atributos, métodos y relaciones.

1.5.2.1 Identificación de clases y objetos

Identificar clases y objetos, al parecer es el común denominador en casi todas las metodologías; sin embargo, existen diferentes formas de llevar acabo esta tarea. Comenzaremos con la forma clásica que nos dice que de nuestro conjunto de requerimientos se identifican los sustantivos y verbos que aparecen, los cuales serán candidatos a clases, después, mediante un proceso de refinamiento, se eliminan aquellas que no son esenciales en el dominio del problema. Otra forma clásica es la que manejan algunos autores como Coad y Jourdon[20] la cual consiste en determinar a los objetos de las distintas fuentes que se deriven de los requerimientos del problema, por ejemplo, de entidades que representan lugares, personas, eventos, otros sistemas, etc., éstos serán de igual manera candidatos a clases y mediante un proceso similar de refinamiento se eliminan las que no son relevantes. El análisis del dominio puede ser muy útil para identificar las clases utilizando estos dos métodos.

Un enfoque no tan dependiente del análisis del dominio del problema es el análisis de comportamiento, que no es otra cosa que realizar una clasificación aplicando el concepto de abstracción en el comportamiento de objetos que se estén analizando, de esta forma, mediante el agrupamiento de objetos por comportamiento y características similares, se determinan las clases. Otro enfoque es el presentado por Jacobson, éste nos dice que de los *casos de uso* se van a tener que mapear a objetos[21], los cuales van a ser de tres tipos

[20] Coad P. y Jourdon, E. Ob. Cti. p. 62.
[21] Jacobson, I. Op. Cit. pp. 174-195.

diferentes, objetos de tipo entidad, de tipo interface o de tipo control. Los *objetos de tipo interface* son los que aparecen en los *casos de uso* cuando se describe una interacción entre el sistema y los actores, típicamente cuando hay un actor comunicándose con un *caso de uso*. Los *objetos de tipo entidad* son los encontrados en el análisis de dominio. Finalmente los *objetos de tipo control* u *objetos de control*, van hacer por lo regular funcionalidades de uno o algunos *casos de uso*, que separan a los objetos de tipo entidad de los objetos de tipo interface. Existen otros métodos para la identificación de objetos que no se mencionan, sin embargo, Booch[22], da un amplio tratamiento a este tema.

Para cada objeto que se haya identificado le corresponderá una clase. Las clases se representan mediante *diagramas de clases*, utilizando la notación *UML*, las clases se representan mediante rectángulos con el nombre de la clase escrito en la parte superior del rectángulo (fig. 1.19). Esta representación de las clases puede ir cambiando cuando se integren otros elementos a las mismas como se verá posteriormente.

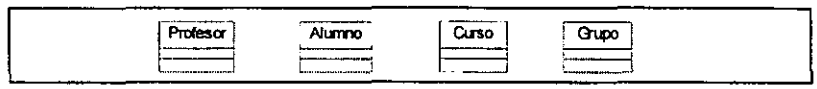


Figura 1.19 Representación de clases

1.5.2.2 Definición de paquetes en el modelo lógico

Un *paquete* es una forma de agrupar elementos de modelado. Un paquete en el *modelo lógico* puede estar integrado por clases o por otros paquetes. Los paquetes fueron propuestos por Jacobson[23] como una forma de representar subsistemas[24] y son incluidos como elementos de modelado del lenguaje *UML*, de hecho los paquetes en el lenguaje *UML* se pueden hacer de *casos de uso*, de clases y posteriormente maparse al *modelo de componentes*[25]. En especial, el poder hacer paquetes de *casos de uso* y de clases, permite una forma natural de representar los marcos de referencia (*frames*), como se mencionó un marco de referencia es una forma de representar componentes generales del sistema, entonces con los paquetes precisamente se esta representando eso. Con *UML*, el

[22] Booch, G. Op. Cit. pp. 172-191.
[23] Jacobson, I. Op. Cit. p. 196.
[24] Los *subsistemas* se mencionan mas adelante (ver apartado1.5.3.4) en este capítulo, y aunque se comparte la idea de Jacobson de que sirven para agrupar varios componentes, se considera como un concepto que se utiliza solo en el diseño y no en el análisis.
[25] *El modelo de componentes* es uno de los cuatro modelos que se puede representar con *UML* y sirve para representar los componentes físicos del sistema.

representar un paquete de *casos de uso* no tiene notación gráfica, pues es solo una lista de los *casos de uso* que forman el paquete, solo hay que asignarle a esa lista un nombre. Un paquete de clases, se representa con un rectángulo, el cual tiene una pestaña en la parte superior de alguna de las esquinas y dentro del rectángulo se especifica el nombre para ese paquete (ver fig. 1.20).

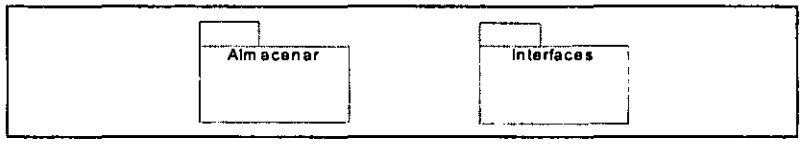


Figura 1.20 Representación de paquetes en el modelo lógico

1.5.2.3 Identificación de atributos, métodos y relaciones

Se definió que las clases están integradas por estructuras de datos y métodos (ver apartado 1.3.2). Estas estructuras de datos están formadas por características propias de un objeto, por ejemplo, la clase "alumno" tiene como características el número que lo identifica dentro de la institución, su nombre, su dirección, fecha de ingreso, etc. A estas características se les conoce como *atributos* y están inmersos en el conjunto de requerimientos. Se identifican cuando se realiza el análisis de la descripción de los *casos de uso* y sus *diagramas de iteración*. Los *atributos* son adjetivos de un objeto y se pueden encontrar o definir en tareas que indiquen el almacenamiento de información, por lo general esta información se puede ver como un conjunto de atributos. Sin embargo, los atributos pueden encontrarse en etapas posteriores como lo son el diseño e implementación, en ambas se identifican los atributos, por cuestiones operativas o cuando se necesita de un atributo especial.

Se definió también que un método es una implementación de una operación que realiza un objeto, el cual consiste en modificar la estructura de datos del objeto, es decir, los atributos (ver apartado 1.3.2). La identificación de los métodos de una clase se hace utilizando los *diagramas de iteración* asociados a los *casos de uso*. Las tareas por realizar definidas en los *diagramas de iteración* y en la descripción de los *casos de uso* en la mayoría de los casos, se convertirán en métodos asociados a algún objeto. Si en el *diagrama de iteración* se tiene una flecha que llega a un objeto, muy probablemente indique la definición de un método que realice lo que indica el mensaje asociado a la

flecha (ver fig. 1.12). Sin embargo, al igual que los atributos, algunos métodos sólo pueden descubrirse en la etapa de diseño o en la de implementación.

Las relaciones entre clases existen de diversos tipos (ver apartado 1.4.4). Son el resultado de la herencia, de la generalización/especialización, de la composición y de conexiones lógicas entre las clases. El identificar las relaciones entre clases, requiere de la aplicación de los conceptos de abstracción de composición, de generalización y especialización, de una revisión muy cuidadosa de los escenarios y descripción de los *casos de uso* para encontrar las conexiones lógicas. Al igual que en los casos anteriores, incluso se podría decir que de forma aun más recurrente, el identificar relaciones se presenta en etapas posteriores del desarrollo del sistema principalmente en el diseño. Es por eso que en el análisis se plantea que se detalle lo más posible a las relaciones, de manera que en el diseño no sea un número muy grande de ellas las que se tengan que agregar, al contrario algunas de las relaciones encontradas se pueden omitir o en su defecto replantear en el diseño.

1.5.2.4 Diagrama de clases

Un *diagrama de clases* va evolucionando hasta estar completo, es decir, en primera instancia sólo se tiene representadas las clases (ver fig. 1.19), si se identifican atributos y métodos se podrán incluir en el diagrama y así continuar hasta poder representar también las relaciones entre las clases, cuando se llega a este punto se dice que se tiene un *modelo estático* de las clases que componen el sistema. Es una representación estática por que sólo se presentan como están formadas y relacionadas las clases, no por su funcionamiento.

Una clase con atributos y métodos utilizando *UML* se representa con un rectángulo, el cual está dividido en tres partes de forma horizontal. La parte superior es donde se introduce el nombre de la clase, la sección media es donde se especifica el nombre de los atributos y la sección inferior donde se define el nombre de los métodos (ver fig.1.21a). Para representar una conexión lógica entre dos clases, se utiliza una línea continua que las une, a la cual se le puede especificar la cardinalidad en cada uno de los extremos de la línea y especificar el nombre de la asociación, el cual aparecerá en la parte central de la línea (ver fig. 1.21b). *UML* permite representar varios casos de cardinalidad entre dos

clases, aparte de los ya mencionados (ver apartado 1.4.4), por ejemplo permite que la cardinalidad de un objeto pueda ser un rango de valores[26].

Las relaciones de composición entre un todo y sus partes se representan entre líneas que tiene un rombo en uno de los extremos de la línea y que debe estar del lado de la entidad que forma el todo (ver fig. 1.21c). Cuando se tiene una relación de dependencia funcional entre dos clases, se representa con una línea discontinua que une a las clases donde un extremo de la línea, contiene una flecha que apunta a la clase dominante o proveedora del servicio (ver fig. 1.12d). La herencia entre clases se representa por líneas que unen a la clase padre con sus hijos, donde en el extremo de la línea que conecta a la clase padre se tiene un triángulo apuntando hacia esa clase (ver fig. 1.21e).

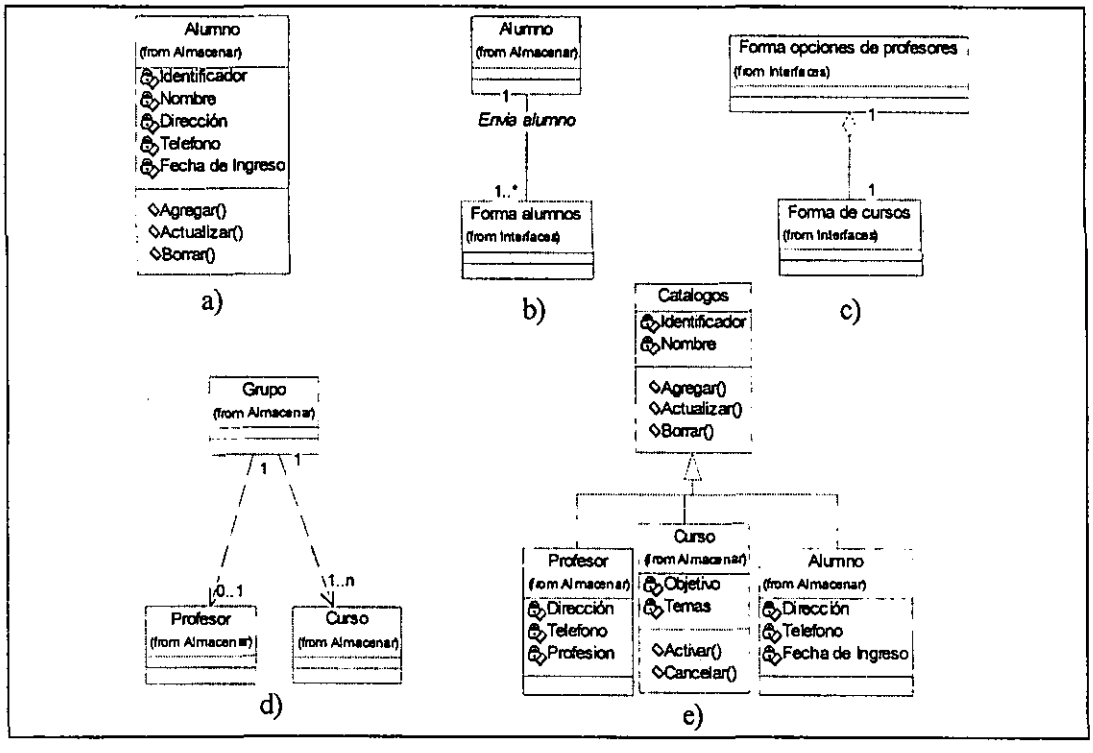


Figura 1.21 Diagramas de clases utilizando UML

1.5.2.5 Escenarios y máquinas de estado finito

Con la obtención del modelo estático se obtiene una descripción de las clases y sus relaciones, sin embargo, todavía no sabemos cómo se comportan esas clases. A la descripción del comportamiento de las clases, se le denomina *modelo dinámico*. Los

[26] Para más información de esta y otras características que permite representar el lenguaje *UML* con respecto a las relaciones entre clases consultar los manuales de especificación de la notación *UML*[20].

diagramas de iteración (representación de *escenarios*) son una buena herramienta para ilustrar el comportamiento dinámico o la iteración de las clases en el tiempo (ver apartado 1.5.1.3), pues nos presenta un número finito de sucesos que ocurre entre los objetos y actores del sistema. Una herramienta que se puede usar para expresar este comportamiento pero enfocado a captarlo sólo en una clase, son las *máquinas de estado finito*. Una *máquina de estado finito* "es una invención hipotética que solo puede existir en un número finito de estados en un momento dado"[27]; representa un conjunto discreto de estados y un conjunto discreto de transiciones de estados. Una transición de estado, es cambiar de una configuración en un conjunto de configuraciones a otra. Las transiciones para ocurrir dependen de un número finito de entradas.

Los *diagramas de iteración* representan escenarios, los escenarios representan el comportamiento de objetos a través del tiempo. Utilizando los diagramas de iteración se pueden construir máquinas de estado finito que representen el comportamiento de un único objeto a través del tiempo. Del *diagrama de iteración*, se identifica a algún objeto que se requiera modelar su comportamiento con una máquina de estados. Se define posteriormente un estado inicial que va a tener el objeto, es decir, el estado en que el objeto se va a encontrar antes de comunicarse con otros objetos u otros actores y que generalmente es el estado en que se debe encontrar el objeto cuando se inicia el sistema. Posteriormente se observan los mensajes que le llegan a ese objeto de actores u otros objetos, cada uno de estos mensajes puede definir un nuevo estado para ese objeto.

Si el mensaje que le llega al objeto le indica a éste realizar alguna operación, es decir, invocar algún método que modifique su estructura de datos, entonces se tiene que definir un nuevo estado para el objeto. Por otra parte, si llega algún mensaje al objeto que indique el fin de algún proceso, como por ejemplo los mensajes de cancelar, abortar o terminar, generalmente le indican al objeto que pase a un estado especial o al estado inicial, aunque esto implique invocar o no alguno de los métodos del objeto. De igual manera, si el mensaje le indica al objeto esperar cierto resultado de algún otro objeto o algún actor, esto también implica que el objeto estará en otro estado. Por lo general, para este último caso, el objeto que le llega el mensaje tiene como consecuencia que enviar otro mensaje a otro objeto o actor y a esperar por la respuesta de éste.

[27] James, M. Op. Cit. p. 429.

Representar con máquinas de estado finito el comportamiento dinámico de todos los objetos que se tengan en el sistema, no es una tarea muy práctica, sin embargo, sí se recomienda llevarla acabo para aquellos objetos que tienen un comportamiento dinámico muy diversificado que requiera de un análisis mas a fondo. Generalmente a estos objetos les llegan más mensajes y/o son los que envían más mensajes en el diagrama de iteración. A continuación se presenta una notación de las máquinas de estado finito, en la cual se pueden definir varias características que no se pueden definir en las representaciones convencionales de máquinas de estados y que permite modelar todas las situaciones que se presentan cuando se utiliza este método en el contexto del análisis orientado a objetos.

1.5.2.6 Diagramas de transición de estados

Un *diagrama de transición de estados*, es una representación del comportamiento dinámico de un objeto a través de una máquina de estado finito. Utilizando la notación *UML*, un estado se representa con un rectángulo con esquinas redondeadas, al cual se le asigna un nombre. La transición de un estado a otro se representa por una flecha que sale de un estado fuente a un estado destino, se puede asociar un nombre a la flecha que indica la transición, u opcionalmente, se puede indicar si se va a realizar alguna acción, o mostrar una condición que indica que la transición se va a realizar sólo si ésta se cumple (ver fig. 1.22).

Las acciones que se pueden llevar acabo en un estado pueden ser de tres tipos; acciones que ocurren cuando se acaba de pasar al estado (*entry*), acciones que se llevan acabo mientras se está en el estado (*do*) y acciones que se llevan acabo cuando se sale del estado (*exit*). Para representar alguna acción que va ocurrir, en el estado se pone el tipo de acción (*entry*, *do* o *exit*) y se escribe el nombre de la acción dentro del rectángulo debajo del nombre del estado (fig. 1.22). En un estado se pueden desencadenar varias acciones y de diversos tipos.

Existen dos tipos de estados especiales dentro del diagrama de transición de estados; el estado inicial, que se representa con un círculo negro y línea con una flecha que une al círculo con el estado que va a ser el inicial, y el (los) estado(s) final(es), que se representa con un círculo negro y blanco y una línea con una flecha que une al círculo con el (los) estado(s) que va(n) a ser el final (ver fig. 1.22). Los diagramas de transición de estados pueden volverse grandes y complejos, es por eso que con *UML* se puede representar el

anidamiento de estados. Un *superestado* es un estado que engloba a otros estados (ver fig. 1.22). Las transiciones pueden ocurrir; entre estados normales, entre superestados y estados normales y entre estados anidados y estados normales (ver fig. 1.22).

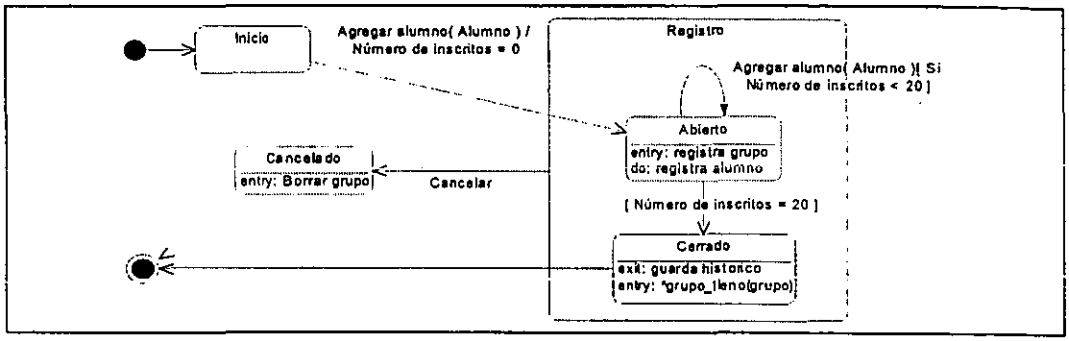


Figura 1.22 Diagrama de transición de estados

Con los *diagramas de transición de estados* y los *diagramas de iteración*, se tiene un *modelo dinámico* de los objetos del sistema que complementado con el *modelo estático* que se tenía con los *diagramas de clases*, se tiene *un modelo lógico* que describe los componentes del sistema y lo que realizan en términos de clases y objetos, es decir, se cumple con el objetivo del análisis. A continuación se verán tareas que se tiene que realizar cuando se pasa a la etapa de diseño.

1.5.3 Diseño (Modelo de Implantación)

El propósito del diseño es obtener una arquitectura del sistema que tenga los suficientes detalles para poder implementarla. La arquitectura de un sistema es un modelo de implantación del sistema que consiste en representar al sistema mediante módulos y/o subsistemas, donde cada uno de estos elementos tiene perfectamente definida su estructura y su definición de lo que va a realizar, así como los elementos físicos y las políticas necesarias para poder llevar acabo su implantación.

Es en esta etapa donde se decide cómo van a estar constituidas las interfaces que quieren los usuarios, las interfaces para comunicación con otros sistemas tales como los protocolos de comunicación que se van a utilizar, tipos de algoritmos a utilizar, si se requiere de algún sistema de gestión de base de datos para almacenar la información, cuántos procesadores se necesitan para los subsistemas, etc.; tareas que de hecho se deben realizar en cualquier diseño. Ya más concretamente para un sistema orientado a objetos, lo que se tendría que realizar en el diseño serían cuestiones como definir los tipos de datos de

las estructuras de los objetos (si es que se necesita), definir los algoritmos de algunos de los métodos que requieran de cierta especialización (si es necesario), definir para cada clase en qué módulo y en qué subsistema de la arquitectura va a estar contenida y especificar su definición de acuerdo a esta selección.

1.5.3.1 Representación del sistema mediante módulos

Los *módulos* de un sistema, son componentes que representan la implantación física de los componentes lógicos del sistema. Un módulo puede representar clases, tareas o funciones que lleva acabo el sistema. El representar el sistema mediante módulos nos da una visión estructurada del sistema así como una representación útil para la implementación, ya que la descripción de cómo están formados estos módulos debe de contener todas las especificaciones que se deben tomar en cuenta al momento de implantar ese módulo.

Cuando se define al sistema a través de los módulos que lo forman, para cada módulo se debe decir cómo estará formado. Si se va a utilizar un lenguaje orientado a objetos, se debe especificar la definición de las clases que forman ese módulo, utilizando los tipos y características de programación orientada a objetos que soportan ese lenguaje. Si algunos objetos se van a representar utilizando una base de datos relacional, se debe realizar una correspondencia de las clases a su definición de entidades en el modelo relacional y definir si algunos métodos se pueden realizar utilizando las herramientas que proporciona el sistema de gestión de base de datos seleccionado (por ejemplo utilizando *triggers* o procedimientos almacenados). De manera similar se tiene que definir cómo serán implementadas las clases que se vayan a realizar utilizando lenguajes basados en otros paradigmas, por ejemplo, utilizando un lenguaje estructurado o un lenguaje de inteligencia artificial. Para cada clase de algún módulo se debe dar una descripción formal de su definición dependiendo de la herramienta con la que se va implementar.

1.5.3.2 Diagramas de módulos para la representación física de componentes

La notación de módulos con *UML* consta de varios componentes, se pueden representar programas, subprogramas, clases y tareas o actividades. Para cada uno de estos componentes se puede representar su especificación y su cuerpo (ver fig. 1.23). La especificación de un componente es la declaración de ese componente, mientras el cuerpo

es la forma en que está definido ese componente. Por ejemplo en los lenguajes *C* y *C++* es muy común separar la especificación y el cuerpo mediante archivos (archivos .h para la especificación y archivos *.c* ó *.cpp* para el cuerpo).

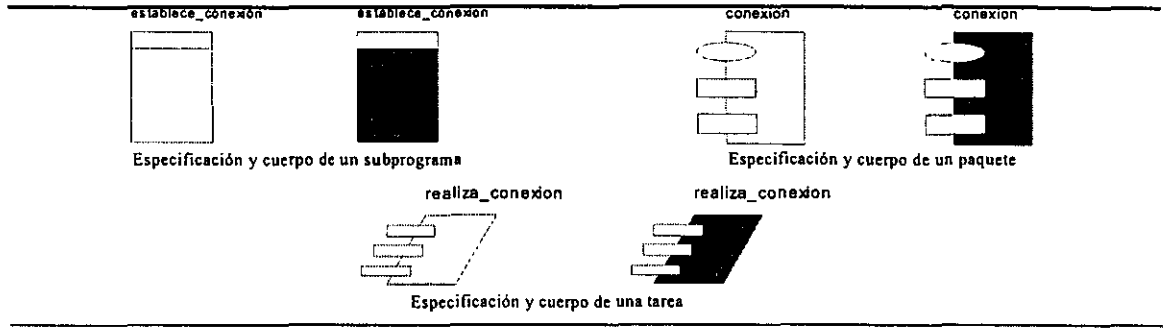


Figura 1.23 Especificación y cuerpo de elementos

Los módulos se pueden agrupar en paquetes (ver fig. 1.24a) que representen otros módulos o que representan unidades las cuales se pueden reutilizar o exportar. Se cuenta con un elemento conocido como componente, (ver fig. 1.24b) el cual se utiliza para representar archivos fuente, librerías o archivos ejecutables dependiendo la herramienta que se quiera utilizar (por ejemplo los archivos *.java* o los archivos *.pbl* de *Power Builder*). También se tiene un icono especial para representar el programa principal (ver fig. 1.24b) si es que la herramienta con la que se va a trabajar lo necesita como en el caso del lenguaje *C*.

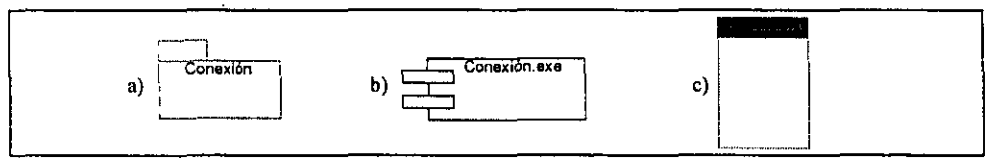


Figura 1.24 Representación de paquete, componente y programa principal

Las clases en el modelo lógico se van a *mapear* a componentes y los componentes puede formar parte de paquetes que a su vez puede representar módulos o subsistemas (ver apartado 1.5.3.3). La relación que se puede presentar entre módulos, es la de dependencia de compilación que puede surgir, lo cual se hace mediante la unión de los componentes por una línea, al igual que en los otros diagramas, la dependencia se indica con una flecha que está al extremo de la línea y apunta al módulo dependiente (ver fig. 1.25).

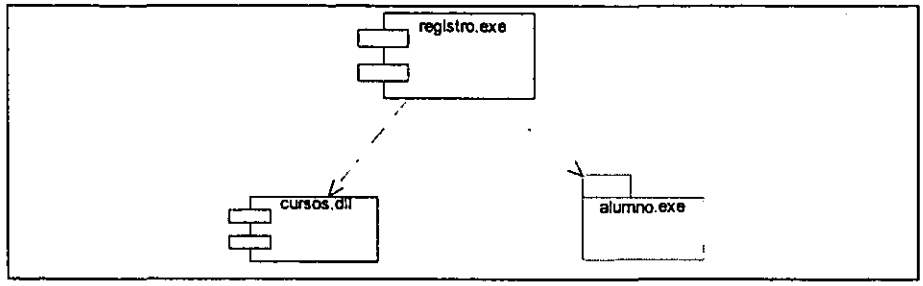


Figura 1.25 Dependencia de compilación entre componentes y paquetes

1.5.3.3 Definición de clases en el diseño

Definir una clase en el diseño va a consistir en integrarla a alguno de los módulos del sistema, esto es, definir sus atributos y métodos de acuerdo a la herramienta de desarrollo que se pretenda utilizar. Realizar esta tarea implica en varias ocasiones, redefinir las clases tanto en su estructura como en su comportamiento, agregando o quitando ciertos atributos o métodos, con la finalidad de que la representación que se tenga de la clase en esta etapa, tenga todas las especificaciones necesarias para su posterior implantación. No es de extrañarse que también se vuelvan a definir algunas de las relaciones entre las clases. Por ejemplo, a veces es necesario agregar una nueva relación entre clases donde no existía tal, por cuestiones de rendimiento o para evitar la redundancia de información.

De hecho se puede afirmar que se vuelve a realizar un análisis de identificación de objetos de sus relaciones y de su comportamiento, tomando en cuenta todos los aspectos relacionados con la herramienta y con la forma en que se piensa implementar la clase físicamente. A este ciclo de constante análisis de objetos, es a lo que Booch llama el "*microproceso del desarrollo*"[22], ya que se repite en todas las etapas de desarrollo de un sistema. Booch afirma que lo que constantemente se está realizando, es un refinamiento de la definición de las clases a través de cuatro actividades que se repiten continuamente hasta llegar a la conclusión del sistema, estas actividades son:

1) identificar clases y objetos,
2) identificar la semántica de clases y objetos,
3) identificar relaciones entre clases y objetos, y
4) especificar interfaces e implantación de clases y objetos.

Aunque la última actividad es una tarea que se va a presentar exclusivamente en el diseño, sí puede traer como consecuencia que se vuelva a realizar algunas de las tres primeras tareas que en un principio pareciera que fueran tareas exclusivas del análisis.

Rambuagh por su parte, afirma que dentro del diseño de objetos se deben de llevar acabo tareas también de refinamiento de clases de sus relaciones y su comportamiento[23] a través de varios pasos:

1) combinar los tres modelos[28] para obtener operaciones aplicables a clases,
2) diseñar algoritmos para implementar las operaciones,
3) optimizar las vías de acceso a los datos,
4) implementar el control para interacciones externas,
5) ajustar las estructuras de clases para incrementar la herencia,
6) diseñar asociaciones,
7) determinar la representación de los objetos, y
8) empaquetar las clases y las asociaciones en módulos.

Se puede observar que los pasos a, c, d, e, f y g no es otra cosa que volver hacer el análisis de objetos, sólo que ahora tomando en cuenta la forma en cómo se van a implantar las clases. Los otros pasos (b y h) quedan comprendidos dentro de las tareas generales que se tienen que realizar para cualquier diseño.

La definición de clases en el diseño aplica volver a identificar clases, atributos, métodos, relaciones, iteración entre objetos y actores, nuevos estados de operación de objetos, etc. Por lo tanto el resultado del diseño de objetos debe ser nuevamente *diagramas de clase*, *diagramas de iteración* y *diagramas de transición de estado* más completos que incluyan detalles y cuestiones más específicas de las herramientas que se seleccionaron para implementar las clases.

1.5.3.4 Subsistemas para la representación de la arquitectura del sistema

Si la representación del sistema es de los componentes más generales, se dice que se tiene la arquitectura del sistema, ya que nos presenta el sistema como una colección de *subsistemas*. Se considera un *subsistema* como un componente general que en la mayoría

[28] Los modelos a los que se refiere son: el *modelo de objetos*, el *modelo dinámico* y el *modelo funcional* que son los tres modelos del análisis de la metodología OMT propuesta por Rambaugh.

de los casos forma parte de la arquitectura del sistema por tener bien delimitado su funcionalidad y sus límites, es decir, un conjunto de módulos se agrupan en un subsistema por que van a realizar tareas que por lo regular van a servir como entradas o salidas de información hacia otros subsistemas.

Un subsistema se puede definir al agrupar a varios módulos que tiene en común la herramienta con la que se van a implantar. Los diagramas de subsistemas utilizados para representar la arquitectura de los sistemas pueden ser de diversas formas y se pueden representar utilizando varias notaciones, por ejemplo; las notaciones de los diagramas de flujo de datos, los diagramas de capas o de bloques o si se trata de representar la arquitectura de un sistema grande y con cierto grado de complejidad se recomienda que se utilicen iconos especiales para representar cada módulo, de manera que estos iconos reflejen la naturaleza de cada susbsistema. La notación que se seleccione para representar la arquitectura del sistema depende de las características que ésta tenga, de esta manera se podrá utilizar la notación que más se adecue a estas características (ver fig. 1.26).

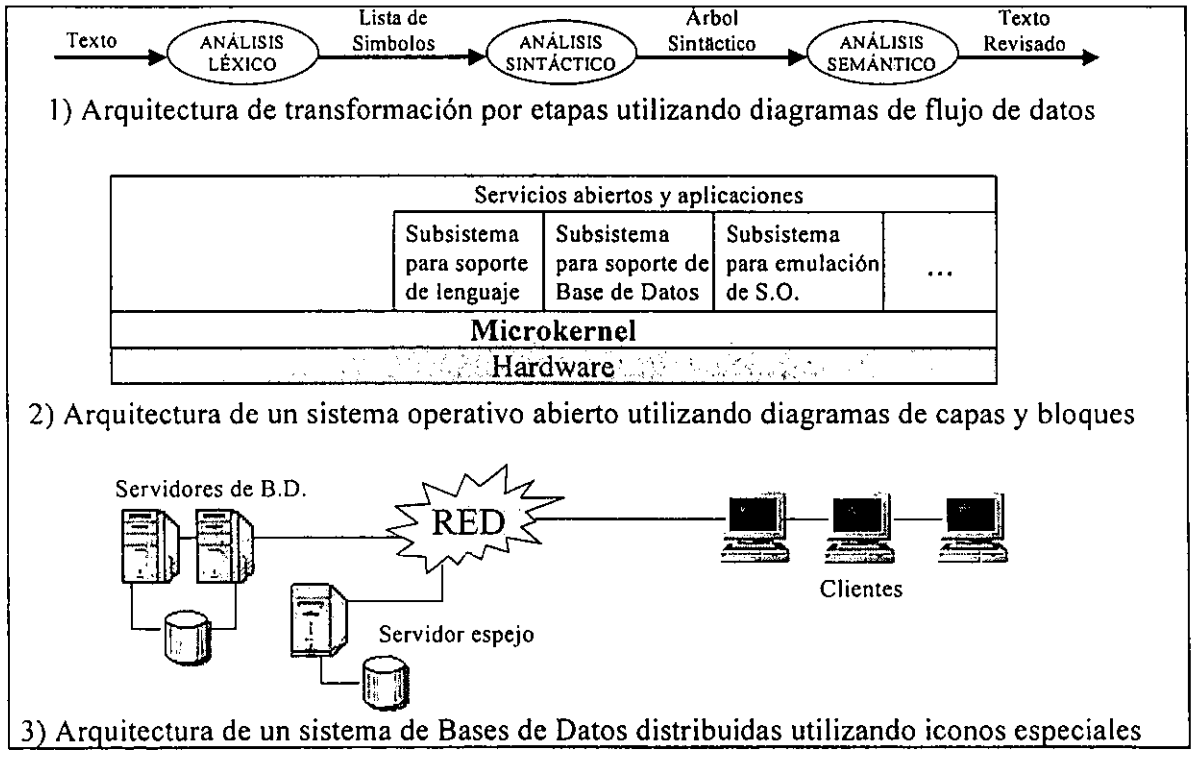


Figura 1.26 Notaciones para representar la arquitectura de sistemas

1.5.3.5 Diagramas de Procesos

Una de las tareas del diseño en general que tiene que ver con el *hardware* y con aspectos de rendimiento del sistema, es la de asignar los distintos subsistemas de *software* que se tiene a procesadores. La asignación de procesos a procesadores es una tarea que debe contemplar diferentes aspectos, por ejemplo, el tipo de tecnología con la que trabajen las herramientas en las que se va a desarrollar el sistema, si es *cliente/servidor* o un esquema centralizado, detectar aquellos subsistemas que necesitan estar interactuando entre sí un número de veces muy grande de tal forma que sea más conveniente asignarlos a un mismo procesador para reducir el tiempo de comunicación que aveces se vuelve mas grande que el tiempo de procesamiento. Aspectos de esta naturaleza que aquí no se tratan a detalle pero que son muy importantes para el rendimiento y funcionamiento adecuado del sistema.

Para representar la asignación de procesos a procesadores con *UML*, se tiene iconos para representar procesos, procesadores y dispositivos. Los tres elementos se representan con cubos a los cuales se les asigna un nombre, solo que para los procesos y procesadores, las caras posteriores están sombreadas y los dispositivos no lo restan. Las posibles comunicaciones que puedan existir entre estos elementos se representan mediante una línea no dirigida (fig. 1.27). Al igual que la representación de la arquitectura, es recomendable que la representación de los diagramas de procesos se implemente con iconos más adecuados si quiere tener una representación con más detalle que refleje la naturaleza de cada uno de los procesadores (una *PC*, una estación de trabajo, un mainframe, etc.), de los dispositivos (una base de datos, una impresora, un *modem*, etc.) o de los procesos (un servidor, un cliente, un hilo o proceso ligero, etc.).

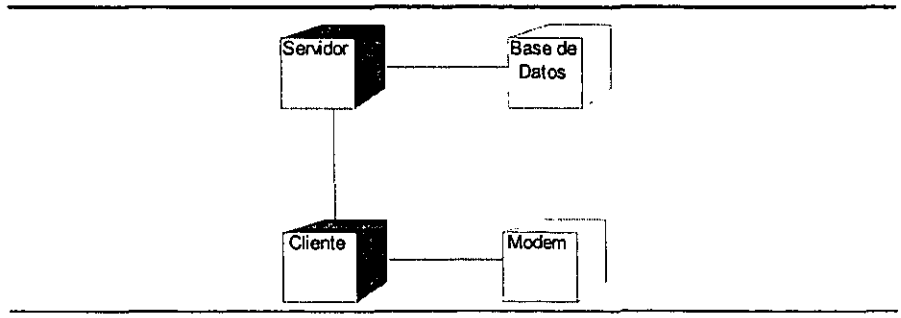


Figura 1.27 Diagrama de procesos

1.5.4 El proceso de desarrollo utilizando el método orientado a objetos

En resumen, se puede decir que el proceso de desarrollo utilizando la orientación objetos consta de tres etapas importantes que podemos dividir en tres modelos; el *modelo de requerimientos*, *el modelo lógico* o análisis del sistema y el *modelo de implantación* o diseño del sistema. En el siguiente esquema se presenta un diagrama de flujo de datos en el cual las actividades se representan como los procesos, las entidades externas son las personas que participan en la realización de esas actividades y los almacenamientos son los productos que obtiene de cada una de las etapas (ver fig. 1.28).

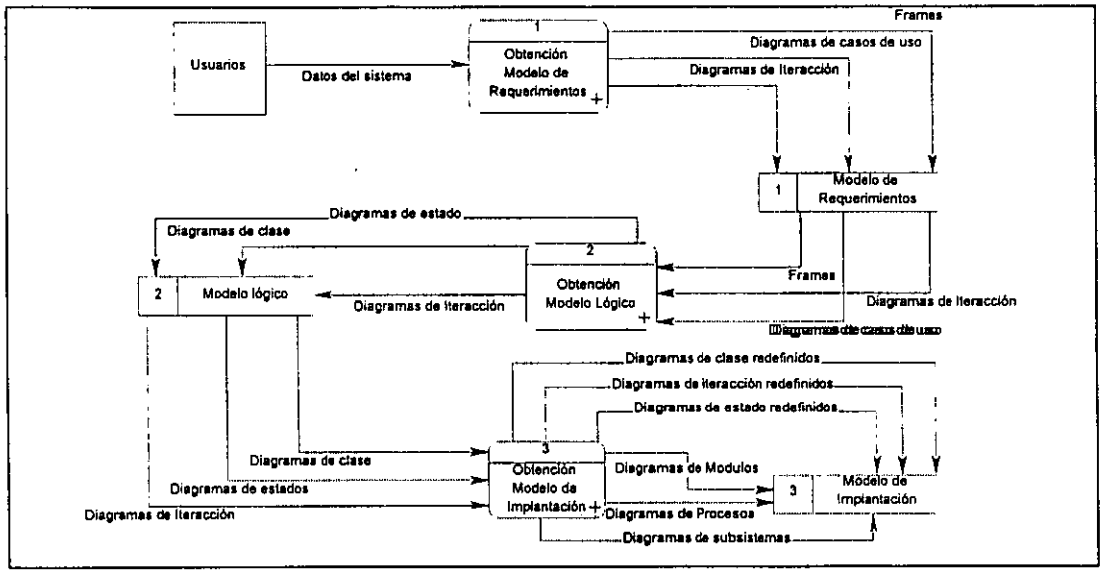


Figura 1.28 Proceso de desarrollo orientado a objetos

Dentro del *modelo de requerimientos*, se identifican los requerimientos del sistema mediante *diagramas de casos de uso*, lo que nos permite delimitar el problema e identificar el dominio del mismo mediante la obtención de los componentes más generales (objetos tareas y actividades). Usando marcos de referencia se puede representar el dominio del problema, pues nos permite agrupar estos elementos. Dentro de esta etapa, ya se puede realizar una primera identificación de objetos, los cuales son los que se puedan identificar en los *casos de uso*. También se podrán obtener *diagramas de iteración* de algunos *casos de uso*, que representan la iteración entre objetos y entre objetos y actores, sobre todo en aquellos *casos de uso* en los cuales se pueden identificar a los objetos.

En el análisis del sistema se obtiene un *modelo lógico* que nos proporciona la descripción estática y dinámica de los objetos. Por ser un modelo lógico, no se incluyen los detalles de la implantación de los componentes que se obtengan. La descripción

estática del sistema consiste en obtener las clases del sistema incluyendo su descripción, es decir sus atributos, métodos y relaciones con otras clases; mediante *diagramas de clase*. La descripción dinámica se hace mediante *diagramas de iteración* y *diagramas de transición de estado* que representan el comportamiento de los objetos a través del tiempo. En esta etapa se vuelve a tener la actividad de identificar los objetos, solo que ahora tomando en cuenta los atributos, los métodos, y las relaciones que se tiene entre los objetos. Los métodos que se tengan, generalmente van a surgir del análisis que se haga de los *diagramas de iteración* y de los *diagramas de transición de estado.*

Finalmente, en el diseño se debe obtener un *modelo de implantación* que represente la arquitectura del sistema. Para esto se deben realizar ciertas tareas que de hecho son válidas para cualquier diseño aunque no sea orientado a objetos. Particularmente para un diseño orientado a objetos, se debe realizar un refinamiento de las clases, ahora tomando en cuenta aspectos de cómo van a ser implantadas estas clases, es decir, las herramientas que se piensa utilizar para las estructuras de datos y la programación de los métodos de esas clases. Se debe especificar a qué módulos y a qué subsistemas de la arquitectura corresponde cada una de las clases. Como ya se mencionó (ver apartado 1.5.3.3), este tipo de tareas nos pueden llevar a realizar todas o algunas de las actividades que se presentaban en el análisis.

1.5.5 Problemática y futuro de la "orientación a objetos"

Para la representación estática de las clases, la notación es muy completa y se podría afirmar que es una de las partes para la cual ya se han propuesto casi todos los casos que se pueden tener. Sin embargo, para la representación dinámica de los objetos todavía se tienen algunos puntos, los cuales se pueden desarrollar, como por ejemplo, utilizar diagramas de flujo de datos se ha cuestionado porque nos puede llevar a realizar un análisis enfocado a tareas y no ha uno enfocado a objetos. Pero puede ser muy útil si en el sistema que se está modelando se tienen tareas que se realizan en forma cooperativa o tareas generales que involucren a varios objetos, y que el representarlos usando diagramas de iteración o diagramas de estados puedan resultar en ocasiones un asunto complejo.

Se cuenta con la notación de diagramas de objetos para representar la semántica de los enlaces propuesta por Booch[24], que pueden modelar muy bien la iteración entre dos objetos que se involucran en un proceso cooperativo, o de sincronización, donde los

objetos se ven como servidores de alguna tarea o se ven como clientes solicitando que se realice algún servicio; variantes de estos diagramas son los que propone Martin[25] como *diagramas de flujo de objetos*, donde los objetos se ven siempre como productores o consumidores de algún producto generado por los propios objetos. Ambas opciones tienen ventajas y desventajas; por ejemplo, con los diagramas propuestos por Booch se puede modelar muy bien todas las llamadas a operaciones que ocurren entre los objetos, pero no se observan los resultados que éstas puedan arrojar, con el modelo de Martin, sí se tiene los resultados obtenidos por las acciones de cierto objeto, pero no se sabe qué operaciones se realizaron para obtener ese resultado. Proponer estrategias para el uso de nuevas herramientas, o integrar las que ya existen de forma que se complementen para modelar de forma completa el comportamiento dinámico de los objetos, es un campo en el cual todavía se puede hacer mucho.

Actualmente los sistemas distribuidos para la comunicación, utilizan *sockets*[29] que son soportados por la mayoría de los lenguajes de propósito general. Sin embargo, para la comunicación entre los objetos a través de una red, se requiere que ésta se de en forma transparente, es decir, sin tener que utilizar las primitivas básicas para el uso de *sockets* cada vez que un objeto se requiera comunicar con otro. En este sentido, se han hecho esfuerzos como el uso de *CORBA*[30], o actualmente el uso de *RMI*[31], que tratan de ser una solución para la comunicación de objetos en forma remota. El uso de estas tecnologías cuando se trabaja con objetos puede ser una buena práctica para que el enfoque de objetos se vea realmente como tecnología de integración.

Sin duda, aquí sólo se han visto algunas de las problemáticas que todavía se tiene que superar en el futuro, pero también a quedado claro que el enfoque orientado a objetos está teniendo una evolución muy importante en cuanto a desarrollo de *software* para sistemas distribuidos que pueden presentar un grado de complejidad alto.

Se pudo constatar que existen bastantes similitudes entre las distintas metodologías orientadas a objetos, pero también existen discrepancias. Hay partes del análisis y del diseño en los cuales una metodología hace mas énfasis que otra. La notación utilizada para

[29] *Socket* es un canal de comunicación virtual a través de una red entre dos procesos.

[30] *CORBA* (*Common Object Request Broker Architecture*) es un estándar para objetos distribuidos desarrollado por *OMG* (*Object Management Group*).

[31] *RMI* (*Remote Method Invocation*) Invocación a un procedimiento remoto usando el lenguaje de programación *Java*.

representar los diagramas conocida como *UML*, es sin duda uno de los intentos más serios para tratar de homogeneizar y estandarizar los diferentes criterios que se tienen en cuanto al análisis y diseño orientado a objetos, así como al análisis y diseño de sistemas en general. Sin embargo, no hay que olvidar que *UML* solo es un lenguaje para el modelado de sistemas y no una metodología, o al menos no se tiene una todavía una propuesta formal por parte de sus creadores.

El enfoque orientado a objetos está siendo adoptado de una u otra forma por las demás tecnologías de cómputo, cuenta con bases conceptuales sólidas así como con las suficientes herramientas para poder implantar cualquier sistema, por complejo que este sea, por lo tanto, el futuro de esta tecnología será un uso cada vez más común de la misma y una *standarizacion* de sus métodos. Como se mencionó al comienzo del capítulo, no se trató de descubrir o promover una nueva metodología, si no de integrar los aspectos comunes que tienen las que ya existen, así como integrar elementos innovadores de las mismas que sirvan para complementar los huecos que se podrían tener si nos enfocamos a una sola. En los siguientes capítulos de este trabajo se van a llevar a la práctica algunos de los métodos y pasos del método orientado a objetos, en particular en el capítulo seis, "Diseño del cliente", se hace un análisis y diseño puramente orientado a objetos.

Capítulo 2. Programación multihilos

En este capítulo hablaremos sobre la programación multihilos como un modelo potente de programación. Usando un modelo de programación multihilos, podremos explotar de manera más eficiente las características que los sistemas operativos nos brindan, como son la multitarea y el multiprocesamiento; podremos también hacer uso de los recursos del sistema operativo en programas que usen mas eficientemente el tiempo de *CPU* y con una relación costo-beneficio más atractiva para la solución de una extensa variedad de problemas de cómputo.

*2.1 Hilos (*Threads*)*

Un hilo o proceso ligero es un flujo de ejecución dentro de un proceso donde cada hilo es un camino diferente de ejecución que puede ejecutar instrucciones de manera independiente, permitiendo al proceso llevar a cabo varias tareas de manera concurrente. Los hilos se inventaron para permitir la combinación del paralelismo con la ejecución secuencial y el bloqueo de las llamadas al sistema.

De lo anterior, un flujo de instrucciones resultante para un programa con un solo hilo de ejecución sería el siguiente: si un proceso 1 ejecuta los enunciados 245, 246 y 247 en un ciclo, su hilo de ejecución puede representarse como la secuencia 245_1, 246_1, 247_1, 245_1, 246_1, 247_1, 245_1, 246_1, 247_1..., donde los subíndices identifican al hilo en ejecución; en este caso sólo hay uno.

Podemos entender a un hilo como un proceso dentro de otro proceso, pues un hilo cuenta con código, datos, y mantiene el estado del *kernel* (*program counter*[1], el *stack pointer*[2], otros registros generales, etc.). Para poder separar la ejecución de un proceso en diferentes flujos, cada hilo requiere de un fragmento de código ejecutable con una pila o *stack* y datos propios. Pero, a diferencia de los procesos distintos que pueden pertenecer a usuarios diferentes y ser hóstiles entre sí, un proceso -que pertenece a un usuario único- puede tener varios hilos para que éstos cooperen y luchen entre sí.

[1] El *Program counter* es el registro del *CPU* que almacena la dirección de la instrucción en ejecución en algún instante dado.

[2] El *Stack pointer* es el registro del *CPU* que guarda la dirección actual de la pila y la dirección de retorno de subrutinas.

En un proceso, todas las partes de su estructura están dentro del espacio del *kernel*, por lo que un programa no puede tener acceso a ningún dato directamente; por el contrario, sí puede tener acceso directamente al código, como son funciones y procedimientos; un proceso es entonces una entidad a nivel *kernel*. Sin embargo, un hilo es una entidad a nivel usuario (*user-level*)[3], pues como hemos mencionado, un hilo tiene su propia pila, su propio conjunto de registros y su propio espacio de direcciones. Todo lo que en un hilo ocurra no requiere de llamadas al sistema, por lo tanto se ejecutan más rápido. Es decir, "*el kernel no se entera de que el hilo existe*"[4].

En la tabla 2.1, se muestran los elementos que pertenecen a un hilo (elementos por hilo) y aquellos que pertenecen al proceso que son compartidos por todos sus hilos (elementos por proceso).

Tabla 2.1 Conceptos por hilos y por proceso

Elementos por hilo	Elementos por proceso
• Contador del programa • Pila • Conjunto de registros • Hilos de los hijos • Estado del Kernel	• Espacio de direcciones • Variables globales • Archivos abiertos • Procesos hijos • Cronómetros • Señales • Semáforos • Información contable

2.1.1 Ventajas de los hilos

La programación multihilos involucra una serie de ventajas significativas con respecto a la programación tradicional de procesos. Entre los principales aspectos que se ven beneficiados por este esquema podemos mencionar los siguientes:

- **Paralelismo-** Diferentes hilos pueden correr en diferentes procesadores de manera simultánea.
- **Ejecución directa-** Cuando un hilo hace una petición de servicio al sistema operativo, el proceso no debe esperar hasta que el servicio se complete, sino que otro hilo puede

[3] Un hilo no siempre se comporta como entidad a nivel usuario, pero por claridad supondremos que siempre lo son. Más adelante se expondrá la causa por la que no siempre se comportan así (Modelo de planificación Uno a Uno).

[4] Lewit, Bil y Berg, J. Daniel, *Threads Primer A Guide to Multithreaded Programming*, p. 11.

continuar. Esto permite varias peticiones en un mismo proceso sin que éste tenga necesariamente que bloquearse.

- **Respuesta-** Como no es necesario bloquear todo el proceso, una aplicación que requiera de largas operaciones puede resolverse por varios hilos independientes. Esto permite a la aplicación estar siempre en actividad, recortando el tiempo de respuesta.
- **Comunicación-** A diferencia de una aplicación que utiliza comunicación interprocesos[5], una aplicación multihilos no solo puede tener varias conexiones *IPC*, sino que puede también compartir datos en el mismo espacio de direcciones.
- **Recursos del sistema-** En aplicaciones que mantienen varios procesos teniendo acceso a la misma área de memoria compartida, el control y sincronización son mucho más costosos tanto en tiempo como en espacio[6], que en la aplicación análoga implementada con hilos.
- **Objetos distribuidos-** Aplicaciones que utilizan objetos distribuidos, son por naturaleza multihilos, pues cada vez que un objeto debe ejecutar alguna acción, ésta se ejecuta en un hilo por separado. Esto incrementa la utilidad de la programación de los objetos.
- **Estructura del programa-** Los programas tradicionales que ejecutan varias tareas, requieren de código complicado para garantizar su control; en cambio un programa multihilos puede ejecutar las mismas tareas, pero con mucho mayor simplicidad de código.

2.1.2 Organización de los hilos en un proceso

Hay tres posibles modelos donde se puede organizar la cooperación de un conjunto de hilos en un proceso que pueden servir como base para el diseño del programa, estos son: Organización cliente-servidor, Organización en equipo y Organización de entubamiento.

2.1.2.1 Organización cliente-servidor

En este modelo, un hilo servidor lee solicitudes de trabajo en el buzón del sistema, después elige a un hilo trabajador inactivo y le envía la solicitud generalmente escribiendo un mensaje en una cola de mensajes asociada al hilo. El hilo servidor despierta entonces al hilo dormido, quien finalmente atenderá la petición en el instante en que pueda realizarlo.

[5] La comunicación interprocesos (*IPC* de *interprocess communications*) se vale de mecanismos del sistema operativo como son los *pipes*, o mas comúnmente de *sockets*.

[6] La sincronía de procesos separados requiere siempre de llamadas al sistema, que como veremos, la ejecución se hace más lenta y la sincronía mucho más complicada.

2.1.2.2 Organización en equipo

En el modelo de equipo todos los hilos son iguales, donde cada uno obtiene y procesa sus propias solicitudes. Como no existe un hilo servidor, se puede utilizar una cola de mensajes común que contenga todas las peticiones pendientes. Así, un hilo debe verificar primero la cola de mensajes antes de buscar en el buzón del sistema.

2.1.2.3 Organización de entubamiento

Finalmente, se puede utilizar el modelo de entubamiento. En éste, el primer hilo genera ciertos datos y los transfiere al siguiente para su procesamiento. Los datos pasan entonces de hilo en hilo y en cada etapa son procesados.

2.1.3 Aspectos del diseño con hilos

El manejo de hilos tiene dos alternativas, una es el uso de hilos dinámicos, la otra es el uso de hilos estáticos.

- **Diseño estático**- En este diseño, se elige el número de hilos a escribir en el programa o durante su compilación. Cada uno de ellos tiene asociada un pila fija. Este método aunque es más simple, es menos flexible.
- **Diseño dinámico**: En este diseño, que es más general, se permite la creación y destrucción de los hilos durante la ejecución. La llamada para la creación del hilo determina el punto de inicio del mismo (como un apuntador a un procedimiento), el tamaño de la pila, su prioridad, así como otros parámetros.

2.1.4 Uso de hilos

Antes de continuar, es importante recalcar que cualquier cosa que pueda hacerse con el uso de hilos puede hacerse sin ellos, utilizando procesos que compartan entre sí espacios de memoria. No obstante, debemos tomar en cuenta la importancia de las ventajas antes mencionadas, pues sobre todo entre más complicada sea la aplicación, mayor será la utilidad del uso de hilos.

En general se recomienda el uso de hilos para programas con tareas independientes, con servidores, con tareas repetitivas y programas para cálculo numérico.

2.2 El Modelo de Hilos

A continuación hablaremos sobre las partes del sistema operativo que intervienen en un modelo de hilos, los cuales permiten comprender la forma en que los hilos trabajan. Para ello abordaremos los conceptos involucrados en la implementación de un sistema operativo *UNIX*, pues éste facilita la comprensión a detalle de la forma en que el *kernel* maneja los hilos a nivel usuario; además, *UNIX* es el sistema operativo con mayor documentación que da mayores facilidades de desarrollo a los programadores.

Debemos considerar también que existen dos formas de implementar hilos. La primera es con bibliotecas a nivel usuario (paquete de hilos); en éstas, el código y todas las estructuras se guardan en el espacio de usuario, la mayoría se ejecutan completamente en el área de usuario sin la necesidad de hacer llamadas al sistema. La segunda, es mediante bibliotecas escritas a nivel *kernel*, que requieran de llamadas al sistema.

2.2.1 La estructura de un proceso

Recordemos que lo único que el *kernel* conoce es la estructura del proceso, pues el *kernel* se vale de esta estructura para controlar la ejecución del proceso.

Sin embargo, en algunas implementaciones más modernas de *UNIX*, como el sistema *Solaris 2* (fig. 2.1), se cuenta con un apuntador a una nueva estructura conocida como proceso ligero o *Lightweight Process* (*LWP*).

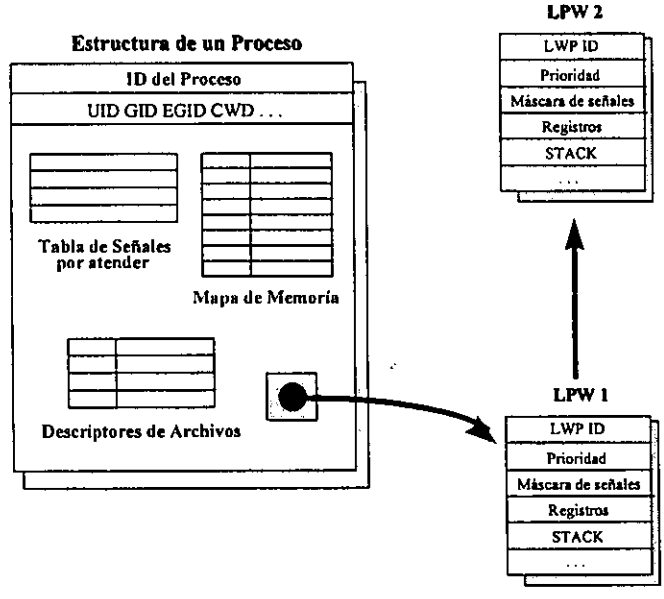


Figura 2.1 La Estructura de un Proceso en UNIX

En este caso, el estado del *CPU*, que incluye el valor de los registros, estadísticas de ejecución (tiempo de usuario, tiempo de sistema), la pila del *kernel* y máscaras de señales, se almacenan en las estructuras *LWP*; ahora un proceso puede contener varios *LWP*[7], de modo que todos los hilos compartan tanto las variables del proceso como su estado, esto gracias a que los *LWP* los comparten también.

Así, un *LWP* puede considerarse como un *CPU* virtual que puede ejecutar un código, llamadas al sistema y ejecutar fallos de página de manera independiente, gracias a que cada *LWP* es atendido por separado por el *kernel*. Esto permite que múltiples *LWP* se ejecuten en un mismo proceso, paralelamente en varios procesadores inclusive, es decir, permiten a las interfaces de hilos la concurrencia a nivel *kernel* y el paralelismo. Como los *LWP* son atendidos por el *CPU* de acuerdo a su clase de planificación (*scheduling*) y prioridad, tal y como lo haría con procesos tradicionales, un proceso con dos *LWP* generalmente requerirá aproximadamente dos veces el tiempo de *CPU* que el mismo proceso con un solo *LWP* para cambiar de contexto (proceso de planificación o *scheduling*).

2.2.2 Hilos y *LWP*

Sabemos que cuando el *kernel* del sistema operativo determina que debe realizar el cambio de contexto de dos procesos, almacena el valor de los registros en la estructura del proceso a suspender, *mapea* algunas de sus direcciones de memoria virtual, carga los valores de los registros y datos del segundo proceso y continua la ejecución.

Para el caso de la implementación de hilos con bibliotecas (paquete de hilos), el cambio de contexto es muy similar, sin embargo, esto se lleva a cabo completamente en el área de usuario por el mismo paquete. En este caso, el proceso consiste en salvar los registros en la estructura de control de un hilo, y remplazarlos por los valores almacenados en la estructura de control de otro hilo. De esta forma, como en el caso de la implementación a nivel *kernel* de *LWP*, podremos tener un sólo programa, en un mismo espacio de memoria, corriendo diferentes partes del programa concurrentemente en muchos *CPU* virtuales.

[7] Un *LWP* para procesos tradicionales y varios *LWP* para procesos con varios hilos.

Como sucede para los *LWP* mencionados en el punto anterior, un hilo tiene su propia pila y apuntador de pila; su contador del programa e información del hilo; así como su prioridad de ejecución y su máscara de señales almacenada en la estructura de control del hilo. Esto permite que varios hilos se ejecuten simultáneamente, e incluso ejecuten la misma sección de código.

Podemos observar en la fig. 2.2 que los hilos, sus pilas, el código, y los datos compartidos, están en el área de usuario, bajo el control del usuario; las estructuras de los hilos también se encuentran en el espacio de usuario, pero bajo control de las bibliotecas; mientras que los *LWP* son parte de la estructura del proceso, bajo el control del *kernel*, y son el medio de ejecución de los hilos.

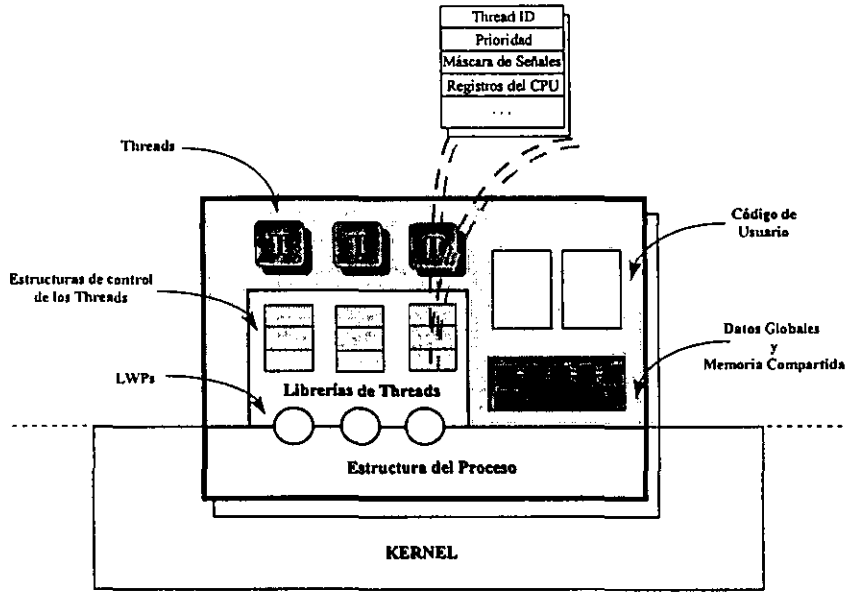


Figura 2.2 Organización de los hilos en un Proceso

Finalmente recordemos que los hilos están completamente en el área de usuario, por lo que el *kernel* no conoce su existencia. Los hilos entonces, no requieren de recursos del *kernel* y ejecutan cualquier operación más rápido.

Recordemos también que los hilos residen en un sólo espacio de direcciones, lo que significa que si un hilo modifica los valores de alguna estructura de datos, mientras otro hilo está leyéndolos, la ejecución del proceso se volverá un caos. Más adelante hablaremos sobre los mecanismos de sincronización entre hilos.

2.2.2 Llamadas al sistema

Una llamada al sistema que sea bloqueante, bloqueará todo el proceso, independientemente de que el proceso tenga uno o varios hilos. Por tanto, si un hilo se bloquea, el paquete de hilos debe optar por ejecutar otro hilo. Esto último representa un problema fundamental, pues en realidad no se puede permitir que el hilo realice la llamada al sistema, puesto que detendría a todos los demás hilos.

La solución a este problema no es fácil, puesto que no todos los sistemas operativos soportan el uso de hilos a nivel *kernel* y, además no siempre es factible hacer cambios a los sistemas operativos existentes. La solución requiere volver a escribir parte de la biblioteca de llamadas al sistema, lo cual es poco eficiente, pero es la única posibilidad.

En esta solución se necesita saber de antemano si la llamada es bloqueante, y reemplazarla por una llamada segura no bloqueante. En algunas versiones de *UNIX*, como las basadas en el estándar *Spec 1170*[8], existe una llamada *select*, la cual permite verificar si algún descriptor de archivo está disponible, es decir, si se puede realizar la llamada para *E/S* sin bloqueo. Con esto, por ejemplo, se puede reemplazar la llamada bloqueante *read* por otro procedimiento que primero realice una llamada *select* y luego la llamada al sistema *read* sólo en el caso en que esta última no se bloquee.

Si la llamada no puede hacerse sin bloqueo, la llamada *read* no se realiza, sino que se ejecuta otro hilo. La siguiente vez que el sistema de planificación (*scheduler*) vuelva a obtener el control, puede volver a verificar si la llamada *read* anterior es segura. A este código que se coloca junto a la llamada al sistema para la verificación de su seguridad, recibe el nombre de *jacket*.

*2.3 Mecanismos de Planificación (*Scheduling*)*

Existen dos formas de realizar la planificación de los hilos, una es la planificación local, la otra es la planificación global. En la planificación local, todo el mecanismo es local al proceso, y el paquete de hilos cuenta con todo el control sobre qué hilo será

[8] *Spec 1170* es el estándar desarrollado por la *X/ Open Foundation.*

ejecutado en un *LWP*[9]. Por su parte, en la planificación global, todo el mecanismo se lleva a cabo por el *kernel*[10].

2.3.1 Modelos de planificación a nivel *kernel*

Existen tres mecanismos principales para llevar a cabo la planificación de los hilos en los recursos del sistema, e indirectamente del *CPU*. Los tres permiten al programador hacer las mismas cosas pero con diferentes niveles de eficiencia.

2.3.1.1 El modelo "Muchos a uno"

En este modelo, varios hilos son creados en el espacio de usuario, pero todos ellos se ejecutan en un sólo *LWP*.

Bajo este esquema, cuando se lleva a cabo un bloqueo por una llamada al sistema, todo el proceso se bloquea. Sin embargo, la creación del hilo, los procesos de planificación y de sincronización, se efectúan completamente en el área de usuario, lo que lo hace rápido, barato y que no requiera de recursos del *kernel*.

Algunas bibliotecas, como en *HP-UX*, utilizan una rutina que reemplaza la llamada bloqueante con una no bloqueante. Así, cuando un hilo hace una llamada bloqueante, el paquete de hilos pone al hilo a dormir y permiten a otro hilo correr; luego, cuando la señal del *kernel* es recibida, las bibliotecas identifican al hilo que hizo la llamada y lo despiertan para que pueda continuar.

2.3.1.2 El modelo "Uno a uno"

Bajo este modelo, el sistema aloja un *LWP* por cada hilo. A diferencia del modelo anterior, éste permite que varios hilos corran simultáneamente en diferentes *CPU*, además permite a uno o más hilos hacer llamadas al sistema en forma bloqueante mientras que otro hilo continúe corriendo.

Este modelo tiene la desventaja de que cada hilo requiere de la creación de un *LWP*, que por supuesto son limitados, además, cada *LWP* utiliza recursos adicionales del sistema. Este modelo es el que utiliza *Windows NT* y *OS/2*.

[9] Esto implica que no se requiere de llamadas al sistema, por lo que no existe *time-slicing* o cuota de tiempo para el hilo.

2.3.1.3 El modelo "Muchos a muchos"

En este otro modelo, cualquier número de hilos se multiplexan en un número igual o menor de *LWP*. La creación de hilos se hace completamente en el espacio de usuario. Tiene la gran ventaja de que el número de *LWP* se puede variar para una aplicación y máquina particulares, la única limitante es el número máximo de hilos que pueden ser creados, lo cual depende del tamaño de la pila de los hilos y de la memoria virtual del sistema. *Solaris* implementa este modelo y lo combina con la posibilidad de utilizar el modelo "*Uno a uno*" para los procesos que así lo requieran.

2.3.2 Planificación de hilos

Cuando dos hilos corren simultáneamente, pueden hacerlo de dos formas. La primera, de manera independiente, en cuyo caso ninguno de los dos requiere nada del otro, por tanto ambos necesitan del uso de tiempo compartido de ejecución. La segunda, cuando ambos procesos dependen directamente el uno del otro, ya que uno de los dos, o bien no puede continuar hasta que el otro haya realizado alguna tarea, o bien el otro no inicie tal tarea hasta que el primero no realice ninguna petición. En este caso al ser dependientes no se requiere del uso de tiempo compartido de ejecución. Casi todos los programas con varios hilos se encuentran en este segundo caso y la planificación se basa en esta dependencia.

En el apartado 2.3.1 hablamos de la planificación global, a partir de aquí, hablaremos sobre la planificación local dentro del proceso, aunque sabemos que cada proceso también debe compartir recursos, ignoraremos su cambio de contexto en el *kernel* del sistema operativo.

2.3.3 Métodos de planificación

Como ya hemos mencionado, la planificación de hilos dependientes entre si no requiere de tiempo compartido, por lo que un hilo que está corriendo en un *LWP* podría hacerlo todo el tiempo, lo que significa que en alguna parte del programa debe existir un código que haga ceder la oportunidad de ejecución a otro hilo. De este modo, existen

[10] En este caso si se permite el uso de *time-slicing*.

cuatro métodos -adoptados por casi todas las bibliotecas- que permiten el cambio de contexto de un hilo, los cuales se explican a continuación:

- **Por sincronización-** Es el método de uso más común, se usa cuando un hilo trata de cerrar un *mutex*[11] sin éxito, entonces el hilo es colocado en la cola de espera (*sleep queue*) hasta obtener el bloqueo, permitiendo a otro hilo su ejecución (normalmente el hilo que colocó el candado). Este método se usa cuando un hilo necesita un recurso compartido y debe bloquearlo.
- **Por suspensión-** En este método, un hilo se suspende así mismo, por ejemplo con *thr_suspend( )*, y se coloca en la cola de suspensión (*stopped queue*) permaneciendo suspendido hasta que otro hilo lo reanude (con *thr_continue( )*).
- **Por preferencia *(Preemtion)*-** En este otro método, un hilo que se encuentra en ejecución realiza algo que provoca que otro hilo con mayor prioridad cambie de contexto. Así, el primer hilo -con menor prioridad- se suspende, y el segundo toma su lugar y se ejecuta. Este método es el más costoso, pues requiere de llamadas al sistema. Tanto el *kernel* (planificación global) como las bibliotecas (planificación local), necesitan mandar una señal al *LWP* en cuestión, para que éste ejecute la rutina (*handler*) que hará el cambio de contexto del hilo en ejecución y permita a uno de mayor prioridad tomar su lugar.
- **Por otorgamiento *(Yielding)***: Finalmente en este método, el hilo en ejecución hace una llamada explícita a una rutina especial como *thr_yield( )*, con lo que el planificador verifica si existe otro hilo con la misma prioridad en espera de ejecutarse, si es así lo cambia de contexto para que se ejecute, si no hubiere, el hilo que hizo la llamada continuará corriendo.

2.3.4 Estados de un hilo

El procedimiento de planificación de hilos se basa en cinco estados (fig. 2.3), y según su estado actual, un hilo podrá estar en ejecución, suspendido o listo para ejecutarse.

[11] Mutex (*Mutual exclusive lock*). Ver la parte referente a *MUTEX*.

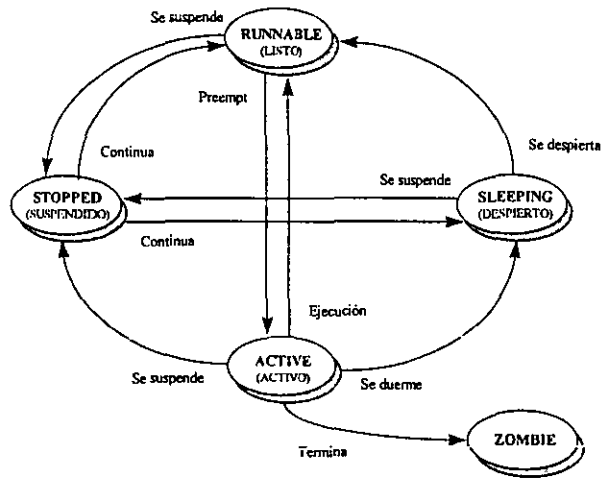


Figura 2.3 Diagrama de estados y transiciones de un Hilo

- **Activo** (*Active*): El hilo está en ejecución
- **Listo** (*Runnable*): El hilo está listo para correr, pero permanece así hasta que un hilo activo libera el *CPU* o hasta que se crea un nuevo *LWP*
- **Dormido** (*Sleeping*): El hilo está en espera de alguna variable de sincronización.
- **Detenido** (*Stopped*): El hilo ha hecho una llamada a *thr_suspend( )* y permanecerá en este estado hasta quc otro hilo haga la llamada a *thr_continue( ).*
- **Zombie**: El hilo ha muerto y está en espera de que sean desalojados sus recursos.

2.3.5 Cambio de contexto de un hilo

Cuando se habla del cambio de contexto de un hilo, significa la acción de tomar ese hilo activo y reemplazarlo por otro hilo que esté en espera de ejecución. Tomaremos como base los pasos para llevar a cabo el cambio de contexto de dos procesos, para posteriormente explicar el procedimiento análogo en hilos. Los pasos son los siguientes:

1) El *CPU* almacena todos los valores de sus registros en la estructura del proceso activo 1.
2) Después, el *CPU* toma todos los valores de los registros almacenados en la estructura del proceso 2 y los carga en sus registros.
3) Finalmente, el *CPU* regresa a modo de usuario, y continua corriendo con los valores de los registros del proceso 2. Se ha realizado el cambio de contexto de los procesos 1 y 2.

Todos los demás datos en la estructura del proceso, como son el directorio de trabajo (*CWD*), los descriptores de archivos, máscaras de señales, etc., se mantienen sin cambios,

de tal forma que cuando el proceso 1 vuelva a cambiar de contexto, podrá hacer referencia a estos datos.

El cambio de contexto de los hilos es muy similar. Si un hilo necesita cambiar de contexto (suspender su ejecución o de irse a dormir), llama a ejecución al planificador, luego el planificador almacena el estado de los registros del *CPU* en la estructura del hilo, posteriormente carga los valores de los registros de la estructura de otro hilo en el *CPU*, finalmente el planificador retorna pero ahora como el segundo hilo, quien puede ahora continuar su ejecución. Todo el cambio se realiza completamente en el área de usuario, con gran rapidez y sin necesidad de ninguna llamada al sistema.

Debemos recordar que el *scheduler* no realiza ninguna alteración a la estructura del proceso o del *LWP*, pues es el sistema operativo quien decide hacer el cambio de contexto del proceso (o *LWP*), es decir, el cambio de contexto de los hilos es completamente independiente al cambio de contexto de los procesos (o *LWP*) y viceversa.

En la fig. 2.4 podemos ver un ejemplo del mecanismo de cambio de contexto de hilos, mismo que a continuación se explica. En el instante 1 tenemos tres hilos listos para correr en dos *LWP*, el hilo T1 mantiene cerrado el *mutex* M, vemos que los hilos T1 y T2 tienen la prioridad más alta, por lo que ambos podrán activarse, mientras que el hilo T3 permanecerá listo en la *runnable queue* en espera de ejecutarse.

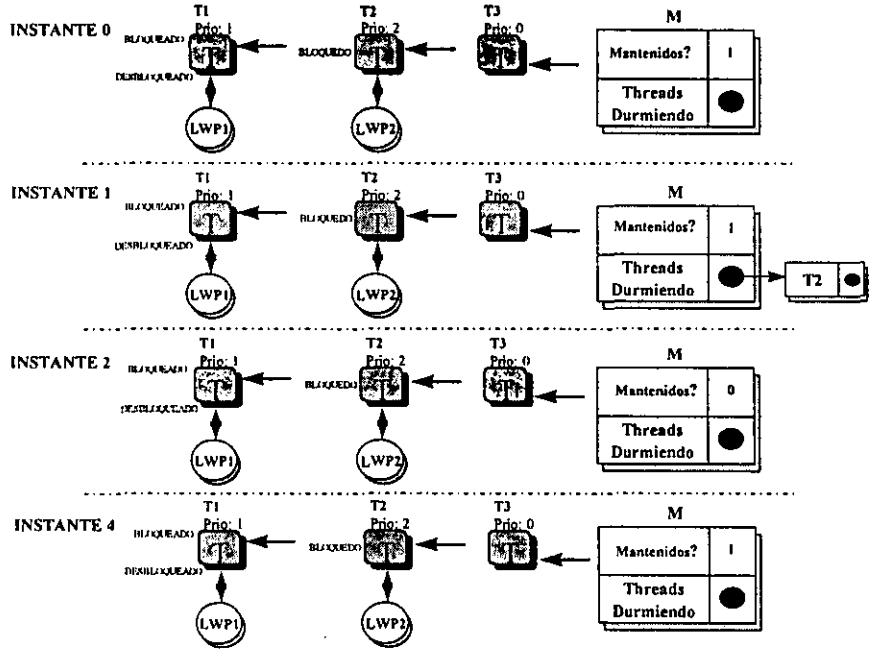


Figura 2.4 Cambio de contexto

En el instante 2, el hilo T2 trata de bloquear el *mutex* M sin éxito, por lo que se coloca a sí mismo en la *sleep queue* y luego llama al planificador. El planificador entra en ejecución aún como el hilo T2 y decide activar al hilo T3, posteriormente el planificador realiza el cambio de contexto de los hilos T2 y T3, termina su ejecución y regresa ahora como el hilo T3 corriendo en el *LPW2*. Ahora, al instante 3, el hilo T1 libera el *mutex*, así que éste toma al primer hilo de la *sleep queue* y cambia su estado a listo (*runnable*) y finalmente llama al planificador.

Esta vez el planificador verifica que el hilo T2 tenga mayor prioridad que el hilo T3, aún activo, así que manda una señal al proceso *LWP2*, termina su ejecución, regresa, y el hilo T1 continua corriendo. El hilo T3 también se mantiene activo, pero en el instante que *LWP2* recibe la señal, el hilo T3 es interrumpido para permitir la ejecución de la rutina de tratamiento de la señal. La rutina vuelve a llamar al planificador, quien cambia el contexto de T3 y T2. Finalmente en el instante 4, el hilo T1 y T2 se activan, T3 está listo en espera de ejecutarse y T2 mantiene el *mutex*.

2.4 Mecanismos de Sincronización

Como hemos visto, en la ejecución concurrente, los hilos guardan una dependencia entre sí. Es decir, un hilo puede depender de la acción de otro hilo y de los datos que éste genere. Entonces, es lógico pensar que si un hilo comienza a cambiar datos mientras otro los está leyendo tendremos problemas. Es por ello que se requiere sincronizar la actividad de todos los hilos, para garantizar la confiabilidad y estabilidad de los datos y del programa, lo que se logra con el uso de variables de sincronización[12].

2.4.1 Secciones críticas

Una sección crítica es un fragmento de código que debe ejecutarse de manera atómica y sin ninguna interrupción que pueda afectar el éxito de su ejecución o la consistencia de los datos involucrados. Cuando un hilo está en su sección crítica -sin importar si está en estado activo o no- y algún otro requiera ejecutar la misma sección crítica, este último será forzado a esperar hasta que el primero termine la ejecución de la sección crítica.

Notemos que si un hilo entra en su sección crítica y dentro de ella requiere algún otro recurso compartido, o bien, el código de la sección es tan grande que utilice mucho tiempo de *CPU*, este hilo afectará decisivamente la concurrencia del programa. Por tal motivo, el uso de secciones críticas requiere que sean lo más pequeñas posibles, además, requiere de un diseño cuidadoso que garantice la optimización de la ejecución.

[12] La implementación de variables de sincronización requiere de la existencia de instrucciones de *hardware* de *Test* y *Set*.

2.4.2 Variables de sincronización

2.4.2.1 Mutex

Un *mutex* (*Mutual Exclusive Lock*) es la variable primitiva más simple para sincronización. Los *mutex* garantizan que cuando un hilo cierra el candado, es decir, se convierte en el dueño del *mutex*, cualquier otro intento por cerrar el candado fallará, provocando que el hilo que lo intentó se duerma; pero cuando el hilo que se apropió del candado lo libere, alguno de los hilos que se encuentran durmiendo será despertado, estando listo para activarse y consecuentemente poder adueñarse del *mutex*.
A continuación se muestra un fragmento de código que ejemplifica el uso de los *mutex*.

Thread 1	**Thread 2**
mutex_lock(&m);	mutex_lock(&m);
global++;	local = global;
mutex_unlock(&m);	mutex_unlock(&m);

2.4.2.2 Candados de Lector/Escritor

Existen programas que requieren lectura de datos compartidos muy a menudo, pero hacen escrituras muy de vez en cuando. Un candado de Lector/Escritor, permite tener muchos hilos leyendo los datos concurrentemente, mientras que los hilos de escritura tengan acceso a la sección crítica de manera serial, uno a uno.

Si un primer hilo hace una petición de lectura, obtiene el candado (fig. 2.5), con lo que si subsecuentes hilos cierran el candado para lectura, todos los hilos podrán leer los datos de manera concurrente. Pero, cuando otro hilo trate de cerrar el candado para escritura, es colocado en la cola de *sleep* para escritores hasta que todos los hilos de lectura lo liberen. Si en ese momento un segundo hilo de escritura intenta obtener el candado, también será colocado en esa cola de *sleep* en orden de prioridad. Posteriormente si se presenta otro hilo para lectura, será colocado en la cola de *sleep* para lectores, y estará dormido hasta que los otros dos escritores hayan terminado.

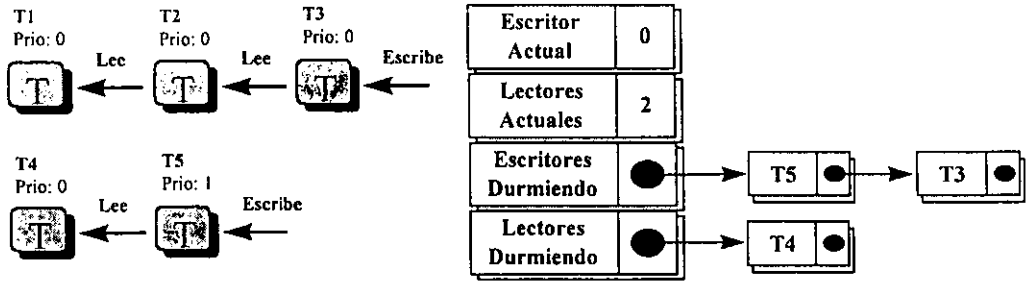


Figura 2.5 Candados de Lector/Escritor

Con esta variable, los hilos escritores obtienen el candado uno a la vez; cuando todos los escritores terminen, todos los hilos lectores que estuviesen dormidos son despertados y podrán adquirir el candado. El método del candado de Lector/Escritor tiene la gran desventaja de que es mucho más costoso que el uso de *mutex*, además de que sólo algunos sistemas lo implementan, como es el caso de *Solaris*.

2.4.2.3 Variables de Condición

Una variable de condición permite establecer condiciones que sean verificables por los hilos. Si la condición es falsa el hilo se dormirá, cuando la condición se cumpla, el hilo se despertará. En la fig. 2.6, un hilo obtiene un *mutex*[13] y verifica la condición. Si ésta es verdadera, el hilo termina su tarea, si es falsa, se libera automáticamente el *mutex* y el hilo se duerme. Cuando otro hilo cambie la condición (llamando a *cond_signal( )*), el primer hilo se despierta, éste, vuelve a adquirir el *mutex* y a evaluar la condición, y dependiendo de si ahora es verdadera o no, el hilo continuará o se irá a dormir.

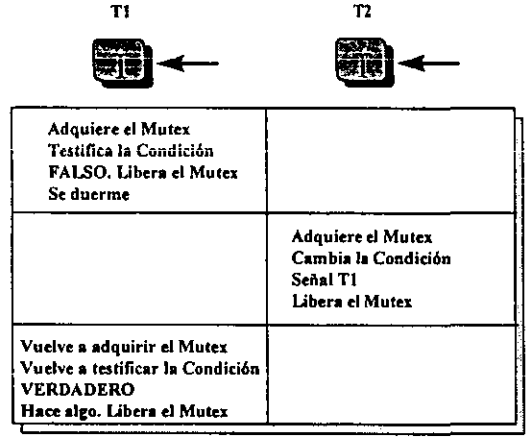


Figura 2.6 Uso de las Variables de Condición

A continuación se muestra un fragmento de código que muestra la forma en que se utilizan las variables de condición.

```
Thread 1                                  Thread 2
mutex_lock(&m);
while( !mi_condicion )
while( cond_wait(&c, &m) != 0);
                                          mutex_lock(&m);
                                          mi_condicion = TRUE;
                                          cond_signal(&m);
                                        mutex_unlock(&m);
hace_algo();
mutex_unlock(&m);
```

[13] Una variable de condición siempre tiene asociada un mutex.

2.4.2.4 Semáforos

Un semáforo es una variable que puede incrementarse progresivamente (llamando a *sema_post( )*), pero decrementarse siempre a cero (llamando a *sema_wait( )*). En este último caso, si el semáforo es mayor a cero, la operación tendrá éxito, si no, el hilo se irá a dormir hasta que otro hilo incremente el semáforo (fig. 2.7).

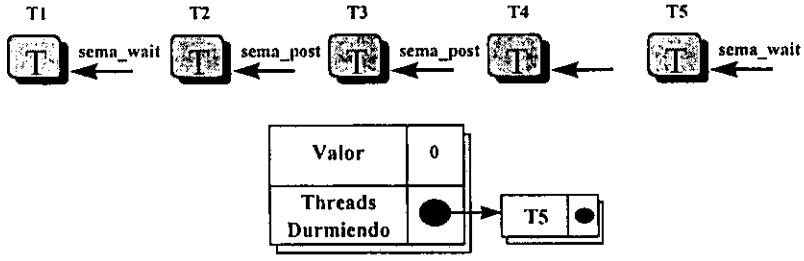


Figura 2.7 Uso de Semáforos

Normalmente, los semáforos son útiles en el uso de *buffers* que deben llenarse y vaciarse, o bien, cuando un hilo deba esperar hasta que algo ocurra.

Algo importante de los semáforos, es que son las únicas variables de sincronización que pueden utilizarse de manera asíncrona desde una rutina de tratamiento para señales. Esto es útil, si se quiere que al ocurrir cierta señal, ésta despierte al hilo. A continuación se muestra la forma para proteger el decremento del semáforo cuando ocurre alguna señal[14].

```
while (sema_wait(&s) == EINTR) { /* Probablemente no hace nada */ }
hace_algo()                      /* Se decrementa el semáforo */
```

2.4.2.5 Barreras (Barriers)

Un *barrier* permite sincronizar a un grupo de hilos para que se detengan en cierto punto de su ejecución y esperen a que los demás hilos lleguen también a ese punto.

2.4.2.6 Candados Spin

Estos candados se utilizan cuando se debe hacer un bloqueo por el menor tiempo posible para permitir a otros hilos correr sin que se bloqueen.

Un *spin lock* funciona cuando se inicia un contador con algún valor, supongamos 30, y se ejecuta la llamada *mutex_trylock( )*, que toma alrededor de 2μs. Si no se pudo cerrar

[14] Un semáforo normalmente regresa el valor *EINTR*, cuando fue interrumpido por una señal o por una llamada a *fork( )*.

el candado, se decrementa el contador y se vuelve a intentar. Si en algún valor del contador se pudo cerrar el candado, se ahorró mucho tiempo, y si no finalmente solo se habrá gastado 60μs (30 * 2μs).

```
spin_lock(mutex_t *m)
{
  int i;
  for (i=0; i > SPIN_COUNT; i++ )
  {
    if (mutex_trylock(m) != EBUSY)
      return;          /* Se obtuvo el candado */
  }
  mutex_lock(m);  /* No obtuvo el spin lock y hace un bloqueo normal */
  return;          /* Bloqueo obtenido */
}
```

2.5 Dificultades del uso de hilos

A continuación se describen algunas de las principales dificultades que se pueden presentar en la programaciòn multihilos:

- **Uso de llamadas al sistema**- La programación multihilos debe evitar el bloqueo de los procesos, por lo que el uso de llamadas al sistema que sean bloqueantes deben ser reemplazadas por otras que no sean bloqueantes.
- **Imposibilidad de utilizar la planificación *Round Robin***- Dentro de un proceso, no existen interrupciones de reloj, lo que imposibilita la planificación de tipo *Round Robin*[15] para la ejecución de los hilos.
- **Aplicaciones que realizan constantes llamadas al sistema**- En ocasiones se requiere desarrollar aplicaciones donde los hilos se bloquean a menudo, lo que requiere de constantes verificaciones de la seguridad de las llamadas al sistema. Por otro lado, existen aplicaciones que tienen limitaciones esenciales para el uso del *CPU* y que pocas veces se bloquea, para lo cual parece difícil identificar la necesidad del uso de hilos.
- **Asignación de memoria**- Los procedimientos para asignación de memoria, como *malloc* de *UNIX*, utilizan tablas de memoria, sin preocuparse por establecer y utilizar regiones críticas protegidas, puesto que fueron escritas para ambientes con un sólo hilo. Esto requiere reescribir toda la biblioteca.

[15] El algoritmo de planificación *Roun Robin*, consiste en asignar a cada proceso (o hilo) un intervalo de tiempo de ejecución, llamado *quantum* y el uso de una lista de procesos ejecutables.

Parte 2

Capítulo 3. Análisis del sistema

3.1 Descripción de las necesidades

Para esta descripción se requiere de un sistema de monitoreo de datos que aplique fórmulas definidas a datos provenientes de diversas actividades. El sistema debe definir variables que se utilicen en la definición de fórmulas, mismas que serán utilizadas como reglas que deben cumplir los datos a monitorear. De este modo, los usuarios podrán definir las reglas que involucren las variables de interés y cuyo cálculo se llevará a cabo continuamente en la recepción de datos.

El sistema deberá sustituir los datos recibidos en las variables correspondientes, que a su vez servirán como parámetros para la definición de las fórmulas. Con los datos sustituidos, deberá calcular cada una de las fórmulas involucradas y, si alguna de estas reglas no se cumplen, el sistema debe ser capaz de mandar mensajes de advertencia a los usuarios que las hayan definido.

3.2 Factores que intervienen

3.2.1 Hecho

Un hecho, representa al conjunto de datos a monitorear. Estos pueden presentarse en cualquier instante. Los hechos identifican las características y valores de cierta operación (datos de variables físicas, datos de mediciones para control de calidad, datos de operaciones bursátiles, etc.) que será delegada al sistema para su análisis.

3.2.2 Disparador

Un disparador representa un agente externo, el cual se ocupa de insertar en el sistema los hechos. Un disparador es en la actividad real, una operación como puede ser una medición de alguna variable física, una operación bursatil, etc.

3.2.3 Operando

Es una variable que tiene asociada un nombre lógico y que puede ser utilizada como parámetro en la definición de fórmulas. Los operandos pueden definirse para que tome el valor de alguno de los cuatro diferentes casos siguientes:

- **Operandos definidos como Valor Fijo**- De valores fijos, como pueden ser π, la fuerza de gravedad, coeficientes físicos, etc.
- **Operandos definidos como datos de un Hecho**- De datos provenientes en los hechos.
- **Operandos definidos como Columnas (1)**- De datos recuperables de una base de datos del mismo *Servidor de SQL* donde está instalado el sistema.
- **Operandos definidos como Columnas (2)**- Del resultado de funciones agregadas de *SQL* aplicada a datos de una base de datos del mismo *Servidor de SQL* donde está instalado el sistema.

3.2.4 Condición

Es la cláusula *WHERE* del *SELECT* necesaria para recuperar el valor de los operandos que sean substituibles de campos o funciones agregadas.

La condición tiene que garantizar la recuperación de un valor, exclusivamente, por cada operando, y en el caso que se utilicen funciones agregadas, se debe generar automáticamente la sentencia *GROUP BY* necesaria para que la sentencia *SELECT* retorne un sólo valor por operando.

A continuación se detalla la manera en que se pueden definir las condiciones:

1) Las condiciones podrán contener *JOINS*.
2) Pueden incluir *SUBQUERIES* a un sólo nivel.
3) Permite la especificación de condiciones a nivel columna, como son las siguientes combinaciones de comparaciones:
    - Dos valores constantes fijos; éstos pueden ser de tipo booleano, entero, real, fecha y cadena.
    - Columnas contra valores fijos y viceversa.
    - Columnas contra valores únicos retornados por funciones.
    - Condiciones *LIKE* para evaluación de cadenas.
    - Comparación de valores nulos mediante las cláusulas *IS NULL* o *IS NOT NULL*.
    - Inclusión de valores en un conjunto de datos especificado por una cláusula *IN*.
4) En el caso de utilizar *SUBQUERIES*, estos pueden retornar valores que serán evaluados mediante los operadores *=ANY*, *!=ANY*, *=ALL*, *!=ALL*, *IN*. Las condiciones

de los *SUBQUERIES* pueden contener prácticamente los mismos tipos de comparaciones que las condiciones más externas arriba mencionadas, incluyendo *JOINS*. Además se pueden definir alias para la especificación de tablas.

5) Un *SUBQUERY* puede recuperar un valor mediante el uso de una función agregada: *MAX, MIN, SUM, AVG* y *COUNT*.

3.2.5 Fórmula base

Una fórmula base es aquella que está definida por el administrador, que identifica un cálculo de tipo general y que puede ser utilizada por los usuarios para la definición de otras fórmulas (fórmulas personalizadas). Este tipo de fórmula puede dar como resultado cualquier tipo de dato considerado en el sistema: Booleano, Entero, Real, Fecha y Caracter. Si el tipo de dato del valor de retorno de la fórmula base resultara ser de tipo booleano, entonces ésta podrá ser utilizada directamente para la generación de alarmas.

3.2.6 Fórmula personalizada

Una fórmula personalizada es aquella que es definida por los usuarios del sistema utilizando como variables los operandos y las fórmulas base, ambos previamente definidos. Este tipo de fórmula puede dar como resultado exclusivamente un tipo de dato booleano, y en el caso en que resulte ser verdadero (*TRUE*) se activará una alarma.

3.2.7 Alarma

Una alarma es una señal que indica el resultado positivo de una fórmula aplicada a determinado hecho. La alarma se mostrará al usuario en una pantalla de monitoreo con todos los datos correspondientes a las variables de la fórmula que la generó.

3.2.8 Usuario

Los usuarios pueden tener tres roles diferentes que son:

- **Administrador-** Es el usuario que tiene acceso completo a todos los módulos.
- **Operador-** Este usuario puede configurar nuevas fórmulas, modificar las actuales, activar y desactivar fórmulas, y monitorear las alarmas.
- **Usuario normal-** Éste puede monitorear las alarmas, activar y desactivar fórmulas.

3.3 Características que debe cumplir el sistema

3.3.1 Alarmas generadas

El sistema puede manejar las alarmas como eventos acumulables o independientes en el tiempo de la siguiente forma:

- Si el usuario define una fórmula de tipo acumulable, la alarma generada será actualizada con los nuevos datos hasta que las series de hechos analizados dejen de producir la alarma, en este caso se notificará la desactivación de la alarma, y al presentarse nuevamente se generará una nueva alarma.
- Si el usuario define una fórmula, la alarma será generada cada vez que el resultado de la fórmula sea positivo (*TRUE*).

3.3.2 Sintaxis para las fórmulas

En la tabla 3.1 se muestra la sintaxis que debe seguir cada uno de los posibles tipos de dato que pueden definir a los operandos. Se muestra también algunos ejemplos que aclaran la forma en que se pueden usar.

Tabla 3.1 Sintaxis de los tipos de datos válidos

Tipos de datos		
Tipo	**Sintaxis**	**Ejemplos**
Booleano	{TRUE}|{FALSE}	TRUE, True, true FALSE, False, false
Real	(-)?{entero}(\.{entero})?([E|e][+|-]?{entero})?	• 10 • 10.02 • 123.323 • -123.323 • 453 E +12 • 453 e -2 • -4564.686 E 23 • 89 E - 34
Fecha	"dd/mm/yyyy" | "dd/mm/yyyy hh:mm"	• "01/12/1997" • "23/05/1997" • "30/07/2005" • "01/12/1997 10:23" • "23/05/1997 10:23" • "30/07/2005 23:59"
Cadena	"cadena"	• "Cadena" • "cadena" • "CADENA" • "123.43" • "122" • "as_12sas" • "a133"

Una vez establecida la sintaxis de los datos, se debe definir el tipo de operadores que se pueden utilizar; estos se muestran en la tabla 3.2 y también se añaden algunos ejemplos.

Tabla 3.2 Sintaxis de los operandos validos

Operadores válidos		
Tipo	**Operadores**	**Operaciones**
aritméticos	^, +, -, *, /, %	• 12+23.43 • -12 + 22.3 • (-12+33.6)*323 • 45% • 45% + 12% • 12% + 12.56 • (45+67^5)/21
relacionales	=, <, >, <=, >=, <>, !=	• 12 = 23 • "cadena1" = "cadena2" • -(34.45 + 24%) >= 12.3
lógicos	NOT, Not, not AND, And, and OR, Or, or XOR, Xor, xor	• ("TELMEX" = "TELMEX" and (3.56/5.6) > 5%) • -(12 + 5.6) < 2 OR (12 + 5.6) >= 0

3.3.3 Administración del sistema

El administrador tiene la responsabilidad de dar mantenimiento a los catálogos necesarios para el funcionamiento del sistema. La información requerida para su completo funcionamiento es la siguiente: disparadores, operandos, condiciones, fórmulas base y usuarios.

El administrador tiene también la responsabilidad de borrar las alarmas que ya no tengan interés para los usuarios.

3.4 Diagramas de Flujo de Datos

En este apartado se especifica el modelo en el que se basa el diseño del sistema. Este modelo está basado en la notación de *Gane & Sarson*, e involucra los factores identificados, las relaciones que guardan entre sí, así como las características de tales factores que el sistema debe cumplir.

Dadas las características de los factores involucrados y las tareas que debe realizar el sistema, hemos dividido el modelo completo en tres diagramas de flujo de datos independientes.

El primero describe todas las tareas que el usuario con el rol administrador debe cumplir; el segundo abarca todo lo necesario para la definición de las fórmulas, su manipulación, activación y desactivación; el tercero es relativo al procedimiento de recepción de los datos, al cálculo de las fórmulas y la ocurrencia de las alarmas de notificación.

3.4.1 Tareas de Administración

A continuación, en la fig. 3.1, mostramos el primer diagrama de flujo de datos obtenido. Éste corresponde a las tareas de administración y mantenimiento del sistema.

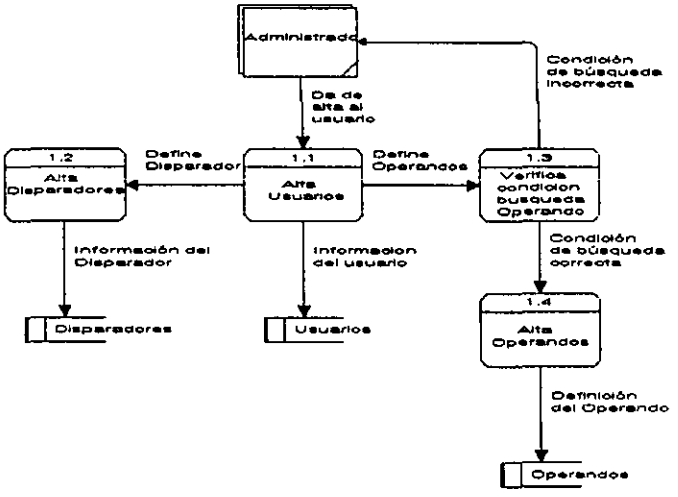


Figura 3.1 Tareas de Administración

Básicamente, en este módulo se definen los elementos preliminares que se utilizarán mas adelante para la definición de fórmulas y para su eventual cálculo.

Estos elementos son los usuarios válidos y su rol en el sistema, los disparadores que insertarán los hechos y que serán asociados a las fórmulas, y los operandos que podrán ser utilizados en la definición de fórmulas.

En seguida, en la tabla 3.3, se especifica cada uno de los elementos que componen el diagrama de flujo de datos anterior. En esta tabla se hace una breve descripción de cada uno de los elementos y la función que desempeñan.

Tabla 3.3 Elementos del *DFD* para la tareas de Administración

Etiqueta	Identificador	Tipo	Descripción
Alta Usuarios	1.1	Proceso	Procedimiento que da de alta al usuario en el sistema.
Alta Disparadores	1.2	Proceso	Procedimiento que registra los nuevos disparadores (agentes externos).
Verifica Condición Búsqueda Operando	1.3	Proceso	Procedimiento que analiza léxica y sintácticamente la condición de búsqueda del operando, si es que se definió como columna.
Alta Operandos	1.4	Proceso	Procedimiento que da de alta la definición de los operandos.
Administrador		Entidad	Usuario con el rol de administrador del sistema.
Disparadores		Almacenamiento	Registro de los disparadores (agentes externos) definidos.
Operandos		Almacenamiento	Registro de los operandos definidos.
Usuarios		Almacenamiento	Registro de los usuarios del sistema.

3.4.2 Definición y manipulación de fórmulas

En el siguiente diagrama de flujo de datos (fig. 3.2), se muestran los procedimientos a través de los cuales se pueden definir, activar y desactivar las fórmulas. Este diagrama está dividido en cinco bloques funcionales relacionados con cinco tareas diferentes cada una: definición de las fórmulas base, activación y desactivación de las fórmulas base, definición de las fórmulas personalizadas, activación y desactivación de las fórmulas personalizadas y especificación de las fórmulas como notificables.

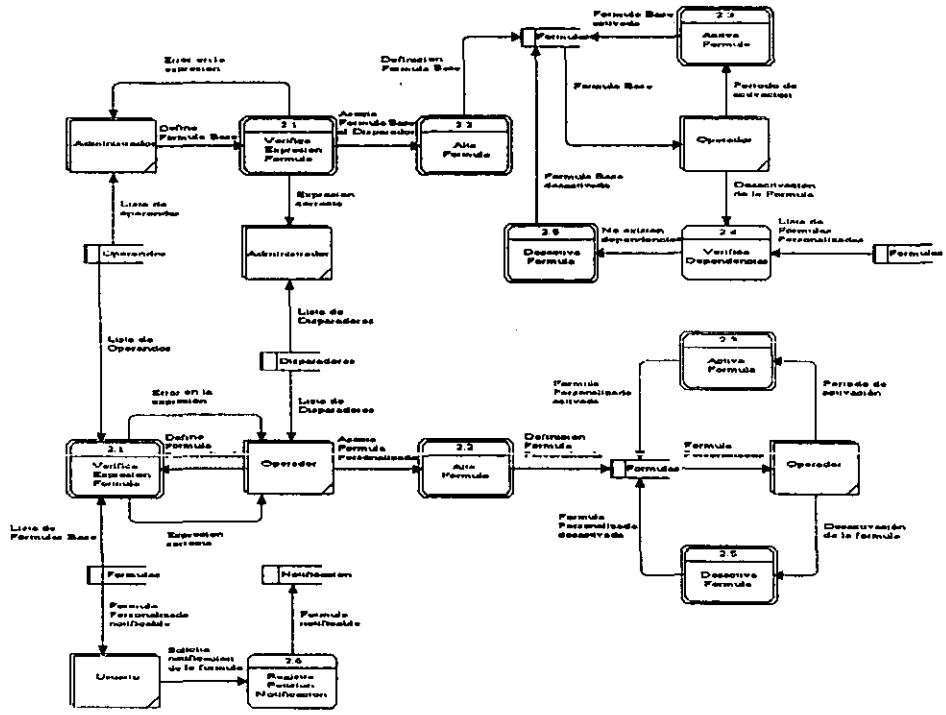


Figura 3.2 Definición y manipulación de Fórmulas

- **Definición de las Fórmulas Base**. En este bloque se define la expresión de la fórmula base. El administrador selecciona de la lista de operandos definidos los parámetros que compondrán la fórmula, se verifica que la expresión sea correcta, se asocia a la fórmula el disparador que desencadenará su cálculo, y finalmente se da de alta.
- **Activación y desactivación de las Fórmulas Base**. En este, se define el periodo de tiempo en el que la fórmula base estará activa. También se especifica cómo se lleva a cabo la desactivación manual de una fórmula base, para lo cual se debe verificar que la fórmula base no esté asociada a ninguna otra fórmula personalizada.
- **Definición de las fórmulas personalizadas**. Aquí se define la expresión de la fórmula personalizada. El operador selecciona de la lista de operandos y de la lista de fórmulas base, los parámetros que compondrán la fórmula personalizada, se verifica también la expresión, se asocia la fórmula al disparador que desencadenará su cálculo, y finalmente se da de alta.
- **Activación y desactivación de las fórmulas personalizadas**. Se define el periodo en el que la fórmula personalizada estará activa. También realiza la desactivación manual de una fórmula personalizada.

- **Especificación de las fórmulas como notificables**. Finalmente, aquí se lleva a cabo el registro de la petición de notificación de una fórmula dada, hecha por algún usuario.

En la tabla 3.4, se especifican los elementos que componen el diagrama de flujo de datos para la definición y manipulación de las fórmulas.

Tabla 3.4 Elementos del *DFD* para la definición y manipulación de fórmulas

Etiqueta	Identificador	Tipo	Descripción
Verifica Expresión Fórmula	2.1	Proceso	Procedimiento en el cual se verifica léxica y sintácticamente la expresión de la fórmula.
Alta Fórmula	2.2	Proceso	Procedimiento que registra una fórmula en la base de datos de control.
Activa Fórmula	2.3	Proceso	Procedimiento que lleva a cabo la activación de una fórmula para que sea notificable.
Verifica Dependencias	2.4	Proceso	Procedimiento en el que se verifica que la fórmula a borrar no sea operando de otra fórmula que este activa.
Desactiva Fórmula	2.5	Proceso	Procedimiento que lleva a cabo la desactivación de las fórmulas, para que no puedan ser notificables.
Registra Petición Notificación	2.6	Proceso	Procedimiento que registra la petición del usuario para que le sean notificadas las alarmas que la fórmula origine.
Administrador		Entidad	Usuario con el rol de administrador del sistema.
Operador		Entidad	Usuario con rol de operador.
Usuario		Entidad	Usuario ordinario sin rol de administrador u operador.
Disparadores		Almacenamiento	Registro de los disparadores (agentes externos) definidos.
Fórmulas		Almacenamiento	Registro de todas las fórmulas definidas.
Operandos		Almacenamiento	Registro de los operandos definidos.

3.4.3 Generación y notificación de alarmas

En el diagrama de la fig. 3.3 se esquematizan las tareas de recepción de datos, el cálculo de las fórmulas, la generación y notificación de las alarmas.

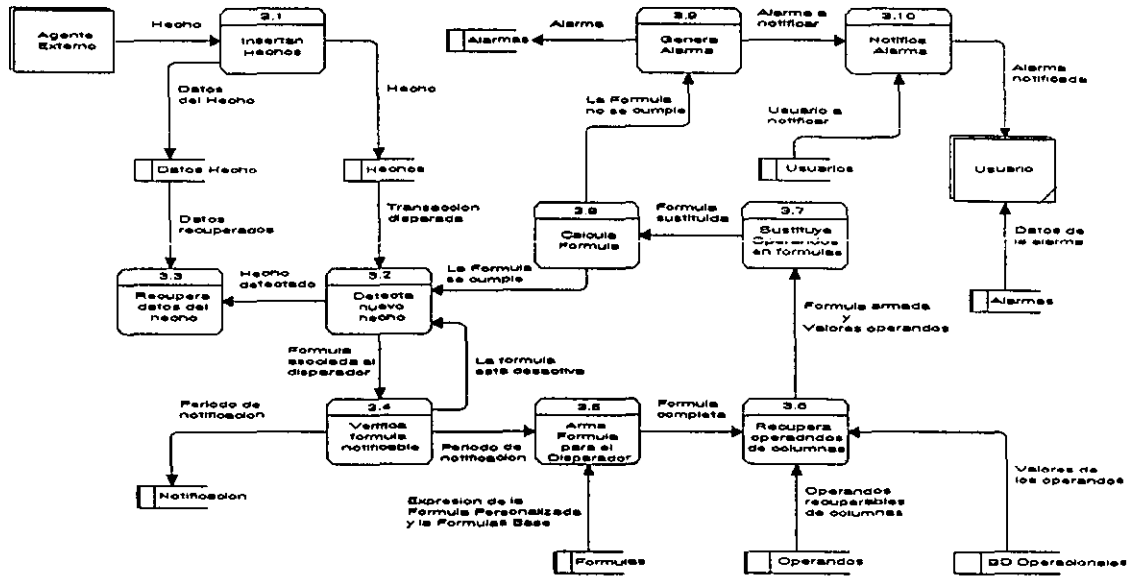


Figura 3.3 Generación y notificación de Alarmas

Este diagrama está formado por tres bloques funcionales que definen las tres tareas mencionadas.

- **Recepción de datos**: En este primer bloque, se detectan los hechos insertados por los agentes externos (disparadores), se recuperan los datos del hecho y se recuperan las fórmulas asociadas al disparador.
- **Cálculo de las fórmulas**: En el siguiente bloque, con las fórmulas recuperadas, se construye la expresión de la fórmula a calcular asociada al disparador. Se recuperan los valores de los operandos definidos como columnas y son sustituidos junto con los valores de los datos de los hechos en la expresión de la fórmula construida. Finalmente, se calcula la fórmula con sus parámetros sustituidos.
- **Generación y notificación de alarmas**: En el último bloque, se evalúa el resultado calculado y, dependiendo de éste, se genera una alarma y se le informa al usuario la ocurrencia de la misma.

La tabla 3.5 contiene la lista de los elementos que componen el *DFD* para la generación y notificación de alarmas.

Tabla 3.5 Elementos del *DFD* para la generación y notificación de alarmas

Etiqueta	Identificador	Tipo	Descripción
Insertan Hechos	3.1	Proceso	Procedimiento que lleva a cabo la inserción de los hechos en la base de datos de control. Este procedimiento lo origina el Agente Externo.
Detecta Nuevo Hecho	3.2	Proceso	Procedimiento que está poleando hasta que detecta la llegada de un nuevo hecho.
Recupera Datos del Hecho	3.3	Proceso	Procedimiento que se encarga de recuperar de la base de datos de control los datos del hecho.
Verifica fórmula Notificable	3.4	Proceso	Procedimiento en el que se verifica que la fórmula deba notificarse.
Arma Fórmula para el Disparador	3.5	Proceso	Procedimiento en el que se construye la Fórmula asociada al Disparador en base a su definición.
Recupera Operandos de Columnas	3.6	Proceso	Procedimiento que lleva a cabo la recuperación de los operandos definidos como columnas. El procedimiento utiliza la condición de búsqueda definida para el operando.
Sustituye Operandos en Fórmulas	3.7	Proceso	Procedimiento se encarga de sustituir cada uno de los valores en los operandos de la fórmula.
Calcula Fórmula	3.8	Proceso	Procedimiento que lleva a cabo el cálculo de la fórmula con los valores sustituidos.
Genera Alarma	3.9	Proceso	Procedimiento que se encarga de generar la alarma a notificarse.
Notifica Alarma	3.10	Proceso	Procedimiento que lleva a cabo la notificación asíncrona al usuario de la ocurrencia de una alarma.
Agente Externo		Entidad	Agente externo (disparador) que inserta los hechos en la base de datos de control.
Usuario		Entidad	Usuario ordinario sin rol de administrador u operador.
BD Operacionales		Almacenamiento	Son las bases de datos donde se almacena la información involucrada en las operaciones de los procesos.
Alarmas		Almacenamiento	Registro de las alarmas generadas que deben informarse a los usuarios.
Datos Hecho		Almacenamiento	Registro de los datos que acompañan a los hechos.
Fórmulas		Almacenamiento	Registro de todas las fórmulas definidas.
Hechos		Almacenamiento	Registro de los hechos insertados que deben ser leídos.
Notificación		Almacenamiento	Registro de las peticiones de los usuarios de notificación de las fórmulas.
Operandos		Almacenamiento	Registro de los operandos definidos.
Usuarios		Almacenamiento	Registro de los usuarios del sistema.

3.5 Modelo conceptual de la base de datos de control

Para el análisis de la base de datos, nos basaremos en el conjunto de requerimientos presentados anteriormente en este capítulo; se realizará un desglose detallado de los mismos, de manera que sirva como base para la definición de un modelo conceptual para la base de datos de control *DBMAD*. El modelo conceptual de la base de datos describe cómo será almacenada la información, a través de un esquema o modelo lógico de los datos. La representación de este modelo será mediante la definición de un diagrama entidad-relación, cuya notación es la que se presenta en la fig. 3.4.

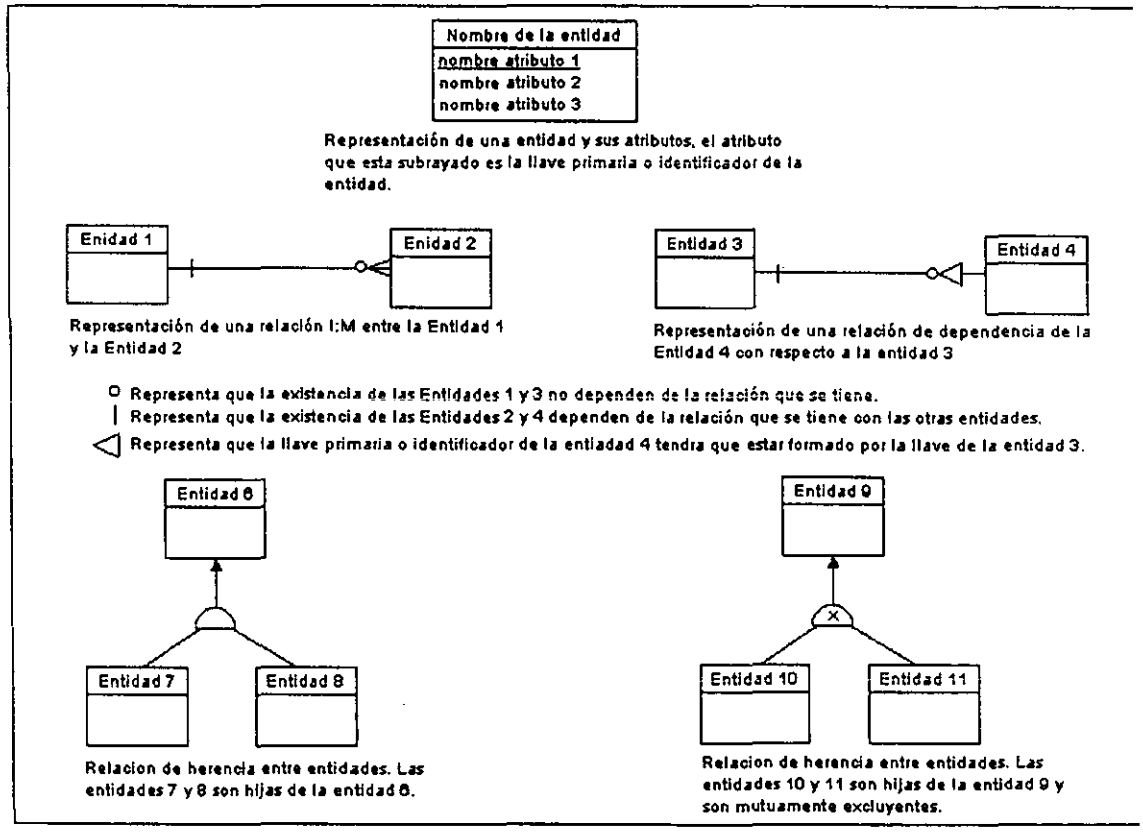


Figura 3.4 Notación de un Diagrama Entidad Relación

La base de datos de control (*Metabase*) de manera general deberá contener la siguiente información.

- Los usuarios del sistema.
- Los operandos involucrados en la construcción de las fórmulas.
- La definición de los disparadores que iniciarán el cálculo de las fórmulas.

- La definición de las fórmulas a utilizar en el sistema.
- El registro de las fórmulas que hayan generado datos por notificar (alarmas).

Cada uno de los elementos mencionados, representarán entidades en el modelo conceptual a elaborar, ya que son componentes independientes y existen dentro del contexto del problema. Sin embargo, se realizará un análisis más detallado de los requerimientos de información que se tiene alrededor de cada uno de los componentes. Esto, para tratar de encontrar los atributos que tienen asociados, así como para encontrar otras posibles entidades y sus relaciones entre sí, de forma que se pueda representar en el modelo lógico, de la manera más exacta posible, todos los datos que se necesitan para poder tener una base de datos de control que cumpla con todos los requerimientos.

3.5.1 Los usuarios del sistema

Para los usuarios, se requiere saber el nombre completo (*nombre* y *apellidos*) de la persona; un *login* o cuenta, que es un nombre corto con el cual se identificará al usuario; así como un *password* o contraseña que la persona debe saber y que se requiere como llave de acceso para validar que efectivamente es un usuario del sistema.

Se necesita asociar a cada uno de los usuarios un *rol* con el fin de tener un control más específico sobre las tareas que podrán realizar en los distintos módulos del sistema. El *rol* de usuario determinará las tareas que le son permitidas a los usuarios, de manera que se manejarán las siguientes categorías: administrador, operador y usuario. Dependiendo del tipo de *rol* que se haya asignado, el usuario podrá llevar acabo distintas tareas en cada uno de los módulos del sistema. Sin embargo, el control sobre las restricciones a los datos, se manejará desde la aplicación. De manera general, las tareas que los usuarios pueden realizar dentro del sistema dependiendo de su *rol* son las siguientes:

- **Administrador**. Tiene acceso total, define a los nuevos usuarios, a los operandos, a los disparadores, y a las fórmulas base y personalizadas.
- **Operador**. Puede definir fórmulas personalizadas y puede configurar fórmulas con acceso restringido.
- **Usuario**. Sólo puede hacer uso de las fórmulas y configurar algunas de las opciones, si la fórmula fuese pública.

Tomando en cuenta estos requerimientos, se define una entidad llamada usuario con los atributos que cumplen con los requerimientos presentados (ver fig. 3.5).

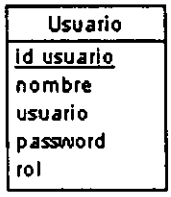


Figura 3.5 Entidad Usuario

3.5.2 Los operandos

Los operandos son los componentes o términos mínimos que sirven para definir a una fórmula. Para cada uno de estos operandos se necesita tener un *nombre lógico* que permita a los usuarios reconocerlos fácilmente; asignarles un *tipo de dato*, es decir, especificar si se trata de un tipo entero, un tipo cadena, un tipo flotante, un tipo booleano o un tipo fecha.

Para cada operando se podrá obtener su valor de tres formas distintas y, precisamente, esta diferencia en la recuperación de su valor, determinará el tipo de operando que se está definiendo.

Primero tenemos los datos recuperables de alguna transacción originada por una inserción en la base de datos de control, a través de un *trigger* o disparador. A estos operandos se les llamará *atributos del hecho* -por la definición que se dio a este término-.

Para este tipo de operandos, se necesita saber el *nombre de la columna* de la cual se va obtener el valor del operando. Esto es importante, ya que cuando se definan los disparadores, se necesitará especificar la columna a la cual hace referencia el operando, de tal forma que se garantice que el *trigger* va a obtener el dato.

Después tenemos los datos recuperables por alguna consulta hecha en las bases de datos del *Servidor de SQL* donde se encuentra la base da datos de control *Metabase*. Para este tipo de operandos se necesitará saber el *nombre de la base de datos*, la *tabla* y la *columna* de donde se recuperará el dato. Para este tipo de operandos hay un requerimiento especial, y es que se les puede asociar una *condición*, es decir, que al momento que se vaya a hacer la consulta, se agregará una condición de búsqueda en la cláusula *where*, por lo que para la condición, se necesitará la *definición* de ésta.

Para evitar que haya errores en su definición, antes de dar de alta la condición de búsqueda, ésta será validada mediante un *parser* que revisará la sintaxis. En caso de que la condición sea correcta, se dará como resultado una expresión traducida o una *definición compilada,* que se necesitará guardar para sustituir otros operandos involucrados en la expresión.

Por último están los datos recuperados de algún *valor constante*. Lo que se necesita saber para este tipo de operandos es, precisamente el *valor* que se utilizará al sustituir en el operando, cada vez que se haga referencia a él, cuando se necesite calcular alguna fórmula que lo utilice.

Tomando en cuenta todos estos requerimientos involucrados con los operandos, el esquema conceptual para manejar los distintos datos que se necesitan se muestran en la fig. 3.6.

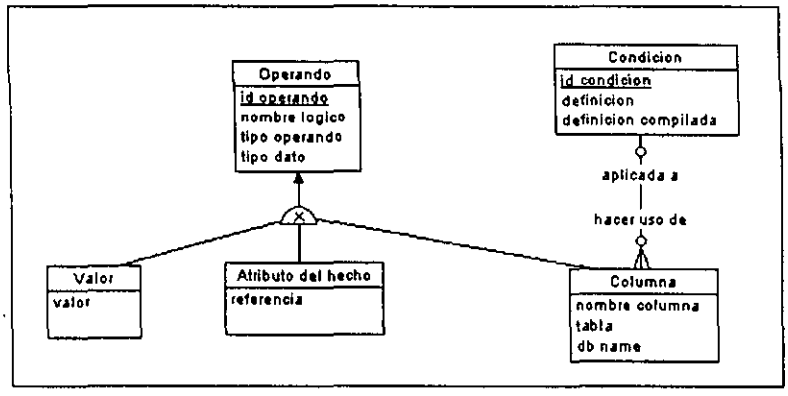


Figura 3.6 Esquema conceptual para el manejo de los tipo de Operandos

3.5.3 Los disparadores

Un disparador representa a un *trigger* que va a recuperar datos provenientes de una transacción que se realiza en alguna base de datos controlada por el *Servidor de SQL*. Estos datos se insertan en la base de datos de control, con ellos se calculan una o varias fórmulas que estén definidas por algunos de estos operandos recuperados vía una transacción. Para poder realizar el cálculo de las fórmulas, éstas deberán contener por lo menos alguno de estos operandos (atributos de hecho) en su definición. Por tanto, al recuperar los datos de alguna transacción, se iniciará el cálculo de las fórmulas.

Este proceso se identificará con el nombre de un hecho -como ya se explicó en la sección anterior-, por tanto, cada uno de los disparadores que se definan podrán generar varios hechos en tiempo de ejecución; éstos a su vez, tendrán que recuperar por lo menos uno o varios datos que corresponderán a operandos definidos como atributos del hecho.

Para el disparador se necesitará asignarle un *nombre lógico*, de manera que sea fácil reconocerlo por los usuarios. Por su parte, para los hechos se tendrá un número que los identifique de forma única (*id_hecho*), y los *valores* recuperados se tendrán que almacenar, así como el *identificador del operando* al que representa, de tal forma que, cuando se necesiten para el cálculo de las fórmulas, se asocien de manera correcta.

Cada vez que se genere un nuevo hecho, el *id_hecho* se incrementará en uno, y se realizará el cálculo de las fórmulas basándose en el *id_hecho* mas pequeño que no haya sido calculado aún. De manera que se necesita un apuntador que le indique a partir de qué hecho utilizar para calcular las fórmulas; una vez que ya se haya utilizado, el apuntador debe incrementarse para el siguiente hecho y de esta manera calcular las fórmulas asociadas al siguiente hecho. De las entidades definidas, el diagrama para representar estos requerimientos se muestra en la fig. 3.7.

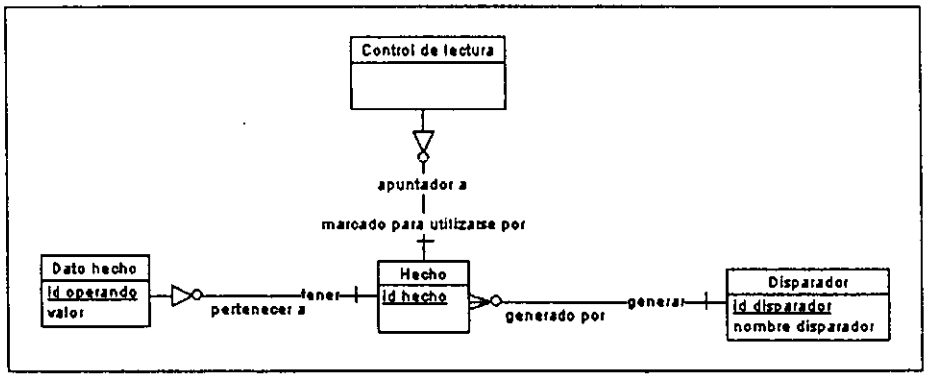


Figura 3.7 Esquema conceptual para el manejo de los Hechos

3.5.4 Las fórmulas

Las fórmulas son reglas que definen los usuarios para validar alguna operación que involucre datos que se quieran analizar. Para su registro se requiere de un *nombre* que las identifique; la *definición* de la fórmula o enunciado que contiene a los operandos, y operadores que la definen; también se requiere el *tipo de dato* que resulta del cálculo de la

misma una serie de parámetros que servirán para darle una configuración y un comportamiento más adecuado. Estas características se explican a continuación.

Se necesitará saber si la fórmula va a ser calculada de manera continua una vez que se haya calculado la primera vez, es decir, determinar si los resultados que genera son *acumulables* en un periodo muy corto. Este tipo de fórmulas se puede dar cuando los valores de los operandos que se están utilizando se obtengan de manera continua y se sepa que una vez que tengan cierto valor, éste nunca disminuirá, lo que generará un cálculo y la generación de alarmas continuas. Esto es necesario saber ya que se requiere de un manejo diferente para este tipo de fórmulas.

Se necesitará saber si la definición de la fórmula va a servir como término para crear nuevas fórmulas (fórmulas base) o no (fórmulas personalizadas), ya que dependiendo de esto, se tendrán comportamiento y restricciones distintas.

Se necesitará especificar el *tipo* de acceso que se tendrá para el manejo de la fórmula. Esto consiste en asignarle un atributo que indique a los usuarios qué es lo que se puede modificar de la fórmula una vez que ha sido creada. En lo que se refiere a su definición, su configuración y su uso, se tendrán las siguientes tipos:

- **Público**- Cualquiera puede configurarla y utilizarla.
- **Reservado**- Puede ser utilizada por cualquier usuario.
- **Privado**- Sólo puede ser utilizada y configurada por el usuario que la creó.

La definición de las fórmulas tendrá que ser validada por un *parser*, el cual además de verificar la sintaxis, deberá traducir la definición de la fórmula de manera que sea fácil de guardar en la base de datos *DBMAD* y que también permita con facilidad identificar los operandos y las expresiones que la forman (*definición compilada*).

Las *expresiones* serán términos formados por uno o más operandos, cuyo valor se presenta al usuario que quiere saber su valor, independientemente del valor de los operandos y del resultado de las fórmulas.

Para las fórmulas base sólo se podrá tener una expresión que las defina; pero para las fórmulas personalizadas, se podrá tener una o más expresiones que formen una fórmula; de manera que el *parser* traductor deberá dejar la definición compilada en términos de claves que se asignarán a los operandos, a claves que se asignen a las expresiones que formen la fórmula y a claves que se asignen a las fórmulas base. De acuerdo con estos

requerimientos se tendrá las siguientes entidades y sus relaciones para modelar la parte de fórmulas.

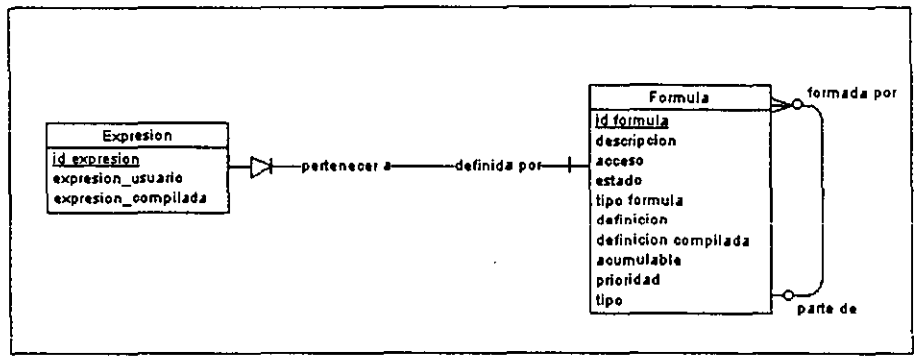


Figura 3.8 Esquema conceptual para el manejo de las Fórmulas

3.5.5 Las alarmas

Registrar los valores generados por el cálculo de una fórmula que cumplió con cierto valor, será registrado como una *alarma* que podrá ser notificada y consultada por los usuarios del sistema. Para cada una de éstas se necesita tener un *identificador*, saber la hora y fecha en que se generó (*tiempo*); en caso de ser acumulable, determinar si ya no se están generando nuevos resultados de forma continua (*activa*). Con la siguiente entidad quedan modelados estos requerimientos.

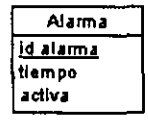


Figura 3.9 Entidad Alarma

3.5.6 Mensajes

Para las alarmas se requiere que los usuarios puedan realizar comentarios acerca de los resultados obtenidos en alguna alarma, y que estos comentarios se puedan enviar a uno o más usuarios del sistema. Para ello, se tendría que saber a quién va dirigido el mensaje (*emisor*), la fecha en que se envio (*tiempo*) y las *notas* o comentarios al respecto. Por lo tanto se requiere de una entidad que contenga estos atributos (fig. 3.10).

Figura 3.10 Entidad Mensaje

Hasta ahora se han definido las principales entidades que se necesitan para cumplir con los requerimientos de información presentados, y aunque ya se han definido ciertas relaciones entre las entidades, se hizo sólo para modelar los elementos principales. Falta definir las relaciones que se tienen entre estos elementos, para lo que se tendrá que recurrir a un análisis mas detallado de los requerimientos, auxiliandonos primero de una matriz de relaciones que nos muestre las entidades que se definieron y que identifique las relaciones entre sí, para posteriormente definir enunciados que nos indiquen qué tipo de relación se tiene entre ellas -de cardinalidad y obligatoriedad-. Además se tendrá que indicar cuáles de estas relaciones tienen atributos.

Como se mencionó anteriormente, los usuarios son los encargados de definir los operandos, las fórmulas y los disparadores. Sin embargo, de todos estos elementos, sólo en el caso de las fórmulas, se necesita saber qué usuario lo definió. Ya que en base a esto, se aplicarán restricciones en cuanto a si se puede o no modificar la definición y la configuración de la fórmula.

Por otro lado, se requiere llevar el control de los usuarios a los que hay que notificar los resultados que cierta fórmula genere, este control se llevará a cabo estableciendo un periodo que indicará si se tiene que notificar o no los resultados a determinado usuario.

Las fórmulas están formadas por varios operandos, y en el caso de las fórmulas personalizadas, estarán formadas por fórmulas base; además como se especificó anteriormente, estarán definidas por expresiones. No obstante, los operandos utilizados en las fórmulas pueden ser diferentes a los que el usuario requiera visualizar al momento que se calcule la fórmula, por lo tanto, se requiere saber además de los operandos que se utilizan en la definición de la fórmula, aquellos que el usuario quiera visualizar su valor una vez que se halla generado una alarma. Estos podrán ser cualquiera de los *atributos de hecho* recuperados por el disparador asociado a la fórmula, o bien cualquiera de los operandos definidos como columna o como valor fijo.

Como ya se mencionó, los disparadores son el medio por el cual se inicia el cálculo de las fórmulas, por lo que se tiene que recuperar los datos que servirán como valores de los operandos (*atributos del hecho*) en la definición de las fórmulas por calcular. Por lo tanto, cualquier fórmula tendrá que tener ligado un *disparador*, además, se deberá conocer los operandos que se recuperarán para cada disparador.

Las fórmulas a su vez podrán generar varias alarmas, cada una de las cuales podrán ser notificadas a varios usuarios; los usuarios pueden mandar mensajes que estén ligados a las alarmas, por lo tanto, se tendrá que llevar la cuenta del *número de mensajes* que se tienen para cada alarma notificada.

Las alarmas generarán varios datos, los cuales deben pertenecer a los valores de los operandos y los valores de las expresiones que forman las fórmulas. Estos valores necesitan ser almacenados de manera que se sepa a qué operando o expresión pertenecen y por cuál fórmula fueron generados.

3.5.7 Matriz de relaciones

Ahora, con base a los requerimientos, se necesita identificar las relaciones entre las entidades. Para esto utilizamos la matriz de relaciones en la tabla 3.5. De las entidades definidas se van a omitir aquellas que ya no tienen relación con las demás entidades, y cuyas relaciones se encontraron en el análisis previo. Las entidades son *Hecho*, *Control de lectura*, *Dato Hecho*, *Valor*, *Columna* y *Condición*.

Tabla 3.5 Matriz de relaciones

	Usuario	**Operando**	**Disparador**	**Fórmula**	**Expresión**	**Alarma**	**Atributo hecho**	**Mensaje**
Usuario				2R		R		R
Operando				2R		R		
Disparador				R			R	
Fórmula	2R	2R	R	R		R		
Expresión						R		
Alarma		R		R				
Atributo hecho			R					
Mensaje	R							

R Existe una relación entre las entidades.
2R Existe dos relaciones entre las entidades.

A continuación se mostrarán los enunciados que ayudan a definir los tipo de relación que existe entre las entidades en cuanto a la obligatoriedad y grado de asociación.

- Relaciones entre entidades *Usuario* y *Fórmula*.

1) Cada usuario puede ser el dueño de una o más fórmulas
2) Cada fórmula debe pertenecer a sólo un usuario

3) Cada fórmula puede ser configurada para notificar a uno o más usuarios
4) Cada usuario puede configurar que se le notifique una o más fórmulas

Para la segunda relación, se tendrán atributos que especifiquen el periodo para hacer la notificación, pues solo así se puede tener control en el conjunto de requerimientos (*fecha de inicio* y *fecha de fin*).

- Relación entre entidades *Usuario* y *Alarma*.

1) Cada usuario puede recibir la notificación de una o más alarmas.
2) Cada alarma puede ser notificada a uno o más usuarios.

Debido a que los mensajes dependen de la alarma a la que hacen referencia y se requiere saber a qué usuario se le envía, el número de mensajes será un atributo de esta relación.

- Relación entre *Usuario* y *Mensaje*.

1) Cada usuario puede mandar uno o más mensajes.
2) Cada mensaje debe ser hecho por sólo un usuario.

- Relaciones entre *Operando* y *Fórmula.*

1) Cada operando puede ser usado en la definición de una o más fórmulas.
2) Cada fórmula debe estar formada por uno o más operandos.
3) Cada operando puede ser visualizado en una o más fórmulas.
4) Cada fórmula puede visualizar uno o más operandos.

- Relación entre *Operando* y *Alarma*.

1) Cada operando puede generar un resultado en una o más alarmas.
2) Cada alarma debe implicar el cálculo de uno o más operandos.
3) Los resultados que se generen del cálculo de las alarmas arrojarán varios valores que corresponderán a operandos y pertenecerán a alguna alarma; el valor que se obtenga será un atributo de esta relación.

- Relación entre *Disparador* y *Fórmula.*

1) Cada disparador puede ser el *trigger* para el cálculo de una o más fórmulas.
2) Cada fórmula debe ser calculada por sólo un disparador.

- Relación entre *Disparador* y *Atributo del hecho.*

1. Cada disparador debe ser el medio por el cual se recuperen uno o más atributos del hecho.

2. Cada atributo del hecho puede ser recuperado por uno o más disparadores.

- Relación entre *Expresión* y *Alarma.*

3. Cada expresión puede generar un resultado en una o más alarmas.
4. Cada alarma debe implicar el cálculo de uno o más expresiones.

Los resultados que se generen del cálculo de las alarmas arrojarán varios valores que corresponderán a expresiones y pertenecerán a alguna alarma; el valor que se obtenga será un atributo de esta relación.

3.5.8 El Modelo conceptual

Con los enunciados del apartado 3.5.7, podemos finalmente unir todas las partes del primer análisis realizado a través de las relaciones que se tiene entre las distintas entidades. El *Modelo Conceptual* final es el presentado en la fig. 3.11.

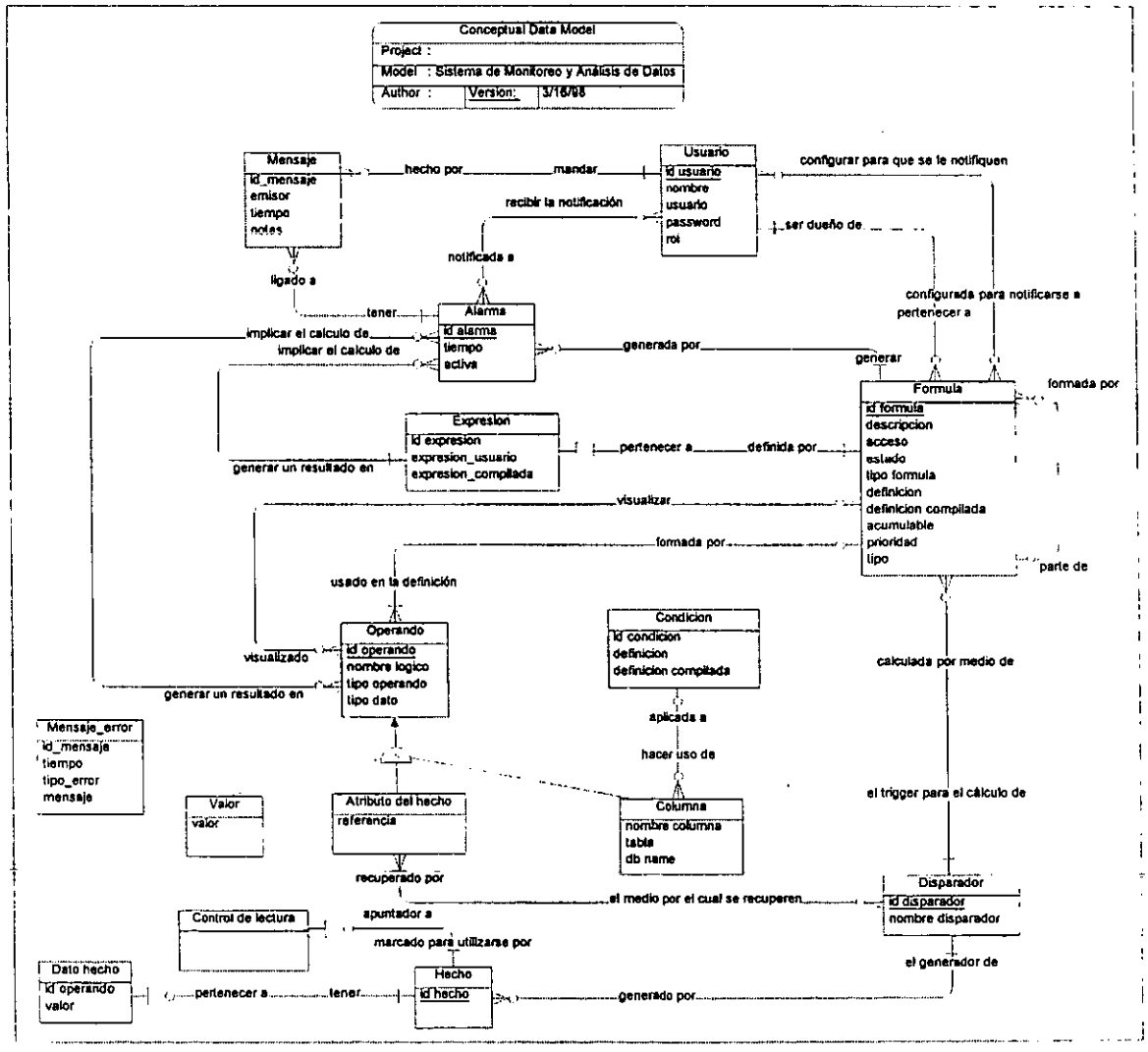


Figura 3.11 Modelo Conceptual final de la Base de Datos de Control

Capítulo 4. Arquitectura global del sistema

En la fig. 4.1 se esquematiza la arquitectura global del sistema, en ésta se muestran todos los componentes que conforman la configuración completa del *SMAD*, mismos que a continuación se explican.

4.1 Componentes

4.1.1 *Servidor de SQL*

Es el servidor de datos que contiene la base de datos de control *DBMAD* que el sistema utiliza para la configuración de usuarios, fórmulas, alarmas, etc. Este servidor también administra las bases de datos operacionales donde se almacenan los datos de interés de las operaciones.

4.1.2 *Servidor Monitor*

Es el servidor desarrollado en *lenguaje C* y librerías *Open Server/Open Client* que efectúa el monitoreo de los datos y el cálculo de las fórmulas. Es además el módulo que notifica a los usuarios la ocurrencia de las alarmas.

4.1.3 Módulo de Administración

Este módulo está desarrollado en *Power Builder* el cual permite administrar el sistema dando mantenimiento a la base de datos de control *DBMAD*. Mediante este módulo, los usuarios con privilegios de administrador podrán dar de alta a usuarios, definir disparadores y definir operandos con sus respectivas condiciones de búsqueda.

4.1.4 Módulo de Configuración

Es el módulo desarrollado en *Power Builder* que permite la definición de las fórmulas, establecer los periodos de vigencia de las fórmulas, así como la activación y desactivación de las mismas.

4.1.5 Módulo de Monitoreo

Este es otro módulo, también desarrollado en *Power Builder,* recibe las alarmas generadas por el *Servidor Monitor*, permite identificar la fórmula asociada a la alarma, así como los valores de los operandos que intervinieron en el cálculo de la fórmula.

4.1.6. Notificador

Es un módulo desarrollado en *Visual C++* y *Open Client* que permite la comunicación entre el *Servidor Monitor* y el *Módulo de Configuración*. Este módulo es el encargado de recibir directamente del *Servidor Monitor* la información de la notificación de las alarmas generadas, para posteriormente enviarla al *Módulo de Configuración*.

4.1.7. Interface

Este módulo está desarrollado en *Visual C++* y *Open Client*, sirve como medio de comunicación entre los módulos de *Administración* y de *Configuración* con el *Servidor Monitor*. Mediante este componente, el *Servidor Monitor* recibe las peticiones de activación, desactivación y definición de fórmulas; así como la definición de las condiciones de búsqueda de operandos, todas provenientes de los módulos en *Power Builder* arriba mencionados.

4.1.8 Base de datos de control (*Metabase*) *DBMAD*

Es la base de datos que mantiene toda la información necesaria para el funcionamiento del sistema. En ella se inscriben los usuarios válidos, almacena la definición de disparadores, operandos y fórmulas. La metabase *DBMAD* es el punto de origen del funcionamiento del sistema, pues son las transacciones que van llegando como hechos, las que desencadenan el análisis de los datos. Además cuando el *Servidor Monitor* genera una alarma, los datos de ésta también se almacenan en la base *DBMAD*.

4.1.9 Bases de Datos Operacionales

Son las bases de datos de propósito general, donde se almacenan los datos referentes a los procesos y sus operaciones por monitorear, los cuales serán el origen de los datos a analizar.

4.1.10 *Parser* Evalúa Condición

Es el analizador léxico y sintáctico que utiliza el *Servidor Monitor* para verificar que las sentencias de definición de las condiciones de búsqueda de los operandos, sean correctas. Este módulo está desarrollado con las herramientas de *UNIX*, *LEX* y *YACC*, para

correctas. Este módulo está desarrollado con las herramientas de *UNIX*, *LEX* y *YACC*, para el análisis léxico y sintáctico respectivamente.

4.1.11 Parser Analiza Fórmula

Este módulo es el analizador léxico y sintáctico con el que el *Servidor Monitor* se auxilia para verificar la correcta definición de las fórmulas. Este módulo revisa la compatibilidad de tipos, validez de operandos y concordancia de paréntesis. También está desarrollado con *LEX* y *YACC* de *UNIX*.

4.1.12 *Parser* Calcula Fórmula

Este es el tercer módulo implementado como analizador léxico y sintáctico. Su función es llevar a cabo el cálculo de las fórmulas que el *Servidor Monitor* requiera en tiempo de ejecución para sus tareas de monitoreo.

A diferencia de los dos analizadores anteriores, éste no sólo verifica la sintaxis de la definición, sino que además calcula la expresión y obtiene su valor final (fig. 4.1). También esta desarrollado en *LEX* y *YACC*.

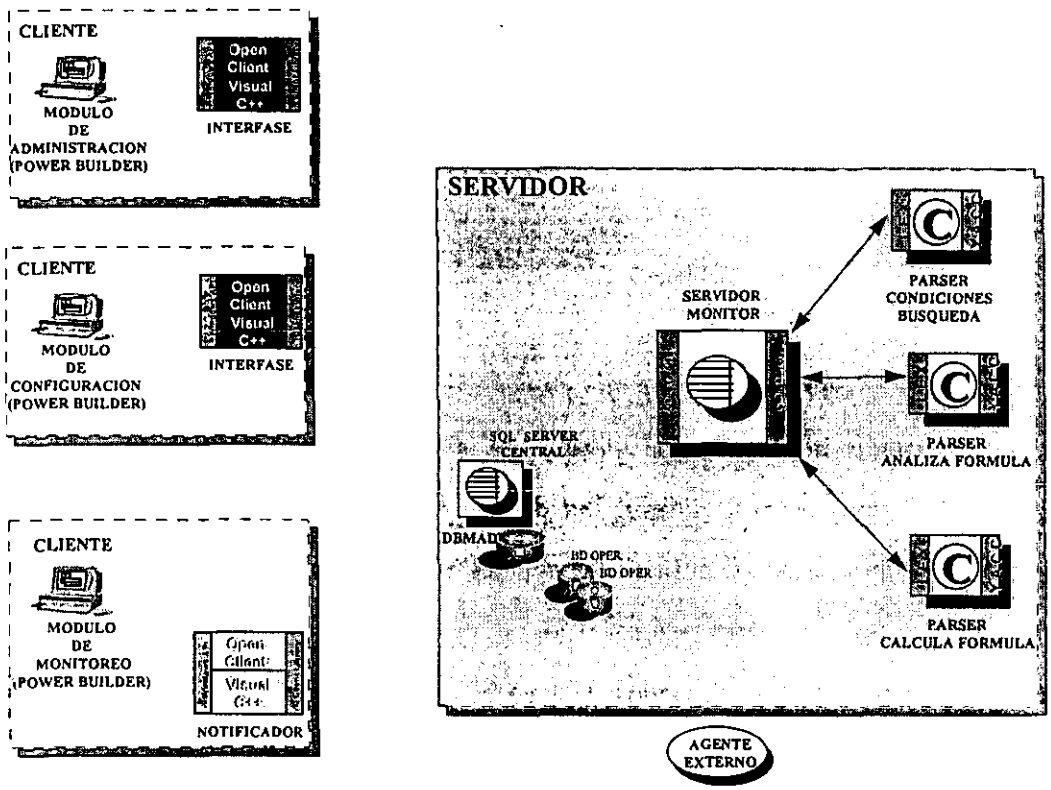


Figura 4.1 Componentes del SMAD

4.2 Procesos funcionales

4.2.1 Interacción con el *Módulo de Administración*

En la fig. 4.2 se muestra el diagrama de flujo de datos entre los componentes que se encuentran del lado del servidor y los componentes del lado del cliente que corresponden al *Módulo de Administración*.

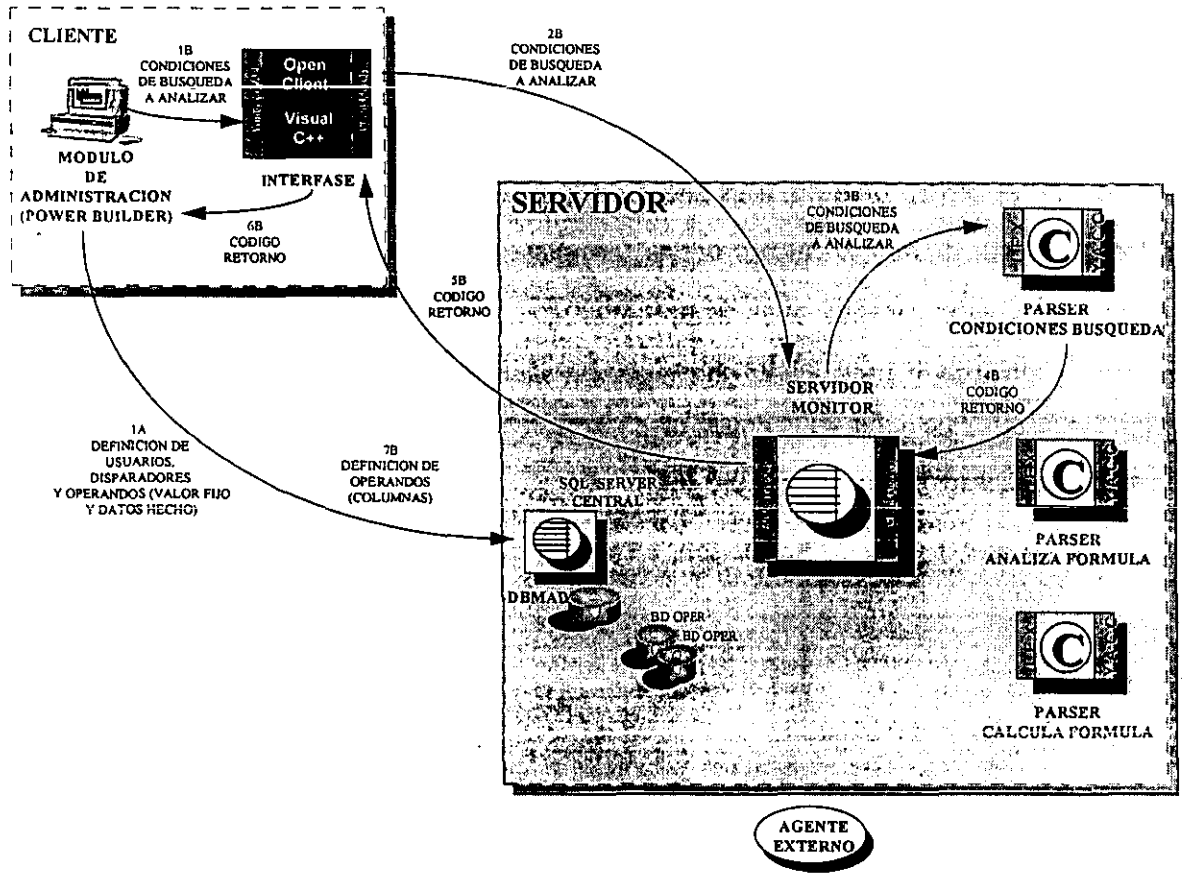


Figura 4.2 Interacción con el Módulo de Administración

Podemos dividir las tareas del *Módulo de Administración* en dos procedimientos diferentes. El primero permite definir, dar de alta y borrar a los usuarios válidos en el sistema, a los disparadores que definen los agentes externos que insertarán los hechos, y a los operandos definidos como valores fijos o valores recuperables de los hechos; el segundo permite definir los operandos definidos como columnas, que más tarde podrán utilizarse para la construcción de las fórmulas, así como darlos de baja.

El primer procedimiento es el más sencillo, pues sólo requiere de la interacción del *Módulo de Administración* con el *Servidor* de *SQL*, ya que para la definición de los usuarios, de los disparadores y de los operandos definidos como valores fijos o como datos del hecho, tan sólo se requiere almacenar su información asociada en la metabase *DBMAD*.

El *Módulo de Administración* cuenta con una pantalla de captura para usuarios, otra para disparadores y otra para operandos; así, una vez que son capturados los datos de interés, el *Módulo de Administración* se conecta como cliente al *Servidor* de *SQL* y le envía, en una sóla transacción, los datos capturados (paso 1A) para que estos se reflejen en la metabase *DBMAD*.

Por su parte, el segundo procedimiento de definición de operandos definidos como datos recuperables de columnas, necesita la ejecución de tareas tanto en el *Servidor* de *SQL* como en el *Servidor Monitor*.

Esto es necesario, puesto que todos los operandos definidos como columnas deben tener asociada una condición de búsqueda en las bases de datos operacionales, y que por supuesto debe analizarse léxica y sintácticamente antes de ser aceptada. Entonces, cuando el administrador da de alta un operando definido como columna, el *Módulo de Administración* envía al *Servidor Monitor*, a través de la *Interfase* (paso 1B y paso 2B), la definición de la condición de búsqueda asociada al operando.

Una vez que el *Servidor Monitor* ha recibido la petición, toma la expresión de la condición de búsqueda y la manda analizar con el *Parser Evalúa Condición* (paso 3B), quien tendrá que verificar que la expresión esté bien definida tanto léxicamente como en sintaxis.

Una vez analizada la expresión, el *parser* obtendrá un código de retorno asociado al tipo de dato final de la expresión o equivalente a error en la definición (paso 4B).

El *Servidor Monitor* devolverá al *Módulo de Administración*, una vez más a través de la interfaces (paso 5B y paso 6B), el código de retorno de la expresión de la condición, cuyo valor dependerá si el operando finalmente se puede dar de alta o no.

Si el código de retorno fuese de error en la expresión, el *Módulo de Administración* mandará al usuario el mensaje de error en la definición; por el contrario, si la expresión

hubiese sido bien definida, el *Módulo de Administración* insertará en la metabase *DBMAD* del *Servidor de SQL*, la definición del operando y de su condición de búsqueda (paso 7B).

4.2.2 Interacción con el *Módulo de Configuración*

A continuación, en la fig. 4.3 se muestra el diagrama de flujo de los datos que intervienen en las tareas de configuración, definición y eliminación de fórmulas. En ella vemos que el único medio a través del cual los usuarios pueden manipular sus fórmulas es el *Módulo de Configuración*. Éste permite la definición, eliminación, activación y desactivación de fórmulas, ya sean base o personalizadas.

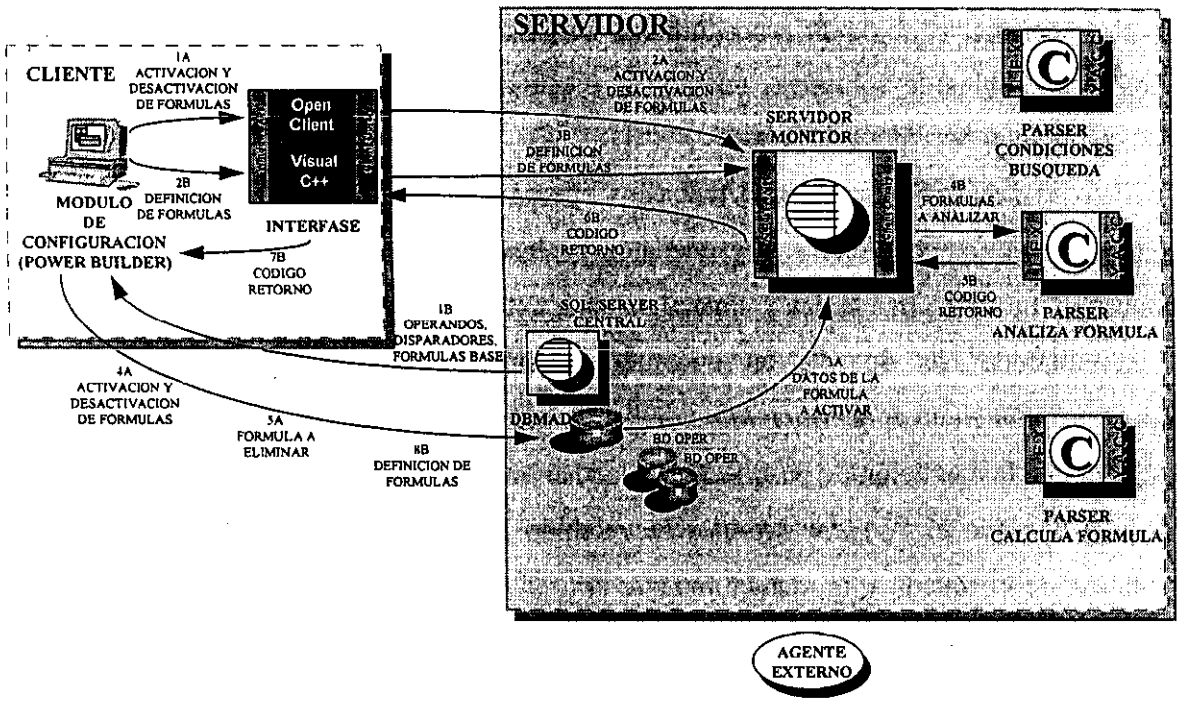


Figura 4.3 Interacción con el Módulo de Configuración

Veamos primero el mecanismo de activación y desactivación de las fórmulas ya definidas en la metabase *DBMAD*. La activación y la desactivación de una fórmula deben hacerse tanto en la memoria compartida del *Servidor Monitor*, como en la metabase *DBMAD* del *Servidor de SQL*. Cuando un usuario selecciona una fórmula para activar o desactivar, el *Módulo de Configuración* envía al *Servidor Monitor*, una vez más a través de la *Interfase*, la petición de activación o desactivación y la fórmula involucrada (paso 1A y paso 2A).

Luego, el *Servidor Monitor* toma la petición, y si ésta es de activación, extrae del *Servidor de SQL* la información asociada a la fórmula, contenida en *DBMAD* (paso 3A), y la carga en su memoria compartida; en cambio si la petición fuese de desactivación, el *Servidor Monitor* tan sólo borra de la memoria compartida toda la información de la fórmula en cuestión.

Finalmente, ya que se ha activado o desactivado la fórmula en el *Servidor Monitor*, se debe hacer también en el *Servidor de SQL*. Si la petición hubiese sido de desactivación, el *Módulo de Configuración* desactiva la fórmula en la base de datos de control *DBMAD* (paso 4A). Pero si hubiese sido de activación, el módulo activa la fórmula también en *DBMAD* por cierto periodo (paso 4A). Si el usuario lo prefiere, puede eliminar una fórmula, pero primero deberá desactivarla y luego eliminarla de *DBMAD* (paso 5A).

Veamos ahora el mecanismo que permite la creación de las fórmulas. Una vez que el *Módulo de Configuración* ha obtenido de *DBMAD* la información de todos los operandos y fórmulas base disponibles para la creación de nuevas fórmulas, y la información de los disparadores a los cuales puede asociarlas (paso 1B), el usuario puede elegir de toda esa lista de operandos y fórmulas base las variables necesarias para definir la nueva fórmula.

Una vez que el usuario ha definido la expresión de la nueva fórmula, ésta se envía al *Servidor Monitor* mediante la *Interfase* (paso 2B y paso 3B).

Cuando el *Servidor Monitor* obtiene la expresión, la evalúa con la ayuda del *Parser Analiza Fórmula* (paso 4B) y obtiene el código asociado al resultado final de la expresión (paso 5B), que como hemos visto, puede ser el código asociado al tipo de dato o el código de error en la definición. El *Servidor Monitor* vuelve a enviar de regreso al *Módulo de Configuración* el valor del código de retorno de la expresión (paso 6B y paso 7B), con lo que este último sabrá si debe o no dar de alta la fórmula.

Posteriormente, con el valor del código de retorno, el *Módulo de Configuración* verifica si la expresión fue correcta, de ser así, da de alta en la metabase *DBMAD* la fórmula recién creada (paso 8B), la expresión que la define y el disparador para el cual se ha asociado. Por el contrario, de no ser correcta la expresión, el módulo simplemente le especifica al usuario que hubo un error en la expresión y que debe corregirla.

4.2.3 Interacción con el *Módulo de Monitoreo*

A continuación se muestra en la fig. 4.4 el flujo de la información involucrada en el proceso de monitoreo de los datos y el cálculo de las fórmulas. Todo el proceso se desencadena al presentarse una transacción originada por algún disparador o agente externo al sistema. Cuando un agente externo realiza una operación que involucre una manipulación de datos en las bases de datos operacionales (paso 1), éstas automáticamente insertan un hecho en la base de datos de control *DBMAD* (paso 2). Este hecho va acompañado de los datos de interés que se vieron involucrados en la transacción realizada por el agente externo.

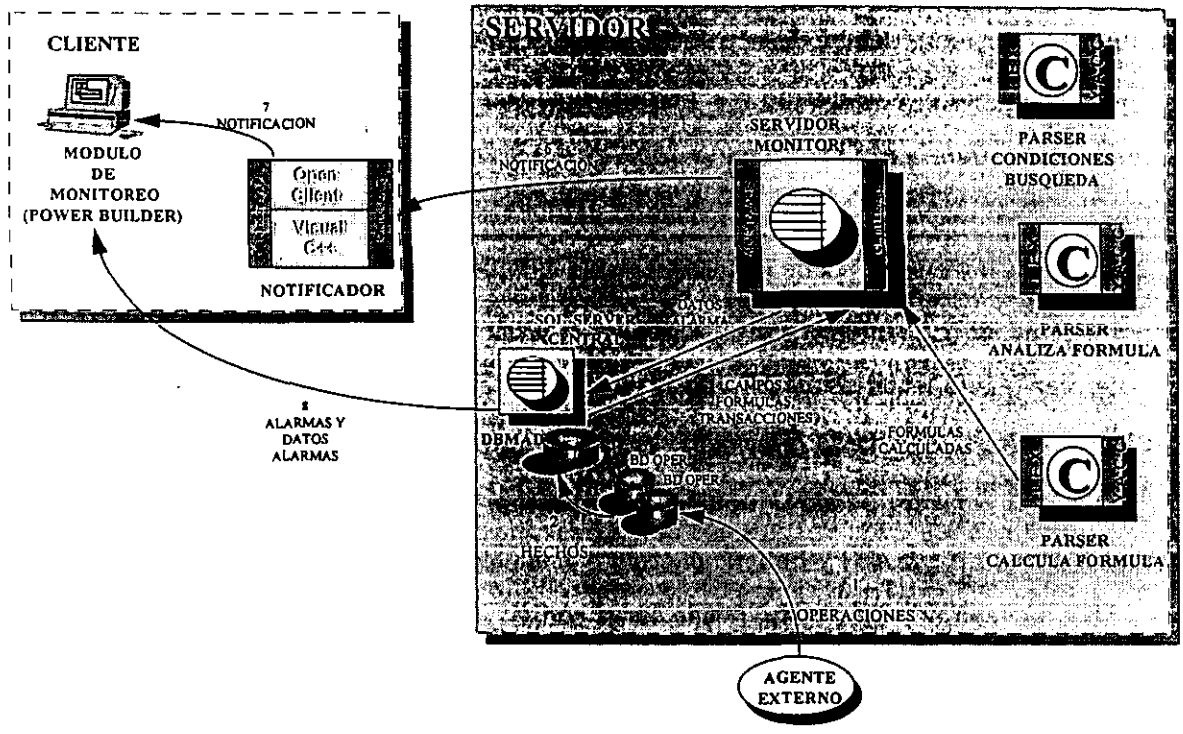


Figura 4.4 Interacción con el Módulo de Monitoreo

Una vez que el hecho ha sido insertado en la base de datos *DBMAD*, el *Servidor Monitor* lo detecta automáticamente, obtiene el disparador asociado al hecho y con él, extrae de la metabase *DBMAD* toda la información sobre la definición de las fórmulas asociadas al disparador, así como los valores de los operandos que definen estas fórmulas (paso 3).

Capítulo 5. *Servidor Monitor*

5.1 Descripción de la arquitectura interna

El *Servidor Monitor* es una aplicación multihilos *Open Server/Open Client* codificado en *lenguaje C* y diseñado para poder ejecutarse prácticamente en cualquier sistema *UNIX*[1].

La función del *Servidor Monitor* es llevar a cabo el control de los procesos de monitoreo de las fórmulas y de notificación a los usuarios sobre las alarmas generadas. El *Servidor Monitor* es el encargado de atender las peticiones de conexión y de ejecución de *RPC* provenientes de los clientes, para definir, activar y desactivar fórmulas, evaluar y calcular las fórmulas definidas por los datos y operandos recuperables, y notificar oportunamente a los usuarios correspondientes las alarmas generadas por el cálculo previo de las fórmulas.

Como ya se ha explicado, la arquitectura interna del *Servidor Monitor* está basada en la tecnología mutihilos, por lo que, cada petición que le sea solicitada, será atendida y procesada por uno o varios hilos de servicio.

El mecanismo de comunicación entre los hilos utiliza colas de mensajes para el intercambio de información y el uso de *mutex* para la administración y control de recursos compartidos, como pueden ser estructuras de datos, memoria compartida y descriptores de archivos.

La arquitectura del *Servidor Monitor* está diseñada para atender los siguientes servicios:

- Lectura y evaluación de los hechos y sus respectivos datos.
- Almacenamiento y notificación de las alarmas generadas.
- Depuración diaria de los hechos evaluados, en el horario que no afecte el desempeño de las operaciones de monitoreo establecido por el administrador.
- Activación y desactivación de fórmulas.
- Carga en memoria de las fórmulas activas y de la información necesaria para el funcionamiento del *Servidor Monitor*.
- Análisis léxico y sintáctico para la definición de las fórmulas.

---

[1] Es necesario que el sistema *UNIX* tenga instaladas las bibliotecas de *Open Server/Open Clien.*

- Análisis léxico y sintáctico de las condiciones de búsqueda de los operandos definidos como campos.
- Evaluación de las fórmulas definidas en la base de datos *DBMAD*, cuyo resultado dependerá la generación o no de una alarma.

5.2 Elementos de la arquitectura interna del Servidor Monitor

En la fig. 5.1 se observan los componentes principales de la arquitectura interna del *Servidor Monitor*, estos son de seis tipos diferentes: hilos de servicio, colas de mensajes (*messages queues*), procedimientos registrados (*RPC*), semáforos de exclusión mutua (*mutex*), *parsers* y estructuras de datos. A continuación se describe la funcionalidad de cada uno de ellos.

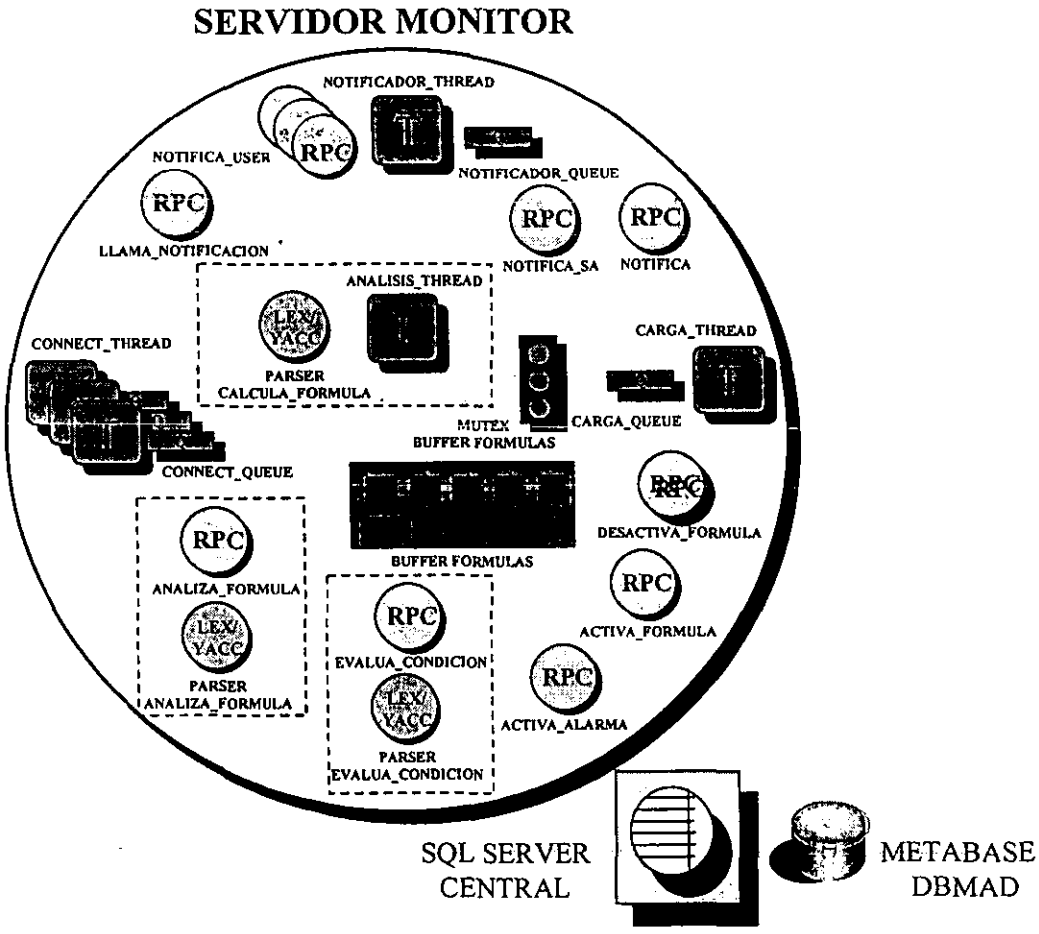


Figura 5.1 Arquitectura interna del Servidor Monitor

5.2.1 Hilos

*5.2.1.1 Hilo de conexión (*CONNECT_THREAD*)*

El *hilo de Conexión* es el hilo que se encarga del control de las conexiones de los clientes y del manejo de los eventos que estos originen.

Cuando un cliente abre una conexión con el *Servidor Monitor*, éste crea automáticamente tanto el *CONNECT_THREAD* como una cola de mensajes asociada (*CONNECT_QUEUE*) para manejar la conexión, luego, coloca en dicha cola el evento *CONEXION* (*SRV_CONNECT*). De esta forma cuando el hilo entra en ejecución, realiza la rutina instalada para el evento *SRV_CONNECT*.

Cada *hilo de Conexion* ejecuta la rutina *connect_handler*, cuya función es registrar al cliente y verificar sus datos (*login* y *password*); crear para cada usuario -en caso de no estar creado- el procedimiento registrado *NOTIFICA_USER* que se utilizará en el esquema de notificación de alarmas, donde *USER* es el *login* del usuario; además, registrar al usuario en el *Servidor Monitor* para que le sean notificadas las alarmas mediante el procedimiento registrado *NOTIFICA_USER* previamente creado. Así cada usuario con el *login USER* tendrá asociado un procedimiento registrado *NOTIFICA_USER* para la notificación de las alarmas.

*5.2.1.2 Hilo de Carga (*CARGA_THREAD*)*

El *hilo de Carga* es el hilo de servicio encargado de mantener en memoria en el *BUFFER_FÓRMULAS* la información de las fórmulas definidas que será útil para su cálculo.

El *hilo de Carga* es el primer hilo de servicio en entrar en ejecución. Como primera tarea, el *Servidor Monitor*, a través del *hilo de Carga*, obtiene de la base de datos *DBMAD* toda la información relacionada con la definición de todas las fórmulas y la almacena en el *BUFFER_FÓRMULAS*.

Una vez cargadas las fórmulas en memoria, el *hilo de Carga* sólo se activará al recibir los siguientes mensajes en la cola de mensajes *CARGA_QUEUE*:

- **Activa fórmula**- Carga la fórmula correspondiente en el *BUFFER_FÓRMULAS*.
- **Desactiva fórmula**- Desactiva la fórmula en el *BUFFER_FÓRMULAS* y la borra de la tabla *Notifica*.

De este modo, será responsabilidad de este hilo:

1) Controlar que la fecha de activación de la fórmula sea la menor entre todas las fechas de activación relativas a cada usuario en la tabla *Notifica*.
2) Controlar que cada fecha de desactivación de la fórmula sea la mayor entre todas las fechas de desactivación relativas a cada usuario en la tabla *Notifica*.

5.2.1.3 Hilo de Análisis (ANÁLISIS_THREAD)

El *hilo de Análisis* por su parte, es el hilo de servicio encargado de aplicar constantemente las fórmulas correspondientes a los nuevos hechos. Dependiendo del resultado del cálculo de las fórmulas, el *hilo de Análisis* generará alarmas y las notificará a los usuarios correspondientes que en ese momento estén conectados.

Debido a que el *hilo de Análisis* necesita obtener la información de las fórmulas del *BUFFER_FÓRMULAS*, al arrancar el *Servidor Monitor*, el *hilo de Análisis* siempre entrará en ejecución después del *hilo de Carga*. Esto garantiza que los datos de las fórmulas estén en memoria antes de que el *hilo de Análisis* los requiera.

Así, al detectar un nuevo hecho, el *hilo de Análisis* debe efectuar las siguientes operaciones:

1) Identificar y recuperar todos los operandos ligados al disparador que insertó el nuevo hecho; incluyendo aquellos valores que hayan sido definidos previamente y aquellos que vengan acompañados en la inserción del hecho.
2) Identificar y calcular con los valores de los operandos antes recuperados, cada una de las fórmulas activas que estén ligadas al disparador.
3) Desactivar las fórmulas cuyo tiempo de activación haya expirado (fórmulas no activas), mandando el mensaje de *Desactiva Fórmula* al *hilo de Carga* a través de la cola de mensajes *CARGA_QUEUE*. En este caso se suspenderá el cálculo completo de la fórmula y simplemente no se generará ninguna alarma por dicha fórmula.
4) Insertar en la base de datos *DBMAD*, los datos referentes a las alarmas que hayan sido generadas por el cálculo previo de las fórmulas; además, ejecutar el *RPC NOTIFICA_USER* para enviar la notificación a los clientes correspondientes que se encuentren conectados.

5.2.1.4 Hilo de Notificación (NOTIFICADOR_THREAD)

El *hilo de Notificación* es el último hilo de servicio en iniciar su ejecución. Este hilo es el encargado de recibir en su cola de mensajes *NOTIFICACIÓN_QUEUE* las peticiones de notificación enviadas por el *hilo de Análisis* (cada vez que se genera una alarma) y ejecutar los *RPC* de notificación correspondientes a cada usuario, que pueden ser: *NOTIFICA_USER* y *NOTIFICA_SA*.

5.2.2 Colas de mensajes (*messages queues*)

5.2.2.1 Cola de mensajes de Conexión (CONEXIÓN_QUEUE)

Existe una cola de mensajes de *Conexión* por cada *hilo de Conexión* creado. Al igual que este último, cada cola de mensajes de *Conexión* es creada automáticamente por el *Servidor Monitor* cuando se presenta el evento *CONEXIÓN* (*SRV_CONNECT*).

La función de cada *CONEXIÓN_QUEUE* es la de recibir cada evento que se envía al hilo asociado. Cada vez que ocurre un evento de un cliente, el *Servidor Monitor* coloca el evento en la cola de mensajes de *Conexión* asociada al *hilo de Conexión* que atiende al cliente; de este modo, la siguiente vez que el *hilo de Conexión* se ejecute, leerá la siguiente petición del evento recibido y llamará a la rutina asociada a dicho evento. Cuando esta rutina termine de ejecutarse, el hilo tratará de leer el siguiente evento en la cola. Si no hubiese otro evento, simplemente el hilo se dormirá.

5.2.2.2 Cola de mensajes de carga (CARGA_QUEUE)

Esta cola de mensajes es la cola asociada al *hilo de Carga*. Es a través de ella que el *hilo de Carga* recibe las peticiones de activación y desactivación de fórmulas.

Para activar o desactivar las fórmulas, el *hilo de Carga* lee de la *CARGA_QUEUE* los siguientes mensajes correspondientes a las peticiones de activación y desactivación respectivamente:

- **Activa fórmula**- Carga la fórmula correspondiente en el *BUFFER_FÓRMULAS*.
- **Desactiva fórmula**- Desactiva la fórmula en el *BUFFER_FÓRMULAS* y la borra de la tabla *Notifica*.

Además recibe simultáneamente a la petición, el identificador de la fórmula que el *hilo de Carga* debe afectar (activar o desactivar). En la *CARGA_QUEUE* se escriben y leen mensajes formados por la estructura *CARGA_REQUEST* que incluye tanto el tipo de petición como el identificador de la fórmula.

*5.2.2.3 Cola de mensajes de notificación (*NOTIFICADOR_QUEUE*)*

La cola de mensajes de notificación está asociada al *hilo de Notificación*. Es a través de ella que se escriben y leen las peticiones de notificación a los usuarios. El *hilo de Análisis* manda escribir mensajes a esta cola cada vez que se presenta una alarma, posteriormente el *hilo de Notificación* toma los mensajes recibidos y ejecuta para cada uno de ellos los *RPC* de notificación correspondientes.

Los mensajes que se leen y escriben en esta cola, están definidos por la estructura *NOTIFICACIÓN_REQUEST*, donde se incluyen el usuario a notificar, el identificador de la alarma generada y el identificador de la fórmula que generó la alarma.

5.2.3 Estructuras de datos

*5.2.3.1 Estructura de datos para la notificación (*NOTIFICACIÓN_REQUEST*):*

La estructura de datos *NOTIFICACIÓN_REQUEST* la utiliza el *Servidor Monitor* para hacer peticiones de notificación al *hilo de Notificación*. Así, cualquier agente tanto interno como externo al servidor, tendrá que hacer peticiones al *hilo de Notificación* utilizando esta estructura. A ésta la definen el identificador del usuario (*id_usuario*) a notificar, el identificador de la alarma (*id_alarma*) que causa la notificación, y el identificador de la fórmula (*id_fórmula*) que originó la alarma.

*5.2.3.2 Estructura de datos para la activación y desactivación de fórmulas (*CARGA_REQUEST*)*

La estructura de datos *CARGA_REQUEST* define las peticiones de activación y desactivación de fórmulas dentro del *Servidor Monitor*. El *hilo de Carga* recibe tales peticiones como mensajes a través de esta estructura de datos. Está definida por el identificador de la fórmula (*id_fórmula*) y la acción a realizar (activación o desactivación).

*5.2.3.3 Buffer de fórmulas y disparadores (*BUFFER_FÓRMULAS*)*

El *BUFFER_FÓRMULAS* es la estructura de datos que contiene la información de todos los disparadores dados de alta en la base de datos *DBMAD*. Contiene además todas las definiciones de las fórmulas asociadas a tales disparadores, los operandos que definen a las fórmulas, así como las sentencias *SQL* de recuperación de estos operandos. El *Servidor Monitor* utiliza esta estructura para obtener toda la información necesaria para el cálculo de las fórmulas.

5.2.4 Semáforos de exclusión mutua (*MUTEX*)

5.2.4.1 MUTEX BUFFER_FÓRMULAS

El *mutex* para el *BUFFER_FÓRMULAS* permite que los hilos de *Análisis* y de *Carga* puedan compartir la misma área de memoria (*BUFFER_FÓRMULAS*). Este semáforo permite que ambos hilos compartan datos, mediante regiones críticas, sin que éstos tengan acceso a los mismos datos al mismo tiempo.

5.2.5. Procedimientos registrados (*RPCS*)

5.2.5.1 Procedimiento registrado NOTIFICA_USER

Es el *RPC* utilizado para la notificación de alarmas a los clientes conectados con un mismo *login* (*USER*). Tiene como parámetros de entrada, el identificador de la alarma y el identificador de la fórmula asociada, y como parámetros de salida, el número de operadores en la fórmula, el número de expresiones en la fórmula, la definición de la fórmula, la lista de operandos en la fórmula y la lista de expresiones en la fórmula.

Existe un *RPC NOTIFICA* para cada grupo de conexiones con el mismo *login (USER)* de usuario válido que esté activo en el *Servidor Monitor*. Cuando un cliente se conecte y sea la primera conexión activa para el *login* del usuario dado, el *Servidor Monitor* creará automáticamente el *RPC* asociado a ese *login*. Por otro lado, cuando el cliente se desconecte, si fuese la última conexión activa asociada al *login*, el *Servidor Monitor* eliminará el *RPC* asociado al *login*.

5.2.5.2 Procedimiento registrado NOTIFICA_SA

EL *RCP NOTIFICA_SA* es utilizado para la notificación de errores al administrador del sistema (*SA*). Este *RCP* a diferencia de los *RCP NOTIFICA_USER* se utiliza cuando por algún motivo, el cálculo de una fórmula no ha podido llevarse a cabo. Entre tales motivos se encuentran:

- fallas en la obtención de los datos de las *Bases de datos operacionales*,
- recuperación fallida de operandos de tipo columna (recuperación de cero o más de un dato) producidos por una definición equivoca de la cláusula de búsqueda, y
- error de tipos en la sustitución de los datos que acompañan a los hechos en las fórmulas relacionadas.

Sólo es posible detectar este tipo de errores en tiempo de ejecución y no al momento de definir las fórmulas; lo que implica una gran responsabilidad del *SA* al momento de definir operandos, fórmulas y disparadores.

Si se presentase cualquiera de los errores mencionados, el *hilo de Análisis* escribirá el mensaje de error en la cola de mensajes *NOTIFICADOR_QUEUE* del *hilo de Notificación*, quien será el que invoque al *RPC NOTIFICA_SA*; Éste recibe como parámetro el número de error producido y lo envía tal cual al cliente.

5.2.5.3 Procedimiento registrado LLAMA_NOTIFICACIÓN

Con el procedimiento registrado *LLAMA_NOTIFICACIÓN*, los usuarios pueden mandar mensajes a otros usuarios, en el momento que se ha notificado una alarma.

Cuando el *Servidor Monitor* genera una alarma y ésta es notificada al usuario que la definió, éste podrá compartir información con otro usuario al cual no le haya sido notificada la alarma. Gracias a este *RPC*, los usuarios pueden comunicarse entre sí cuando se presenten las alarmas, mediante el envío de pequeños mensajes escritos (hasta 255 caracteres).

Este *RPC* recibe como parámetros, el identificador del usuario, el identificador de la alarma a la cual se relaciona el mensaje y el identificador de la fórmula que causó la generación de la alarma. Con estos datos, el *RPC* escribe un mensaje en la cola de mensajes *NOTIFICADOR_QUEUE* del *hilo de Notificación*, quien finalmente será el que

mande una nueva notificación -junto con el mensaje escrito- al usuario destino, pese a que éste no sea el dueño de la fórmula.

5.2.5.4 Procedimiento registrado ANALIZA_FÓRMULA

El *RCP ANALIZA_FÓRMULA* se vale de un analizador léxico y sintáctico (*PARSER ANALIZA_FÓRMULA*) para llevar a cabo su tarea.

El *RPC ANALIZA_FÓRMULA* efectúa el análisis léxico y sintáctico de la definición de las fórmulas; para ello, sustituye los nombres lógicos de los operandos o de las fórmulas base por el código asociado a su tipo de datos de acuerdo a la tabla 5.1.

Tabla 5.1 Código asociado al tipo de datos de los operandos

Código	Descripción
#b&	donde #b& indica operando de tipo booleano.
#r&	donde #r& indica operando de tipo real.
#c&	donde #c& indica operando de tipo cadena.

Esta nueva expresión, a la cual llamaremos *expresion_tipos*, es la definición que de hecho se analiza por el *PARSER ANALIZA_FÓRMULA*, el cual evaluará semánticamente de acuerdo a los tipos de datos de la *expresion_tipos*, si la fórmula está bien definida.

Si el *parser* no encontró error alguno, el *RPC* lleva a cabo otra sustitución en los operandos de la fórmula. Esta vez sustituye los nombres lógicos de los operandos o de las fórmulas base por sus identificadores como en el formato de la tabla 5.2.

Tabla 5.2 Código del formato de los operandos en base a sus identificadores

Códigos	Descripción
#O&XXXXXX	donde XXXXXX es el *id_operando*
#F&XXXXXX	donde XXXXXX es el *id_fórmula*
#E&XXXXXX	donde XXXXXX es el *id_expresión*

Esta otra expresión, a la que llamaremos *expresión_compilada*, se regresa al cliente, quien finalmente será el encargado de guardar la definición de la fórmula en la metabase *DBMAD* para su futuro cálculo.

El *RPC* devuelve el tipo del resultado que generará la fórmula (obtenido del *parser*), que puede ser de tipo real, fecha, booleano, cadena, o bien, un código de error producido por una fórmula mal definida.

5.2.5.5 Procedimiento registrado EVALUA_CONDICIÓN

Es el *RPC* que efectúa el análisis léxico y sintáctico de una condición (cláusula *WHERE*) para la búsqueda de operandos de tipo columna; además sustituye los nombres lógicos de los operandos por sus identificadores con el siguiente formato:

#O&XXXXXX donde XXXXXX es el *id_operando*

El *RCP EVALUA_CONDICIÓN* utiliza un analizador léxico y sintáctico (*PARSER EVALUA CONDICIÓN*), que le permite establecer si la condición de búsqueda es válida.

Al momento de la definición de la condición, el *Servidor Monitor* no conoce el tipo de dato asociado a una columna, por lo que el *parser* ignorará el tipo de dato asociado a las columnas, y tan sólo verificará los tipos de datos de los operandos que hayan sido dados de alta previamente en la base de datos de control *DBMAD* y los tipos de los valores constantes.

Por lo anterior, el administrador debe conocer con certeza el tipo de dato de la columna a recuperar, de lo contrario, el error en la definición de la condición de búsqueda se presentará precisamente en el momento en que se trate de recuperar el dato. Además, es responsabilidad del administrador definir la condición de búsqueda del campo, de tal forma que sólo se obtenga un dato; de lo contrario, el *Servidor Monitor* sólo detectará el error en la recuperación de datos en el momento de realizar la búsqueda.

Pese a lo anterior, si el *Servidor Monitor* llegase en tiempo de ejecución a detectar algún error relacionado con lo arriba mencionado, mandará una notificación de error (con el *RPC NOTIFICA_SA*) únicamente al administrador del sistema, y escribirá las posibles causas en el archivo de *log* del *Servidor Monitor* y en la tabla *MENSAJE_ERROR*, misma que podrá ser consultada por el administrador.

5.2.5.6 Procedimiento registrado ACTIVA_FÓRMULA:

El *RPC ACTIVA_FÓRMULA* es utilizado para enviar un mensaje de activación en la cola de mensajes *CARGA_QUEUE* del hilo *Carga*. De este modo, se carga la fórmula en la memoria del *Servidor Monitor*.

5.2.5.7 Procedimiento registrado DESACTIVA_FÓRMULA

El *RPC DESACTIVA_FÓRMULA* se utiliza para enviar un mensaje de desactivación en la cola de mensajes *CARGA_QUEUE* del *hilo de Carga*. Con esto, se elimina de la memoria del *Servidor Monitor* la información de la fórmula.

5.2.6. Analizadores Léxicos y Sintácticos (*Parsers*)

5.2.6.1 Análisis léxico

A continuación se explica la forma en que definimos la especificación de aceptación y reconocimiento de patrones de cadenas válidas para la construcción de fórmulas.

Para ello definimos el conjunto de las expresiones regulares, que a su vez definen a cada uno de los componentes léxicos que integran los patrones válidos[2].

Sabemos que una expresión regular puede construirse a partir de expresiones regulares más simples, utilizando un conjunto de reglas de definición o reglas léxicas. Así, utilizamos la notación *EBNF* (*Extended Backus-Naur form)* para definir el conjunto de reglas léxicas. Esta notación consta de símbolos como se muestra en la tabla 5.3.

Tabla 5.3 Símbolos de la notación *EBNF*

→	*Se define como*
<,>	*Establece los símbolos no terminales*
|	*Operador de exclusión*
*	*Cero, una o más ocurrencias*
+	*Una o más ocurrencias*
?	*Cero o una ocurrencia*

Así por ejemplo, **<entero>** → **<digito>** +, es una regla de producción. En este caso **<entero>** es producido por una o más ocurrencias de miembros del conjunto de todos los dígitos definido por el símbolo no terminal **<digito>**.

[2] Para más información sobre el diseño de los *Analizadores Léxicos*, véase el Apéndice G. Codificación de las expresiones regulares y gramáticas.

Por otra parte, debemos aclarar que con el fin de simplificar la notación y de *mapaer* directamente el diseño del analizador léxico a su implantación, haremos referencia a los símbolos no terminales como las palabras en negritas.

Además introducimos los símbolos metalingüísticos de asociación *(,)*, los para la definición de rango de valores símbolos **[, - y]**, el punto **(.)** para especificar la clase de todos los caracteres *ASCII* (excepto el caracter de nueva línea), y \ como símbolo de escape.

De esta forma, por ejemplo, podemos definir al conjunto de todas las cadenas, que a su vez definen a los números reales a través de las siguientes reglas léxicas:

dígito → [0-9]
entero → **dígito** +
real → **entero** (\.**entero**) ? ([E | e] [+ \ -] ? **entero**) ?

5.2.6.2 Análisis sintáctico

Ahora vamos a explicar cómo se diseñaron los analizadores sintácticos, es decir, la especificación formal del lenguaje diseñado para la definición de fórmulas, la cual está basada en el cálculo de expresiones.[3]

Para ello construimos un metalenguaje a base de gramáticas independientes del contexto, pues éstas nos permiten la especificación de lenguajes no regulares, como por ejemplo, aquellos que aceptan las cadenas formadas por estructuras anidadas, tales como paréntesis anidados.[4]

Igual que para el diseño de los analizadores léxicos, nos basámos en una versión adaptada de la notación *EBNF* para la especificación de las reglas sintácticas. Así por ejemplo, la siguiente regla sintáctica especifica la clase de las expresiones aritméticas, como un conjunto de producciones recursivas de *expr*, y donde **real** es el símbolo terminal que forma el conjunto de todos los números reales.

expr → *expr* + *expr* | *expr* - *expr* | *expr* * *expr* | *expr* / *expr* | (*expr*) | **real**

[3] Para más información sobre el diseño de los *Analizadores Sintácticos*, véase el Apéndice G. Codificación de las expresiones regulares y gramáticas.

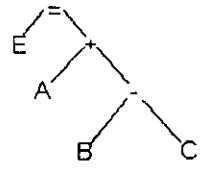
[4] Los lenguajes no regulares no pueden construirse a partir de autómatas finitos ni a partir de expresiones regulares.

Como podemos observar, la regla puede producir derivaciones de símbolos no terminales tanto por la derecha, como por la izquierda. Veámos el caso de la siguiente expresión, donde la operacion de suma puede ser asociativa hacia la izquierda o la derecha:

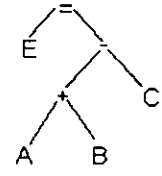
$$E = A + B - C$$

Podemos darnos cuenta de que primero podría realizarse la operación B - C y el resultado sumarse a A, en este caso la asociación y la derivación serían hacia la derecha; por otro lado, también pudo haberse realizado primero la operación A + B y al resultado restarle C, entonces la asociación y la derivación son hacia la izquierda.

Las gramáticas que especifican la clase de expresiones de este tipo se les conoce como gramáticas ambiguas, puesto que el árbol de derivación correspondiente puede ramificarse hacia ambos lados del símbolo no terminal (fig. 5.2).



Derivación hacia la derecha



Derivación hacia la izquierda

Figura 5.2 Ejemplo de dos árboles de derivación para la expresión E = A + B - C

Lo anterior podría resultar en análisis erróneos de las expresiones. Por ejemplo en expresiones que involucren los operadores * y +, donde el orden de precedencia es importante; o bien en expresiones que involucren el operador -, pues este puede comportarse como operador unario y asociarse a la derecha (números negativos), o actuar como operador binario (resta). Para evitar este tipo de ambiguedades, se tuvo que considerar tanto el orden de precedencia de los operandos, así como el sentido de su asociación.

Con esto indicamos además el sentido de la derivación para cada tipo de operando, evitando en la práctica la ambigüedad de las gramáticas en su especificación formal matemática.

5.2.6.3 Parser ANALIZA_FÓRMULA

El *PARSER ANALIZA_FÓRMULA* es el componente encargado de realizar directamente el análisis léxico y sintáctico de la definición de las fórmulas. El *RPC ANALIZA_FÓRMULA* se auxilia de este *parser* para llevar a cabo tal análisis.

Cuando se ejecuta el *RCP*, éste recibe la fórmula a ser analizada y sustituye los operandos por el código asociado a su tipo de datos para así obtener la fórmula expresada en tipos de datos (*expresion_tipos*).

El *PARSER ANALIZA FÓRMULA* toma la *expresion_tipos* y evalúa si su definición es correcta; tal definición se lleva a cabo en dos partes, la primera es el análisis léxico y está implementada en un programa en *LEX*, y la segunda es el análisis sintáctico y está implementada en un programa en *YACC*.

Una vez hecho el análisis, el *parser* devuelve el código de retorno correspondiente al tipo de dato que genera la fórmula (booleano, real o fecha), o bien, un código de error de definición.

5.2.6.4 Parser CALCULA_FÓRMULA

A diferencia del *parser* anterior, el *PARSER CALCULA_FÓRMULA*, no sólo analiza la fórmula, sino que también la calcula. De hecho, este *parser* es un componente complementario del *hilo de Análisis*, ya que este último es quien en tiempo de ejecución sustituye los valores en las fórmulas cargadas en el *BUFFER_FÓRMULAS* y después las manda calcular una a una con *PARSER CALCULA_FÓRMULAS*.

El *hilo de Análisis* ejecuta al *parser* cada vez que ha sustituido los valores de una fórmula y desea calcularla. El *parser* recibe la expresión de la fórmula con sus operandos sustituidos por valores y la calcula.

Como resultado de este cálculo, el *parser* devuelve el código asociado al tipo de dato final de la fórmula (booleno, real o fecha).

En la primera etapa del *parser*, el analizador léxico - programado en *LEX*- separa en componentes léxicos los valores de los operandos y los operadores; en la segunda etapa - programada en *YACC* - el analizador sintáctico toma los componentes léxicos y hace los cálculos pertinentes, dependiendo del tipo de operador y de los valores de los operandos.

5.2.6.5 Parser EVALUA_CONDICIÓN

El último *parser*, *PARSER EVALUA_CONDICIÓN*, es el componente encargado de realizar el análisis léxico y sintáctico de la definición de las sentencias de búsqueda (condiciones) de operandos en las *Bases de Datos Operacionales*.

El *RPC EVALUA_CONDICIÓN* se auxilia de este *parser* para llevar a cabo tal análisis. Como en el caso del análisis de las fórmulas, al ejecutarse el *RCP*, este recibe la condición de búsqueda a ser analizada, sustituye los operandos por el código asociado a su tipo de dato, y obtiene la condición expresada en tipos de dato (*condición_tipos*).

El *PARSER EVALUA_CONDICIÓN* toma la *condición_tipos* y evalúa su definición; la definición también se lleva a cabo en dos partes, la primera -programada en *LEX*- es el análisis léxico, y la segunda -programada en *YACC*- es el análisis sintáctico.

Una vez hecho el análisis, el *parser* devuelve el código de retorno de definición correcta, o bien, el código de error de definición.

5.3. Procesos funcionales del Servidor Monitor

5.3.1. Arranque del *Servidor Monitor* e inicialización de hilos

Antes de arrancar su ejecución normal, el servidor debe ser inicializado. El proceso de inicialización consiste en alojar las estructuras de datos de control de hilos; inicializar el estado del *Servidor Monitor* y configurar parámetros como: número máximo de conexiones, número máximo de colas de mensajes, número máximo de *mutex*, tamaño del *stack* del servidor, etc.

Ya inicializado, el *Servidor Monitor* inicia su ejecución; con esto, el servidor crea un hilo temporal que será el responsable de crear y alojar memoria para los diferentes componentes, entre los cuales están los hilos de servicio, colas de mensajes, *mutex*, procedimientos registrados y manejadores de eventos.

En la fig. 5.3 se muestra el proceso de creación y arranque de los hilos de servicio del servidor. En ésta se observa que el *SCHEDULER* es el encargado de la administración de la ejecución de los hilos de servicio, y es quien lleva a cabo la planificación de los hilos. El *SCHEDULER* se vale de una colección de colas de mensajes denominadas *RUN_QUEUES* y *SLEEP_QUEUE* para llevar acabo la planificación de los hilos y su cambio de contexto.

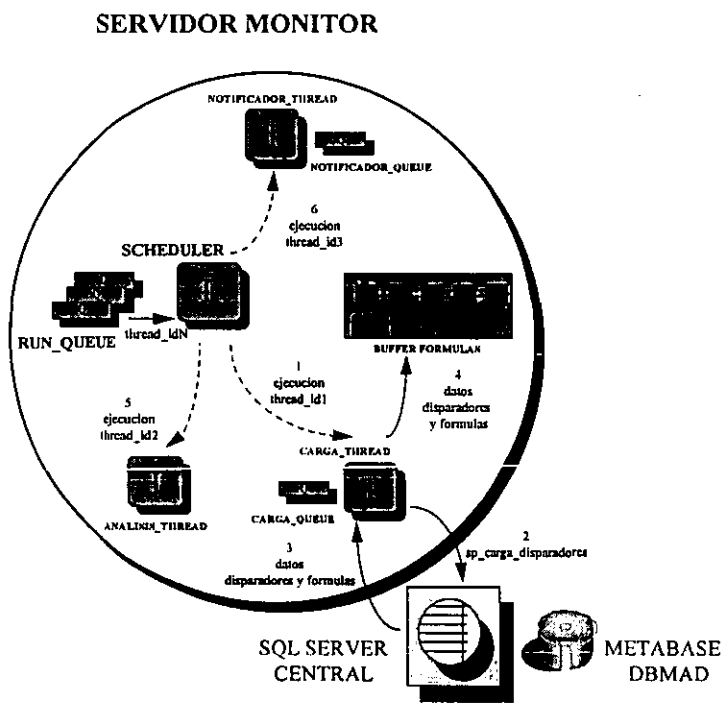


Figura 5.3 Arranque del Servidor Monitor *e inicialización de hilos*

El orden de creación e inicio de ejecución de los hilos de servicio dentro del *Servidor Monitor* es: *hilo de Carga* (*CARGA_THREAD*), *hilo de Análisis* (*ANÁLISIS_THREAD*) e *hilo de Notificación* (*NOTIFICADOR_THREAD*).

Inicialmente, entra en ejecución el *hilo de Carga* (paso 1); una vez que el *hilo de Carga* ha entrado en ejecución por primera vez, abre una conexión estática (permanente) hacia el *Servidor de SQL Central*, de donde constantemente obtendrá la información necesaria para la definición y monitoreo de las fórmulas.

Después para obtener la información de todas las fórmulas definidas en la metabase *DBMAD*, el *hilo de Carga* manda ejecutar en el *Servidor de SQL* el procedimiento almacenado *sp_carga_disparadores*; con este procedimiento obtiene todos los datos referentes a la definición de los disparadores y fórmulas en *DBMAD* (paso 3). El *hilo de Carga* coloca toda esa información recibida del *Servidor de SQL* en la memoria del *Servidor Monitor* en el *BUFFER_FÓRMULAS* (paso 4).

Hasta aquí ha sido inicializada la memoria compartida (*BUFFER_FÓRMULAS*) por los hilos de *Carga* y *Análisis*. En este *buffer* se han cargado todos los disparadores creados, las fórmulas relacionadas con cada disparador, los operandos que serán sustituidos en las fórmulas al recuperar sus valores, y las sentencias *SQL* para la

recuperación de los operandos definidos como columnas. El *hilo de Carga* ha concluido hasta ahora su primera tarea y termina momentáneamente su ejecución para dejar a otros hilos iniciar.

Como el *Servidor Monitor* tiene ya en el *BUFFER_FÓRMULAS* los datos que el *hilo de Análisis* necesita, este puede entonces arrancar su ejecución inicial (paso 5) y, al igual que el *hilo de Carga*, como primer paso, abre una conexión estática con el *Servidor de SQL Central* que le permitirá detectar los hechos que se vayan presentando.

Así, el *hilo de Análisis* simplemente se mantiene en espera de la llegada de nuevos hechos y datos que monitorear. Finalmente, se inicia la ejecución del *hilo de Notificación* (paso 6), el cual tan sólo entra en estado de espera de alguna petición de notificación en su cola de mensajes *NOTIFICADOR_QUEUE*.

Iniciados los hilos de servicio, estos se encuentran en espera de alguna petición que atender. El *hilo de Carga* esperará por alguna petición de activación o desactivación de una fórmula, el *hilo de Análisis* esperará la llegada de algún hecho acompañado de datos que analizar, y el *hilo de Notificación* esperará alguna petición de notificación de alarma a un usuario o de error al administrador.

5.3.2 Manejo de las conexiones de los clientes

Cuando el servidor está corriendo, el hilo *NETWORK_LISTENER* se pone a escuchar constantemente las peticiones de los clientes. Si un cliente manda una petición de conexión, el *NETWORK_LISTENER* crea un *hilo de Conexión* para atender las subsecuentes peticiones y genera el evento *CONEXIÓN* (*SRV_CONNECT*) asociado a la estructura de control del hilo creado (apuntador al hilo), con esto, el nuevo hilo manda llamar a la rutina asociado para el manejo del evento *SRV_CONNECT*, al cual llamaremos *CONNECT_HANDLER* y cuya función se explica más adelante.

En la fig. 5.4 se observa que cuando un usuario entra en sesión al sistema, en la máquina donde trabaja se ejecutan dos procesos: uno es la pantalla de monitoreo (*Módulo de Monitoreo*), y otro, el que realmente tiene interacción de manera directa con el *Servidor Monitor*, el *Notificador*. De aquí en adelante haremos referencia al *Notificador* simplemente como *el cliente*, y como tal, el *Notificador* es el que manda peticiones al *Servidor Monitor* y recibe las respuestas (alarmas y errores).

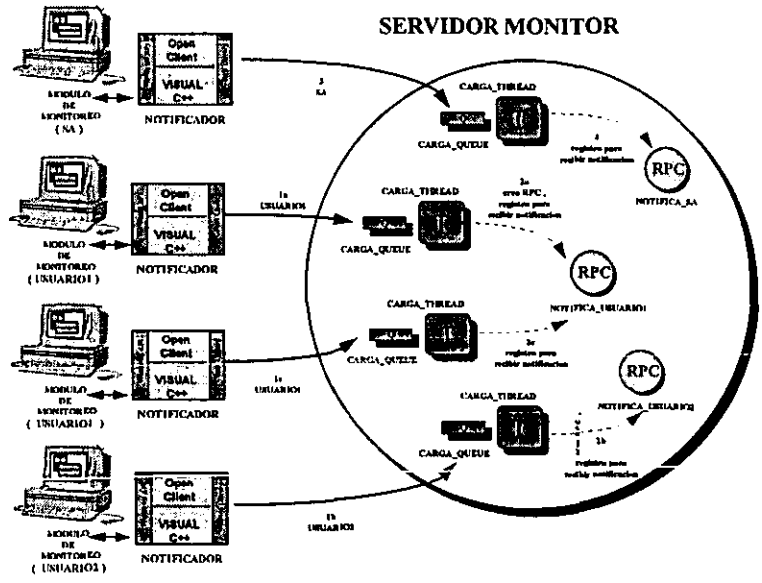


Figura 5.4 Manejo de las conexiones de los clientes

Siempre que un cliente abra una conexión, el *Servidor Monitor* lo inscribirá en la lista de notificación. Así, cada vez que ocurra una alarma de una fórmula definida por un usuario específico, el servidor notificará la ocurrencia de esa alarma absolutamente a todos los clientes conectados con el mismo *login* de acceso que el del dueño de la fórmula. El proceso de inscripción en la lista de notificación se detalla a continuación.

Supongamos que el usuario con el *login USUARIO1* entra en sesión, entonces el *Notificador* asociado al *USUARIO1* abre una conexión con el *Servidor Monitor* (paso 1a). El *hilo de Conexión* asignado mandará llamar a la rutina *CONNECT_HANDLER*; en este caso al ser la primera conexión que el *USUARIO1* ha abierto, el *CONNECT_HANDLER* crea el *RPC NOTIFICA_USUARIO1* -perteneciente al grupo de *RPC NOTIFICA_USER-* (paso 2a) mediante el cual se efectuará la notificación exclusivamente a los clientes con ese *login*. Sin embargo, no basta con crear el *RPC*, sino que el cliente debe ser agregado a la lista de notificación del *RPC* -mediante el apuntador al *hilo de Conexión* que lo referencia-. Así mismo, si otro usuario (*USUARIO2*) abre una primera conexión, se creará el *RPC NOTIFICA_USUARIO2* y se inscribirá en su lista de notificación (paso 1b y 2b).

Supóngase ahora que otro usuario entra en sesión con el mismo *login USUARIO1* (paso 1c), entonces, el nuevo *hilo de Conexión* volverá a llamar a la rutina *CONNECT_HANDLER*; en ésta, se detecta que ya existe una conexión abierta con el *login*

USUARIO1, por lo que en este caso ya no se crea un nuevo *RPC*, únicamente se inscribe al nuevo cliente -a través del apuntador al *hilo de Conexión*- en la lista de notificación para el *RPC NOTIFICA_USUARIO1* (paso 2c). Así se garantiza que al ocurrir una alarma para una fórmula definida por *USUARIO1*, el *Servidor Monitor* mandará la notificación a todos los clientes que estén conectados como *USUARIO1*.

Existe otro *RPC* para notificación, el *RPC NOTIFICA_SA*, que a diferencia de los pertenecientes al grupo *NOTIFICA_USER*, sirve para la notificación de errores. Éste se crea al iniciarse el *Servidor Monitor* y se mantiene así hasta que el servidor se da de baja -evento *stop* (*SRV_STOP*)-. Supóngase ahora el caso en que un cliente se conecta como usuario administrador (paso 3). En este caso, el *Notificador* identifica que el usuario cuenta con derechos de administrador por lo que debe estar habilitado no sólo para recibir alarmas, sino también eventuales errores, lo que quiere decir que debe estar inscrito en las listas de notificación del *RPC NOTIFICA_SA* y en algún *NOTIFICA_USER*. Para este último, al conectarse al *Servidor Monitor*, el *hilo de Conexión* asociado se encarga de crear el *RPC NOTIFICA_USER* -si fuese necesario- e inscribirlo en su lista de notificación. Pero para el caso del *RPC NOTIFICA_SA*, como éste ya ha sido creado y como el *Notificador* ha identificado previamente al usuario como administrador, será el *Notificador* quien mandará directamente la petición al *Servidor Monitor* para que lo inscriba en la lista de notificación *NOTIFICA_SA* (paso 4). De esta forma el cliente es inscrito a nivel servidor en *NOTIFICA_USER*, y a nivel cliente en *NOTIFICA_SA*.

Finalmente una vez que cada *hilo de Conexión* termina de ejecutar la rutina *CONNECT_HANDLER*, estará en disposición de recibir eventos en su cola de mensajes *CONNECT_QUEUE* para la atención de las peticiones.

5.3.3 Análisis de la definición de las fórmulas

A continuación, en la fig. 5.5 explicamos el procedimiento a través del cual se definen las fórmulas. El único medio por el cual los usuarios pueden definirlas es a través del *Módulo de Configuración*, pero éste se auxilia de la *Interface* para trabajar como cliente del *Servidor Monitor*[5]. Cuando un usuario quiere definir una fórmula, la *Interface* manda una petición al servidor con el *RPC rp_analiza_fórmula* (paso 1). El *hilo de Conexión*

[5] Recordemos que la *Interface* es el módulo que comunica al *Servidor Monitor* con el *Módulo de Configuración* y el *Módulo de Administración*. La *Interface* es quien hace las peticiones al *Servidor Monitor*.

recibe la petición y llama a la rutina para *rp_analiza_fórmula*. En está se recibe la *fórmula* a ser analizada (paso 2). Ya con la definición de la fórmula, el *RPC* obtiene del *Servidor de SQL* dos expresiones, la *expresion_tipos* y la *expresión_compilada*; la primera se obtiene sustituyendo de la variable *fórmula* todos los operandos por la clave correspondiente a su tipo definido en la tabla de *OPERANDOS* de la metabase *DBMAD*, y la segunda se obtiene de sustituir de *fórmula* todos los operandos por la clave de su identificador también cn la tabla *OPERANDOS* (paso 3).

Por ejemplo, para la expresión de la fórmula *(valor_final - valor_inicial)%* > 5%, si los operandos *valor_inicial* y *valor_final* tuvieran los *id_operando* 1 y 2 respectivamente y ambos fueran de tipo real, la *expresion_tipos* y la *expresión_compilada* serían como se muestra en la tabla 5.4.

Tabla 5.4 Expresión compilada y de tipos para la fórmula (valor_final - valor_inicial)% > 5%

fórmula:	(valor_final - valor_inicial)% > 5%
expresion_tipos:	(#r& - #r&)% > 5%
expresion_compilada:	(#O&000001 - #O&000002)% > 5%

Una vez hechas las sustituciones, el *RPC* manda analizar en el *PARSER ANALIZA_FÓRMULA* la cadena obtenida en *expresion_tipos*. Si esta expresión fuese correcta en cuanto al uso de los operadores, concordancia de paréntesis y compatibilidad de tipos, el *parser* generará como retorno el código asociado al tipo de dato que resulte en la evaluación de la fórmula (*BOOLEANO*, *FECHA* o *REAL*); o bien el código de error en la definición (*FÓRMULA_ERROR*).

Finalmente el *RPC* devuelve al cliente la *expresion_compilada* y el *valor* del código asociado al tipo de dato, esto para el caso en que la fórmula esté bien definida; por el contrario si no lo estuviese, regresaría en el parámetro *valor* el código asociado a *FÓRMULA_ERROR* (paso 4).

En nuestro ejemplo vemos que la fórmula está bien definida, y que su tipo de dato final es *booleano*. En este caso el *RPC* devuelve al cliente en *expresion_compilada* la cadena *(#O&000001 - #O&000002)% > 5%* y en *valor* el código de retorno *BOOLEANO*.

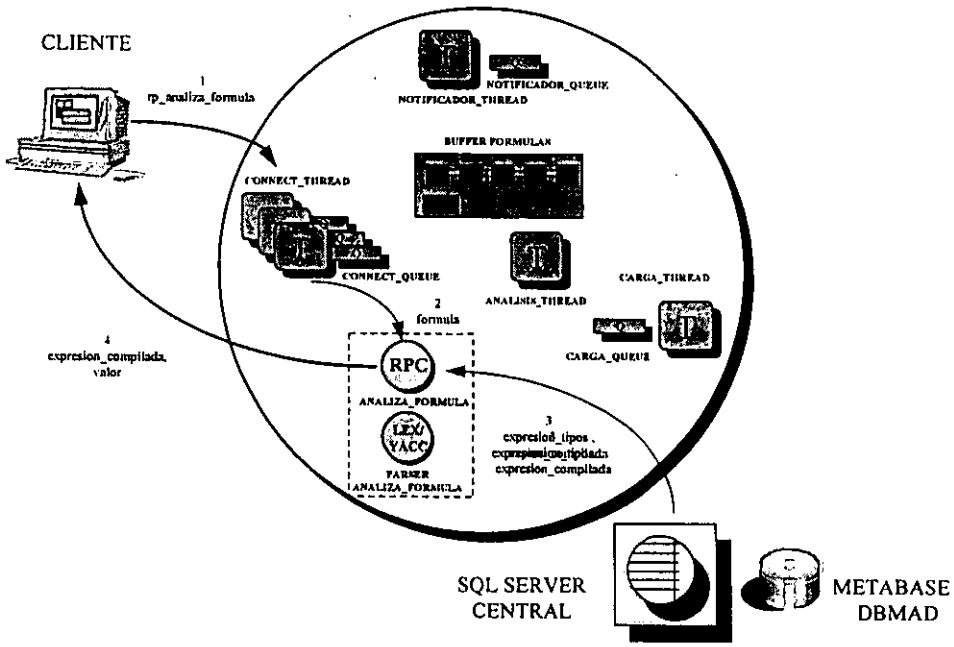


Figura 5.5 Análisis de la definición de las fórmulas

5.3.4 Activación de las fórmulas

Para analizar una fórmula no basta con definirla en la base de datos *DBMAD* del *Servidor de SQL*, es necesario que esté activa en el *Servidor Monitor*, lo que significa que su definición debe estar cargada en su memoria en el *BUFFER_FÓRMULAS*.

En la fig. 5.6 se muestra el diagrama del procedimiento de activación de fórmulas. En él, observamos que el cliente (*Interface*) manda la petición de activación con el *RPC activa_fórmula* (paso 1). El *hilo de Conexión* respectivo recibe la petición y ejecuta la rutina asociada. Ésta recibe como parámetro de entrada el identificador *id_fórmula* de la fórmula a activar (paso 2), y con ello el *RPC* escribe el *id_fórmula* obtenido y el mensaje *ACTIVA_FÓRMULA* en la cola de mensajes *CARGA_QUEUE* del *hilo de Carga* (paso

3)[6]. Finalmente la rutina del *RPC activa_fórmula* manda a la *Interface* el mensaje de petición concluida (paso 4)[7].

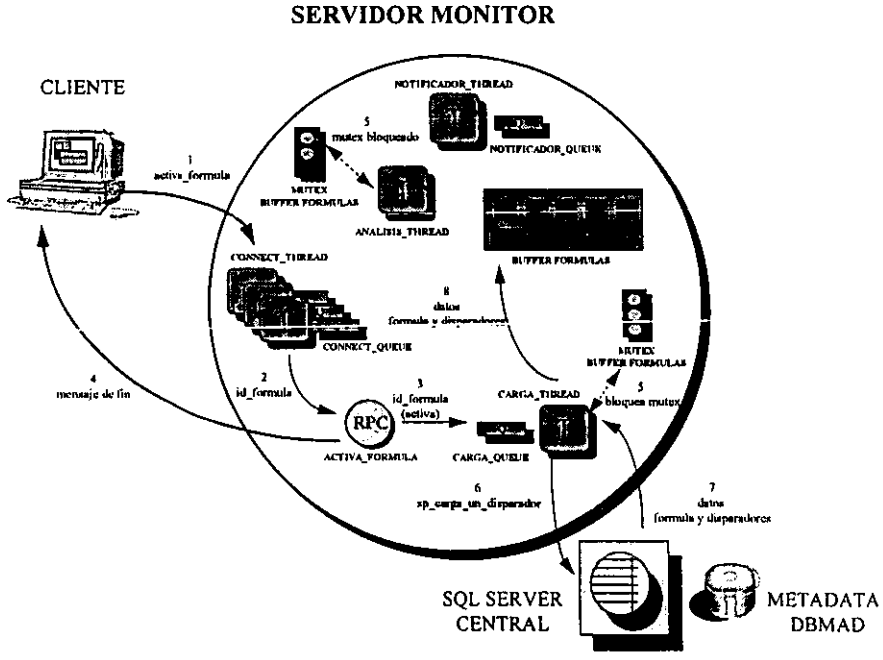


Figura 5.6 Activación de las fórmulas

Hasta este punto la tarea del *RPC activa_fórmula* se ha cumplido, toca entonces al *hilo de Carga* hacer su labor. El *hilo de Carga* permanece durmiendo (estado de *sleeping*)[8], pero despierta al recibir el mensaje proveniente del *RPC activa_fórmula*, inmediatamente después de activarse (estado de *active*), toma la señal *ACTIVA_FÓRMULA* junto con el *id_fórmula* involucrado y se prepara para cargar en el *BUFFER_FÓRMULAS* la información almacenada en la metabase *DBMAD*. Así, el *hilo de Carga* tratará de bloquear el *mutex MUTEX_BUFFER_FÓRMULAS*, pero si ya estuviese bloqueado (por el *hilo de Análisis*) el *hilo de Carga* se detendrá (estado de *stopped*) en espera de que el *mutex* sea liberado. Una vez que ha podido bloquear al *mutex* (paso 5), el *hilo de Carga* volverá a activarse y podrá alterar el contenido del *BUFFER_FÓRMULAS* sin causar ningún conflicto[9].

[6] La rutina escribe el mensaje en la cola de mensajes *CARGA_QUEUE* y continua su ejecución sin esperar a que el mensaje sea leído por el *hilo de Carga*.

[7] Si este mensaje (*SRV_DONE_FINAL*) no es enviado, el cliente se quedará suspendido esperando la respuesta final.

[8] Después de realizar la carga inicial, el hilo de *Carga* se duerme en espera de algún mensaje (activación o desactivación de una fórmula) en su cola de mensajes.

[9] Hay que recordar que la transición del estado de *stopped* a *active* se realiza pasando por el estado de *runnable* (listo para entrar en actividad).

Posteriormente el *hilo de Carga* a su vez realiza una petición, a través del procedimiento almacenado *sp_carga_un_disparador*, al *Servidor de SQL* por cada uno de los disparadores que utilizan la fórmula (paso 6). Una vez que el *Servidor Monitor* ha obtenido de la metabase *DBMAD* del *Servidor de SQL* los datos de la fórmula y los disparadores asociados (paso 7), finalmente los coloca en el *BUFFER_FÓRMULAS* (paso 8). Por último el *hilo de Carga* desbloquea el *mutex* y se vuelve a dormir en espera de algún otro mensaje de activación o desactivación, permitiendo así la ejecución de otros hilos.

5.3.5 Desactivación de las fórmulas

El mecanismo de desactivación de fórmulas en el *Servidor Monitor* es muy similar al mecanismo de activación descrito en el punto anterior. No obstante su similitud, la desactivación se realiza más rápido, puesto que no necesita ninguna intervención del *Servidor de SQL*, pues una vez hecha la petición del cliente, todos los pasos consecuentes se realizan a nivel de usuario (*user level*)[10] del *Servidor Monitor*.

En la fig. 5.7 esquematizamos la desactivación de las fórmulas. En ésta, observamos que el cliente realiza la petición al servidor utilizando el *RPC desactiva_fórmula* (paso 1). En él se especifica el *id_fórmula* por desactivar, mismo que es recuperado por la rutina *desactiva_fórmula* (paso 2). Con el *id_fórmula*, el *RPC* escribe el mensaje *DESACTIVA_FÓRMULA* y el *id_fórmula* en la cola de mensajes del *hilo de Carga* (paso 3), y termina su labor mandando el mensaje de petición concluida al cliente (paso 4).

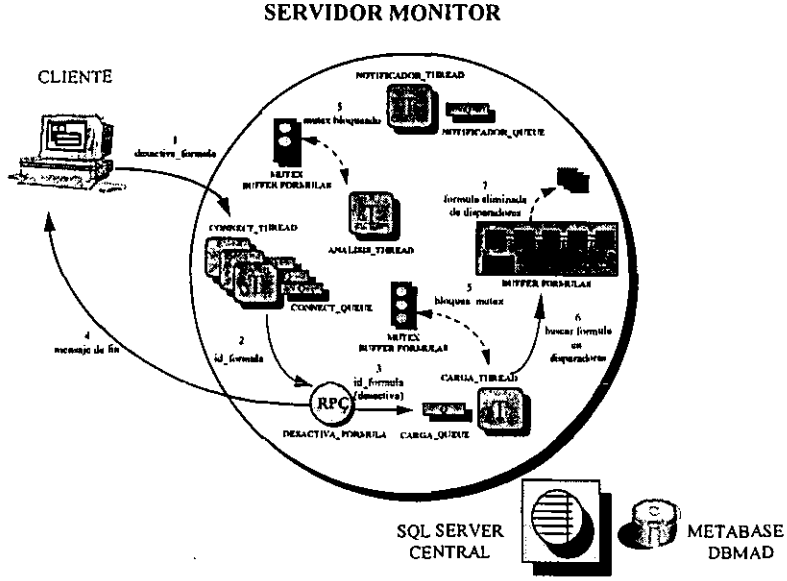


Figura 5.7 Desactivación de las fórmulas

[10] A nivel de usuario, virtualmente todo lo que en el hilo pase no requiere la intervención de llamadas al sistema (*system calls)* y por tal es más rápido.

Como ocurre para la desactivación, el *hilo de Carga* se activa al recibir el mensaje en su cola de mensajes y luego tratará de cerrar el *mutex*. Una vez que el *mutex* se haya liberado y consecuentemente haya logrado cerrarlo (paso 5), el *hilo de Carga* busca en el *BUFFER_FÓRMULAS* todas las ocurrencias del *id_fórmula* (paso 6) y simplemente las elimina (paso 7)[11]. Para concluir, el *hilo de Carga* una vez más desbloquea el *mutex* y se pone a dormir en espera de algún otro mensaje de activación o desactivación.

5.3.6 Análisis de las condiciones de búsqueda de operandos

El análisis de las condiciones de búsqueda de los operandos es, conceptualizando a nivel de componentes, muy similar al análisis de la definición de fórmulas descrito en el punto 5.3.3. A continuación en la fig. 5.8 mostramos el procedimiento análogo para la definición de las condiciones.

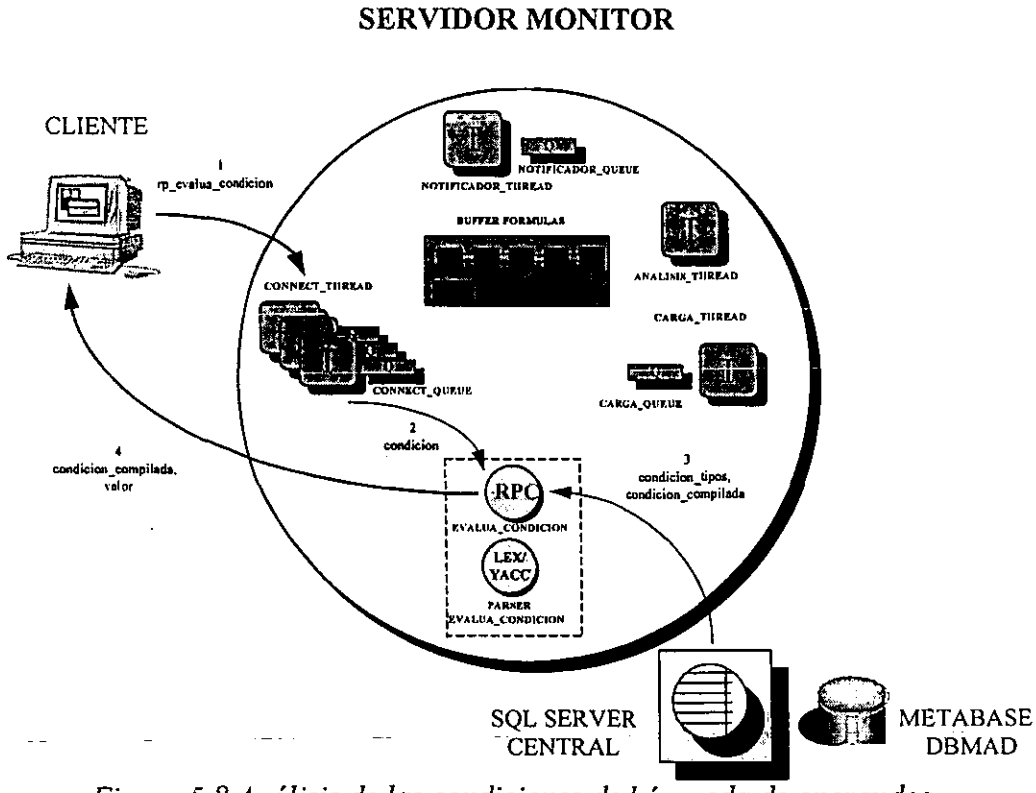


Figura 5.8 Análisis de las condiciones de búsqueda de operandos

[11] Se libera el espacio en memoria ocupado por cada ocurrencia de la fórmula.

En este caso el medio por el cual los usuarios definen los operandos recuperables de columnas es el *Módulo de Administración*, que también se auxilia de la *Interface* para actuar como cliente del *Servidor Monitor*. Cuando se define un operando como recuperable de una columna de alguna base de datos, invariablemente debe definirse su condición de búsqueda. Para ello, la *Interface* manda una petición al servidor con el *RPC rp_evalua_condición* (paso 1).

El *hilo de Conexión* recibe la petición y llama a la rutina para *rp_evalua_condición*, donde se envía la expresión de la *condición* de búsqueda a ser analizada (paso 2). Posteriormente, el *RPC* obtiene -similar al punto 5.3.3- del *Servidor de SQL* dos expresiones, la *condición_tipos* y la *condición_compilada*.

La primera se obtiene sustituyendo de la variable *condición*, los operandos por la clave correspondiente a su tipo definido en la tabla de *OPERANDOS* de la metabase *DBMAD*; además del tipo de dato de retorno de las funciones utilizadas; la segunda se obtiene de sustituir de *condición* todos los operandos por la clave de su identificador también en la tabla *OPERANDOS* (paso 3).

En seguida, vemos el ejemplo para el caso de la expresión de condición de búsqueda definida como: *(valor_final < 2000 and valor_inicial > 500) and fecha >= getdate( )*. Como los operandos *valor_inicial, valor_final* y *fecha* tienen los *id_operando* 1, 2 y 3 respectivamente y los dos primeros son de tipo real y el tercero de tipo fecha, la *condición_tipos* y *condición_compilada* serían como se muestran en la tabla 5.6.

Tabla 5.6 Expresiones de tipos y compilada para la fórmula:

(valor_final < 2000 and valor_inicial > 500) and fecha >= getdate( )

condición:	(valor_final < 2000 and valor_inicial > 500) and fecha >= getdate( )
condición_tipos:	(#r& < 2000 and #r& > 500) and #f& >= #f&
condición_compilada:	(#O&000001 < 2000 and #O&000002 > 500) and #O&000003 >= getdate( )

Con estas nuevas expresiones, el *RPC* manda analizar en el *PARSER EVALUA_CONDICIÓN* la cadena obtenida en *condición_tipos*. En el caso de que la condición estuviese mal definida, retorna el código de *CONDICIÓN_ERROR*; pero si estuviese bien definida puede regresar el tipo de dato final (*BOOLEANO, REAL, FECHA* y *CADENA*)[12] -en caso de que la condición finalmente se reduzca a un valor-, o bien, puede regresar el código de retorno *COLUMNA* -para los casos en que la evaluación no

[12] Nótese que a diferencia del *PARSER ANALIZA_FÓRMULA*, el *PARSER EVALUA_CONDICIÓN* acepta que el resultado final de la condición sea de tipo *CADENA*.

pueda determinar el tipo de dato final por depender éste del tipo de dato recuperado (en tiempo de ejecución) de las columnas-.

Finalmente el *RPC* devuelve al cliente la *condición_compilada* y el *valor* del código asociado al tipo de dato. Esto para el caso en que la condición esté bien definida; por el contrario si no lo estuviese, regresaría en el parámetro *valor* el código asociado a *CONDICIÓN_ERROR* (paso 4).

5.3.7 Cálculo de las fórmulas y generación de las alarmas

A continuación describimos el proceso principal en el *Servidor Monitor*, el proceso que involucra la sustitución de datos en las fórmulas, el monitoreo de las mismas y la consecuente generación de alarmas.

Todo este mecanismo es controlado por el *hilo de Análisis* y, puesto que la mayoría de las otras tareas del servidor son asíncronas que pueden ocurrir no con mucha frecuencia, el *hilo de Análisis* es el que mantiene la mayor actividad.

En la fig. 5.9 se observan las tareas de control que lleva a cabo el *hilo de Análisis*, así como su funcionalidad. En ésta, vemos que el inicio del ciclo de monitoreo ocurre al presentarse un hecho externo insertado por un disparador definido.

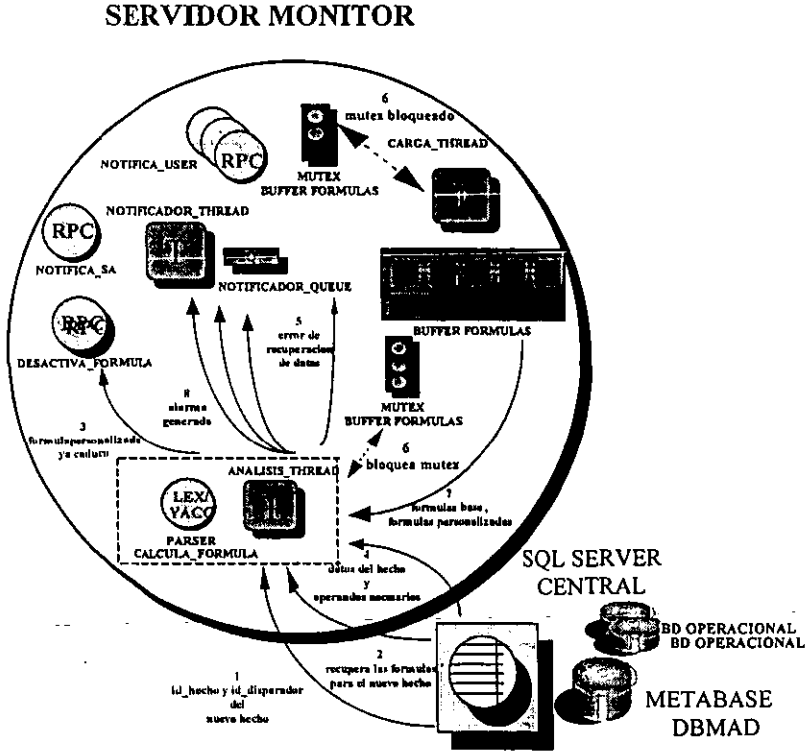


Figura 5.9 Cálculo de las fórmulas y generación de alarmas

En ese instante, el *hilo de Análisis* detecta la llegada de un nuevo hecho (*id_hecho*) en la metabase *DBMAD* para determinado disparador *id_disparador* (paso 1).

Sabemos que al definir las fórmulas tenemos que asociarlas a cierto disparador que sea el que propicie la evaluación de las mismas, por lo que al detectar el nuevo hecho, el *hilo de Análisis* toma de la metabase *DBMAD* las fórmulas (*id_fórmulas*) que debe evaluar (paso 2). Si para alguna de estas fórmulas, el tiempo de activación hubiese concluido, el *hilo de Análisis* simplemente mandará ejecutar el *RPC desactiva_fórmula* (paso 3), quien hará la petición de desactivación de la fórmula al *hilo de Carga* y luego suspenderá su ejecución en espera de algún hecho nuevo.

Por otra parte, cada nuevo hecho involucra la presencia de varios datos ligados al disparador y que deben sustituirse en las fórmulas que le fueron asociadas, además sabemos que cada fórmula puede tener en su definición algún operando con valor fijo ya establecido en la metabase *DBMAD* (en la tabla de *VALORES*), por lo que el siguiente paso (paso 4) del *hilo de Análisis* es recuperar - de la tabla *DATOS_HECHO* - todos los valores de los datos que acompañan al nuevo hecho (*id_hecho*) y aquellos que sean fijos en la metabase y que estén en la definición de las fórmulas.

Si por algún motivo ocurriese un error en la recuperación de los datos (véase la parte *Procedimiento registrado EVALUA_CONDICIÓN*), entonces el *hilo de Análisis* mandará una petición de notificación de error al *hilo de Notificación* (paso 5); sin no hubiese error, el *hilo de Análisis* continuará con su ejecución normal.

Hasta este punto el *hilo de Análisis* ha obtenido todos los valores que debe sustituir en las fórmulas. Recordemos que al hablar de fórmulas, significa que el disparador puede estar ligado a varias fórmulas base que a su vez definen una fórmula personalizada cuya evaluación final sea la que origine eventualmente una alarma.

Recordemos también que el *Servidor Monitor* mantiene en memoria en el *BUFFER_FÓRMULAS* la definición compilada de todas estas fórmulas, por lo que el *hilo de Análisis* debe obtener de memoria la definición compilada de las fórmulas involucradas por los *id_fórmulas* recuperados.

Así, como ocurre con el *hilo de Carga*, el *hilo de Análisis* tratará de cerrar el *mutex MUTEX_BUFFER_FÓRMULAS*, de no lograrlo, esperará a que se libere (estado de

stopped); una vez que logre colocar el candado al *mutex* (paso 6) podrá tener acceso a la memoria compartida en *BUFFER_FÓRMULAS*, así podrá continuar su ejecución en su zona crítica y recuperar las definiciones compiladas de las fórmulas base y la fórmula personalizada (paso 7).

El *hilo de Análisis* cuenta ya con todos los elementos para poder realizar la evaluación de la fórmula personalizada, con ellos, el *hilo de Análisis* sustituye cada uno de los valores de los operandos recuperados en las definiciones compiladas de las fórmulas; evalúa con el *PARSER CALCULA_FÓRMULA* una a una las nuevas expresiones de las fórmulas base y sustituye los resultados respectivos en la definición de la fórmula personalizada final. Luego, la expresión final de la fórmula personalizada es evaluada también con el *parser* y, finalmente se reduce para obtener un resultado de tipo booleano (*TRUE o FALSE*).

Se ha obtenido un resultado final del cálculo de las fórmulas, y si este resultado fuese *TRUE*, el *hilo de Análisis* debe generar una nueva alarma *id_alarma* para esa fórmula personalizada (*id_fórmula*). Posteriormente, el *hilo de Análisis* almacena en la metabase *DBMAD* cada uno de los valores de los operandos recuperados en el hecho, para que posteriormente el usuario -de no estar en sesión en ese momento-, pueda consultar en su siguiente sesión los datos de la alarma generada.

Para concluir, el *hilo de Análisis* manda una petición en la cola de mensajes *NOTIFICA_QUEUE* al *hilo de Notificación* (paso 8) para que éste informe al cliente la ocurrencia de la alarma. Termina así el ciclo de monitoreo y el *hilo de Análisis* se duerme (estado de *sleeping*) en espera de nuevos hechos que reinicien el ciclo.

5.3.8 Notificación de alarmas y errores a los usuarios

En la fig. 5.10 se muestra cómo se realiza la notificación de alarmas y errores.

El proceso de notificación se desencadena siempre por el *hilo de Análisis* cuando ha originado alguna alarma. En la fig. 5.10 podemos constatar que el *hilo de Análisis* puede generar tanto alarmas como errores.

Expliquemos primero el mecanismo de notificación de alarmas. Cuando el *hilo de Análisis* ha calculado con éxito una fórmula personalizada[13], obtiene como resultado *TRUE* o *FALSE* (tipo *BOOLEANO*), con lo cual, si el resultado fuese *TRUE* deberá generar una alarma.

[13] Una fórmula personalizada esta definida por una o más fórmulas base.

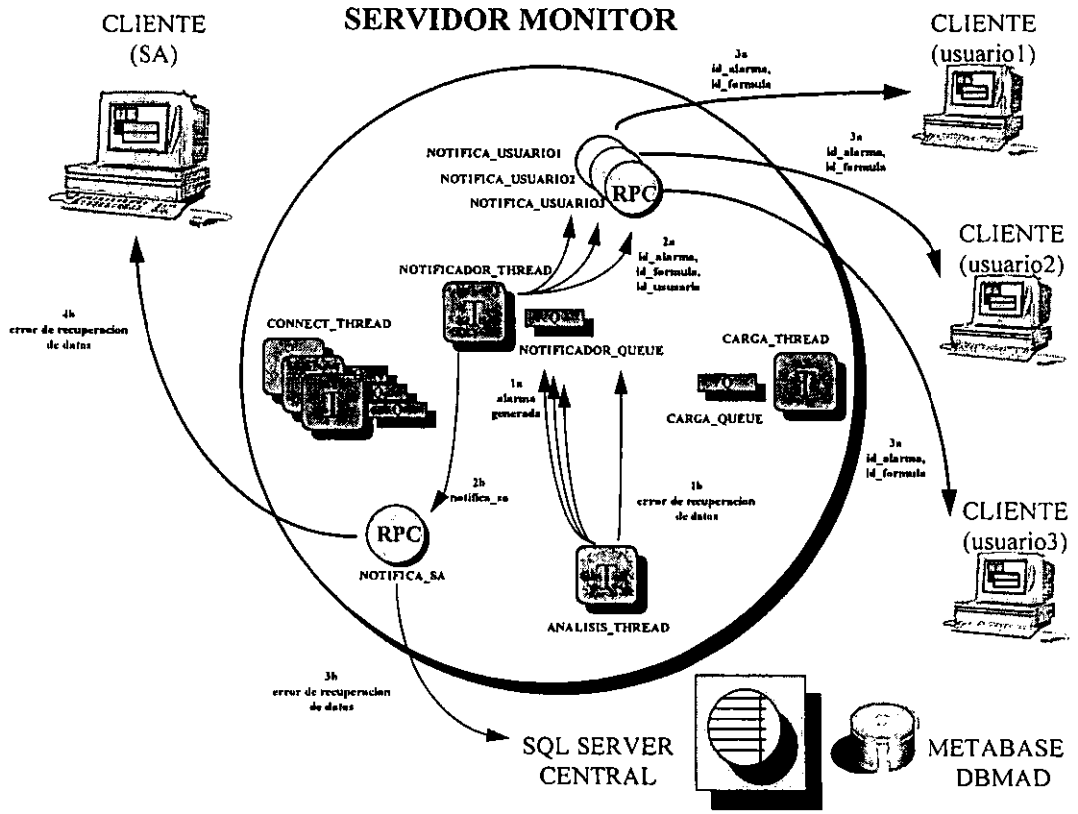


Figura 5.10 Notificación de alarmas y errores a los usuarios

Así, supongamos que en un instante cualquiera, el *hilo de Análisis* ha calculado una fórmula definida por el *USUARIO1*, cuyo resultado final es *TRUE*. Entonces el *Servidor Monitor* deberá notificar a todos los usuarios conectados con el *login USUARIO1* la ocurrencia de la alarma.

En el momento en que ha ocurrido la alarma, el *hilo de Análisis* detecta el usuario que definió la fórmula, y envía un mensaje al *hilo de Notificación* en su cola de mensajes *NOTIFICA_QUEUE* (paso 1a).

En este mensaje se especifica el identificador de la alarma generada (*id_alarma*), la fórmula que generó la alarma (*id_fórmula*), y el usuario (*id_usuario*) al cual se debe notificar la alarma; el *hilo de Notificación* -que hasta este punto se encuentra dormido- se activa y manda ejecutar internamente el RPC[15] de notificación correspondiente al *id_usuario*, que en este caso es el *RPC NOTIFICA_USUARIO1* (paso 2a).

[14] Una fórmula personalizada esta definida por una o más fórmulas base.

[15] Un *RPC* puede ser ejecutado como una petición de un cliente al servidor, o bien, ejecutarse dentro del mismo servidor por algún otro de sus componentes.

Luego, la rutina para el *RPC NOTIFICA_USUARIO1* notifica de manera asíncrona al cliente con el *USUARIO1* la alarma y la fórmula que la desencadenó (paso 3a), incluyendo la información de los valores de los operandos, el valor final de la fórmula y un *bit* para la bandera de *MENSAJE* (para mayor información sobre este *bit*, véase el punto 5.3.9).

Lo mismo sucederá al generarse alarmas para las fórmulas definidas por los usuarios *USUARIO2*, *USUARIO3*, etc., quienes finalmente sólo conocerán la ocurrencia de las alarmas para las fórmulas definidas por ellos.

Expliquemos ahora el caso en que el *hilo de Análisis* no haya generado una alarma, sino un error. Supongamos que el *hilo de Análisis* por alguna circunstancia no haya podido realizar el cálculo de la fórmula (error en la recuperación de los datos, incompatibilidad de tipos entre los datos recuperados y los operandos de las fórmulas, o recuperación de cero o más de un dato para un operando definido como columna), entonces, el *hilo de Análisis* escribirá en la cola de mensajes *NOTIFICADOR_QUEUE* el código asociado al error detectado (paso 1b).

Una vez más, al recibir el mensaje, el *hilo de Notificación* se activará y determinará que el mensaje recibido corresponde a un código de error, por lo que debe notificar no al dueño de la fórmula, sino al administrador.

El *hilo de Notificación* manda entonces a ejecutar el *RPC NOTIFICA_SA* (paso 2b). Por su parte la rutina para el *RPC NOTIFICA_SA* tiene, por una parte, que registrar en la metabase *DBMAD* el error ocurrido (paso 3b), y por otra, que notificar en tiempo de ejecución al administrador la ocurrencia del error (paso 4b).

Con esto garantizamos que si el administrador no estuviese en sesión en el momento de presentarse el error, la siguiente vez que entre, el sistema le mostrará la lista de todos los errores ocurridos.

5.3.9 Comunicación entre usuarios

Para poder lograr la comunicación entre usuarios, el *Servidor Monitor* se apoya en el mecanismo de notificación de alarmas explicado en el punto anterior. En la fig. 5.11

vemos cómo el *Servidor Monitor* acopla tal mecanismo para implementar el intercambio de mensajes entre usuarios.

Al ocurrir una alarma, el usuario a través del *Módulo de Monitoreo* puede mandar un mensaje escrito a otro usuario, esto lo hace almacenando los mensajes en la metabase *DBMAD*, para que posteriormente de manera asíncrona, otro usuario pueda consultarlo. En la fig. 5.11 vemos que un cliente conectado en la máquina 1 hace una petición al servidor, usando el *RPC rp_llama_notificación* (paso 1). En esta petición se envían los parámetros *id_alarma*, *id_fórmula* e *id_usuario*. La rutina *rp_llama_notificación* recibe los parámetros (paso 2), y con éstos escribe un mensaje (*NOTIFICADOR_REQUEST*) en la cola de mensajes *NOTIFICADOR_QUEUE* (paso 3).

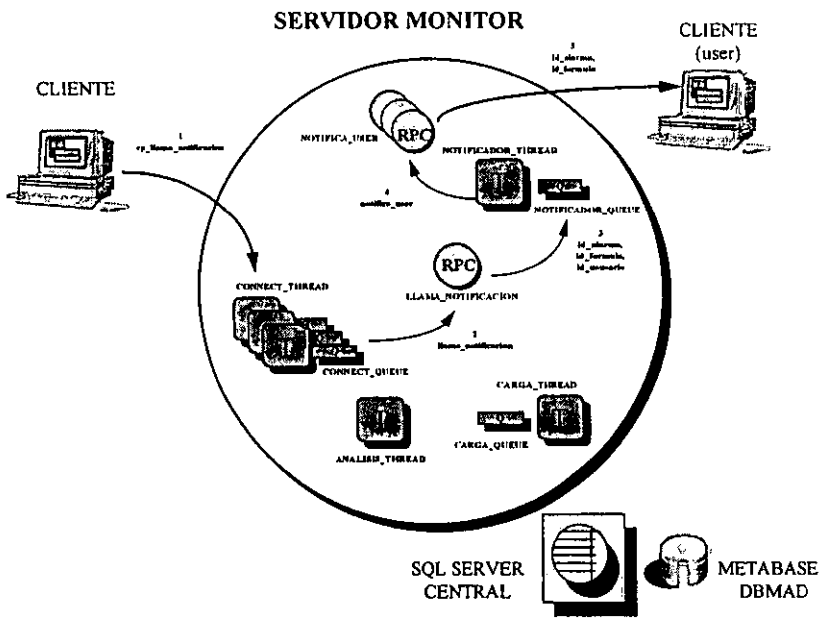


Figura 5.11 Comunicación entre usuarios

Al recibir el mensaje, el *hilo de Notificación* se activa y con el *id_usuario* recibido ordena la ejecución del *RPC* de notificación correspondiente al usuario (para más detalles véase el punto 5.3.8).

Al ejecutarse el *RPC* de notificación correspondiente, se notificará al usuario indicado por *id_usuario*, que se ha generado la alarma *id_alarma* para la fórmula *id_fórmula* (paso 4); dentro de la información recibida, el cliente detectará que se ha encendido el *bit* de la

bandera de *MENSAJE*, con lo que el usuario sabrá que tal notificación ha sido originada por la alarma de otro usuario, y decidirá si desea o no consultar el mensaje previamente almacenado en la metabase *DBMAD*.

Lo anterior significa que cuando ocurra una alarma, usuarios diferentes que no sean los dueños de la fórmula podrán ser notificados de la ocurrencia de tal alarma. Por supuesto que esto sólo podrá ocurrir una vez que el dueño de la fórmula haya sido notificado por el servidor la ocurrencia de una alarma y, que el usuario envíe el aviso a otro usuario sobre la ocurrencia de la alarma para la fórmula que le pertenece.

Capítulo 6 Diseño del cliente

Para el diseño del cliente se utilizará una metodología que incluye algunos de los conceptos usados para el diseño de bases de datos relacionales, sobre todo varias de las ideas expuestas en el capítulo uno de este trabajo que se ubican dentro del contexto de análisis y diseño orientado a objetos. Se realizarán tres modelos (*Modelo de implantación*, *Modelo lógico* y *Modelo de requerimientos*) que en conjunto determinarán el diseño de las aplicaciones que se necesitan implementar para la parte que se refiere al cliente del *Sistema de Monitoreo y Análisis de Datos* (*SMAD*).

La metodología orientada a objetos sólo se utiliza en el diseño del cliente por que se cuenta con más oportunidades de poner en práctica varios de los puntos que la forman, ya que se cuenta con una herramienta que soporta características de este enfoque, lo que no se puede realizar completamente si se contempla el uso de un manejador de base de datos relacional como es el caso de la base de datos de control, o el caso del *Servidor Monitor*, para el cual se utilizó bibliotecas específicas que soportan el paradigma de procedimientos, lo cual no hace tan fácil que se pueda considerar un diseño orientado a objetos. Sin embargo, creemos que al usar la metodología orientada a objetos para el diseño del cliente, varios aspectos importantes y pasos a seguir de este enfoque quedan bien ejemplificados.

6.1 Modelo de requerimientos para las aplicaciones cliente del SMAD

En este modelo se describen los requerimientos de información y componentes que se necesitan para el desarrollo de las aplicaciones, que en este caso, formarán parte del cliente para el *SMAD*; los requerimientos de información y los componentes descritos a su vez, determinarán la funcionalidad que la aplicación deberá proveer en el sistema mediante una descripción y descomposición detallada de la misma.

Esto se llevará acabo través de un análisis de *casos de uso*, con lo cual quedará cubierto la parte de la obtención de los requerimientos y la funcionalidad, posteriormente con estos resultados se realizará un análisis de dominio para determinar los componentes generales del sistema utilizando marcos de referencia.

6.1.1 Análisis de *casos de uso*

El análisis de *casos de uso* consiste en primera instancia, en determinar a los actores del sistema, es decir, los diferentes roles que los usuarios del sistema adoptan; después, determinar la funcionalidad del mismo mediante la obtención de *casos de uso*. Se debe partir de una serie de datos que originalmente se obtienen de entrevistas y formatos que los analistas deben de hacer a los clientes, en este caso se utilizarán como datos del sistema, los datos ya recabados en los capítulos tres y cuatro.

6.1.1.1 Actores del sistema

Lo que nos debemos preguntar para determinar a los actores del sistema es ¿quiénes son los que van a utilizar el sistema?. De la información indagada en el capítulo tres, se observa que existen tres tipos de actores que intervienen directamente con el cliente del *SMAD* (ver apartado 3.2.8), éstos son administradores, operadores y usuarios normales, cada uno va a definir a un actor en el análisis de *casos de uso*. También existen actores que surgen de la pregunta ¿qué otros susbsistemas interactuan con el cliente del *SMAD*?, se observa que en el capítulo cinco se tiene dos subsistemas con los cuales hay intercambio de información, éstos son la base de datos de control y el *Servidor Monitor*, aunque en realidad la comunicación con este último componente es a través de dos aplicaciones ya definidas, la *interface* y el *notificador* (ver apartados 4.1.6 y 4.1.7). Estos tres elementos (*metabase*, *interface* y *notificador*) por lo tanto también van a definir actores en el análisis de casos de uso.

6.1.1.2 Casos de uso del sistema

Los *casos de uso* representan la funcionalidad del sistema al mostrar la interacción entre los actores y el propio sistema. Para obtener los *casos de uso* se deben responder preguntas encaminadas a descubrir las tareas que realizan los actores y si van a realizar cambios que modifiquen la información del sistema (ver apartado 5.1.2). Del capítulo tres se determina que las tareas a realizar por el cliente son:

a) manejo de cuentas de usuario,

b) manejo de los operandos,

c) *manejo de disparadores,*

d) *manejo de fórmulas,*

e) *configuración de fórmulas,*

f) *consulta y monitoreo de alarmas,*

g) *creación y mantenimiento de bitácora de alarmas, y*

h) *creación y mantenimiento de bitácora de errores.*

Cada una de estas tareas obliga a actualizar los datos en el sistema, o la necesidad de informar de la modificación de cierta información a los distintos actores, por lo tanto cada una de estas tareas definirá un *caso de uso*. La información que se manejará en la mayoría de los *casos de uso* será la misma que se tiene en la definición del esquema de la base de datos de control (*DBMAD*); sin embargo, hay que especificar y explicar en qué consisten las funciones u operaciones que se llevarán en *cada caso de uso*, es decir, describir con detalle la funcionalidad de los *casos de uso* (ver apartado 1.5.1.1). Para algunos *casos de uso*, lo conveniente será realizar diagramas de interacción (ver apartado 1.5.1.2) que muestren la secuencia de intercambio de información entre los actores y los posibles objetos que se puedan llegar a determinar.

6.1.1.3 Descripción del caso de uso Manejo de cuentas de usuario

El manejo de las cuentas de usuario queda restringido sólo a los administradores del sistema (ver fig. 6.1). La información que se maneja para las cuentas de usuarios es la siguiente: el identificador, el *login*, el *password* y el nombre completo del usuario, así como el rol asociado a esa cuenta de usuario, el rol marcará los privilegios que tendrá el usuario para poder realizar ciertas funciones en los diferentes módulos del cliente del *SMAD* (ver apartados 3.2.8 y 3.5.1).

Figura 6.1 Diagrama de caso de uso para el manejo de cuentas de usuario

Las funciones a realizar para el caso de uso *manejo de cuentas* son:

- Consultar los usuarios que se tengan definidos.
- Agregar, actualizar o borrar registros de usuarios en la base de datos de control.

La especificación de cómo se llevarán acabo estas tareas se presenta en las tablas 6.1 y 6.2.

Tabla 6.1 Especificación de la tarea de *consulta de datos*

Consulta de datos
• Establecer una conexión con el *Servidor SQL*.
• Realizar una consulta que permita traer los datos que se requieran
• Almacenar los datos del *Servidor SQL* y desplegarlos en pantalla.

Tabla 6.2 Especificación de las tareas *agregar actualizar o borrar registros*

Agregar, actualizar o borrar registros
• Establecer una conexión con el *Servidor SQL*.
• Si se requiere seleccionar algún registro.
• Si se requiere proporcionar los datos para agregar o actualizar algún registro.
• Seleccionar la operación a realizar y mandarla a ejecutar al *Servidor SQL*.
• Esperar la respuesta del servidor.
• Desplegar mensaje en pantalla de los resultados obtenidos.

6.1.1.4 Descripción caso de uso Manejo de operandos

Los operandos sólo podrán ser definidos por los administradores, para ello se necesita el identificador del operando, el nombre de éste, el tipo de operando y el tipo de dato, y dependiendo del tipo de operando, se tendrá que especificar distinta información; si el tipo de operando es *atributo de la transacción*, se tiene que dar el nombre de la columna (referencia) a la cual va a representar dicho operando, si el tipo de operando es *valor fijo*, se tiene que especificar el valor que representa ese operando conforme al tipo de dato especificado, y si el tipo de operando es *campo*, se tendrá que especificar la base de datos, la tabla y la columna de donde se va obtener el valor de dicho operando. Si el tipo de operando es *campo* en caso de que se requiera, se podrá aplicar alguna función agregada a dicha columna y se podrá especificar una condición.

Si se necesita dar de alta alguna condición, se requiere conocer la definición de ésta, así como el valor de la traducción del *parser* (definición compilada) de esa condición.

Debido a que el funcionamiento para el manejo de operandos requiere más detalle, se tiene que asociar un *extend* (ver apartado 5.1.2) en el diagrama de *casos de uso* (ver fig. 6.2).

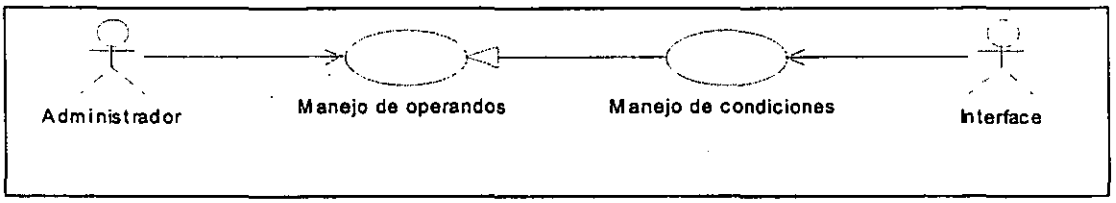


Figura 6.2 Diagrama de casos de uso *para el manejo de operandos*

Las funciones que se tienen que realizar para el *caso de uso manejo de operandos* son:

- Consultar los operandos que se tengan definidos.
- Agregar, actualizar o borrar los operandos en la base de datos de control.

La especificación de estas tareas es igual a las del manejo de usuarios, sólo que ahora con la información para operandos (ver tablas 6.1 y 6.2). Las funciones que se tienen que realizar para el *caso de uso manejo de condiciones* son:

- Consultar las condiciones que se tengan definidas.
- Agregar, actualizar o borrar las condiciones en la base de datos de control.

Se observa que el manejo de las condiciones es similar a los anteriores, de hecho la tarea de consulta tiene la misma funcionalidad (ver tabla 6.1), sin embargo, se requiere que las condiciones sean evaluadas léxica y sintácticamente por un *parser* en el *Servidor Monitor* antes de ser creadas o modificadas (ver apartado 4.1.10), en este caso se utiliza la interface para realizar esta comunicación con el *Servidor Monitor* como se expresa en el diagrama de *casos de uso* (ver fig. 6.2). La especificación de la tarea para borrar un registro es similar a las anteriores (ver tabla 6.2). Por lo tanto las tareas de agregar y actualizar condiciones se llevarán acabo de acuerdo a la tabla 6.3.

Tabla 6.3 Especificación de tareas *actualizar* y *borrar* utilizando la *interface*

Agregar o actualizar registros utilizando la interface

- Establecer una conexión con el servidor *SQL*.
- Si se requiere seleccionar algún registro.
- Si se requiere proporcionar los datos para agregar o actualizar algún registro.
- Establecer una conexión con la interface.
- Mandar la definición de la cadena a ser evaluada, vía la interface.
- Esperar la respuesta y los datos correspondientes de la evaluación a través de la interface, si la evaluación fue incorrecta abortar la operación.
- Seleccionar la operación a realizar y mandarla a ejecutar al servidor *SQL*
- Esperar la respuesta del servidor.
- Desplegar mensaje en pantalla de los resultados obtenidos.

6.1.1.5 Descripción caso de uso Manejo de disparadores

Los administradores también serán los únicos en definir disparadores, para éstos se requiere del identificador del disparador, el nombre lógico que el usuario le da cuando lo define, además de una lista de todos los operandos que se podrán recuperar con ese disparador y de la posibilidad de generación de un *script* que contenga la definición de el *trigger* que representa al disparador. Estos últimos dos puntos serán extensiones del caso de uso principal (ver fig. 6.3).

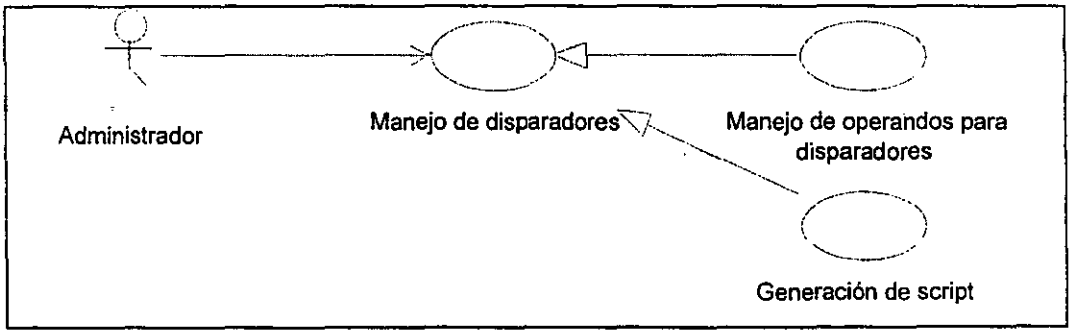


Figura 6.3 Diagrama de casos de uso para el manejo de disparadores

Las funciones a realizar para el caso de uso *manejo de disparadores* son:

- Consultar los disparadores que se tengan definidos.
- Agregar, actualizar y borrar los disparadores en la base de datos de control.

Las especificaciones de las dos primeras tareas es similar a las ya descritas (ver tablas 6.1 y 6.2). Las funciones a realizar para el caso de uso *manejo de operandos* para *disparadores* son:

- Manejo de lista de operandos que pueden ser asociados a un disparador.
- Manejo de lista de operandos seleccionados para ser recuperados por el disparador.

La especificación de estas tareas queda definida en las tablas 6.4 y 6.5.

Tabla 6.4 Especificación de tarea *manejo de lista de operandos para disparadores*

Lista de operandos para disparadores
• Establecer una conexión con el *Servidor SQL*. • Seleccionar todos los operandos del tipo *atributo de hecho.* • Almacenar los datos que envía el *Servidor SQL* y desplegarlos en una lista.

Tabla 6.5 Especificación de la tarea *manejo de lista de operandos por recuperar*

Lista de operandos por recuperar
• Seleccionar un disparador ya existente o crear uno nuevo. • Escoger de la lista de operandos para disparadores aquellos que serán recuperados por el disparador seleccionado y colocarlos en una lista de operandos por recuperar. • Cuando de tenga que agregar actualizar o borrar un disparador establecer una conexión con el *Servidor SQL*, actualizar en la base de datos de control la tabla que contiene la relación entre los operandos del tipo *atributos del hecho* y el disparador que se está seleccionando (ver apéndice E).

La función a realizar para el *caso de uso generación de script* es:

- Generar la definición del *script* que representa al disparador. La especificación de esta tarea se da en la siguiente tabla.

Tabla 6.6 Especificación de la tarea *generar trigger*

Generar definición de *trigger*

- Establecer conexión con el *Servidor SQL*.
- Seleccionar los nombres reales de los operandos seleccionados, los cuales serán recuperados vía el *trigger* a definirse.
- Generar el *script* para el *trigger* utilizando el nombre que el usuario dió para definirlo y los nombres reales de los operandos seleccionados.
- Mostrar la definición del *trigger* dando la posibilidad de guardarla en un archivo.

6.1.1.6 Descripción caso de uso Manejo de fórmulas

Las fórmulas pueden ser definidas por administradores (*fórmulas base*) o por operadores (*fórmulas personalizadas*, ver apartado 3.4.2). Para las fórmulas se requiere del identificador de la fórmula, el nombre, la definición de la fórmula que introduce el usuario, la definición compilada que regresa el *parser* después de validar la definición de la fórmula, el tipo de fórmula, el acceso, el estado, la prioridad y el tipo de dato que se obtendrá como resultado de su cálculo.

Para definir una fórmula se necesita también saber quién es el usuario que la creó, el disparador asociado que indique el inicio de su cálculo, además de elaborar listas de los operandos que constituyen a la fórmula y de los operandos que el usuario considera importantes, así como visualizar al momento de que la fórmula genere una alarma.

También la definición de una fórmula implica la creación de expresiones que la constituyen, las cuales podrán tener valores cuando se genera una alarma; para las expresiones se requiere de un identificador, la definición tecleada por el usuario y la definición compilada que regresa el *parser* (todos estos requerimientos surgen del análisis de relaciones entre entidades de la base de datos *DBMAD*, apartado 3.5.7). Estos últimos aspectos se contemplarán como casos de uso extendidos (ver figura 6.4)

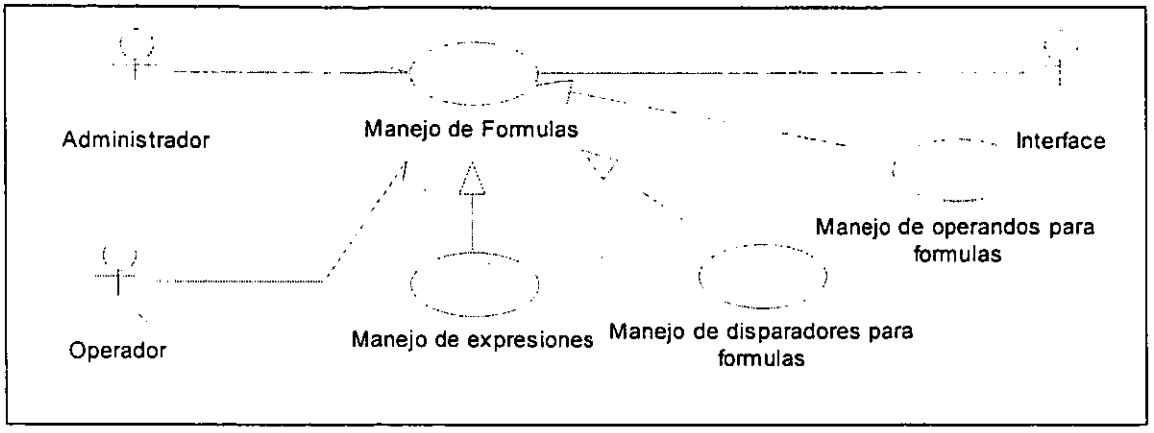


Figura 6.3 Diagrama de casos de uso para el *manejo de fórmulas*

Las funciones a realizar para el caso de uso *manejo de fórmulas* son:

- Consultar las fórmulas que se tengan definidas.
- Agregar, actualizar y borrar fórmulas en la base de datos de control.

La especificación para la tarea de consulta es similar a las ya definidas (ver tabla 6.1), para las tareas de agregar y actualizar una fórmula el procedimiento es similar al de las condiciones, ya que se necesita de un *parser* (ver apartado 4.1.11) para hacer valida su definición utilizando la *interface* para lograr la comunicación con el *Servidor Monitor* (ver tabla 6.3). La tarea de borrar fórmula también es similar a las ya definidas (ver tabla 6.2).

La función a realizar para el caso de uso *manejo de disparadores para fórmulas* es:

- Manejar una lista de disparadores, los cuales pueden ser asociados a las fórmulas. La especificación de esta tarea esta en la tabla 6.7

Tabla 6.7 Especificación de la tarea *lista de disparadores para fórmulas*

Lista de disparadores para fórmulas

• Establecer una conexión con el *Servidor SQL*. • Seleccionar todos los disparadores. • Almacenar los datos que envía el *Servidor SQL* y desplegarlos en una lista. • Seleccionar de la lista de disparadores, el disparador que estará asociado a la fórmula que se esté definiendo, en caso de que sea una fórmula personalizada que utilice una fórmula base, el disparador de la fórmula base será también el disparador de la fórmula personalizada. • Actualizar en la base de datos de control la tabla que contiene la relación entre el disparador que se escogió y la fórmula que se está utilizando cuando se tenga que agregar, actualizar o borrar una fórmula (ver apéndice E).

Las funciones a realizar para el caso de uso *manejo de operandos para fórmulas* son:

- Manejo de los operandos que forman la definición de la fórmula.
- Manejo de lista de operandos que pueden ser visualizados por una fórmula.
- Manejo de lista de operandos seleccionados para ser visualizados por una fórmula.

La especificación para estas tareas se presenta en las tablas 6.8, 6.9 y 6.10.

Tabla 6.8 Especificación de la tarea *manejo de operandos que forman la definición de una fórmula*

Manejo de operandos que forman la definición de una fórmula
• Establecer una conexión con el *Servidor SQL*. • Cuando se tenga que agregar, actualizar o borrar una fórmula, **actualizar en la base de datos de control la tabla** que contiene la relación entre la fórmula que se escogió y los operandos que la constituyen (ver apéndice E). • Para determinar esta lista de operandos se tendrá que utilizar la definición compilada de la fórmula que regresa el parser y en la cual cada operando es etiquetado con una cierta sintaxis (ver apartado 5.2.**54**).

Tabla 6.9 Especificación de la tarea *manejo de lista de operandos que se pueden visualizar*

Manejo de lista de operandos que se puede visualizar

• Establecer una conexión con el *Servidor SQL*. • Seleccionar todos los operandos de tipo *campo* y *valor* que son los que sus valores se pueden obtener directamente, obtener también los operandos del tipo *atributo de hecho* que deben ser recuperados por el disprador asociado o seleccionado a la fórmula utilizada. • Almacenar los datos que envía el *Servidor SQL* y desplegarlos en una lista.

Tabla 6.10 Especificación de la tarea *manejo de lista de operandos a ser visualizados*

Manejo de lista de operandos a ser visualizados
• Seleccionar una fórmula ya existente o crear una nuevo. • Escoger de la lista de operandos que se pueden visualizar aquellos que se considere se necesite desplegar su valor cuando la fórmula genere alarmas. • Cuando se tenga que agregar, actualizar o borrar una fórmula, hay que establecer una conexión con el *Servidor SQL*, actualizar en la base de datos de control la tabla que contiene la relación entre los operandos cuando se quieren visualizar y la fórmula que se está seleccionando (ver apéndice E)

6.1.1.7 Descripción caso de uso Configuración de fórmulas

La configuración de las fórmulas consiste en modificar ciertas de sus propiedades, las cuales son la prioridad, el acceso, la disponibilidad y decir si es acumulable o no. También configurar una fórmula será establecer un rango de fechas dentro de la cual, la fórmula estará activa y podrá ser calculada, es decir, un periodo en el cual la fórmula podrá generar alarmas; para esto se tiene que informar al *Servidor Monitor* vía la *interface* cada vez que se active una fórmula, de forma que a partir de ese momento éste inicie el cálculo de la fórmula. La configuración de las fórmulas podrá ser hecha por administradores, usuarios u operadores, dependiendo del tipo de acceso que la fórmula tenga y del tipo de usuario que la quiera configurar se contemplan casos de uso extendidos para representar las dos tareas para la configuración de fórmulas (ver apartados 3.5.1 y 3.5.4) (ver fig. 6.4).

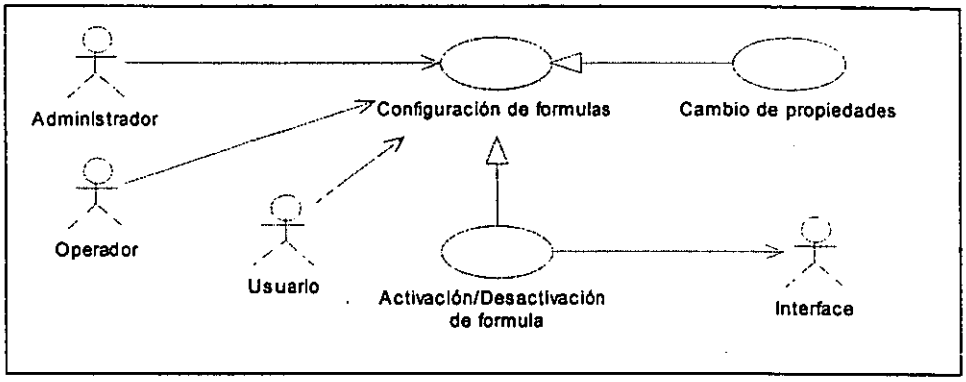


Figura 6.4 Diagrama de casos de uso para la configuración de fórmulas

La función a realizar para el caso de uso *configuración de fórmulas* es:

- Manejo de una lista de fórmulas que muestre su configuración. La especificación de esta tarea se muestra en la tabla 6.12.

Tabla 6. 12 Especificación de la tarea *Manejo de la lista de configuración de fórmulas*

Manejo de lista de configuración de fórmulas
• Establecer conexión con el *Servidor SQL* • Seleccionar el nombre de la fórmula, así como sus características configurables y si es posible su rango de activación de cada una de ellas. • Almacenar los datos que envía el servidor y mostrarlos en pantalla en una lista.

La función a realizar para el caso de uso *cambio de propiedades* es:

- Modificar las propiedades configurables de las fórmulas. La especificación de esta tarea se muestra en la tabla 6.13.

Tabla 6.13 Especificación de la tarea *Modificar propiedades configurables de fórmulas*

Modificar propiedades configurables de fórmulas
• Seleccionar una fórmula de la lista de configuración de fórmulas y especificar que se quiere modificar sus propiedades. • Verificar que el rol del usuario que seleccionó la fórmula tenga permisos para modificarla, de acuerdo al acceso que ésta tenga y verificar que la fórmula no está activada, de no cumplirse con estas condiciones hay que abortar la operación. • Establecer conexión con el *Servidor SQL*. • Cambiar las propiedades que se requieran de la fórmula que se seleccionó vía *Sevidor SQL*. • Esperar por los resultados que envíe el servidor y desplegarlos en pantalla.

La función a realizar para el caso de uso *Activar/Desactivar fórmula* es:

- Establecer o borrar el rango de tiempo para la notificación de una fórmula. La especificación de esta tarea se muestra en la tabla 6.14.

Tabla 6.14 Especificación de la tarea *Establecer o borrar rango de notificación*

Establecer o borrar rango de notificación
• Seleccionar una fórmula de la lista de configuración de fórmulas que no esté activa y **especificar que se quiere activar**. • Verificar que el rol del usuario que seleccionó la fórmula tenga permisos para activarla, de acuerdo al acceso que ésta tenga, de no cumplir con ésta condición, habrá que abortar la operación. • Establecer conexión con el *Servidor SQL* • Establecer el **rango de tiempo** en el cual la fórmula seleccionada va a estar activa. • Mandar los datos para activar la fórmula actualizando la tabla que contiene la relación de notificación entre la fórmula y el usuario (ver **apéndice E**). • Establecer comunicación con la *interface*. • Mandar vía *interface* el identificador de la fórmula para que ésta pueda ser calculada por el *Servidor Monitor*. • Esperar la respuesta que envíe la *interface* con respecto a la activación, si es negativa habrá que abortar toda la operación. • Desplegar los resultados obtenidos en pantalla.

6.1.1.8 Descripciónde l caso de uso Consulta y monitoreo de alarmas

La consulta y el monitoreo de alarmas se podrá llevar acabo por los tres actores que usan el sistema (administrador, operador y usuario). La información que se necesita es el *id* de la alarma que se generó al calcular la fórmula, así como la hora y fecha (tiempo) en que ocurrió; para alarmas acumulables se necesita saber si se van a recibir más datos (activa), la lista de operandos a visualizar, su definición y su valor al momento en que se genere la alarma; por último, para la lista de expresiones se necesitará su definición y su

valor también al momento en que se genere la alarma. Sólo se mostrarán los datos de alarmas generadas por fórmulas que el usuario activó.

Con los datos obtenidos de las alarmas se podrán realizar distintas operaciones, entre ellas, permitir mostrar las alarmas de acuerdo a criterios de búsqueda y asociar mensajes a las alarmas que se generaron, en este caso la información que se necesita para manejar los mensajes es el *id* del mensaje, la fecha y hora en que se envió (tiempo), el *id* del usuario a quien va dirigido el mensaje (emisor) y por último el contenido del mensaje (notas); por último, cualquier actor tiene la posibilidad de borrar las alarmas que se le notificaron.

El monitoreo de alarmas consiste en recibir alarmas en tiempo de ejecución, es decir, al mismo tiempo que se estén consultando las alarmas se pueden recibir nuevas alarmas. Para esto se requiere que se establezca una comunicación asíncrona con la aplicación *notificador* que es la encargada de recibir las alarmas del *Servidor Monitor* al momento que éstas se generan. El usuario debe configurar el modo en que recibirá la notificación de nuevas alarmas. Los distintos requerimientos que se mencionan requieren del uso de varios casos de uso extendidos (ver fig.6.5).

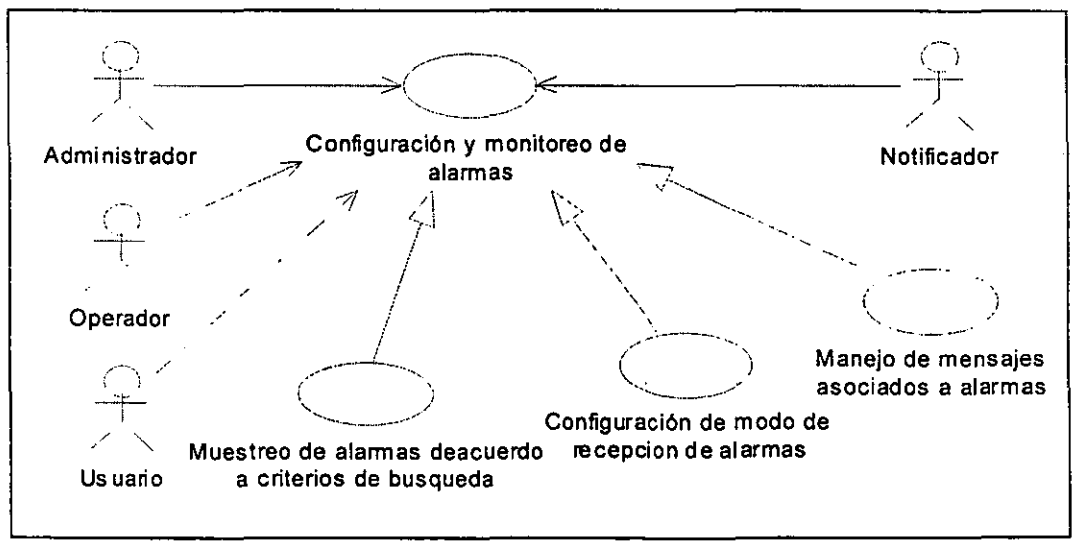


Figura 6.5 Diagrama de casos de uso para la configuración y monitoreo de alarmas

Las funciones a realizar para el caso de uso *Configuración y monitoreo de alarmas* son:

1) Manejo de una lista de alarmas que muestre los datos y propiedades de las mismas.

2) Inicialización y manejo de la comunicación con el notificador, con el fin de poder monitorear las alarmas en tiempo de ejecución.

3) Borrado personalizado de alarmas.

La especificación de estas tareas queda expuesta en las tablas 6.15, 6.16 y 6.17.

Tabla 6.15 Especificación de la tarea *Manejo de lista de alarmas*

Manejo de lista de alarmas
• Establecer una conexión con el *Servidor SQL*. • Seleccionar todas las alarmas que se hallan notificado al usuario, recordando que sólo se mostrarán los datos de alarmas generadas por fórmulas que el usuario activó. • Almacenar los datos que envía el *Servidor SQL* y desplegarlos.

Tabla 6.16 Especificación de la tarea *Inicialización y manejo de comunicación con el notificador*

Inicialización y manejo de la comunicación con el notificador
• Mandar a ejecutar el *notificador*. • Establecer comunicación con el *notificador*. • Esperar respuesta de confirmación de comunicación, en caso de ser negativa abortar la operación. • Interrumpir la ejecución de lo que se esté haciendo cuando el *notificador* envíe una nueva alarma, atender la petición y recibir los datos. • Almacenar los datos que envió el *notificador* y anexarlos a la lista de alarmas, mandar el mensaje de la notificación al usuario de acuerdo al criterio de configuración del modo de recepción de alarmas que el usuario halla seleccionado. • Finalmente cuando se termine la tarea de consulta y monitoreo se deberá mandar a detener al *notificador*.

Tabla 6.17 Especificación de la tarea *Borrado personalizado de alarmas*

Borrado personalizado de alarmas
• Establecer conexión con el *Servidor SQL*. • Seleccionar las alarmas que se quieren borrar de la lista de alarmas. • Mandar borrar las alarmas en la tabla que representa la relación entre las alarmas y el usuario.

La función a realizar para el caso de uso *Muestreo de alarmas de acuerdo a criterios de busqueda* es:

Establecer criterios de búsqueda para filtrar el número de alarmas en la lista de alarmas. La especificación de esta tarea se da en al siguiente tabla.

Tabla 6.18 Especificación de tarea *Criterios de busqueda para alarmas*

Criterios de busqueda para alarmas
• Cuando algún actor lo solicite, establecer filtros y búsquedas por criterio para a la consulta de los datos mostrados en la lista de alarmas. • Cuando algún actor lo solicite, establecer rango de fechas para la consulta de los datos mostrados en la lista de alarmas.

La función a realizar para el caso de uso *Configuración del modo de recepción de alarmas* es:

- Establecer un mecanismo para la forma en que se deben de avisar la recepción de una nueva alarma en tiempo de ejecución.

Tabla 6.19 Especificación de tarea *Mecanismos para la recepción de nuevas alarmas*

Mecanismos para la recepción de nuevas alarmas
1) Mostrar cuando alguno de los actores (usuarios del sistema) lo soliciten, los criterios de aviso cuando se genera una alarma, los cuales quedan definidos como: *a)* Avisar con un *Beep* (*Default*). *b)* Avisar con un mensaje que se despliegue. *c)* Maximizar la pantalla. *d)* No notifica Según la selección del usuario, aplicar la política de aviso de nueva alarma cuando ésta se presente, **guardar la configuración al final en archivo de configuración**.

Las funciones a realizar para el caso de uso *Manejo de mensajes asociados a alarmas* son:

- Crear y borrar mensajes asociados a las alarmas generadas.

- Manejo de lista de mensajes asociados a alarmas del actor que este en ese momento consultando las alarmas.

La especificación de la primera tarea es similar a la presentada anteriormente (ver tabla 6.1), solo que con la información de mensajes y sin tener la operación de actualización. La especificación de la segunda tarea se presenta en las siguiente tablas.

Tabla 6.20 Especificación de la tarea *Manejo de lista de alarmas*

Manejo de lista de mensajes
• Establecer una conexión con el *Servidor SQL*. • Seleccionar todos las mensajes que se hallan notificado al usuario, esto es todos los mensajes de la tabla de mensajes que contengan el identificador del usuario que esta utilizando el sistema en ese momento. • Almacenar los datos que envía el *Servidor SQL* y desplegarlos.

6.1.1.9 Descripción caso de uso Creación y matenimiento de bitácora de alarmas

La bitácora de alarmas es una lista que muestra todas las alarmas generadas, la cual sólo puede ser consultada por los administradores (ver fig. 6.6). Información necesaria para la bitácora de alarmas es el identificador de las alarmas, la fecha y hora (tiempo) en que fueron generadas, identificador, nombre, tipo de fórmula y prioridad de las fórmulas que generaron esas alarmas.

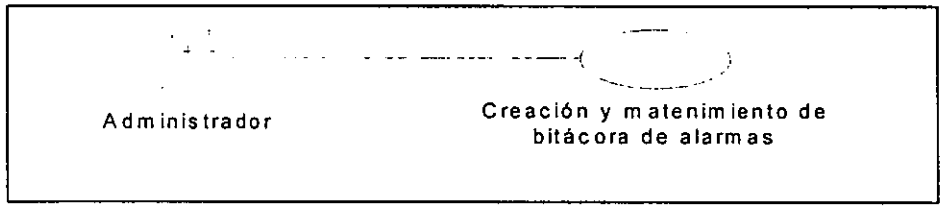


Figura 6.6 Diagrama de caso de uso para creación y mantenimiento de bitácora de alarmas

Las funciones por realizar del caso de uso *Creación y mantenimiento de bítacora de alarmas* son:

- Consulta de todas las alarmas generadas.
- Borrado de alarmas.

La especificación de estas tareas es similar a las ya definidas (ver tablas 6.1 y 6.2) solo que ahora tomando en cuenta los datos de las alarmas y actualizando todas las tablas con las que tiene relación la entidad alarma cada vez que una de éstas es borrada.

6.1.1.10 Descripción caso de uso Creación y mantenimiento de errores

La bitácora de errores es una lista que muestra todos los errores generados por el *Servidor Monitor* y sólo puede ser consultada por administradores (ver fig. 6.7). Para la bitácora de errores se necesita saber el mensaje, el tipo de error (de sintaxis o de recuperación de algún parámetro), la fecha y la hora en que ocurrió el error (tiempo).

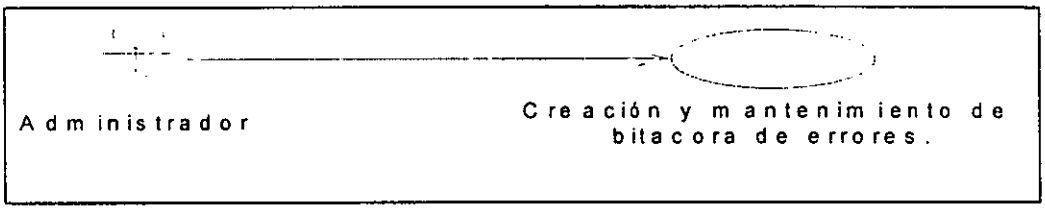


Figura 6.7 Diagrama de caso de uso para creación y mantenimiento de bitácora de errores

Las funciones por realizar del caso de uso *Creación y mantenimiento de bitacora de errores* son:

- Consulta de todas las alarmas generadas.
- Borrado de alarmas.

La especificación de estas tareas es similar a las ya definidas (ver tablas 6.1 y 6.2) solo que ahora tomando en cuenta los datos de los errores.

6.1.1.11 Diagramas de iteración.

Los diagramas de iteración sirven para representar una secuencia de sucesos (mensajes) entre un agente externo (actor) y los componentes que se puedan llegar a detectar del sistema (objetos) (ver apartado 1.5.1.3).

Basándonos en los *casos de uso* ya determinados, podemos detectar que en lo casos de uso *manejo de cuentas de usuario*, *manejo de operandos*, *manejo de disparadores* y *manejo de fórmulas*, lo que se está requiriendo es un objeto *forma de captura* y un objeto *menú de opciones*, donde las operaciones a realizar son agregar, actualizar y borrar los registros en la base de datos *DBMAD* vía el *Servidor SQL* de cada uno de los distintos datos que se manejan en cada caso de uso.

También una de las operaciones comunes es la creación de una lista que muestre los datos de los registros de la base de datos *DBMAD*. El siguiente diagrama de iteración muestra los pasos a seguir para realizar estas tareas (ver fig. 6.8).

Al agregar como al actualizar una condición o una fórmula, se tiene que hacer validando su definición a través de una *Parser* que es una función del *Servidor Monitor*, en este caso la comunicación con éste se hace vía la aplicación *interface*, este proceso es el mismo para cuando se tiene que agregar, actualizar u obtener condiciones, fórmulas o expresiones de fórmulas; el siguiente diagrama de iteración muestra los pasos que se desarrollan cuando se presenta (ver fig. 6.9).

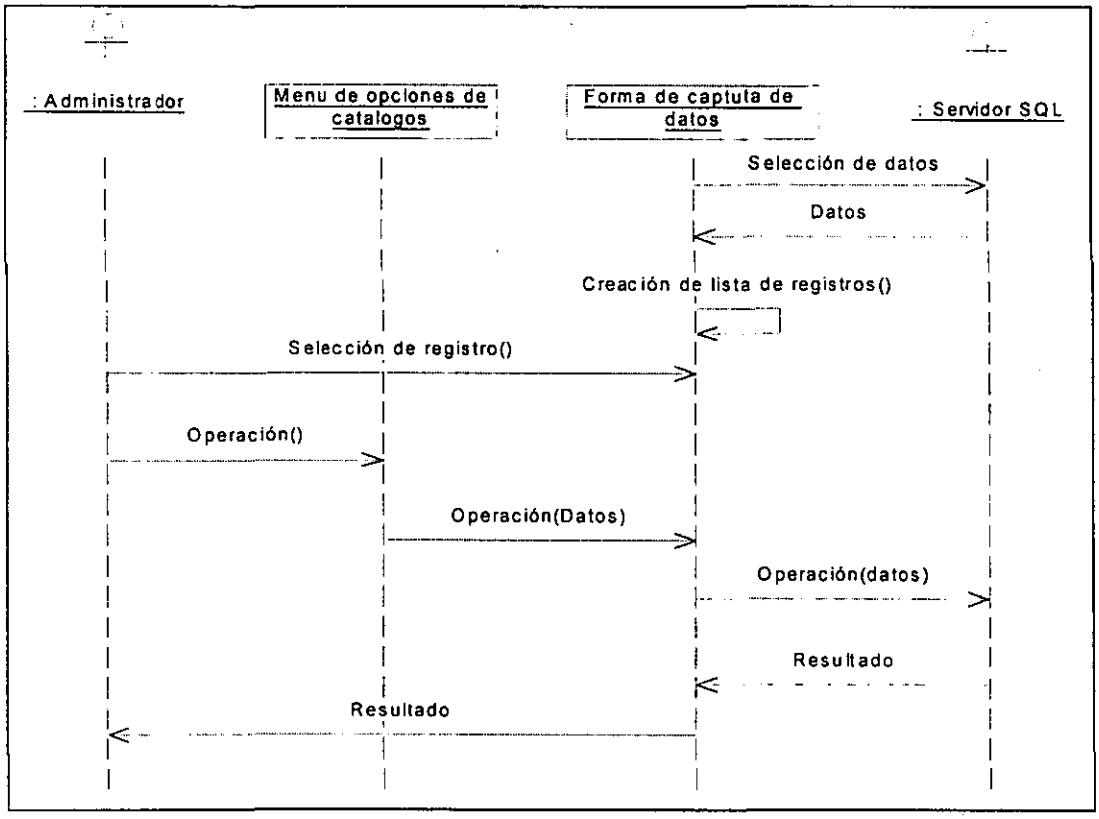


Figura 6.8 Diagrama de iteración para representar operaciones con catálogos

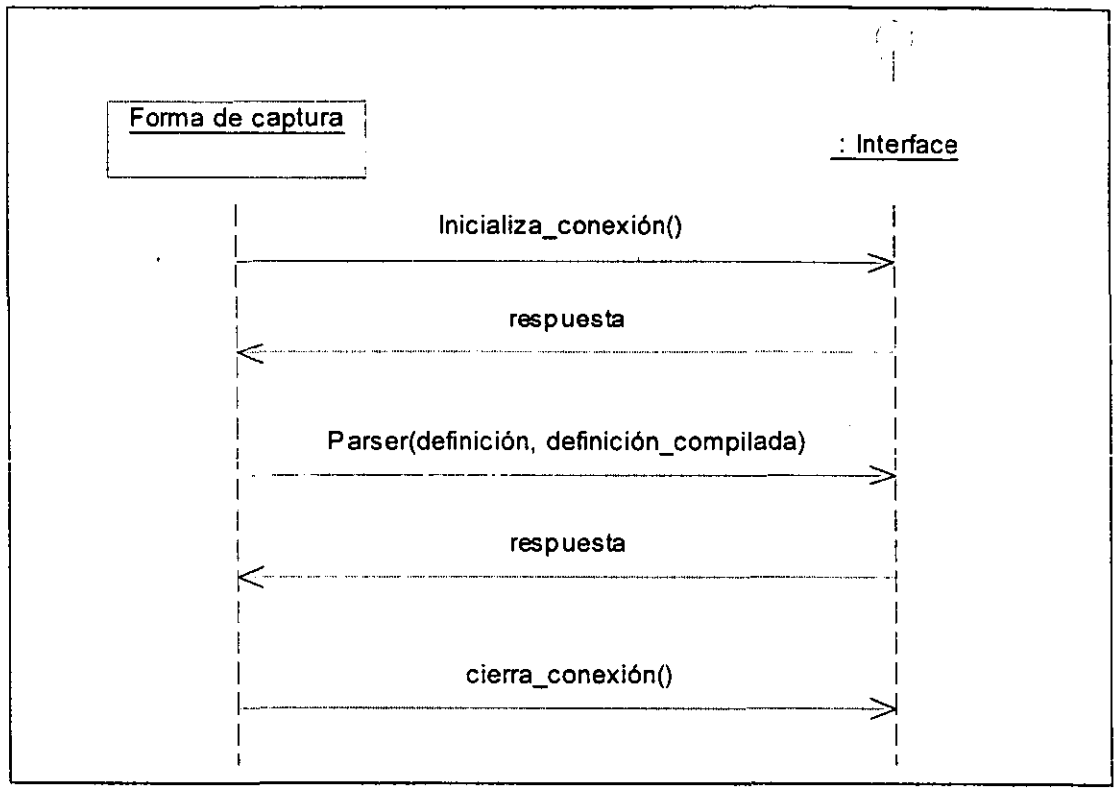


Figura 6.9 Diagrama de iteración para representar la comunicación con la interface

6.1.2 Análisis de dominio

Con *análisis de casos de uso* tenemos una descripción de la funcionalidad del cliente del *SMAD*, ahora lo que tenemos que plasmar son los componentes generales que intervienen (ver apartado 1.5.1.4), ésto se conoce como *análisis de dominio*. El análisis de dominio nos permite determinar de manera más precisa el tipo de sistema que se requiere. En este caso nos enfocaremos a determinar los objetos generales, las tareas generales y los subsistemas que integran al cliente del *SMAD*.

6.1.2.1 Objetos generales

Dentro de los objetos generales que intervienen para la realización del cliente del *SMAD* tenemos formas de captura o formas para despliegue de datos, así como menús de opciones y listas de despliegue de datos; estos objetos se necesitan en casi todos los casos de uso que analizados dependiendo de la funcionalidad que éstos tengan. Por ejemplo tenemos que para los casos de uso que representan la funcionalidad de catálogos (Manejo

de cuentas de usuario, manejo de disparadores, manejo de operandos y manejo de fórmulas) los que vamos a tener de forma general va a ser un *objeto forma de captura*, un *objeto menú de opciones* y un *objeto lista de despliegue de datos*. También se pueden determinar objetos de acuerdo a las tareas en común que se tengan en los casos de uso que se esté analizando, por ejemplo, se observa que de manera general en la mayoría de los casos de uso y particularmente en los que representan la funcionalidad de catálogos, se necesita realizar una conexión con el *Servidor SQL*, por lo tanto se necesita de un objeto que establezca esta comunicación. Resumiendo éstos, serían los objetos que se necesitan para casos de uso que representan la funcionalidad de catálogos (ver fig. 6.11).

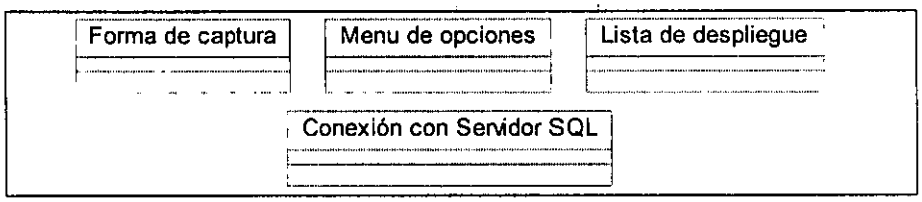


Figura 6.11 Objetos generales para catálogos

Por otro lado para los casos de uso que representan sólo el despliegue de datos (Configuración de fórmulas, consulta y monitoreo de alarmas, creación y mantenimiento de bitácoras de alarmas y errores) tenemos que los objetos que se necesitan son: Un *objeto forma de despliegue*, un *objeto menú de opciones* y de igual manera un *objeto lista de despliegue*. También para estos casos de uso se necesita de una conexión con el *Servidor SQL* por lo que se requiere de un objeto que se encargue de esta tarea (ver fig. 6.12).

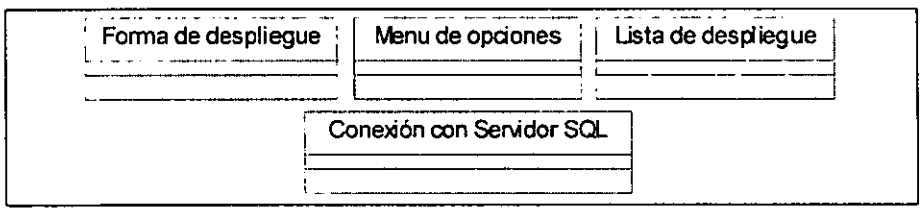


Figura 6.12 Objetos generales para despliegue de datos

6.1.2.2 Tareas generales

Las tareas generales también dependen de los casos de uso que estemos tratando, por ejemplo para los casos de uso que representan catálogos (Manejo de cuentas de usuario, manejo de disparadores, manejo de operandos y manejo de fórmulas) las tareas que se

tiene en común son: obtener, almacenar y desplegar registros, agregar, actualizar y borrar registros y por supuesto establecer comunicación con el *Servidor SQL* (ver fig. 6.13).

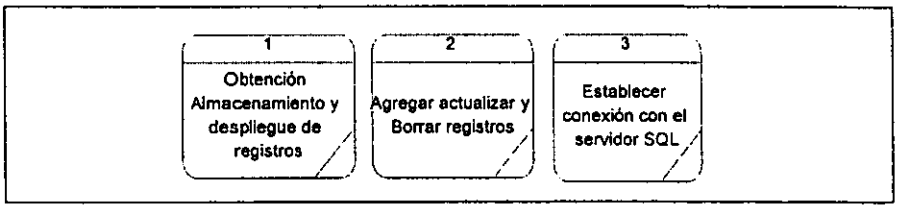


Figura 6.13 Tareas generales para catálogos

Por otra parte, para los casos de uso que representan la consulta especializada de información, tenemos que las tareas comunes a realizar son: obtener, almacenar y desplegar información específica; borrar ciertos registros que representan relaciones o actualizar los que ya se tienen; de igual forma establecer una conexión con el servidor *SQL* y además establecer criterios personalizados para la consulta de la información. (ver fig. 6.14).

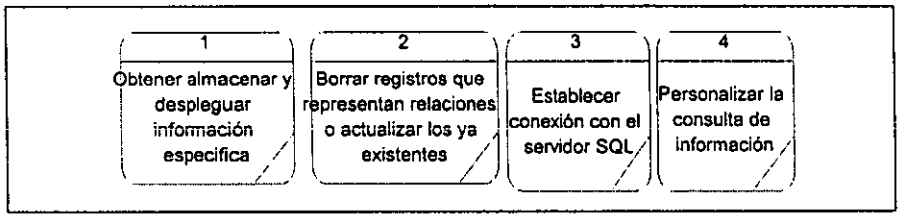


Figura 6.14 Tareas generales para el despliegue de datos

6.1.2.2 Subsistemas para el cliente

Los subsistemas o componentes generales que forman el cliente para el *SMAD* ya se definieron en el capítulo cuatro (ver apartados 4.1.3, 4.1.4 y 4.1.5) y se tratarán con más detalle en el modelo de implantación; como parte del análisis de dominios aquí sólo se dará una pequeña definición de cada uno de ellos:

- Módulo de Administración. Interfase gráfica de usuario para la definición y consulta de los usuarios, de las fórmulas base, de los operandos y de los disparadores.
- Módulo de Configuración. Interfase gráfica de usuario para la definición, consulta, configuración y activación de fórmulas base y personalizadas.
- Módulo de Monitoreo. Interfase gráfica de usuario para la consulta y recepción de alarmas generadas por el cálculo de fórmulas.

- *Interfase*. Librería dinámica (*DLL*[1]) encargada de ser el medio por el cual cualquiera de los módulos pueda mandar a ejecutar alguna función específica que realice el *Servidor Monitor*.
- *Notificador*. Aplicación capaz de establecer una conexión permanente con el *Servidor Monitor* y a través de ésta poder recibir de manera asíncrona mensajes y datos que deberá posteriormente enviar al módulo de monitoreo.

Una vez determinados los *objetos tareas* y subsistemas generales para el cliente del *SMAD*, ya se tendrá el análisis de dominio del sistema. Conclusiones importantes de este análisis de dominio es por ejemplo que el sistema que se trata de realizar es una *interfaz interactiva*. Este tipo de sistemas está dominado por la iteración con agentes externos como se pudo constatar en el análisis de casos de uso. Estos actores externos son independientes del sistema, por lo tanto no es posible controlar sus entradas, aun cuando el sistema pueda solicitar respuesta de ellos. En este caso, como es un sistema que va a partir de cero, lo que se recomienda es aislar los objetos que forman la interfaz de los que definen la semántica de la aplicación y utilizar objetos predefinidos para interactuar con los agentes externos, utilizando un ambiente de desarrollo que lo permita.

6.2 Modelo lógico para las aplicaciones cliente del SMAD

El modelo lógico es una descripción de lo que realiza el sistema (ver apartado 1.5.2) a través de la definición de los objetos que lo van a constituir. La definición de los objetos queda plasmada a través de dos modelos de los objetos. El modelo estático de los objetos que consiste en definir la estructura y métodos de los objetos, así como las relaciones que puedan existir entre ellos. El modelo dinámico representa el comportamiento de los objetos en el tiempo. En los siguiente apartados se obtiene estos dos modelo utilizando los distintos diagramas de la notación UML que existen para ello.

6.2.1 Modelo estático.

Como se menciono el modelo estático nos muestra la estructura de los objetos, es decir, los atributos y métodos que los componen, así como las relaciones que existen entre

[1] *DLL*: (*Dinamic Link Library*) Archivo que contiene una o más funciones que son compiladas, ligadas y almacenadas de manera separada de los procesos, de manera que cuando algún proceso ejecuta alguna función definida en un *DLL*, el sistema operativo liga el código dentro del espacio de memoria del

ellos. Sin embargo la primera tarea es identificar los objetos y después encontrar sus características. En el siguiente apartado se realiza esta primera tarea.

6.2.1.1 Identificación de objetos

El identificar los objetos requiere de volver analizar los diagramas de casos de uso y sus descripciones y también de analizar los diagramas de iteración, en donde de hecho ya se encuentran identificados algunos de estos objetos. Existen diversas formas para identificar los objetos (ver apartado 1.5.2.1), una de las cuales la que sugiere Jacobson consiste en convertir los casos de uso en diferentes tipos de objetos, según la funcionalidad que sé este analizando o que realice el caso de uso. Aunque existen algunos autores como Edward V. Berard[2] y Donal G. Firesmith[3] que afirman que el utilizar esta técnica de *mapera* directamente de los casos de uso a objetos nos puede llevar a realizar un análisis por procedimientos y no uno orientado a objetos, como la aplicación que sé esta analizando en este caso es una *interfaz interactiva* la mayoría de los casos de uso son objetos de tipo interfase por lo tanto si se puede realizar una conversión directa a este tipo de objetos, sin embargo, se utilizarán los conceptos de abstracción, generalización y composición para reafirmar la identificación de los objetos cuando sea necesario.

Para todos los casos de uso principales se observa que es necesario una conversión de cada uno de ellos a un tipo de objeto interfase. El tipo de objeto interfase no es otra cosa que un objeto ventana el cual contiene otros objetos con los cuales podrá llevar acabo las tareas indicadas. Esencialmente se van a estar utilizando dos tipos de objetos ventanas; las ventanas para catálogos que contendrán objetos para representar formularios y las ventanas para consulta que contendrán objetos para llevar acabo la consulta y el despliegue de información personalizada. No nos enfocaremos a definir con exactitud los objetos que estarán contenidos en cada una de los objetos ventana sólo aquellos en los que se requiera especificar debido a que son parte esencial para explicar algún funcionamiento que se indico en algún caso de uso y que se verá más adelante cuando se identifiquen las relaciones, los atributos y métodos y el modelo dinámico.

---

proceso.

[2] "Be carefull with use cases" Edward V. Berand. The object Agency , Inc.

[3] "Use Cases the Pros and Cons" Donald G. Firesmith. Knoledge Systems Sorporation.

Los objetos ventana para catálogos serán los que se identificaron como formas de captura en los diagramas de iteración del análisis de casos de uso y se necesitan para el manejo de: usuarios, disparadores, operandos, fórmulas, condiciones, mensajes. Los objetos ventanas para la consulta y despliegue de datos son los que sé identificarón como formas de despliegue de datos y se necesitan para el monitoreo de alarmas, la bitácora de errores y la bitácora de alarmas. Finalmente se necesita un objeto ventana que nos permita llevar acabo las tareas de configuración de fórmulas. Cada uno de estos objetos definirá una clase ventana en la que se puedan llevar acabo las operaciones especificadas. (fig. 6.15)

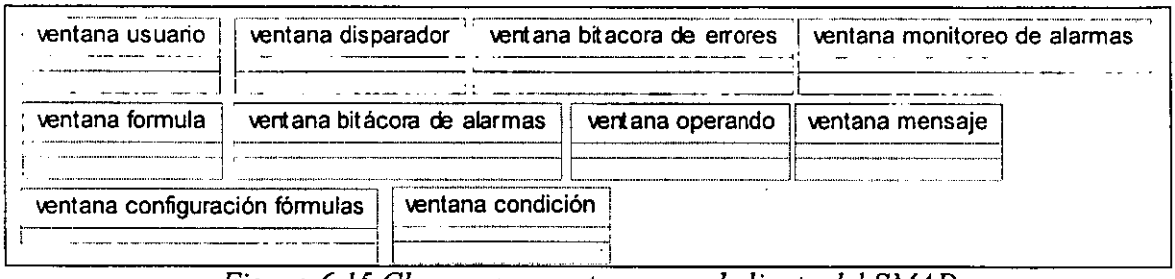


Figura 6.15 Clases para ventanas en el cliente del SMAD

Se necesitan para algunas de las clases ventana, otro objeto que sea el que les indique que operación tienen que realizar, y el cual es el que es manipulado directamente por alguno de los actores usuarios del sistema. El tipo de objeto menú es el identificado en los diagramas de iteración como "Menú de opciones" El objeto menú se necesita para aquellos objetos ventana que tienen una funcionalidad importante en los casos de uso, es decir, para los objetos que aparecen en los casos de uso principales, no en los extendidos. Por lo tanto se requerirá de un menú para las ventanas: usuario, disparador, operandos, formula, configuración de formulas, monitoreo de alarmas, bitácora de errores y bitácora de alarmas. Cada uno de estos objetos definirá una clase menú distinta (fig. 6.16).

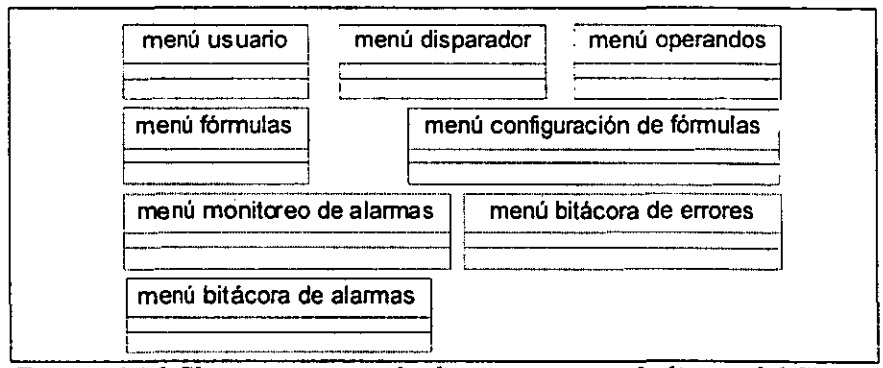


Figura 6.16 Clases para menús de opciones en el cliente del SMAD

Finalmente se tiene al objeto que establece la comunicación con el *Servidor SQL* y el cual se especifico en el análisis de dominios. Este objeto definirá una clase la cual llamaremos *conexión SQL*.

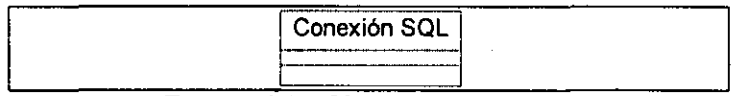


Figura 6.17 Clase conexión SQL

6.2.1.2 Identificación de atributos y métodos

La identificación de atributos y métodos se podrá llevar acabo utilizando la descripción de los casos de uso que se realizo en el modelo de requerimientos, los atributos son características de los objetos mientras que los métodos son las operaciones que pueden realizar (ver apartado 1.3.2). La información que se definió para los casos de uso se convertirá en los atributos para las distintas clases y las tareas que se establecieron para los mismos definirán los métodos también para los distintas clases identificadas.

Hay que aclarar que para los objetos del tipo *menú de opciones* no se tendrán definidos atributos ya que como se observa en la figura 6.8 estos solo sirven para permitir a los actores usuarios del sistema el escoger una opción que después deberán canalizar a los objetos ventana, los cuales en su mayoría si tendrán atributos. Es decir, los objetos menú de opciones sólo se dedicarán a enviar mensajes a otros objetos. Quizás en el diseño si se contemplen algunos atributos para los objetos menú de opciones pero por el momento solo los métodos que invocarán las operaciones en los objetos ventana. (fig. 6.18)

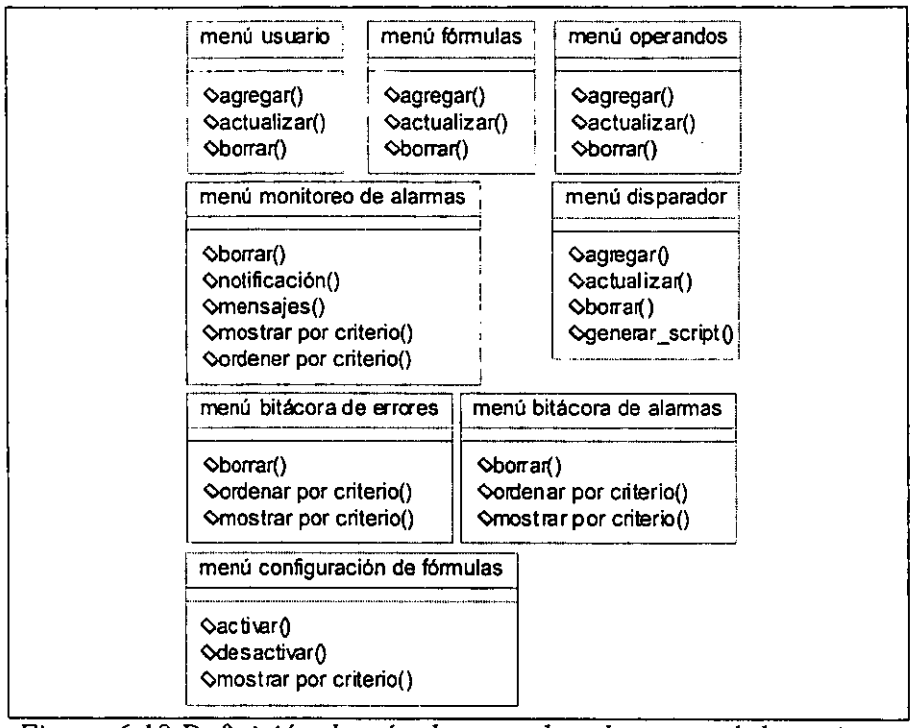


Figura 6.18 Definición de métodos para las clases menú de opciones

Los objetos ventana tendrán los mismos métodos que los objetos menús de opciones, sólo que estos objetos si tendrán definidos atributos. Los atributos de cada objeto serán parte de la información que se obtuvo en la descripción de los casos de uso. Solo se obtendrán los atributos que puedan corresponderse como características de los objetos hasta este memento definidos, si parte de la información aun no puede asignarse como parte de la estructura de algún objeto, posiblemente más adelante se asignará como atributo de otro objeto que en este momento todavía no se ha definido. Para el caso de los objetos ventana condición y mensaje no se tiene asociado un menú de opciones, pero las operaciones a realizar son las mismas que para las ventanas; agregar, actualizar, y borrar registros en la base de datos de control (fig. 6.19)

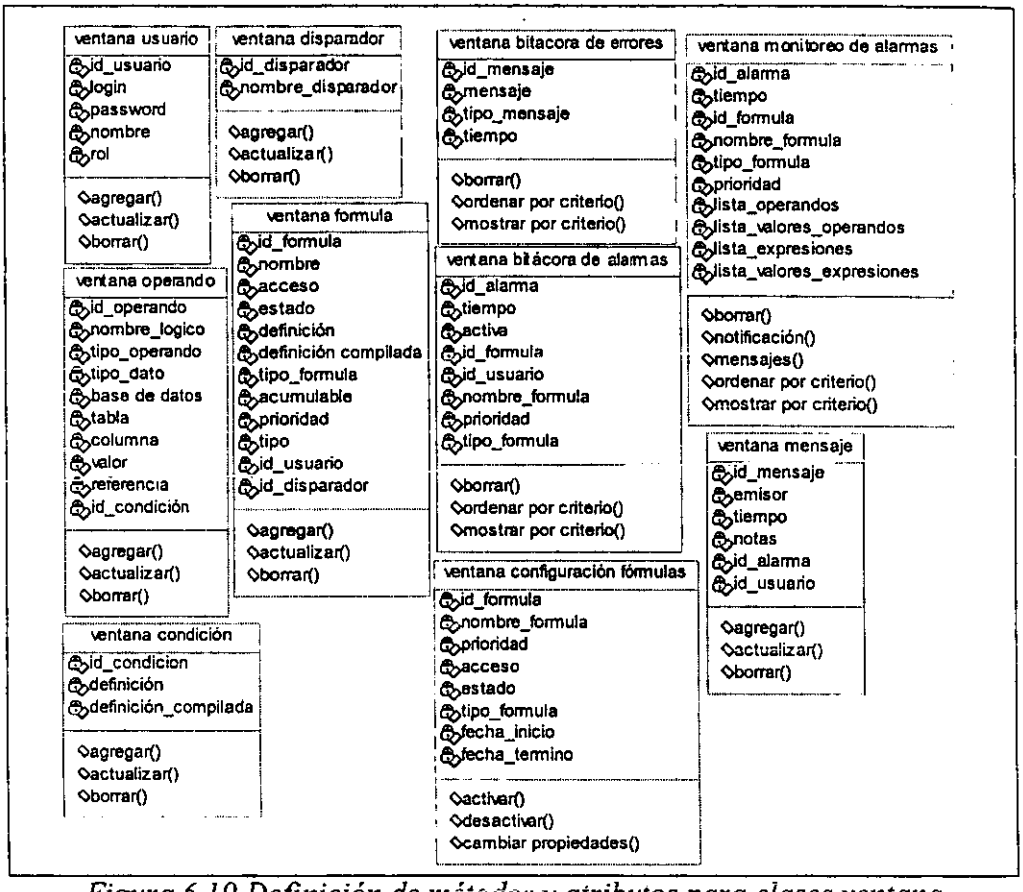


Figura 6.19 Definición de métodos y atributos para clases ventana

Finalmente tenemos que para el objeto *conexión SQL*, los atributos que se requieren son: el nombre del servidor, la dirección ip del mismo así como el número de puerto por donde atiende las peticiones, el nombre de la base de datos de control y el login y password validos para poder accesar a la base de datos de control. Hay que aclarar que estos atributos pueden variar en el modelo de implantación, por el momento se contemplan estos atributos que son los que generalmente se necesitan para establecer una conexión. Las operaciones a realizar son la establecer una conexión y terminarla (fig. 6.20).

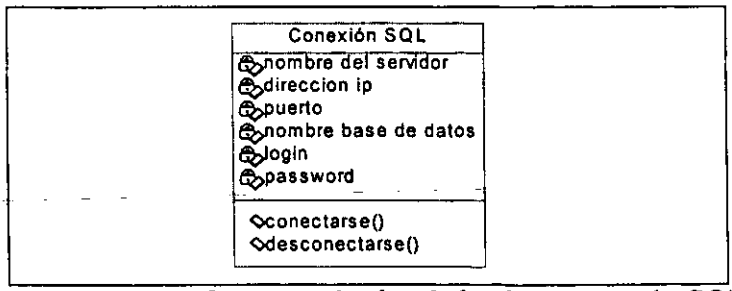


Figura 6.20 atributos y métodos de la clase conexión SQL

6.2.1.3 Identificación de relaciones

Las relaciones son vínculos que pueden llegar ha establecerse entre las distintas clases identificadas (ver apartado 1.4.4). El determinar relaciones entre clases puede tener como resultado el determinar nuevas clases genéricas o compuestas, según el concepto que sé este aplicando (ver apartados 1.4.2 y 1.4.3). Particularmente podemos aplicar los conceptos de abstracción y generalización para definir nuevas clases más generales que engloben el comportamiento similar que tengan las distintas clases hasta el momento definidas. Así se observa que para las clases ventanas se puede establecer una jerarquía de generalización (fig. 6.21)

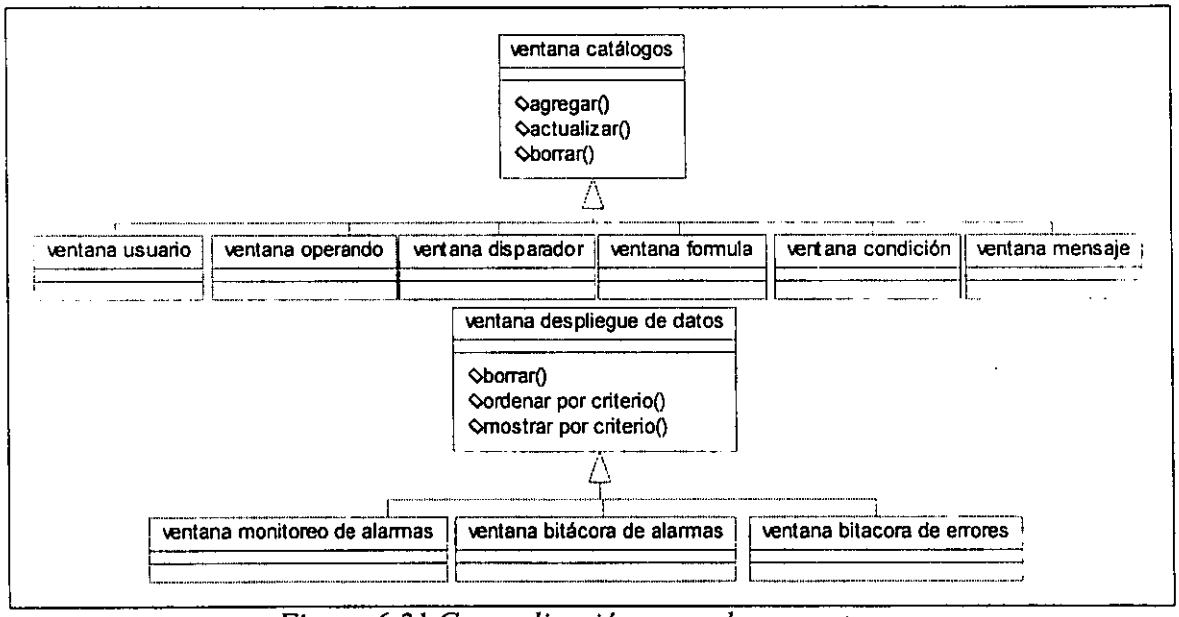


Figura 6 21 Generalización para clases ventanas

De forma similar se obtiene una jerarquía de clases para las clases menú de opciones. Existe otra funcionalidad que todavía no se ha definido en cuanto a que objetos van a contener esa información y las operaciones que van a realizar y estos son las listas de registros que van ha estar presentes en prácticamente todas las ventanas y que van a servir para representar la información en forma de lista de registros y desplegarla, de hecho en algunas ventanas se van a requerir de una o más listas que contengan diferente información, según lo expuesto en la descripción de los diagramas de casos de uso. De manera general en todas las ventanas por lo menos se necesita de una lista que contenga todos los registros de la información que va a manejarse en esa ventana, ya sea datos que

correspondan a una sola tabla en la base da datos de control, como es el caso de las ventanas para catálogos, o datos que correspondan a múltiples tablas en la base de datos de control como es el caso de las ventanas de despliegue de datos. Por lo tanto podríamos establecer una clase ventana más general que este formada por una lista, la cual dentro de sus atributos estará definido un arreglo que puede crecer de manera indefinida y que va a servir para contener y mostrar registros. (fig. 6.22)

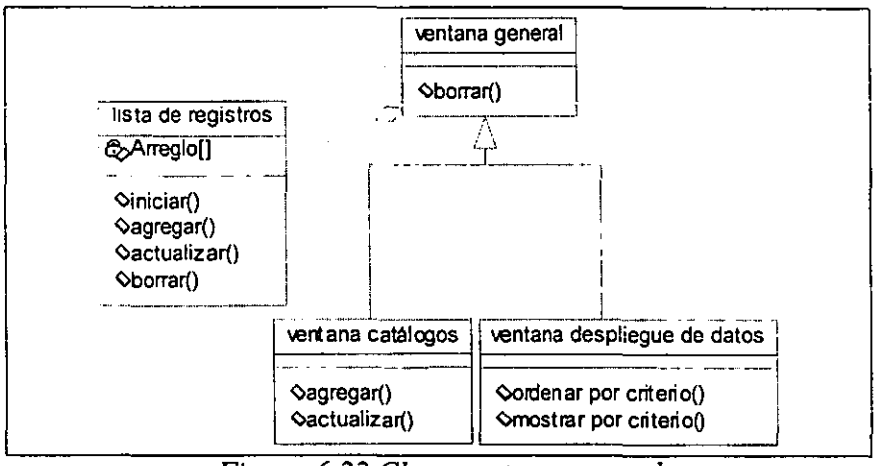


Figura 6.22 Clase ventana general

Se puede observar que la clase *lista de registros* tiene una relación *parte de* con la clase ventana general, es decir, estamos utilizando el concepto de composición (ver apartado 1.4.2) para definirla como parte de la clase ventana general. La clase listas de registros tiene definidos los métodos agregar, actualizar o borrar, estos son para realizar estas operaciones sobre el arreglo que se definió como parte de sus atributos y no sobre la base de datos de control como es el caso de las operaciones definidas para las ventanas, sin embargo, el arreglo si será afectado por los cambios que sufra algún registro en la base de datos de control por lo tanto se tendrá que actualizar, esto se verá con más detalle en el modelo dinámico. El método *iniciar()* es la operación que se realiza cuando se tiene que llenar por primera vez el arreglo con la distinta información que maneja cada ventana. Por último el método *borrar()* es el único que tiene en común las clases *ventana catálogos* y *ventana despliegue de datos*, por lo tanto será un método que hereden de la *clase ventana general.*

Algunas ventanas como ya se menciono necesitarán de estar compuestas de otras listas de registros además de la lista principal que cuenta con la información que va

manipular. En primer lugar tenemos a la ventana de disparadores, la cual requiere de dos objetos listas de operandos, uno que contenga una lista de todos los operandos que se puedan recuperar por un disparador, es decir una lista con todos los operandos del tipo *atributo del hecho* y otro que contenga la lista que muestre los operandos que ya estén asociados al disparador seleccionado en la lista principal o que se quieran asociar en caso de ser un disparador nuevo (fig. 6.23). La lista de operandos se llenará con todos los operandos del tipo atributo del hecho con el método *iniciar()*, por el contrario la lista operandos por recuperar será diferente para cada disparador, y los operandos que se anexen a ella serán los escogidos de la lista de operandos, de forma que un operando no puede estar en las dos listas al mismo tiempo o un operando o se agrega en la lista de operandos por recuperar o permanece en la lista de operandos, con los métodos actualizar lista se realiza este control en ambas listas. La lista de operandos por recuperar representa la tabla de la relación entre los operandos que son recuperados por un disparador en la base de datos de control, por lo tanto, cada vez que haga alguna modificación de algún disparador sé tendrá que actualizar también la tabla *operando_disparador* a través de los métodos *agregar()*, *actualizar()* y *borrar()*.

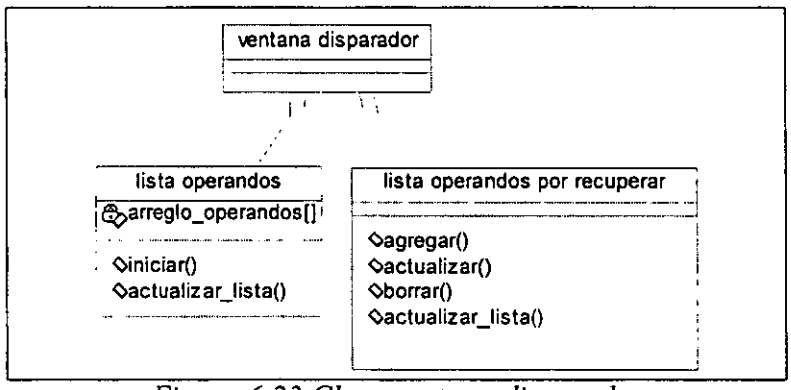


Figura 6.23 Clase ventana disparador

[4] "Be carefull with use cases" Edward V. Berand. The object Agency , Inc.
[5] "Use Cases the Pros and Cons" Donald G. Firesmith. Knoledge Systems Sorporation.

Conclusiones

El desarrollo de sistemas poco a poco se va volviendo una tarea más complicada, debido a la necesidad de integrar varias tecnologías diferentes entre sí cada vez más específicas. Por esto, se acentúa la importancia de usar una metodología que ayude a reducir la complejidad que se presenta en el desarrollo de grandes sistemas de cómputo.

En la presente tesis proponemos una metodología de análisis y diseño basada en el diseño orientado a objetos, pero que engloba herramientas de otros paradigmas como son la programación estructurada, los diagramas de flujo de datos, el análisis y diseño de bases de datos relacionales y el modelado de objetos, procesos y eventos.

Hoy en día no es posible hablar de metodologías aisladas que propongan una solución tomando como base únicamente un paradigma. En el desarrollo del *SMAD* se comprobó lo anterior, utilizando diversas herramientas y tecnologías como la programación orientada a objetos, la programación multihilos, la arquitectura *cliente-servidor*, el diseño de bases de datos relacionales, de gramáticas de contexto libre y autómatas, así como de conceptos de estructuras de datos.

Una consecuencia de lo anterior, es que en la actualidad no se pude hablar de sistemas de cómputo aislados que estén restringidos a una plataforma. Por el contrario, como responsables del diseño y desarrollo de un sistema, debemos tener una visión global, que nos permita abstraer la solución como un conjunto de componentes relacionados entre sí, donde cada uno de ellos esté basado en el paradigma y tecnología que mejor se adapte a sus funciones dentro del conjunto, es decir, del sistema.

En este proyecto, por ejemplo, se partió de un análisis de objetos o entidades para obtener una arquitectura global del proyecto, para esta primera aproximación, se eligieron los paradigmas de diseño, de programación, y de tecnología que más se adaptaron a cada uno de estos componentes. Así, para las interfaces de monitoreo y administración se utilizó el diseño y programación orientada a objetos; para el servidor monitor la tecnología multihilos; para los analizadores léxicos y sintácticos, la teoría de diseño de lenguajes formales y de autómatas; finalmente para la base de datos de control se utilizó la teoría de diseño de bases de datos relacionales; todo esto integrado bajo el modelo cliente-servidor.

Un aspecto importante de tener una visión global e integradora, es que permite a los ingenieros y científicos de informática, conocer, aprender, aplicar y desarrollar nuevas tecnologías y modelos. De otra manera, teniendo la visión de un sólo paradigma, cuyo alcance de aplicación y estudio es limitado, no sería posible lo anterior. De hecho, uno de los resultados importantes de esta tesis fue involucrar el uso de conceptos establecidos desde hace ya varios años, como la *teoría de autómatas* y *bases de datos relacionales*, con el uso de tecnología más recientes, como la *programación orientada a objetos* y algunas más nuevas como la *programación multihilos*.

En cuanto a la metodología orientada a objetos se llegó a los siguientes resultados o conclusiones:

- Una *metodología orientada a objetos* ayuda al diseño e inclusive para la implantación de sistemas que no necesariamente utilicen herramientas *orientadas a objetos* ya en el desarrollo. Sin embargo, sí sería una tarea difícil y aveces imposible; por el contrario podemos afirmar que en lo que respecta al *análisis orientado a objetos*, sí puede ser una herramienta útil para ayudarnos a realizar una visión global, debido a que la misma naturaleza que se establece en un *análisis de orientación a objetos* es la de enfrentar la complejidad del problema y elaborar un modelo en el cual se tengan una serie de objetos los cuales realicen ciertas funciones individualmente, pero que en conjunto persiguen uno o varios objetivos. Podemos decir que cualquier sistema en sus primeras etapas busca precisamente esto.
- A pesar de que han transcurrido casi tres décadas desde la primera vez que se utilizó el concepto *objeto* y que se ha venido estableciendo una muy fuerte teoría dentro del diseño y desarrollo de *software* orientado a objetos, aun no se puede afirmar que exista una metodología estándar definitiva que sea la que se tenga que aplicar. Esto es en parte a que existen discrepancias e irregularidades en cuanto a la notación a utilizar, respecto a cuestiones del comportamiento de los objetos dentro del modelo, es decir, si deben representarse mediante diagramas de estados, de procesos, tareas, eventos o de combinaciones de éstos.
- El hecho de estar estudiando un "nuevo" paradigma, como en su momento se consideró a la *orientación a objetos*, no implica olvidarse de otros; más aún, se puede decir que

distintos modelos de programación pueden ser utilizados como partes de este nuevo paradigma. Por ejemplo, el modelado de diseño de bases de datos relacionales es útil para establecer los componentes estáticos de los objetos, en un *diseño orientado a objetos*; el estudio de las máquinas de estado finito, en especial la implantación de diagramas de estados, puede ser una herramienta útil para poder modelar el comportamiento de objetos si lo vemos desde el punto de vista de su comportamiento en el tiempo.

Proponemos además el uso de la programación multihilos, con lo que podremos explotar de manera más eficiente las características que los sistemas operativos ofrecen, como la multitarea y el multiprocesamiento, haciendo uso de los recursos del sistema más eficientemente en tiempos de respuesta menores.

Entre las ventajas del uso del esquema multihilos podemos mencionar las siguientes:

- Se facilita el paralelismo con varios hilos corriendo en diferentes procesadores simultáneamente.
- La ejecución de los procesos es más rápida, pues cuando éstos hacen peticiones al sistema operativo, no tienen que esperar a que el servicio se complete, sino que permiten a otro hilo continuar. Esto permite a las aplicaciones estar siempre en actividad, recortando su tiempo de respuesta.
- En aplicaciones que mantienen varios procesos que tienen acceso a la misma área de memoria compartida, el control y sincronización son mucho más costosos tanto en tiempo como en espacio, que la aplicación análoga implementada con hilos. Esto facilita el diseño e implementación de aplicaciones concurrentes, pues en lugar de dividir sus tareas en varios procesos con espacios de memoria diferentes, se pueden dividir las mismas tareas en hilos pertenecientes a un mismo proceso, pero en espacios de direcciones compartidos.
- El uso de hilos, facilita el diseño de la aplicación con metodologías orientadas a objetos, pues en aplicaciones que utilizan objetos distribuidos, éstas son por naturaleza multihilos, debido a que cada vez que un objeto debe ejecutar alguna acción, ésta se ejecuta por separado. Además los programas tradicionales que ejecutan varias tareas, requieren de código complicado para garantizar su control, en cambio un programa

multihilos puede ejecutar las mismas tareas, pero con mucho mayor simplicidad de código.

Finalmente, debemos aclarar que todo lo que pueda hacerse con hilos puede hacerse sin ellos, utilizando procesos que compartan entre sí espacios de memoria. No obstante, debemos tomar en cuenta la importancia de las ventajas mencionadas, ya que entre más complicada sea la aplicación, mayor será la utilidad del uso de hilos.

Durante el desarrollo del proyecto se presentaron algunos problemas que arrojan ciertas conclusiones. Primero, debemos destacar que algunos de los errores se debían a la falta de conocimientos de la herramienta de *software*; esta falta de conocimiento se presentaba en cuestiones muy específicas y en realidad la mayoría de las veces se solucionaba buscando en manuales o en la ayuda de los programas, pero muchas otras veces, gracias a la experiencia de expertos en el manejo de las herramientas.

Algunas de las mejoras que se podrían lograr en algunos de los módulos son las siguientes:

- Integrar reportes que presentará la información de forma general, resumida y con el mayor número de gráficas posibles, de manera que pueda servir para tomar decisiones en las interfaces de usuario, debido a la gran cantidad de información que se estaría manejando en la base de datos de control.
- Que los reportes se puedan realizar de manera personalizada en tiempo de ejecución, lo que haría que el proyecto fuera más adaptable al proceso que se pretende monitorear.
- Establecer una interfaz gráfica para el *Notificador* donde se pueda observar todos los datos que recibe tanto de las interfaces como del *Servidor Monitor*, de esta manera si ocurre algún error, se sabrá a cual de las partes se debió.
- Realizar una aplicación capaz de obtener los datos de algún dispositivo periférico de la computadora y enviarlos al servidor de base de datos donde se encuentra la metabase. Este dispositivo sería muy útil sobre todo si se quisiera analizar algún proceso que requiera convertir los datos de alguna variable física a datos binarios, para posteriormente enviarlos al periférico de la computadora. En este caso prodría desarrollarse un módulo de adquisición de datos en tiempo real.

- Finalmente, la información generada por las alarmas podría servir como información para realizar diagnósticos y soluciones de problemas más específicos de manera automática. En este caso podría realizarse un análisis más profundo de la naturaleza y semántica de los datos monitoreados, para la posible adición de un sistema inteligente, quedando esta opción como un posible proyecto a futuro.

Apéndice A Funcionamiento del sistema

En la tabla A1-1 se especifican los requerimientos del *Servidor Monitor* que son válidos, si se considera 30 operandos por cada disparador que tenga fórmulas activas asociadas y que la media de los dígitos que componen el valor de un operando sean 50 (convertido en ASCII).

Módulo	Plataforma	Memoria *Ram*	Disco duro	Conexiones al servidor de *SQL*
Servidor Monitor	*UNIX*	5 MB de base + Numero de disparadores activos * (25Kb + 1.5Kb * (2 * Numero de fórmulas activas por disparador + 30))	2 MB	2
Administración	*Windows* (32 bits)	8 MB (Recomendable 16 MB)	20 MB	1
Configuración y Monitoreo	*Windows* (32 bits)	8 MB (Recomendable 16 MB)	20 MB	2 máx
Notificación	*Windows* (32 bits)	8 MB (Recomendable 16 MB)	1 MB	1

Triggers para la inserción de los hechos

Los *triggers* interesados en la generación de alarmas, tienen que actualizar las tablas *Hecho* y *Dato_hecho* de la siguiente manera:

1. Declaración de la variable *id_hecho* y *id_disparador*:

declare @id_disparador numeric(8,0)

declare @id_hecho numeric(8,0)

Inicio de la transacción (*begin transaction*).

2. Inserción en la tabla *Hecho* de *id_disparador* con el procedimiento almacenado *sp_inserta_hecho @id_disparador, @id_hecho output.*

El *id_disparador* será fijo por cada disparador que esté escribiendo hechos y deberá corresponder al identificador del mismo que fue proporcionado por el *Modulo de administración.*

Inserción en la tabla *Dato_hecho* de los valores asociados con el hecho.

El administrador deberá estar consciente del significado de cada dato y del *id_dato_hecho* asignado, donde *id_dato_hecho* será utilizado por el administrador para la definición de estos parámetros como operandos utilizables en la construcción de fórmulas.

Así, por ejemplo para un hecho con *N* operandos:

insert Dato_hecho values(@id_hecho,id_dato_hecho1)

insert Dato_hecho values(@id_hecho,id_dato_hecho2)

...

insert Dato_hecho values(@id_hecho,id_dato_hechoN)

3. Fin de la transacción (*commit transaction*).

   Para cada hecho, el *Servidor Monitor* tiene que realizar lo siguiente:

   1. Leer los nuevos hechos en base a *ptr_lectura* en la tabla *Control_lectura*.
   2. Evaluar las fórmulas ligadas al hecho por medio del *id_disparador*.
   3. En caso de resultado positivo (*TRUE*) generar una alarma.

   Por otra parte, la generación de una alarma se lleva a cabo en los siguientes cuatro pasos:

   1. Almacenamiento por el *Servidor Monitor* de la alarma en las tablas *Alarma, Dato_operando* y *Dato_expresion* de la base de datos *dbapc*.
   2. Inserción por el *Servidor Monitor* en la tabla *alarma_notifica* de los usuarios que se tienen que notificar para dicha alarma.
   3. Notificación del *Servidor Monitor* a todos los usuarios conectados correspondientes a la alarma.
   4. Visualización en el *Modulo de monitoreo* de la alarma a aquellos usuarios registrados (en la tabla *alarma_notifica* de la base de datos *dbapc*) para recibir la notificación de la misma.

Apéndice B Glosario de términos

abstracción Es definida como una actividad mental que permite conceptualizar un problema a un nivel de detalle necesario bajo el contexto que sé este manejando.

agregación Es la relación entre objetos donde un objeto esta compuesto de varios objetos.

alarma Es una señal que indica el resultado positivo de una formula aplicada a determinado hecho.

analisis léxico Es la primera fase de compilación. En esta se hace un análisis lineal, en el que la cadena de caracteres que constituyen el programa fuente se lee de izquierda a derecha y se agrupa en componentes léxicos.

Análisis orientado a objetos Método encaminado a encontrar e identificar las clases y objetos que se encuentran inmersos dentro del contexto de nuestro problema.

analisis sintáctico Es la segunda fase de compilación. En esta se hace un análisis jerárquico, en el que los caracteres o los componentes léxicos se agrupan jerárquicamente en colecciones anidadas con un significado colectivo.

analizador léxico Es la primera fase de un compilador.

analizador sintáctico Es la segunda fase de un compilador.

área del *kernel* Es el área de memoria que el *kernel* utiliza para sí mismo. Los programas de usuario no pueden accesar esta área.

área de usuario Es el área de memoria dedicadas a los programas del usuario. El *kernel* establece este espacio pero generalmente nunca lo bloquea.

arquitectura Se refiere a la manera en la cual el sistema es dividido en subsistema y como estos subsistemas iteractuan uno con otro.

atributo Es una propiedad de una clase para la cual un objeto o instancia de esa clase contendría un valor. Notar que esta definición es análoga al definir el campo de una tabla.

barriers Un *barrier* permite sincronizar a un grupo de hilos para que se detengan en cierto punto de su ejecución y esperen a que los demás hilos lleguen también a ese punto.

Base de datos relacional Base de Datos que almacena y accesa los datos basándose en el esquema relacional, en el cual se tiene tablas que representan conjuntos y atributos los elementos de esos conjuntos.

Base de datos Orientada o Objetos Es aquella base de datos que almacena los datos en forma de encapsulamiento de objetos y la Interfase que nos proporciona el acceso a los datos es una aplicación orientada a objetos.

callbacks Son rutinas que se ejecutan internamente en el *Open Server* al recibir la llamada a un *RPC*.

cambio de contexto Es el procedimiento de mover un proceso fuera del *CPU* y colocar otro.

clase Implantación de un tipo abstracto de dato que soporta un conjunto de propiedades designado. Una clase describe a un grupo de objetos con propiedades similares Note que la definición de una clase provee los medios de programar los métodos de un objeto. El objeto solo existe en tiempo de ejecución.

clase abstracta Es una clase que tiene solo subclases pero no instancias. Las subclases derivadas de las clases abstractas si pueden tener instancias.

cliente Es un ente (proceso o *thread*) que se está ejecutando y que realiza peticiones de recursos o servicios e interpreta los resultados obtenidos.

cola de mensajes Las colas de mensajes son mecanismos de comunicación entre procesos (o hilos) que posibilitan el intercambio de datos con formato determinado.

componentes léxicos Son secuencias de caracteres que tienen un significado colectivo.

comportamiento Como actúa y reacciona un objeto, desde el punto de vista de sus cambios de estado y por invocación de mensajes.

comunicación interprocesos Véase ***IPC***.

descomposición El proceso de dividir un sistema en partes, se puede ver desde una perspectiva de procesos o tareas donde cada una representa una pequeña parte de una procesomas grande, o desde una perspectiva de objetos, en donde se tiene un conjunto de objetos que cooperan para realizar un fin común.

Diseño Orientado a Objetos Metodología que permite la definición, clasificación y organización de objetos.

disparador Representa un agente externo, el cual se ocupa de insertar en el sistema los hechos. Es en la actividad real una operación

encapsulamiento Separar de la intromisión externa la implantación interna (atributos y métodos) de un objeto.

espacio del kernel Véase **área del *kernel***.

espacio de usuario Véase **área de usuario**.

esquema externo Define la manera en la cual la organización de los datos es presentada al usuario.

esquema interno Describe las estructuras internas que sirven para el acceso obtención y manejo de los datos. El esquema interno comúnmente se refiere al esquema de la base de datos.

estructura de un proceso Es una estructura a nivel *kernel* que describe todos los aspectos relevantes de un proceso.

fórmula Forma establecida para expresar alguna cosa o modo convenido para ejecutarla o resolverla.

fórmula base Es aquella que es definida por el administrador, que identifica un cálculo de tipo general y que puede ser utilizada por los usuarios para la definición de fórmulas personalizadas.

fórmula personalizada Es aquella que es definida por los usuarios del sistema utilizando como variables los operandos y las fórmulas base.

generalización Es la relación entre una clase y una y más versiones redefinidas de esa clase.

handlers Son rutinas que se ejecutan en el proceso al presentarse algún evento o señal.

hecho Un hecho representa al conjunto de datos a monitorear e identifican las características y valores de cierta operación

herencia Relación que permite a una clase ser definida con parte o en su totalidad con la estructura y el comportamiento de otra clase (herencia simple) u otras clases (herencia múltiple).

hilo Es el código que se coloca junto a una llamada al sistema para la verificación de su seguridad, de manera que se ejecute en forma no bloqueante.

identidad Propiedad que permite o un objeto u identidad ser destingible de todos los demás.

instancia Véase **Objeto**.

IPC Es la idea de que diferentes procesos puedan comunicarse a traés de diferentes medios.

jacket Es el código que se coloca junto a una llamada al sistema para la verificación de su seguridad, de manera que se ejecute en forma no bloqueante.

kernel El *kernel* o núcleo del sistema es un programa que siempre está resitente en memoria y es el encargado de controlar los recursos del *hardware*, controlar los dispositivos periféricos, administrar archivos, ejecutar programas y controlar a los usuarios.

llamada al sistema Es una rutina que sirve como interfaz entre el sistema operativo y los programas del usuario. Las llamadas al sistema crean, eliminan y utilizan varios objetos del software, controlados por el sistema operativo, como son los procesos y los archivos.

llamada al sistema bloqueante Es una llamada al sistema que bloquea al proceso en el kernel mientras espera a que algo ocurra. Típicamente son llamadas de lectura/escritura de datos.

llamada a un procedimiento almacenado (*Remote Procedure Call RPC*). Es una llamada a un procedimiento almacenado que se encuentra en otro nodo de la red.

librería Es una colección de rutinas que muchos programas pueden utilizar. Estas rutinas están agrupadas en un mismo archivo conocido como librería.

LWP *Light Weight Process*. Entidad a nivel kernel que debe ser planificada.

máscara de señales Es una máscara lógica que le indica al kernel (o librería de hilos) cuales siñales serán aceptadas y cuales deberán ser colocadas en una cola de espera.

memoria compartida Es la memoria que es compartida por más de un proceso. Cuando cualquier proceso realice un cambio, todos los demás procesos verán el cambio.

mensaje Operación que un objeto invoca o manda que se realice sobre otro objeto.

metabase Es una base de datos que permite la definición y control de otras bases de datos o de la especificación de un conjunto de datos.

método Operación sobre un objeto definida como parte de la declaración de una clase.

modelo Es la presentación de un diseño que representa la solución a algún problema.

modelo cliente-servidor Es el modelo estándar de ejecución de aplicaciones en una red.

modelo conceptual de datos Especifica las reglas según las cuales se estructuran los datos, y también las operaciones asociadas permitidas. Incluye los esquemas: interno, externo y conceptual.

modelo de objetos Es una representación que utiliza los conceptos del diseño orientado a objetos (clases, encapsulamiento, modulariadad, agregación. etc.), para proporcionar una solución a algún problema. En el modelo de objetos se definen de manera detallada los componentes estructurales y de comportamiento cada objeto.

modulariadad Propiedad de un sistema que ha sido descompuesto un conjunto de módulos.

multihilos Es un paradigma de programación in el cual un solo proceso puede tener mas de un camino de ejecución.

mutex Un *mutex* (*Mutual Exclusive Locks*) es la variable primitiva más simple para sincronización y protección de secciones criticas.

objeto Es un elemento, unidad o entidad individual, ya sea real o abstracto, identificable, que tiene una estado y comportamiento en el tiempo y con un papel bien definido en el dominio del problema.

operación Véase **método**.

operando Es una variable que tiene asociada un nombre lógico y que puede ser utilizada como parámetro en la definición de fórmulas.

paquete de hilos El es conjunto de librerías que implementan a nivel usuario el uso de *threads*.

parser Véase **analizador sintáctico**.

planificación Es el algoritmo (algoritmo de planificación) utilizado por el sistema operativo (o libreria de hilos) para decidir que proceso (hilo) debe ejecutarse.

planificador Es la parte del sistema operativo (o librería de hilos) que debe llevar a cabo el algotitmo de planificación.

polimorfismo Es la propiedad que permite que una operación, pueda interactuar de manera diferente con objetos de diferentes clases, en otras palabras, objetos de diferentes clases pueden responder en su muy particular forma a un mensaje común.

POSIX *Portable Operating System Interface*. El el estándar acordado por la *IEEE* para la creación de *APIs* que sean portables a todos los sistemas *UNIX*. Existe un apartado en *POSIX* concerniente a la creración de programas multihilos.

preemption Es el acto de forzar a un thread a detener su ejecución.

procedimiento almacenado Es un código procedural que reside en el servidor reduciendo trafico en la red, utilizando comandos *SQL* para accesar a las bases de datos. Son importantes elementos en una aplicación Cliente/Servidor ya que proporciona performance e integridad de datos

procedimiento registrado Véase ***RPC***.

proceso Es un programa en ejecución y todo su estado asociado a ello.

proceso ligero Véase **Hilo**.

Programación Orientada a Objetos Implantación de programas que se organizan como colecciones cooperativas de objetos, los cuales representan una instancia de algún tipo, y cuyos tipos son miembros de una jerarquía de tipos unidos mediante relaciones que son la herencia.

program counter Es el registro en el *CPU* que define cual instrucción será ejecutada.

privada Declaración que se hace sobre atributos u operaciones y que indica que solo pueden ser accesados por métodos de esa clase.

relación Asociación que representa una conexión semántica entre dos objetos, entidades o módulos.

RPC Son procedimientos que estan contruidos como rutinas internas del *Open Server*.

sección crítica Una sección crítica es un fragmento de código que debe ejecutarse de manera atómica y sin ninguna interrupción que pueda afectar el éxito de su ejecución o la consistencia de los datos involucrados.

semáforo Un semáforo es un mecanismo de sincronización consistente en una variable que puede incrementarse progresivamente, pero decrementarse siempre a cero.

señal Son interrupciones de software que pueden ser enviadas a un proceso para informarle de algún evento asíncrono o situación especial.

servidor Es un ente (proceso o thread) que se está ejecutando y que recibe peticiones de recursos o servicios a ser atendidas y que genera resultados.

sincronización Es el procedimiento que lleva a cabo el control de la ejecución de procesos (o hilos) que con memeoria compartida.

***Spec* 1170** Es la especificación de un *API* para todos los sistemas *UNIX*, adoptado por la mayoría de los fabricantes. Constituye el conjunto de rutinas de programasción soportadas por casi todos los sistemas.

spin Es un tipo de mecanismo de sincronización que se utilizan cuando se debe hacer un bloqueo por el menor tiempo posible, para permitir a otros hilos correr sin que se bloqueen.

stack Es un área de memoria usada para almacenar resultados intermedios y direcciones de retorno de funciones mientras estas se encuantran llamando a otras funciones.

stack pointer Es el registro en el *CPU* que le dice al *CPU* cual es la siguiente localidad de memoria libre en el *stack*.

thread Véase **Hilo**.

tipo de dato abstracto Definición de una estructura de datos que describe solo el servicio que ofrece hacia el exterior. Este nivel de abstracción permite el reutilizar la estructura de datos en todos los casos que requieren exactamente el mismo conjunto de servicios.

trigger Son un tipo de procedimientos almacenados en el servidor que se ejecutan automáticamente cuando ocurre un evento predefinido (altas, bajas y actualizaciones de datos.

variables de clase Parte del estado de una clase, las variables de clase constituyen la estructura de una clase.

variable de condición Una variable de condición es un mecanismo de sincronización que permite establecer condiciones que sean verificables por los hilos.

variables de instancia Valor o deposito que forma parte del estado de un objeto, colectivamente las variables de instancia de un objeto constituyen la estructura de ese objeto.

yielding Es el acto de suspender temporalmente la ejecución de un thread de manera voluntaria.

Apéndice C Diccionario de datos de los diagramas de flujo de datos del *SMAD*

Tareas de Administración

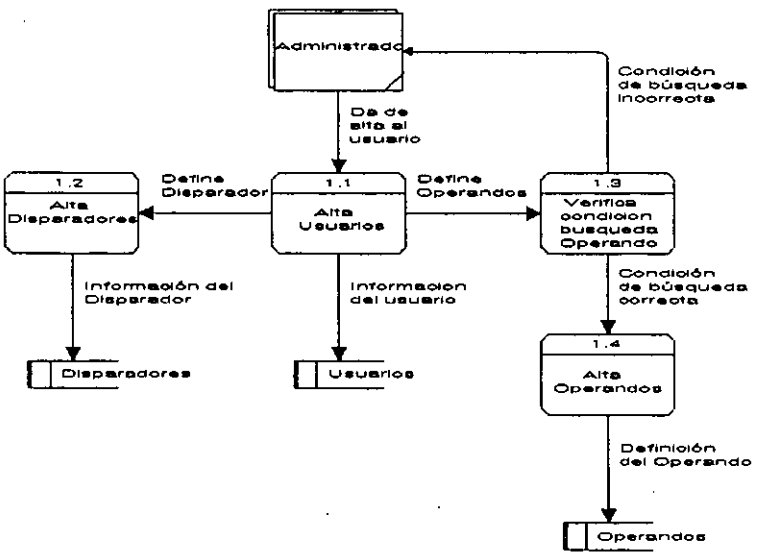


Figura 1 *Tareas de Administración*

ETIQUETA	IDENTIFICADOR	TIPO	DESCRIPCION
Alta Usuarios	1.1	Proceso	Procedimiento que da de alta al usuario en el sistema.
Alta Disparadores	1.2	Proceso	Procedimiento que registra los nuevos disparadores (agentes externos).
Verifica Condición Búsqueda Operando	1.3	Proceso	Procedimiento que analiza léxica y sintácticamente la condición de búsqueda del operando, si es que se definió como columna.
Alta Operandos	1.4	Proceso	Procedimiento que da de alta la definición de los operandos.
Administrador		Entidad	Usuario con el rol de administrador del sistema.
Disparadores		Almacenamiento	Registro de los disparadores (agentes externos) definidos.
Operandos		Almacenamiento	Registro de los operandos definidos.
Usuarios		Almacenamiento	Registro de los usuarios del sistema.

Tabla 1 *Elementos del DFD para la tareas de Administración*

Definición y manipulación de Fórmulas

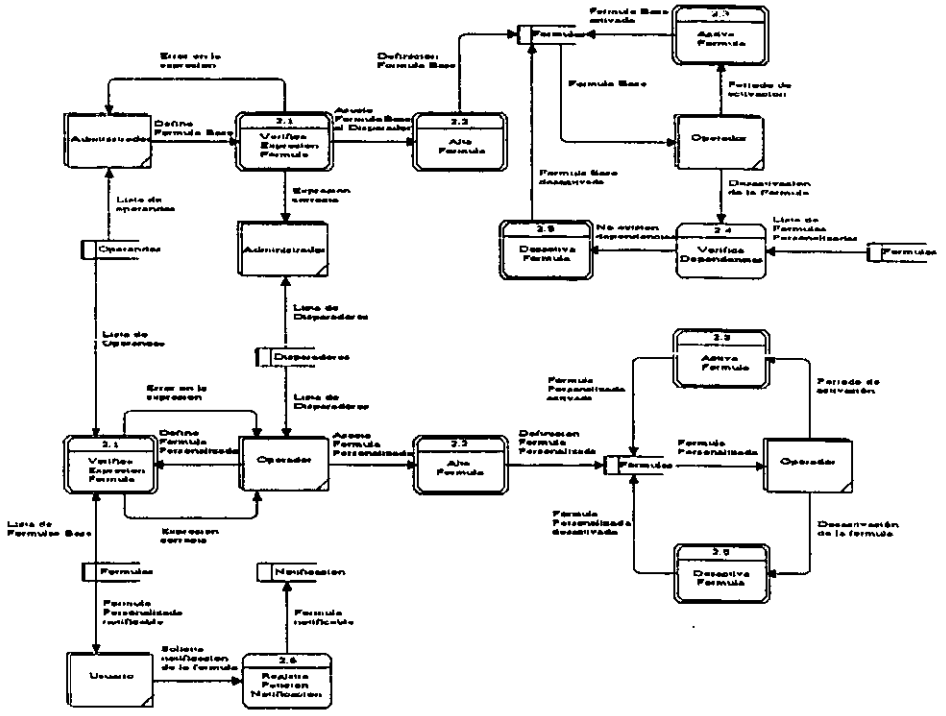


Figura 2 *Definición y manipulación de Fórmulas*

ETIQUETA	IDENTIFICADOR	TIPO	DESCRIPCION
Verifica Expresión Formula	2.1	Proceso	Procedimiento en el cual se verifica léxica y sintácticamente la expresión de la formula.
Alta Formula	2.2	Proceso	Procedimiento que registra una formula en la base de datos de control.
Activa Formula	2.3	Proceso	Procedimiento que lleva a cabo la activación de una formula para que sea notificable.
Verifica Dependencias	2.4	Proceso	Procedimiento en el que se verifica que la formula a borrar no sea operando de otra formula que este activa.
Desactiva Formula	2.5	Proceso	Procedimiento que lleva a cabo la desactivación de las fórmulas, para que no puedan ser notificables.
Registra Petición Notificación	2.6	Proceso	Procedimiento que registra la petición del usuario para que le sean notificadas las alarmas que la formula origine.
Administrador		Entidad	Usuario con el rol de administrador del sistema.
Operador		Entidad	Usuario con rol de operador.
Usuario		Entidad	Usuario ordinario sin rol de administrador u operador.
Disparadores		Almacenamiento	Registro de los disparadores (agentes externos) definidos.
Fórmulas		Almacenamiento	Registro de todas las fórmulas definidas.
Operandos		Almacenamiento	Registro de los operandos definidos.

Tabla 2 *Elementos del DFD para la definición y manipulación de Fórmulas*

Generación y notificación de Alarmas

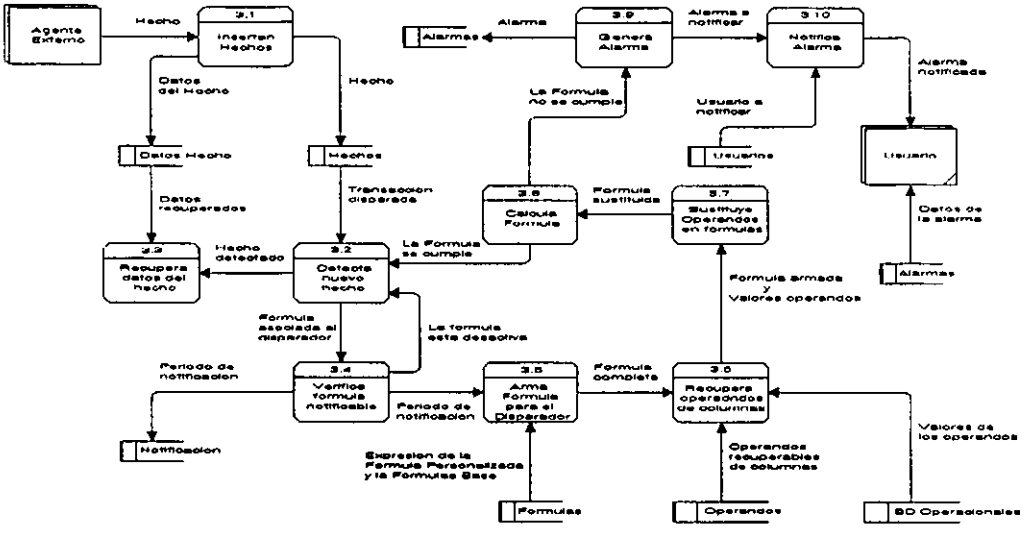


Figura 3 *Generación y notificación de Alarmas*

Etiqueta	Identificador	Tipo	Descripcion
Insertan Hechos	3.1	Proceso	Procedimiento que lleva a cabo la inserción de los hechos en la base de datos de control. Este procedimiento lo origina el Agente Externo.
Detecta Nuevo Hecho	3.2	Proceso	Procedimiento que esta poleando hasta que detecta la llegada de un nuevo hecho.
Recupera Datos del Hecho	3.3	Proceso	Procedimiento que se encarga de recuperar de la base de datos de control los datos del hecho.
Verifica Formula Notificable	3.4	Proceso	Procedimiento en el que se verifica que la formula deba notificarse.
Arma Formula para el Disparador	3.5	Proceso	Procedimiento en el que se construye la Formula asociada al Disparador en base a su definición.
Recupera Operandos de Columnas	3.6	Proceso	Procedimiento que lleva a cabo la recuperación de los operandos definidos como columnas. El procedimiento utiliza la condición de búsqueda definida para el operando.
Sustituye Operandos en Fórmulas	3.7	Proceso	Procedimiento se encarga de sustituir cada uno de los valores en los operandos de la formula.
Calcula Formula	3.8	Proceso	Procedimiento que lleva a cabo el calculo de la formula con los valores sustituidos.
Genera Alarma	3.9	Proceso	Procedimiento que se encarga de generar la alarma a notificarse.
Notifica Alarma	3.10	Proceso	Procedimiento que lleva a cabo la notificación asíncrona al usuario de la ocurrencia de una alarma.
Agente Externo		Entidad	Agente externo (disparador) que inserta los hechos en la base de datos de control.
Usuario		Entidad	Usuario ordinario sin rol de administrador u operador.
BD Operacionales		Almacenamiento	Son las bases de datos donde se almacena la información involucrada en las operaciones de los procesos.
Alarmas		Almacenamiento	Registro de las alarmas generadas que deben informarse a los usuarios.
Datos Hecho		Almacenamiento	Registro de los datos que acompañan a los hechos.
Fórmulas		Almacenamiento	Registro de todas las fórmulas definidas.
Hechos		Almacenamiento	Registro de los hechos insertados que deben ser leídos.
Notificación		Almacenamiento	Registro de las peticiones de los usuarios de notificación de las fórmulas.
Operandos		Almacenamiento	Registro de los operandos definidos.
Usuarios		Almacenamiento	Registro de los usuarios del sistema.

Tabla 3 *Elementos del DFD para la generación y notificación de Alarmas*

Apéndice D Diccionario de datos del modelo conceptual de la base de datos *DBMAD*

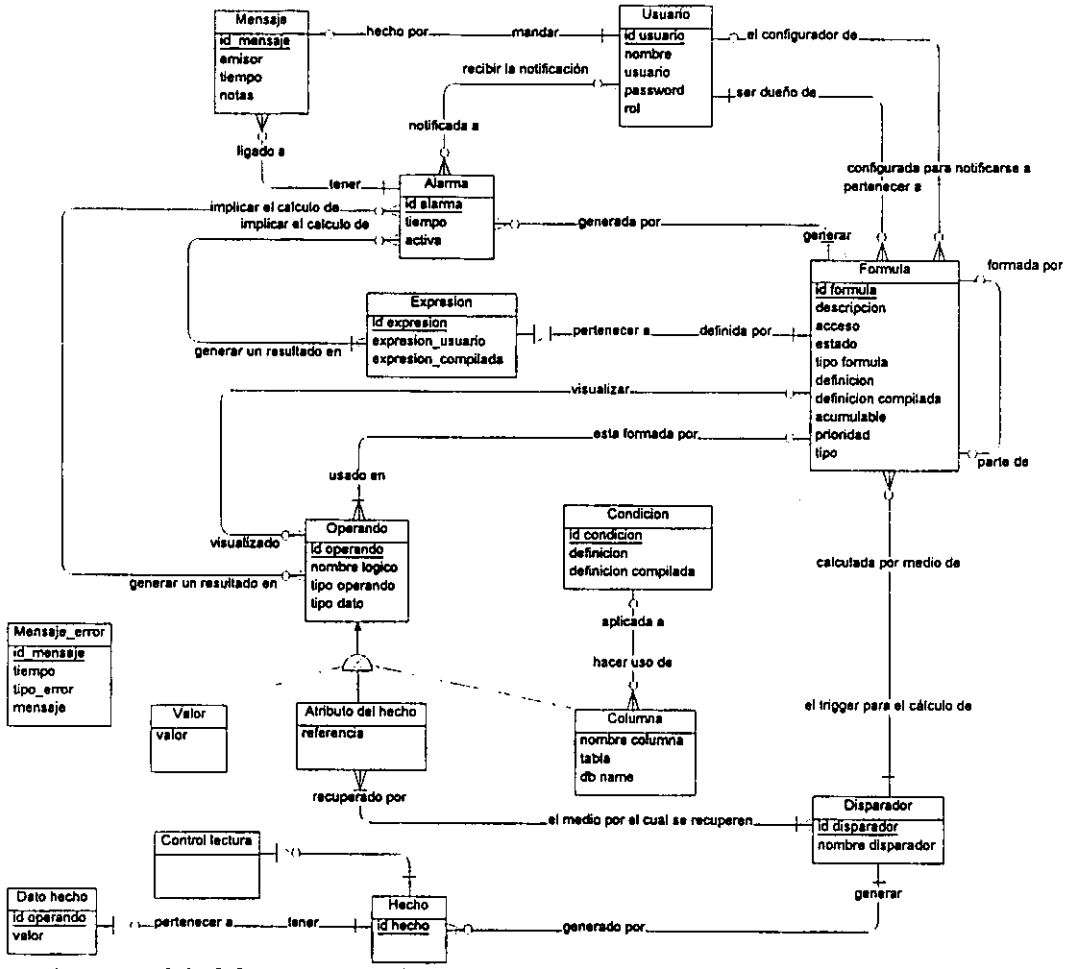


Figura 1 *Modelo conceptual de la Base de Datos DBMAD*

Lista de Entidades

NOMBRE	CÓDIGO
Alarma	alarma
Atributo del hecho	atributo_hecho
Columna	columna
Condicion	condicion
Control lectura	control_lectura
Dato hecho	dato_hecho
Disparador	disparador
Expresion	expresion
Expresion	expresion
Formula	formula
Formula	formula
Hecho	hecho
Hecho	hecho
Mensaje	mensaje
Mensaje	mensaje

NOMBRE	CÓDIGO
Mensaje_error	mensaje_error
Operando	operando
Usuario	usuario
Valor	valor

• Alarma

Descripción
Contine el registro de las alarmas ocurridas.

Lista de Atributos

NOMBRE	CÓDIGO	TIPO	I	M	DESCRIPCIÓN
id alarma	id_alarma	NO6	Si	Si	Identificador de la alarma.
tiempo	tiempo	DT	No	Si	Fecha y hora en que ocurrió la alarma.
activa	activa	BL	No	Si	Identifica si la alarma de tipo acumulable esta todavia activa.

• Atributo del hecho

Descripción
Identifica el valor de un operando que se recupera de un hecho.

Lista de Atributos

NOMBRE	CÓDIGO	TIPO	I	M	DESCRIPCIÓN
referencia	referencia	VA50	No	Si	Es el nombre de la columna a la que hace referencia el operando.

• Columna

Descripción
Columnas que se tendran que recuperar desde las base de datos indicadas utilizando las condiciones registradas en la formula asociada.

Lista de Atributos

NOMBRE	CÓDIGO	TIPO	I	M	DESCRIPCIÓN
nombre columna	nombre_columna	VA50	No	Si	Nombre de la columna (funciones agregadas del Servidor de SQL).
tabla	tabla	VA30	No	Si	Nombre de la tabla a que pertenece la columna.
db Nombre	db_Nombre	VA30	No	Si	Nombre de la base de datos donde resiede la columna.

• Condicion

Lista de Atributos

NOMBRE	CÓDIGO	TIPO	I	M	DESCRIPCIÓN
id condicion	id_condicion	NO6	Si	Si	Identificador de la condición.
definicion	definicion	VA255	No	Si	Definición escrita por el usuario.
definicion	definicion	VA255	No	Si	Definición escrita por el usuario.
definicion compilada	definicion_compilada	VA255	No	Si	Definición escrita por el parser.
definicion compilada	definicion_compilada	VA255	No	Si	Definición escrita por el parser.

• Control lectura

Descripción
Tabla de control para la lectura de hechos nuevos.

• Dato hecho

Descripción
Contiene los datos referentes al hecho.

Lista de Atributos

NOMBRE	CÓDIGO	TIPO	I	M	DESCRIPCIÓN
id operando	id_operando	N6	Si	Si	Identificador del dato.
valor	valor	VA100	No	Si	Es el valor convertido en ASCII.

• Disparador

Descripción
Identifica los triggers que pueden registrar transacciones.

Lista de Atributos

NOMBRE	CÓDIGO	TIPO	I	M	DESCRIPCIÓN
id disparador	id_disparador	NO6	Si	Si	Identificador del trigger.
nombre disparador	nombre_disparador	VA100	No	Si	Nombre del trigger.

• Expresion

Descripción
Expresiones utilizadas para la evaluacion de formulas.

Lista de Atributos

NOMBRE	CÓDIGO	TIPO	I	M	DESCRIPCIÓN
id expresion	id_expresion	I	Si	Si	Identificador de la expresion.
expresion_usuario	expresion_usuario	VA255	No	Si	Mantiene la expresion escrita por el usuario.
expresion_compilada	expresion_compilada	VA255	No	Si	Mantiene la expresion compilada por el parser.

• Formula

Descripción
Identifica las formulas necesarias para el funcionamiento del sistema:
Formulas base
Formulas personalizadas
Clausulas where para la localizacion de columnas.

Lista de Atributos

Nombre	Código	Tipo	I	M	Descripción
id formula	id_formula	NO6	Si	Si	Identificador de la formula.
descripcion	descripcion	VA100	No	Si	Nombre asignado por el usuario para identificar la formula.
descripcion	descripcion	VA100	No	Si	Nombre asignado por el usuario para identificar la formula.
acceso	acceso	VA1	No	Si	Describe el tipo de acceso que tienen los usuarios no dueños de la formula: Publico : Baja, Modificación, Utilizo Reservado : ninguno Accesible : Utilizo
estado	estado	BL	No	Si	Identifica si es una formula disponible.
estado	estado	BL	No	Si	Identifica si es una formula disponible.

Nombre	Código	Tipo	I	M	Descripción
tipo formula	tipo_formula	A1	No	Si	Identifica el tipo de formula (Clausula where, formula base, formula personalizada).
definicion	definicion	VA255	No	Si	Definicion de la formula escrita por el usuario.
definicion compilada	definicion_compilada	VA255	No	Si	Definicion compilada por el parser.
acumulable	acumulable	BL	No	Si	Identifica si la alarma generada por la formula es acumulable en el tiempo.
prioridad	prioridad	SI	No	Si	Identifica en nivel de prioridad para la presentacion de alarmas (1- 5). No es utilizado como criterio de evalucion.
tipo	tipo	A1	No	Si	

• Hecho

Descripción
Transacciones que se tienen que procesar.

Lista de Atributos

NOMBRE	CÓDIGO	TIPO	I	M	DESCRIPCIÓN
id hecho	id_hecho	NO8	Si	Si	Identifica la transaccion.

• Mensaje

Descripción
Mantiene los mensajes asociados a alguna alarma que un usuario envia a otros desde el modulo de monitoreo.

Lista de Atributos

Nombre	Código	Tipo	I	M	Descripción
id_mensaje	id_mensaje	NO8	Si	Si	Identificador del mensaje.
emisor	emisor	N6	No	Si	Identificador del usuario que recibe el mensaje.
tiempo	tiempo	DT	No	Si	Fecha y hora en que se envia el mensaje.
notas	notas	VA255	No	Si	Mensaje enviado.

• Mensaje_error

Descripción
Mantiene los mensajes de error producidos durante el procedimiento de monitoreo de datos.

Lista de Atributos

Nombre	Código	Tipo	I	M	Descripción
id_mensaje	id_mensaje	NO8	Si	Si	Identificador del mensaje.
id_mensaje	id_mensaje	NO8	Si	Si	Identificador del mensaje.
tiempo	tiempo	DT	No	Si	Fecha y hora en que ocurrió el error.
tiempo	tiempo	DT	No	Si	Fecha y hora en que ocurrió el error.
tipo_error	tipo_error	I	No	Si	Tipo de error producido.
tipo_error	tipo_error	I	No	Si	Tipo de error producido.
mensaje	mensaje	VA255	No	Si	Mensaje del error producido.
mensaje	mensaje	VA255	No	Si	Mensaje del error producido.

• Operando

Descripción
Identifica los operandos utilizados en las formulas.

Lista de Atributos

Nombre	Código	Tipo	I	M	Descripción
id operando	id_operando	NO6	Si	Si	Identificador del dato.
nombre logico	nombre_logico	VA100	No	Si	Nombre asignado por el administrador para referenciar el operando en las formulas.
tipo operando	tipo_operando	A1	No	Si	Identifica el tipo del operando (valor fijo, campo, parametro del hecho).
tipo dato	tipo_dato	A1	No	Si	Identifica el tipo de dato del operando (real, entero, character, boolean, fechas).

• Usuario

Descripción
Contiene los usuarios que tienen acceso al sistema.

Lista de Atributos

Nombre	Código	Tipo	I	M	Descripción
id usuario	id_usuario	NO6	Si	Si	Identificador del usuario.
nombre	nombre	VA50	No	Si	Nombre del usuario.
usuario	usuario	VA20	No	Si	Login del usuario.
password	password	VA20	No	Si	Password del usuario.
rol	rol	A1	No	Si	Identifica el rol del usuario: Administrador: acceso total Operador: modulo de configuración personalizada alta formulas - acceso permitido modificación formulas - acceso permitido solo si es dueño o si la formula es publica baja formulas - acceso permitido solo si es dueño o si la formula es publica modulo de notificación formulas publicas y accesibles - acceso permitido formulas reservadas - acceso permitido al dueño Usuario : modulo de configuración personalizada no tiene acceso modulo de notificación formulas publicas y accesibles - acceso permitido formulas reservadas - acceso permitido al dueño

• Valor

Descripción
Identifica un operando con valor fijo.

Lista de Atributos

NOMBRE	CÓDIGO	TIPO	I	M	DESCRIPCIÓN
valor	valor	VA100	No	Si	Es el valor convertido en ASCII.
valor	valor	VA100	No	Si	Es el valor convertido en ASCII.

Apéndice E Diccionario de datos del esquema físico de la base de datos *DBMAD*

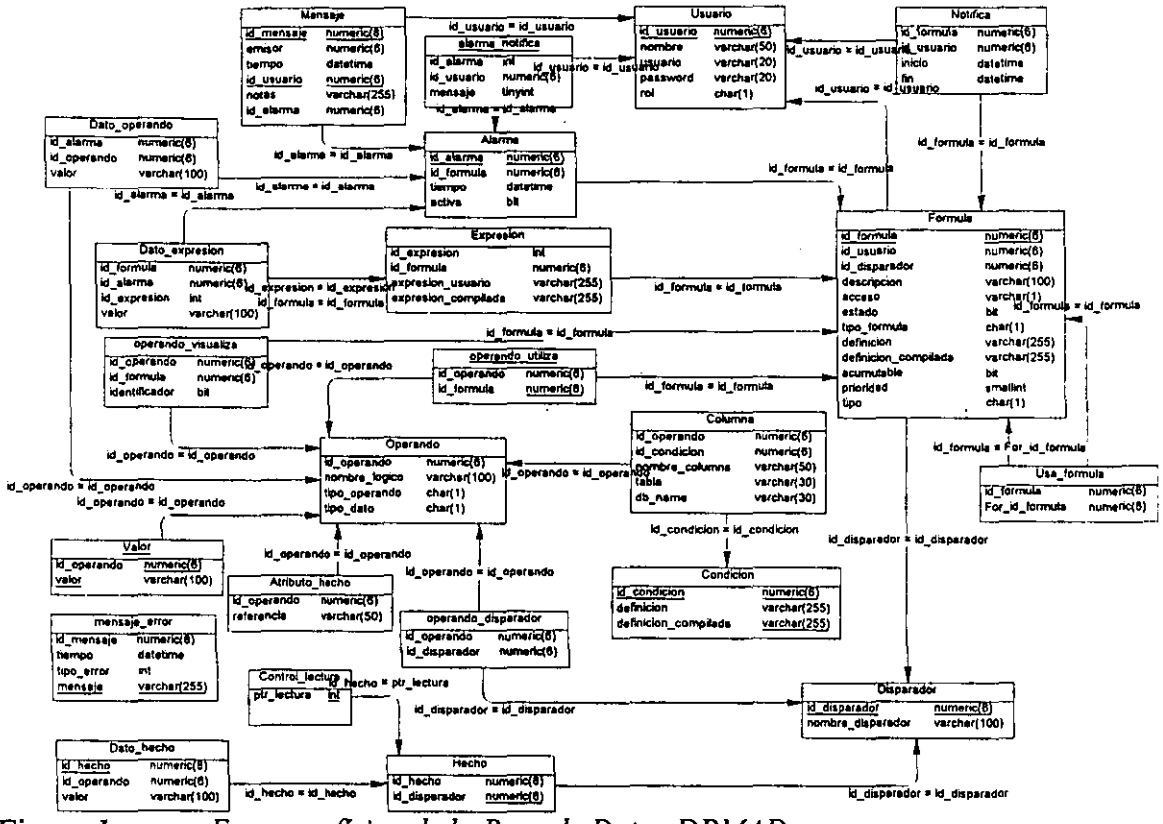


Figura 1 *Esquema físico de la Base de Datos DBMAD*

- **Alarma**

Descripción

Identifica las alarmas ocurridas.

Columnas

Columna	Descripcion
activa	Identifica si la alarma de tipo acumulable esta todavia activa.
id_alarma	Identificador de la alarma.
id_formula	Identificador de la formula
tiempo	Fecha y hora en que ocurrió la alarma

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Formula	id_formula	id_formula

Referenciada por

Referenciada por	Llave primaria	Llave foránea
alarma_notifica	id_alarma	id_alarma
Dato_expresion	id_alarma	id_alarma
Dato_expresion	id_alarma	id_alarma
Mensaje	id_alarma	id_alarma

Referenciada por	Llave primaria	Llave foránea
Dato_operando	id_alarma	id_alarma

• alarma notifica

Descripción

Identifica las alarmas que ya fueron notificadas.

Columnas

Columna	Descripcion
id_alarma	Identificador de la alarma.
id_usuario	Identificador del usuario
mensaje	Representa el número de mensajes que el usuario ha recibido para esa alarma

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Usuario	id_usuario	id_usuario
Alarma	id_alarma	id_alarma

• Atributo del hecho

Descripción

Identifica el valor de un operando que se recupera de un hecho.

Columnas

Columna	Descripcion
id_operando	Identificador del operando
referencia	Es el nombre de la columna a la que hace referencia el operando.

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Operando	id_operando	id_operando

• Columna

Descripción

Identifica a los operandos que son columnas de alguna base de datos indicada y se tendrán que recuperar a través de una consulta.

Columnas

Columna	Descripcion
db_name	Nombre de la base de datos donde reside la columna
id_condicion	Identificador de la condición
id_operando	Identificador del operando
nombre_columna	Nombre de la columna (puede tener funciones de agregacion del Servidor de SQL)
tabla	Nombre de la tabla a que pertenece la columna

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Operando	id_operando	id_operando

Referenciando a	Llave primaria	Llave foránea
Operando	id_operando	id_operando
Condicion	id_condicion	id_condicion

• Condicion

Descripción

Identifica la condicion que puede asociarse a los operandos recuperados de alguna consulta.

Columnas

Columna	Descripcion
definicion	Definición de la condición tal como la tecleo el usuario
definicion_compilada	Definición de la condición regresada por el parser.
id_condicion	Identificador de la condición

Referenciada por

Referenciada por	Llave primaria	Llave foránea
Columna	id_condicion	id_condicion

• Control lectura

Descripción

Tabla de control para la lectura de hechos nuevos.

Columnas

Columna	Descripcion
ptr_lectura	Mantiene el apuntador de lectura

Referenciando a

REFERENCIANDO A	LLAVE PRIMARIA	LLAVE FORANEA
Hecho	id_hecho	ptr_lectura

• Dato expresion

Descripción

Referencia los resultados de las expresiones intermedias usadas para la evaluacion de formula que produco la alarma.

Columnas

Columna	Descripcion
id_alarma	Identificador de la alarma.
id_expresion	Identificador de la expresion.
id_formula	Identificador de la formula
valor	Es el resultado de la expresion convertido en ASCII

Referenciando a

Referenciando a	Llave primaria	Llave foránea

Referenciando a	Llave primaria	Llave foránea
Alarma	id_alarma	id_alarma
Expresion	id_expresion	id_expresion
	id_formula	id_formula

• Dato hecho

Descripción
Contiene los datos referentes al hecho.

Columnas

Columna	Descripcion
id_hecho	Identifica la transaccion
id_operando	Identificador del dato
valor	Es el valor convertido en ASCII.

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Hecho	id_hecho	id_hecho

• Dato operando

Descripción
Referencia los datos relativos a la alarma.

Columnas

Columna	Descripcion
id_alarma	Identificador de la alarma.
id_operando	Identificador del operando
valor	Es el valor convertido en ASCII

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Alarma	id_alarma	id_alarma
Operando	id_operando	id_operando

• Disparador

Descripción
Identifica los triggers que pueden registrar transacciones.

Columnas

Columna	Descripcion
id_disparador	Identificador del trigger
nombre_disparador	Nombre del trigger

Referenciada por

Referenciada por	Llave primaria	Llave foránea
Formula	id_disparador	id_disparador
operando_disparador	id_disparador	id_disparador
Hecho	id_disparador	id_disparador

• Expresion

Descripción

Expresiones utilizadas para la evaluacion de formulas.

Columnas

Columna	Descripcion
expresion_compilada	Mantiene la expresion compilada por el parser
expresion_usuario	Mantiene la expresion escrita por el usuario
id_expresion	Identificador de la expresion
id_formula	Identificador de la formula

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Formula	id_formula	id_formula

Referenciada por

Referenciada por	Llave primaria	Llave foránea
Dato_expresion	id_expresion id_formula	id_expresion id_formula

• Formula

Descripción

Identifica las formulas necesarias para el funcionamiento del sistema:

Formulas base

Formulas personalizadas

Columnas

Columna	Descripcion
acceso	Describe el tipo de acceso que tienen los usuarios no dueños de la formula: Publico : Baja, Modificación, Utilizo Reservado : ninguno Accesible : Utilizo
acumulable	Identifica si la alarma generada por la formula es acumulable en el tiempo.
definicion	Definicion de la formula escrita por el usuario
definicion_compilada	Definicion compilada por el parser
descripcion	Nombre asignado por el usuario para identificar la formula
estado	Identifica si es una formula disponible
id_disparador	Identificador del trigger
id_formula	Identificador de la formula
id_usuario	Identificador del usuario
prioridad	Identifica en nivel de prioridad para la presentacion de alarmas (1- 3). No es utilizado como criterio de evalucion.
tipo	Nos dice el tipo de dato que queda después de calculada la formula. Para formulas personalizadas siempre queda un tipo booleano.
tipo_formula	Identifica el tipo de formula (Clausula where, formula base, formula personalizada)

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Disparador	id_disparador	id_disparador
Usuario	id_usuario	id_usuario

Referenciada por

Referenciada por	Llave primaria	Llave foránea
Alarma	id_formula	id_formula
Notifica	id_formula	id_formula
Usa_formula	id_formula	id_formula
Usa_formula	id_formula	For_id_formula
Expresion	id_formula	id_formula
operando_utiliza	id_formula	id_formula
operando_visualiza	id_formula	id_formula

• Hecho

Descripción

Transacciones que se tienen que procesar.

Columnas

Columna	Descripcion
id_disparador	Identificador del trigger
id_hecho	Identifica la transaccion

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Disparador	id_disparador	id_disparador

Referenciada por

Referenciada por	Llave primaria	Llave foránea
Control_lectura	id_hecho	ptr_lectura
Dato_hecho	id_hecho	id_hecho

• Mensaje

Descripción

Mantiene los mensajes que un usuario envia a otros desde el modulo de monitoreo.

Columnas

COLUMNA	DESCRIPCION
emisor	Identificador del usuario que envia el mensaje
id_alarma	Identificador de una alarma que se quiere notificar a otro usuario
id_mensaje	Identificador del mensaje
id_usuario	Identificador del usuario que recibe el mensaje.
notas	Mensaje enviado
tiempo	Fecha y hora en que se envia el mensaje

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Alarma	id_alarma	id_alarma
Usuario	id_usuario	id_usuario

• Mensaje_error

Descripción

Identifica los mensajes de error que ocurrierón al momento de calcular o enviar alrguna alarma.

Columnas

Columna	Descripcion
id_mensaje	Identifuicador del mensaje de error
mensaje	Una breve explicación del error
tiempo	Fecha y hora en que se registro el mensaje
tipo_error	

• Notifica

Descripción

Esta tabla indica que usuarios hay que notificar en caso de alarma.

Columnas

Columna	Descripcion
fin	Identifica la fecha de fin del periodo en que el usuario desea la notificacion de la alarma.
id_formula	Identificador de la formula
id_usuario	Identificador del usuario
inicio	Identifica la fecha de inicio del periodo en que el usuario desea la notificacion de la alarma.

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Formula	id_formula	id_formula
Usuario	id_usuario	id_usuario

• Operando

Descripción

Identifica los operandos utilizados en las formulas.

Columnas

Columna	Descripcion
id_operando	Identificador del operando
nombre_logico	nombre asignado por el administrador para referenciar el operando en las formulas
tipo_dato	Identifica el tipo de dato del operando (real, entero, character, boolean, fechas)
tipo_operando	Identifica el tipo del operando (valor fijo, campo, parametro del hecho).

Referenciada por

Referenciada por	Llave primaria	Llave foránea
Valor	id_operando	id_operando
Valor	id_operando	id_operando
Columna	id_operando	id_operando

Referenciada por	Llave primaria	Llave foránea
		erando
operando_utiliza	id_operando	id_operando
operando_disparador	id_operando	id_operando
Atributo_hecho	id_operando	id_operando
Dato_operando	id_operando	id_operando
operando_visualiza	id_operando	id_operando

• Operando_disparador

Descripción

Identifica los operandos que serßn recuperados por cierto disparador.

Columnas

Columna	Descripcion
id_disparador	Identificador del trigger.
id_operando	Identificador del operando.

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Operando	id_operando	id_operando
Disparador	id_disparador	id_disparador

• operando_utiliza

Descripción

Identifica los operandos que forman cierta formula.

Columnas

Columna	Descripcion
id_formula	Identificador de la formula
id_operando	Identificador del operando

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Operando	id_operando	id_operando
Formula	id_formula	id_formula

• operando_visualiza

Descripción

Identifica los operandos asociados a cierta formula, los cuales se visulizara su valor en el modulo de monitoreo al momento en que se consulte o notifique el calculo de la formula.

Columnas

Columna	Descripcion
id_formula	Identificador de la formula
id_operando	Identificador del operando
identificador	Identifica si el operando se tiene que presentar en la pantalla de monitoreo

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Formula	id_formula	id_formula
Operando	id_operando	id_operando

• Usa formula

Descripción

Tabla que indica cuales formulas base son utilizadas por una formula personalizada.

Columnas

Columna	Descripcion
For_id_formula	Identificador de la formula
id_formula	Identificador de la formula

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Formula	id_formula	id_formula
Formula	id_formula	For_id_formula

• Usuario

Descripción

Contiene los usuarios que tienen acceso al sistema.

Columnas

Columna	Descripcion
id_usuario	Identificador del usuario
nombre	Nombre del usuario
password	Password del usuario
rol	Identifica el rol del usuario: Administrador: acceso total Operador : modulo de configuración personalizada alta formulas - acceso permitido modificación formulas - acceso permitido solo si es dueño o si la formula es publica baja formulas - acceso permitido solo si es dueño o si la formula es publica modulo de notificación formulas publicas y accesibles - acceso permitido formulas reservadas - acceso permitido al dueño Usuario : modulo de configuración personalizada no tiene acceso modulo de notificación formulas publicas y accesibles - acceso permitido formulas reservadas - acceso permitido al dueño
usuario	Login del usuario

Referenciada por

REFERENCIADA POR	LLAVE PRIMARIA	LLAVE FORANEA
alarma_notifica	id_usuario	id_usuario
Notifica	id_usuario	id_usuario
Formula	id_usuario	id_usuario
Mensaje	id_usuario	id_usuario

- **Valor**

Descripción

Identifica un operando con valor fijo.

Columnas

COLUMNA	DESCRIPCION
id_operando	Identificador del operando
valor	Es el valor convertido en ASCII

Referenciando a

Referenciando a	Llave primaria	Llave foránea
Operando	id_operando	id_operando

Apéndice F Codificación de las expresiones regulares y gramáticas

Parser Analiza Formula y parser Calcula Formula

Análisis Léxico

TRUE	→	True | true | TRUE
FALSE	→	False | false | FALSE
boolean	→	**TRUE | FALSE**
col_bol	→	#b&
col_cad	→	#c&
col_fec	→	#f&
col_real	→	#r&
opfuncion	→	abs
opneg	→	no | not | No | NO | Not | NOT
opdisy	→	Y | y | and | AND | And
opconj	→	OR | Or | or | O | o
opxor	→	XOR | Xor | xor
IF	→	if | IF | If | SI | Si | si
digito	→	[0-9]
letra	→	[A-Za-z]
simbolo	→	_ | - | # | % | $ | ? | ¿ | ¡ | !
id	→	**letra (letra | digito | simbolo)***
cadena	→	" (**id | real**) "
hora	→	((([0 | 1] **digito**) | (2 [0-3])) : ([0-5] **digito**))
mes31dias	→	((0 ? [1 | 3 | 5 | 7 | 8]) | 10 | 12)
mes30dias	→	((0 ? [4 | 6 | 9]) | 11)
febrero	→	02 | 2
siglo_20	→	19 **digito digito**
siglo_21	→	20 **digito digito**
anyo	→	**siglo_20 | siglo_21**
fec31dias	→	(**digito** | ([0-2] **digito**) | (3 [0 | 1])) / (**mes31dias / anyo**)
fec30dias	→	(**digito** | ([0-2] **digito**) | 30) / (**mes30dias / anyo**)
fecfebrero	→	(**digito** | ([0-2] **digito**)) / (**febrero / anyo**)
fecha	→	(**fec31dias | fec30dias | fecfebrero**) (**blanco hora**) ?
oprelacion	→	= | < | > | <= | <> | >= | !=
oparitmetico	→	^ | + | - | * | / | %
oplogico	→	**opneg | opdisy | opconj | opxor**
entero	→	**digito** +
real	→	**entero** (\.**entero**) ? ([E | e] [+ \ -] ? **entero**) ?
columna	→	**id**

Análisis Sintáctico

```
linea   → lineas expr '\n' | lineas expr4 '\n'
   s    | lineas expr5 '\n' | lineas '\n'
        | error '\n'

 expr   → expr OP_DISY expr | expr OP_CONJ expr
        | expr OP_XOR expr | OP_NEG expr
        | expr OP_IGU expr | expr OP_DIF expr
        | '(' expr ')' | '[' expr ']' | '{' expr '}'
        | expr2
        | expr3

expr2   → expr4 OP_MEN expr4 | expr4 OP_MAY expr4
        | expr4 OP_MENI expr4 | expr4 OP_MAYI expr4
        | expr4 OP_IGU expr4 | expr4 OP_DIF expr4
        | expr5 OP_MEN expr5 | expr5 OP_MAY expr5
        | expr5 OP_MENI expr5 | expr5 OP_MAYI expr5
        | expr5 OP_IGU expr5 | expr5 OP_DIF expr5
        | DATO_BOOLEAN

expr3   → DATO_CADENA OP_IGU DATO_CADENA
        | DATO_CADENA OP_DIF DATO_CADENA

expr4   → expr4 OP_SUMA expr4 | expr4 OP_RESTA expr4
        | expr4 OP_MULT expr4 | expr4 OP_DIV expr4
        | expr4 OP_POT expr4 | expr4 OP_PORC
        | OP_RESTA expr4 | OP_ABS '(' expr4 ')'
        | '(' expr4 ')' | '[' expr4 ']' | '{' expr4 '}'
        | DATO_REAL
        | DATO_ENTERO

expr5   → '"' DATO_FECHA '"'
        | DATO_FECHA
```

Parser Evalua Condicion

Análisis Léxico

```
where    → where | WHERE
TRUE     → True | true | TRUE
FALSE    → False | false | FALSE
boolean  → TRUE | FALSE
col_bol  → #b&
col_cad  → #c&
col_fec  → #f&
col_real → #r&
opneg    → not | Not | NOT
opdisy   → and | AND | And
opconj   → OR | Or | or
digito   → [0-9]
letra    → [A-Za-z]
simbolo  → _ | - | # | % | $ | ? | ¿ | ¡ | ! | @
hora     → ( ( [0 | 1] digito ) | ( 2 [0-3] ) ) : ( [0-5] digito )
mes31dias → ( 0 ? [1 | 3 | 5 | 7 | 8 ] ) | 10 | 12
```

```
mes30dias  →  ( 0 ? [4 | 6 | 9] ) | 11
febrero    →  02 | 2
siglo_20   →  19 ( digito digito )
siglo_21   →  20 ( digito digito )
anyo       →  siglo_20 | siglo_21
fec31dias  →  ( digito | ( [0-2] digito ) | (3 [0 | 1] ) ) / ( mes31dias / anyo)
fec30dias  →  ( digito | ( [0-2] digito ) | 30 ) / ( mes30dias / anyo)
fecfebrero →  ( digito | ( [0-2] digito ) ) / ( febrero / anyo)
fecha      →  \ ( fec31dias | fec30dias | fecfebrero ) ( blanco hora ) ? \
oprelacion →  = | < | > | <= | <> | >= | !=
oparitmetico → ^ | + | - | * | / | %
entero     →  digito +
real       →  entero ( \. entero ) ? ( [E | e] [+ -] ? entero ) ?
id         →  letra ( letra | digito | simbolo ) *
cadena     →  " (id | real) "
null       →  "is null" | "is not null"
like       →  ( ( opneg blanco) ? like cadena ) ( escape " ( simbolo | letra ) " ) ?
tabla      →  ( ( id . ) ? id ) | ( id . id . id ) | ( id .. id )
columna    →  ( tabla blanco ) ? id
funcag     →  ( ( sum | min | max | avg | count ) ( columna ")" ) | "count(*)"
columnas   →  columna ( , ( columna | funcag ) ) *
tablas     →  tabla ( , tabla ) *
select     →  select columnas from tablas
any        →  oprelacion ( [A | a] ny ) | ANY
all        →  oprelacion ( [A | a] ll ) | ALL
in         →  opneg ? blanco ( In | in | IN )
cin        →  in "(" ( cadena | real | entero ) ( , ( cadena | real | entero ) ) * ")"
part       →  year | month | week | day | weekday | hour | minute | second | yy | mm | wk | dd | dw | hh | mi | ss
datatype   →  int | ( "numeric(" entero "," entero ")" ) | ( "float(" entero ")" ) | real | money | datetime | ( "char(" entero ")" ) | ( "varchar(" entero ")" ) | bit
fec_func   →  "getdate()" | ( "dateadd(" part , entero , ( fecha | "getdate()") ")" )
num_func   →  ( "abs(" ( entero | real ) ")" ) | ( "datepart(" part , ( fecha | fec_func ) ")" ) | ( "datediff(" part , ( fecha | fec_func ) , ( fecha | fec_func ) ")" )
str_func   →  "lower(" ( cadena | columna ) ")" ) | ( "right(" ( cadena | columna ) , entero ")" ) | ( "substring(" ( cadena | columna ) , entero "," entero ")" ) | ( "upper(" ( cadena | columna ) ")" ) | ( "datename(" part , ( fecha | fec_func ) ")"
```

Análisis Sintáctico

```
lineas → lineas subquery '\n' | lineas join '\n'
       | lineas expr '\n' | lineas expr4 '\n'
       | lineas expr5 '\n' | lineas expr7 '\n'
       | lineas expr8 '\n' | lineas '\n'
       | error '\n'

subquery → subquery OP_DISY subquery | subquery OP_CONJ subquery
         | join OP_DISY subquery | join OP_CONJ subquery
         | subquery OP_DISY join | subquery OP_CONJ join
         | expr OP_DISY subquery | expr OP_CONJ subquery
         | subquery OP_DISY expr | subquery OP_CONJ expr
         | expr ANY sql | expr ALL sql
         | expr IN sql | expr7 ANY
         | expr7 ALL sql | expr7 IN sql
         | expr8 ANY sql | expr8 ALL sql
         | expr8 IN sql | misc ANY sql
         | misc ALL sql | misc IN sql
         | '(' subquery ')'

sql → '(' SELECT WHERE expr ')' | '(' SELECT WHERE join ')' | '(' SELECT ')'

join → join OP_DISY join | join OP_CONJ join
     | '(' join ')' | '[' join ']' | '{' join '}'
     | misc

misc → expr7 OP_IGU expr4 | expr7 OP_DIF expr4
     | expr7 OP_SUMA expr4 | expr7 OP_RESTA expr4
     | expr7 OP_MULT expr4 | expr7 OP_DIV expr4
     | expr7 OP_POT expr4 | expr4 OP_IGU expr7
     | expr4 OP_DIF expr7 | expr4 OP_SUMA expr7
     | expr4 OP_RESTA expr7 | expr4 OP_MULT expr7
     | expr4 OP_DIV expr7 | expr4 OP_POT expr7

expr → expr OP_DISY expr | expr OP_CONJ expr
     | OP_NEG expr | expr OP_IGU expr
     | expr OP_DIF expr | expr CIN | expr NULO
     | '(' expr ')' | '[' expr ']' | '{' expr '}'
     | expr2 | expr3

expr2 → expr4 OP_MEN expr4 | expr4 OP_MAY expr4
      | expr4 OP_MENI expr4 | expr4 OP_MAYI expr4
      | expr4 OP_IGU expr4 | expr4 OP_DIF expr4
      | expr5 OP_MEN expr5 | expr5 OP_MAY expr5
      | expr5 OP_MENI expr5 | expr5 OP_MAYI expr5
      | expr5 OP_IGU expr5 | expr5 OP_DIF expr5
      | DATO_BOOLEAN

expr3 → expr8 OP_IGU expr8 | expr8 OP_DIF expr8
      | expr7 OP_IGU expr7 | expr7 OP_DIF expr7
      | expr8 OP_IGU expr7 | expr8 OP_DIF expr7
      | expr7 OP_IGU expr8 | expr7 OP_DIF expr8
      | expr7 CIN | expr7 NULO
      | expr7 LIKE | expr8 CIN
      | expr8 NULO | expr8 LIKE

expr4 → expr4 OP_SUMA expr4 | expr4 OP_RESTA expr4
      | expr4 OP_MULT expr4 | expr4 OP_DIV expr4
      | expr4 OP_POT expr4 | expr4 OP_PORC
      | OP_RESTA expr4 | OP_ABS '(' expr4 ')'
      | expr4 CIN | expr4 NULO
      | '(' expr4 ')' | '[' expr4 ']' | '{' expr4 '}'
      | expr6

expr5 → '"' DATO_FECHA '"' | DATO_FECHA

expr6 → DATO_ENTERO | DATO_REAL

exp7 → expr7 OP_SUMA expr7 | expr7 OP_RESTA expr7
     | expr7 OP_MULT expr7 | expr7 OP_DIV expr7
     | expr7 OP_POT expr7 | expr7 OP_PORC
     | OP_RESTA expr7 | OP_ABS '(' expr7 ')'
     | '(' expr7 ')' | '[' expr7 ']' | '{' expr7 '}'
     | expr9

expr8 → expr8 OP_SUMA expr8
      | '(' expr8 ')' | '[' expr8 ']' | '{' expr8 '}'
      | DATO_CADENA

expr9 → COLUMNA
```

Apéndice G Pseudocódigo

Servidor Monitor

Programa principal

Manejadores de Eventos

```
connect_handler(ssp)
{
        Obtener los datos del cliente conectado y registralos en el archivo de log
        Crear el RPC notifica_USER utilizado para la notificacion de alarmas a los clientes conectados
        if( El RPC ya existe )
           Tan solo se registra al cliente en la lista de notificacion
        else if( EL RPC si fue creado )
           Se registra al cliente en la lista de notificacion
        Se actualiza la informacion de los ususarios conectados
        if( Existe otro cliente conectado con el mismo login (offset != -1))
                Se incrementa el numero de conexiones del cliente
        else if (No existe ningun cliente conectado con el mismo login)
                Se crea la estructura de control del cliente
        userList.num_users++
        Decirle al cliente "Eso es todo por ahora."
}

disconnect_handler(ssp)
{
        Obtener los datos del cliente conectado y registralos en el archivo de log
        Brrar al cliente de la lista de notificaciones
        Actualizar la informacion de los usuarios conectados
        Si es la ultima conexion libera el espacio
        userList.num_users--;
}
```

Procedimientos Registrados

```
rp_evalua_condicion(srvProc)
{
        Describir el parametro de entrada @condicion y ligarlo a la variable condi
        Transferir los valores de los parametros de entrada a las variables
        Abrir una conexion al SQL Server
        Sustituir operandos por su tipo de datos (expr_tipos) y obtener la expresion con los id's de los operandos (expr_compilada)
        Cerrar la conexion al SQL Server
        Verificar con el PARSER EVALUA CONDICION la clausula where (expr_tipos) y obtiener su valor (en valorCondicion)
        Describir el parametro de retorno @expresion y ligarlo a la variable expresion_compilada
        Definir el parametro de retorno @valor
        Transferir los parametros de salida al cliente
        Decirle al cliente "eso es todo por ahora"
}

activa_formula(ssp)
{
        Describir el parametro de entrada @formula y ligarlo a la variable idf
        Transferir los valores de los parametros de entrada a las varibales (con srv_xferdata(..., CS_GET, SRV_RPCDATA, ...) )
        Mandar la peticion de ACTIVACION (ACTIVA_FORMULA) de la formula (idf) a la cola CARGA_QUEUE
        Decirle al cliente "eso es todo por ahora" (con srv_senddone(..., SRV_DONE_FINAL, ...) )
}
```

```
desactiva_formula(ssp)
{
        Describir el parametro de entrada @formula y ligarlo a la variable idf
        Transferir los valores de los parametros de entrada a las varibales
        Mandar la peticion de DESACTIVACION (DESACTIVA_FORMULA) de la formula (idf) a la cola CARGA_QUEUE */
        Decirle al cliente "eso es todo por ahora"
}

rp_llama_notificacion(ssp)
{
        Describir los parametros de entrada @idalarma, @idformula, @idusuario y ligarlo a las variables
        Transferir los valores de los parametros de entrada a las varibales
        Abrir una conexion con el SQL SERVER
        Obtener el nombre del usuario (USER)
        Notificar al i-esimo usuario
        Cerrar la conexion con el SQLServer
        Decirle al cliente "eso es todo por ahora"
}

activa_alarma(ssp)
{
        Describir los parametros de entrada @idalarma, @idformula y ligarlos a las variables
        Transferir los valores de los parametros de entrada a las varibales
        Abrir una conexion con el SQL SERVER
        Notificar a todos los usuarios
        Cerrar la conexion con el SQLServer
}
```

Hilos

```
Carga_handler()
{
        Bloquer al thread de Analisis
        Abrir una conexion al SQL Server (c_ptr)
        Carga inicial de los disparadores, formulas y operandos
        Desbloquear el acceso a la estructura BUFFER_FORMULAS y al thread de Analisis
        /* Codigo del Thread de Carga que se estara ejecutando continuamente */
        while (1==1)
        {
          Esperar el mensaje de activacion o desactivacion de alguna formula
          if (Se hizo una peticion de ACTIVACION )
          {
             Bloquear el acceso a la estructura BUFFER_FORMULAS
             Activar la formula (idf) en memoria en el BUFFER_FORMULAS
             Desbloquear el acceso a la estructura BUFFER_FORMULAS
          }
          else if ( Se hizo una peticion de DESACTIVACION )
          {
             Bloquear el acceso a la estructura BUFFER_FORMULAS
             Desactivar la formula (idf) en memoria en el BUFFER_FORMULAS
             Desbloquear el acceso a la estructura BUFFER_FORMULAS
          }
          Colocar al thread el la cola de runnable y permitir la ejecucion de otros threads
        }
}

Analisis_handler()
{
     Bloquear el thread de Analisis hasta que el thread de Carga termine la carga inicial */
     Abrir una conexion estatica con el SQL Server ( conn_ptr )
     /* Codigo del Thread que se estara ejecutando continuamente */
     while(1==1)
     {
       Efectuar el control para la depuracion de los hechos
```

```
if ( Hay un nuevo hecho )
{
  Verificar si hay formulas para el hecho recuperado */
  for( Para cada uno de los disparadores )
    if ( Existen formulas asociadas al disparador asociado al hecho recuperado )
      for ( Para cada una de las formulas en el disparador )
        if( La formula es personalizada )
          if ( La formula esta activa y su periodo de activacion ya inicio )
            if ( El periodo de activacion no ha caducado.)
              ptr_form_act[nform] = ptr_formula;
              nform++;
            else if (El periodo de activacion de la formula caduco )
              Llamar all rpc desactiva_formula
        else if ( Es formula base (simpre esta activa) )
        {
          ptr_form_act[nform] = ptr_formula;
          nform++;
        }
    if( Existen formulas activas para el hecho )
      for( Cada uno de los datos del hecho )
        for( Cada uno de los operandos en el disparador )
          if ( Ya esta en memoria en el BUFFER_FORMULAS la definicion del operando )
              Liberar el valor anterior y almacenar el nuevo valor
          if ( El operando no existe en memoria en el BUFFER_FORMULAS)
              Alta en memoria del nuevo operando
        RECUPERA LOS OPERANDOS NECESARIOS
        for ( Cada uno de las sentencias select del disparador )
        {
          Sustituir los operandos en la definicion compilada del select
          if( Hubo error en los operandos del hecho )
          {
            Notificar al administrador la ocurrencia del error y registrarlo en DBMAD
            break;
          }
          Si no hubo error en los operandos del hecho, recuperar los operandos
          if( Hubo error en la recuperacion de operandos )
          {
            Notificar al administrador la ocurrencia del error y registrarlo en DBMAD
            break;
          }
          if ( No hubo error en la recuperacion de operandos )
            for( Cada operando (i) del j-esimo select )
              Guardar el valor del i-esimo operando recuperado
          else if( Se ha recuperado mas de un valor para el operando )
          {
            Notificar al administrador la ocurrencia del error y registrarlo en DBMAD
            break;
          }
          else if( No se recupero ningun valor para el operando )
          {
            Notificar al administrador la ocurrencia del error y registrarlo en DBMAD
            break;
          }
        } /* For */
      if( No se presento ningun error de recuperacion de operandos )
      {
        CALCULA CADA FORMULA BASE
        for( Cada formula ( f ) en el disparador[ptr_disp_act] )
          if( La formula es base )
          {
            Calcula cada expresion por separado
            if( Hubo algun error )
              Salir del ciclo for y no hacer el calculo de la formula
            Sustituir los valores de las expresiones calculadas en la definicion de la Formula base (en expr)
            Calcular con el PARSER CALCULA_FORMULA la Formula base y obtener su resultado ( en result_for )
            if ( Hubo error en los tipos del hecho y de la Formula base)
            {
```

```
                    Notificar al administrador la ocurrencia del error  y registrarlo en DBMAD
                    break;
                  }
                  Si hubo error en los tipos del hecho y de la Formula base, almacenar el resultado de la Formula base
                } /* if */
              if( No hubo ningun error en el calculo de las Formulas base )
              {
                CALCULA CADA FORMULA PERSONALIZADA
                for( Cada una de las formulas ( f ) del disparador[ptr_disp_act] )
                  if( La formula es Formula personalizada )
                    Calcular cada expresion por separado
                    Si no hubo error de tipos entre los datos del hecho y la expresion, almacenar el resultado de la expresion
                    if( Hubo algun error en las expresiones )
                      Salir del ciclo for y no realizar el calculo de la Formula personalizada
                    for( Cada una de las expresiones ( i ) de la f-esima Formula personalizada)
                      Almacenar en la i-esima expresiones el valor del resultado obtenido
                    Sustituir los valores de las expresiones en la definicion de la Formula personalizada
                    Calcular con el PARSER CALCULA_FORMULA la Formula personalizada y obtener su resultado (result_for)
                    if ( Hubo error de tipos en la Formula personalizada )
                    {
                       Notificar al administrador la ocurrencia del error  y registrarlo en DBMAD
                       break;
                    }
                    Si no hubo error de tipos en la Formula personalizada, almacenar el resultado final de la Formula personalizada
                    if( La Formula debe notificarse a algun usuario )
                    {
                      if( El resultado de la formula es positivo (TRUE) )
                        Notificar al usuario la ocurrencia de una alarma ( con notifica_alarma(..., ptr_disp_act, ptr_form_act[f], ...) )
                    }
                    else if( No se debe notificar )
                      Desactivar la alarma acumulable ( con desactiva_alarma(..., ptr_disp_act, ptr_form_act[f], ...) )
                  } /* if */
                } /* for */
              } /* if */
            } /* if */
          } /* if */
          Actualizar el apuntador de lectura de los hechos en DBMAD
        } /* if */
        Desbloquear el acceso a la estructuras BUFFER_FORMULAS
        Colocar al thread el la cola de runnable y permitir la ejecucion de otros threads
      } /* while */
}
```

Algoritmos generales

```
notifica_alarma(conn_ptr,tp,ptrd,ptrf)
{
        Obtener el numero de usuarios a notificar
        for( Cada uno de los usuarios a notificar )
          Obtener el id_usuario y el usuario
        if ( La Formula es acumulable )
        {
          Obtener el id_alarma de la formula a notificar
          if ( Solo existe un id_alarma para la formula )
          {
            Iniciar la transaccion en el SQL Server (begin tran)
            Actualizar la fecha de ocurrencia de la alarma
            for( Cada uno de los operandos ( i ) del disparador ( ptrd ) )
              Actualizar el valor del Dato_operando de la alarma ocurrida con el valor del i-esimo operando
            for( Cada expresion ( i ) en la formula ( ptrf ) )
              Actualizar el valor del Dato_expresion de la alarma ocurrida con el valor de la i-esima expresion
            Terminar la transaccion en el SQL Server (commit tran)
            Se envia la notificacion a los clientes
        }
        else if ( Hay mas de un _id_alarma para la formula)
            return FALLA;
```

```
        /* La formula no es acumulable */
        Insertar la alarma en la base de datos DBMAD
        for( Cada uno de los usuarios por notificar )
          Insertar en alarma_notifica la notificacion del usuario
        Iniciar la transaccion en el SQL Server (begin tran)
        for( Cada uno de los operandos ( i ) del disparador (ptrd) )
          Insertar el valor del Dato_operando de la alarma ocurrida con el valor del i-esimo operando
        for( Cada expresion ( i ) en la formula ( ptrf ) )
          Insertar el valor del Dato_expresion de la alarma ocurrida con el valor de la i-esima expresion
        Terminar la transaccion en el SQL Server (commit tran)
        Se envia la notificacion a los clientes
}

envia_notificacion(ssp,conn_ptr,procName,idal,idf,login)
{
        La alarma sera notificada despues de que haya pasado minimo un segundo respecto al ultimo envio
        while (El tiempo actual sea igual al tiempo del ultimo envio )
          Colocar al thread el la cola de runnable y permitir la ejecucion de otros threads
        Obtener los datos de la notificacion
        Iniciar el RPC de la notificacion
        Definir los parametros de entrada @idalarma, @idformula, @numopers, @numexprs, @ formula
        for( Cada uno de los operandos ( N ) en notificaInfo->num_opers )
          Definir el parametro de entrada @ parametroN
        for( Cada una de las expresiones ( K ) en  notificaInfo->num_exprs )
          Definir el parametro de entrada @ parametroK
        for(i=j; i <= 20; i++)
          Definir el parametro de entrada @ parametroI
        Ejecutar el RPC de la notificacion (con srv_regexec(...) )
}

envia_notificacion_sa(ssp,num_error)
{
        Iniciar el RPC de la notificacion al administrador (con srv_reginit(..., 'hotifica_sa", ..., SRV_M_PNOTIFYALL....) )
        Definir el parametro de entrada @num_error (con srv_regparam(...,"@num_error", CS_INT_TYPE, ...,num_error,...) )
        Ejecutar el RPC de la notificacion al administrador (con srv_regexec(...) )
}
```

BIBLIOGRAFIA

1) **Aho, Alfred U., Sethi, Ravi y Ulman, Jeffrey D.** *Compiladores, principios, técnicas y herramientas*. Addison Wesley Iberoamericana. 1990.

2) **Andrews, Gregory R.** *Concurrent Programming, principles and practice*. The Benjamin/Cummings Publishing Company, Inc. 1991.

3) **Booch, Grady**. *Análisis y diseño orientado a objetos con aplicaciones*. 2a ed.. Addison-Wesley / Diaz de Santos. 1996.

4) **Biderdorf, Daryl, Gliden, Keith y Powers, Shelley**. *Power Builder How-to*. Waite Group Press. 1996.

5) **Brookshear, J. Glenn**. *Teoría de la computación, lenguajes formales, autómatas y complejidad*. Addison-Wesley Iberoamericana. 1993.

6) **Coad, P. y Yourdon, E.** *Object-Oriented Analysis*. Prentice Hall. 1990.

7) **Collins, Dave.** *Design Object-Oriented user interfaces*. Benjamin/Cummings Publishing. 1995.

8) **Coulouris G., Dollimore Jean, Kindenberg T.** *Distributed Systems*. 2a ed.. Addison –Wesley. 1995.

9) **Ghezzi, Carlo y Jazayeri, Mehdi**. *Programming Language Concepts*. 3era ed. John Wiley & Sons. 1998.

10) **GretZinger, M. y Prabhat, K.** *Distributed Object-Oriented Data-System Design,* Prentice Hall. 1992.

11) **Jacobson, Ivar y otros**. *Object-oriented software engineering.* Addison-Wesley. 1992.

12) **James, Martin. y James J.** *Análisis y diseño Orientado a Objetos.* Prentice Hall Hispanoamericana. 1994.

13) **Lester, Bruce P.** *The Art of Parallel Programming*. Prentice Hall. 1993.

14) **Lewis, Ted G.** *Foundations of Parallel Programming, A machine–independent approach.* IEEE Computer Society Press. 1994.

15) **Lewit, Bil y Berg, J. Daniel.** *Threads Primer A Guide to Multithreaded Programming,* Prentice Hall, 1996.

16) **Márquez García, Francisco Manuel.** *UNIX Programación avanzada.* Addison-Wesley Iberoamericana.

17) **Northrup, Charles J.** *Programming with UNIX threads.* John Wiley & Sons, Inc. 1996.

18) **Perrott, R. H.** *Parallel Programming.* Addison-Wesley. 1987.

19) **Rambaugh, James. y otros.** *Modelado y diseño orientado a objetos.* Prentice Hall. 1991.

20) **Rational Rose Corporation.** *Unifield Modeling Language Notation guide.* 1997.

21) **Reference Manual.** *Open Client and Open Server, Common Libraries.* 1994.

22) **Reference Manual.** *Open Client Client-Library/C.* 1993

23) **Robbins, Kay A.**, Robbins Steven. *UNIX Programación práctica, Guía para la Concurrencia, la Comunicación y los Multihilos.* Prentice Hall. 1997.

24) **Sybase SQL Server, Refence Manual Volume I,** *Commands, Funtions and Topics.* 1993

25) **Sybase SQL Server, Refence Manual Volume II,** *System Procedures and Catalog Stored Procedures.* 1993.

26) **Tanenbaum, Andrew S.** *Sistemas Operativos Modernos.* Prentice Hall. 1993.