INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS ESTADO DE MÉXICO

METODOLOGÍA PARA ANÁLISIS Y DISEÑO DE SISTEMAS DISTRIBUIDOS

TESIS PARA OPTAR EL GRADO DE
MAESTRO EN CIENCIAS DE LA COMPUTACIÓN
PRESENTA

DANIEL ARENAS SELEEY

Coasesores: Dr. JESÚS SÁNCHEZ VELÁZQUEZ
Dr. ROBERTO GÓMEZ CÁRDENAS

Comité de Tesis: Dr. JESÚS VÁZQUEZ GÓMEZ
M.C. FRANCISCO CAMARGO SANTACRUZ

Jurado:	Dr. JESÚS VÁZQUEZ GÓMEZ	Presidente
	M.C. FRANCISCO CAMARGO SANTACRUZ	Secretario
	Dr. JESÚS SÁNCHEZ VELÁZQUEZ	Vocal
	Dr. ROBERTO GÓMEZ CÁRDENAS	Vocal

Atizapán de Zaragoza, Edo. México, Junio de 1998

CONTENIDO

1. INTRODUCCION

La aparición de las PC's en los ochenta, ha permitido tener un ambiente individualmente configurado para cada usuario. Toda la capacidad de cómputo, la capacidad de almacenamiento, datos y programas han llegado a ser propios de cada PC. Se han implementado interfaces de usuario muy convenientes y soluciones individualizadas. Esto ha sido soportado por sistemas operacionales orientados a ventanas, sistemas de procesamiento de texto, hojas de cálculo, programas gráficos, etc.

Con la idea de dar acceso a los datos compartidos por los usuarios, las PC's han sido conectadas a computadores maestros (host). PC's, estaciones de trabajo (workstations) y computadores estros han sido combinados en sistemas cliente-servidor. La parte principal de la aplicación corre sobre el cliente pero algunas partes corren sobre servidores especializados. Vía servicios de comunicación, los clientes pueden accesar las facilidades suministradas por los servidores.

El costo de los sistemas computacionales (especialmente el de computadores pequeños de escritorio o PC's) disminuye, la facilidad de conectividad de los computadores personales de diferentes rendimientos a supercomputadores también se ha incrementado. Este es el caso de las redes de computadores muy comunes hoy en día y en las cuales se encuentran todo tipo de computadoras. Estas redes son ya usadas en muchas aplicaciones y sus principales ventajas son el incremento en el rendimiento y confiabilidad, y la facilidad de crecimiento del sistema con la simple adición de nodos. Los usuarios encuentran que se puede hacer una considerable reducción de costos reemplazando los sistemas grandes centralizados con sistemas distribuidos. Sin embargo la implementación de cada sistema distribuido involucra diferentes problemas de diseño; es por eso que uno de los propósitos de esta tesis es suministrar a los especialistas con información detallada acerca de métodos para el análisis y diseño de aplicaciones sobre dichos sistemas.

Es entonces muy importante establecer una relación más precisa entre la ingeniería de software y los sistemas distribuidos. La ingeniería de software es una disciplina valiosa en el desarrollo de sistemas, soportando la creación y mantenimiento de los mismos. En esta tesis se definirá la

relación entre ciclos de vida, métodos y herramientas, y cómo podrían constituir el marco de un ambiente de ingeniería de software más abierto, especialmente en el desarrollo de sistemas de software distribuido.

Además se busca establecer si hay una diferencia crucial entre las técnicas de la ingeniería clásica del software (técnicas para la producción de software secuencial) y las técnicas necesarias para la producción de sistemas distribuidos. Esta inquietud viene del hecho de que ciertas nociones como seguridad (safety)/vida (liveness), asumir (assume)/comprometer (commit) y sistema (system)/ambiente (environment) son esenciales para sistemas distribuidos solamente. Más aún, en general, los sistemas distribuidos tienden a ser más complejos y difíciles de desarrollar que el software secuencial.

El presente trabajo tiene como objetivo plantear un esquema metodológico para el análisis y diseño de sistemas distribuidos. Para lograr este objetivo se han desarrollado los siguientes capítulos:

Capítulo 2. En este capítulo se hace una introducción a los sistemas distribuidos con el fin de tener un contexto general sobre los mismos. Se describen las características principales y sus ventajas. Por otra parte se cubren conceptos relacionados con la computación en red, tales como topología, el modelo OSI, el paradigma cliente-servidor entre otros. Finalmente se describen los ambientes de computación distribuida y de computo cooperativo.

Capítulo 3. Una vez considerados los aspectos principales de los sistemas distribuidos, en este capítulo aspectos de Ingeniería de Software y los diferentes enfoques de análisis y diseño son tratados. Las principales actividades del desarrollo de software son explicadas. Se detallan los conceptos relacionados con paradigmas de desarrollo o ciclos de vida, los enfoques funcionales, de datos, de procesos, y objetos entre otros.

Capítulo 4. Los anteriores capítulos dan una visión general sobre los sistemas distribuidos y la ingeniería de software. En este capítulo se describen métodos más específicos para el análisis y diseño de sistemas en general, así como otros que cubren aspectos de los sistemas de software distribuido.

Capítulo 5. En este capitulo se integran los conceptos enunciados a lo largo de este trabajo de tesis y se enuncian ideas básicas en relación con el desarrollo de sistemas de software haciendo énfasis en el caso de los sistemas distribuidos. En resumen se enuncian cuales son los retos que se presentan para este tipo de sistemas y que oportunidades hay para solucionarlos.

Capítulo 6. Los patrones de diseño son una alternativa tentadora para la creación de un ambiente de ingeniería de software más abierto. Ellos pueden ser reusados para la solución de problemas relacionados con el desarrollo de software. En este capítulo, se presentan las principales características de los patrones de diseño y se relacionan y describen algunos de los principales patrones existentes a la fecha.

Capítulo 7. En este capítulo se hace una aplicación de los patrones de diseño en dos casos reales: Sistemas de Manufactura Flexible y Bases de Datos Distribuidas. De esta manera se evalúa su aplicabilidad y algunas de las características promulgadas por los patrones de diseño.

Capítulo 8. Una vez aplicados y evaluados los patrones de diseño, en este capítulo se plantea un esquema metodológico basado en los mismos. Este esquema busca cubrir en forma general los principales aspectos (resultado de la presente investigación) a tener en cuenta en el análisis y diseño de sistemas de software distribuido e indica que patrones deberían ser creados para cubrir dichas aspectos y obtener un sistema acorde a los requerimientos especificados.

Capítulo 9. Corresponde a las conclusiones de esta tesis y, a ideas respecto a trabajos de investigación que pueden ser ralizados en el futuro, tomando como punto de partida el final de esta investigación.

2 LOS SISTEMAS DISTRIBUIDOS

Durante las dos últimas décadas la principal aplicación del procesamiento de información y de la tecnología de cómputo se ha hecho con sistemas centralizados. Los usuarios en dichas terminales comparten el procesador central, los equipos conectados para almacenamiento de datos, y los programas disponibles. Cada usuario dispone de un intervalo de tiempo del procesador central, es decir, el computador es compartido por los diferentes usuarios. Este tipo de sistema se conoce como sistema de tiempo compartido y puede considerarse también la ejecución paralela o concurrente de varios programas sobre varios procesadores. Si el mismo programa es ejecutado por diversos usuarios, cada ejecución está en un estado, es decir, cada usuario tiene su propio conjunto de datos y la instrucción del programa a ser ejecutada es específica para cada usuario. La ejecución de un programa para un usuario particular es llamado un proceso. A cada proceso se le da su propio intervalo de tiempo de procesador.

La mayoría de los programas involucran acceso a bases de datos (los programas procesan una o más bases de datos). La principal tarea de las aplicaciones de bases de datos es recuperar, procesar, actualizar y reemplazar datos almacenados en las bases. Uno de los problemas a resolver en sistemas distribuidos ha sido desarrollar un modelo de datos que cubra todos los aspectos de una organización con la idea de evitar que el mismo dato esté en más de una base de datos.

Varios procesos pueden accesar concurrentemente una misma base de datos. Para mantener los datos en un estado consistente, el acceso compartido de los mismos deberá ser sincronizado. La sincronización permite que la ejecución de los procesos esté controlada: un proceso está detenido hasta que otro proceso ha alcanzado cierto estado.

En sistemas distribuidos, las aplicaciones se basan en computadoras interconectadas. Un programa distribuido es implementado como un conjunto de procesos corriendo en diferentes sistemas computacionales que conforman una red. Estos procesos cooperan estrechamente para lograr un objetivo común. Para lograr dicho objetivo común los procesos deben intercambiar y sincronizar su ejecución. Así, un ambiente de ingeniería de software para el desarrollo de

programas distribuidos debe soportar procesos, su distribución en varias computadoras, y la comunicación y sincronización de los procesos.

En los sistemas llamados cliente-servidor el paralelismo descrito anteriormente está oculto para el usuario. Los clientes requieren soporte de los servidores y están conectados (clientes y servidores) por una red de computadoras. Los clientes corren una interfase amigable (por ejemplo, X-windows) y el programa principal de la aplicación, el cual usa los servicios ofrecidos por los servidores (por ejemplo, servicios para bases de datos). Los servicios son usados e invocados como procedimientos en programas secuenciales. La comunicación en los sistemas cliente-servidor puede hacerse en términos de las operaciones de **intercambio de mensajes** (*send y* receive), pero más comúnmente, y debido a que estos sistemas son ejecutados en sistemas remotos, su procedimiento de invocación es denominado **Llamado a Procedimiento Remoto** (Remote Procedure Call - RPC). Por más de veinte años, los sistemas operativos, sistemas de control de plantas nucleares, sistemas de comunicación, etc., han sido estructurados como programas cooperativos.

En el pasado las investigaciones y la aplicación de la ingeniería de software han sido influenciados por el deseo de desarrollar un ambiente de ingeniería de software que cubra todos los aspectos del desarrollo de software y sirva para todos los tipos de programas y programadores. Sin embargo, hay importantes diferencias entre el desarrollo de aplicaciones para manejo de bases de datos y aplicaciones comerciales, sistemas para el control de procesos de plantas nucleares, sistemas flexibles de manufactura y de automatización y sistemas de telecomunicaciones, por mencionar algunos.

Los sistemas distribuidos han sido el resultado de una evolución del manejo de información centralizada, a un posterior esquema de servicio utilizando redes de computadoras. Un sistema distribuido consiste de una familia de componentes distribuidos, interactuando conceptualmente o espacialmente. Su comunicación se hace por medio de mensajes a través de una red, lo cual permite un trabajo conjunto. Igualmente los sistemas distribuidos permiten compartir recursos e información en áreas extensas. "El software de un sistema distribuido le permite a las computadoras coordinar sus actividades y compartir sus recursos de hardware, software y datos. Los usuarios de un sistema distribuido bien diseñado deben percibir una facilidad única e

integrada de cómputo a pesar de que ésta sea implementada por muchas computadoras en lugares diferentes" [Couloris 1994].

Algunas de las características de los sistemas distribuidos son:

- Ausencia de un estado global fácilmente perceptible por un observador
- Recursos compartidos a través de un sistema de comunicación
- Escalabilidad. El sistema funciona efectiva y eficientemente sin modificar su estructura aunque se añadan nuevos usuarios y nodos.
- Transparencia. El usuario debe percibir un sistema integrado.
- Tolerancia a fallas. El sistema debe seguir funcionando, a pesar de que uno de sus componentes falle.
- Confiabilidad. Control en la posibilidad de pérdida y modificación de los datos transmitidos a través de las vías de comunicación.
- Consistencia. Debido a que los diferentes procesos accesan y actualizan datos concurrentemente, dichos cambios deben aparecer igual a todos los otros procesos que integran el sistema distribuido.

Algunas de las ventajas más importantes de los sistemas distribuidos son [Shatz, Wang, 1989]:

- *Incremento del rendimiento*: El rendimiento está generalmente definido en términos de tiempos promedio de respuesta y de la carga (throughput). Si la capacidad de procesamiento se puede localizar donde es requerida, el tiempo de respuesta será altamente reducido.

- *Incremento en la disponibilidad*: Si el sistema distribuido es bien diseñado, nodos en buen estado pueden tomar las tareas de otros nodos que estén fallando. Esto significa que el sistema distribuido sigue trabajando con un rendimiento reducido pero sin perder su funcionalidad.

- *Incremento de la flexibilidad*: Funcionalidades adicionales pueden agregarse al sistema distribuido o el número de usuarios puede incrementarse permanentemente.

Es importante en sistemas distribuidos considerar los conceptos de procesos y programas concurrentes. Los programas concurrentes se caracterizan por tener un conjunto de instrucciones interrelacionadas por hilos de control múltiple, y que se ejecutan al mismo tiempo. Un proceso es una secuencia de instrucciones ejecutadas por uno o más hilos de control.

La siguiente figura (figura 2.1) muestra la relación entre hilos (procesos) y un proceso objeto (secuencia de instrucciones ejecutadas por uno o mas hilos de control):

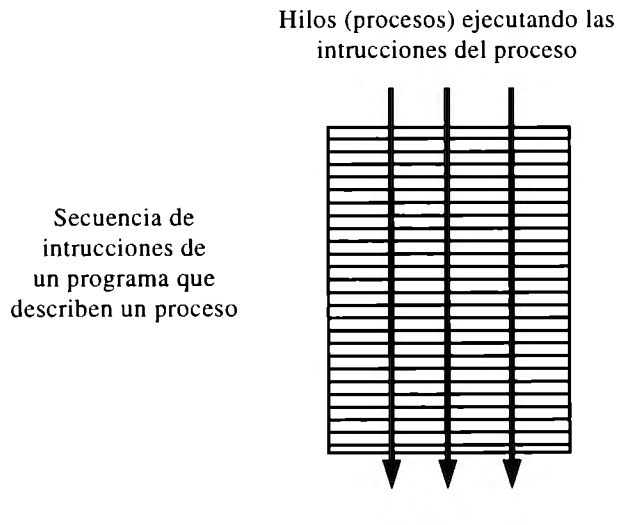


Figura 2.1 Hilos y procesos

Cuando se considera el software de ejecución sobre un sistema distribuido debemos distinguir entre sistemas de cómputo en red y sistemas de cómputo cooperativos [Fleischmann, 1994].

2.1 Sistemas de cómputo en red

2.1.1 Estructura de red

Las redes de computadoras son el medio necesario para la comunicación entre los componentes de un sistema distribuido. Dichos sistemas pueden ser implementados sobre **Redes de Área Local (LAN)** o **Redes de Área Amplia (WAN)**, las cuales a su vez funcionan sobre diferentes topologías, de las cuales las más comunes son : **bus, anillo, estrella, árbol y completa**.

Los sistemas de cómputo en red se caracterizan por la secuencia de trabajos que llegan independientemente a los nodos. Los trabajos se asignan e implementan más o menos en forma independiente los unos de los otros y están débilmente ligados. El sistema distribuido sirve inicialmente como una red para compartir recursos.

2.1.2 Paradigma Cliente-Servidor y el concepto de llamado a procedimientos remotos (Remote Procedure Call - RPC)

En un sistema distribuido normalmente se usa el paradigma cliente/servidor, en donde los programas servidores administran los datos y servicios compartidos entre los clientes. El servidor encapsula recursos, que son accesados por medio de operaciones invocadas por los clientes. La figura 2.1.2.1 muestra el concepto de los sistemas cliente-servidor.

El uso de hilos en un servidor ofrece acceso concurrente a los clientes. Cuando un servidor tiene múltiples hilos, debe asegurar la atomicidad de sus operaciones y su efecto sobre sus datos. Algunos servidores pueden incrementar la cooperación entre clientes haciéndolos esperar hasta que otro cliente ha suministrado un recurso necesario.

En algunos casos, como el de la transferencia de un flujo de datos a un cliente, la interacción entre un cliente y un servidor es más una conversación que una sencilla operación de invocación.

Cuando los clientes necesitan usar un servidor para almacenar datos por un periodo extenso de tiempo, el servidor debe garantizar que los datos sobrevivirán aún cuando él falle.

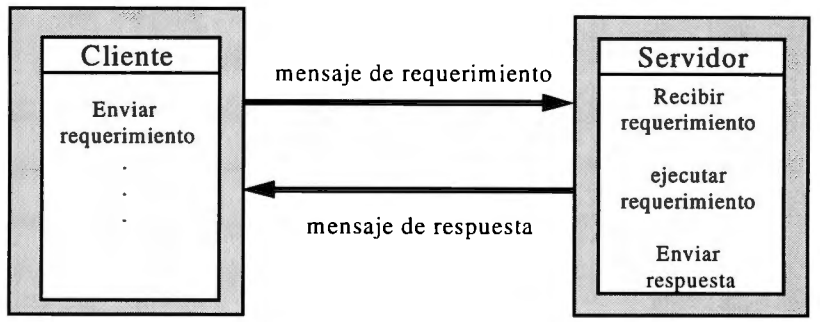


Figura 2.1.2.1. El concepto de los sistemas cliente-servidor

Muchas aplicaciones requieren transacciones entre el cliente y el servidor. Una transacción se define como una secuencia de operaciones del servidor que deben ser atómicas. Es decir, las transacciones deben garantizar atomicidad en la presencia de múltiples clientes y aún en caso de fallas del servidor.

El anidamiento permite estructurar las transacciones como conjuntos de otras transacciones. Las transacciones anidadas son unas herramientas muy poderosas en sistemas distribuidos debido a que permiten concurrencia y tolerancia a fallas.

Una forma muy común de compartir recursos es con un servidor de archivos. Todos los archivos están localizados en un nodo dedicado en el sistema distribuido. Componentes de software corriendo sobre otros nodos envían sus requerimientos de acceso al archivo al software servidor de archivos. El servidor de archivos ejecuta los requerimientos y regresa los resultados (al cliente). Desde el punto de vista del usuario un sistema cliente/servidor puede ser distinguido de un sistema central, porque un usuario no puede ver si un archivo está localizado sobre el sistema local o sobre un nodo servidor de archivos remoto. Para el usuario; el sistema cliente/servidor

aparece como un muy conveniente y flexible sistema central de cómputo. Los sistemas cliente servidor son muy flexibles. Para una nueva aplicación un nuevo servidor dedicado puede ser adicionado, esto es, los sistemas de bases de datos corren sobre servidores de bases de datos especializados los cuales tienen un tiempo de respuesta pequeño. La aplicación corriendo sobre el cliente, llama las funciones requeridas y que son suministradas por los servidores. Esto se hace principalmente con llamados a procedimientos remotos (RPC - Remote Procedure Calls). Un RPC se asemeja a un llamado a procedimientos excepto que éste es usado en sistemas distribuidos.

2.1.3 Ambiente de computación distribuida (DCE)

Un ambiente de computación distribuida es un conjunto integrado de herramientas que soportan computación en red en un ambiente de computación heterogéneo. Esta tecnología ha sido definida por la Fundación de Sistemas Abiertos (OSF) para soportar el desarrollo de aplicaciones distribuidas para redes de computadoras heterogénea. La siguiente figura 2.1.3.1 muestra la arquitectura OSF DCE:

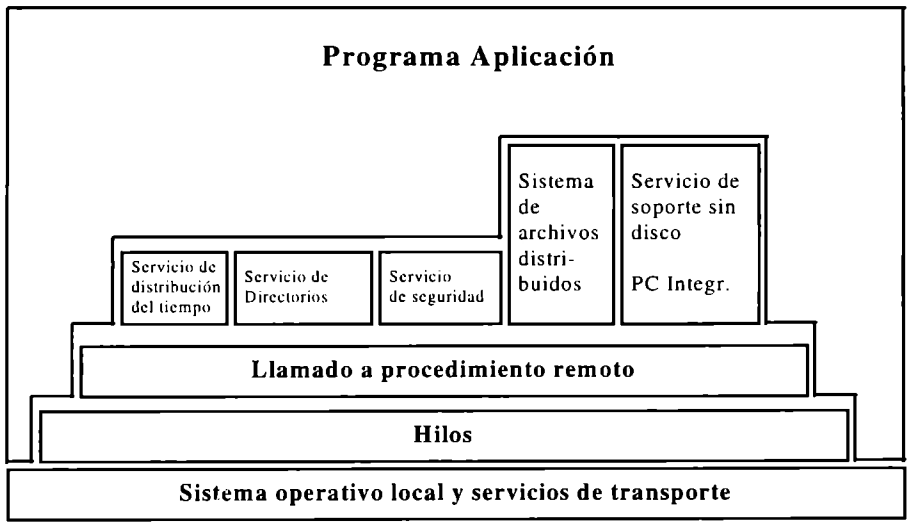


Figura 2.1.3.1. Arquitectura de OSF DCE

En DCE, los programas cliente y servidores son ejecutados por hilos, ésto es, procesos. Los hilos usan RPC's para comunicarse entre sí, y semáforos binarios y variables condicionales para sincronización. Los RPC's son soportados por directorios de servicio (DCE Call Directory Service) y servicios de seguridad (DCE Security Service). Los directorios de servicios mapean nombres lógicos a direcciones físicas. Si un cliente hace llamados a un proveedor particular de servicios, el directorio de servicios es usado para encontrar el servidor apropiado. El servicio de seguridad de DCE da capacidades para asegurar comunicación y acceso controlado a los recursos. El servicio de distribución de tiempo permite la sincronización de relojes en un sistema distribuido. Es requerido para eventos de login, recuperación de errores, etc. El servicio de archivos distribuidos permite compartir los archivos a través del mismo sistema. Finalmente el servicio de soporte sin disco permite a las estaciones de trabajo (workstations) usar archivos de disco en segundo plano (background) sobre servidores de archivos, como si fueran discos locales.

2.2 Sistemas de cómputo Cooperativo

En sistemas de cómputo cooperativo, un conjunto de procesos corren sobre diversos nodos de procesamiento. Estos nodos cooperan para lograr un objetivo común y en conjunto forman un programa distribuido. Esta es la diferencia con el esquema cliente-servidor. En sistemas cooperativos, las diferentes secciones de un programa se ejecutan sobre diferentes computadoras involucradas en un único programa; ésto se puede ver en el nivel de programación, pues esas secciones son ejecutadas concurrentemente. Estas diferentes secciones de un programa son también procesos. La realización de un objetivo común involucra un intercambio de datos y la sincronización de la ejecución. Los sistemas cooperativos son principalmente usados para la automatización de procesos técnicos, la implementación de software de comunicaciones, etc.

2.2.1 Software para los sistemas de comunicaciones

Un sistema de comunicaciones consiste de una red de comunicaciones y el software de comunicaciones que corre sobre los diferentes nodos de procesamiento. El software da los servicios de comunicación para el software de aplicación, el cual los usa para intercambiar

mensajes con el software de aplicación que está corriendo sobre otros nodos. El servicio de comunicación está basado en los niveles internos de la red (una red usualmente está compuesta de líneas y de diversos nodos de conmutación aunque algunas redes de área local no contienen nodos de conmutación). Para poder suministrar un servicio de comunicación conveniente existe un intercambio de mensajes entre los componentes de software de los sistemas de comunicación. La comunicación se basa en **protocolos**, que son un conjunto de reglas y formatos que permiten tener un método uniforme para la comunicación entre procesos en un sistema distribuido. El software de redes está organizado normalmente en una estructura jerárquica de capas. Hay una interfase entre cada par de capas y cada capa funcional suministra un conjunto de *servicios* a la capa arriba de ésta. Una capa está representada por un módulo en cada computadora conectada a la red.

El modelo para la interconexión de sistema abiertos (OSI) adoptado por la Organización Internacional de Estándares (ISO), es el modelo de referencia para asegurar estándares en el desarrollo de protocolos de acuerdo a los requerimientos de los sistemas abiertos [Couloris 1994].

El propósito de cada nivel o capa del protocolo es el siguiente:

CAPA	**DESCRIPCIÓN**
Aplicación	Protocolos designados para encontrar los requerimientos de comunicación de una aplicación específica, a menudo definiendo la interfase para un servicio.
Presentación	Transmisión de datos en una representación de red que es independiente de la usada en computadoras individuales. También se hace encriptamiento en esta capa.
Sesión	Se establece la comunicación entre procesos y realiza recuperación de errores .
Transporte	Direccionamiento de los mensajes a los puertos de comunicación
Red	Transferencia de paquetes de datos entre computadoras de una red determinada. Generación de ruteo.

CAPA	DESCRIPCIÓN
Enlace de datos	Responsable de la transmisión libre de errores de paquetes entre computadoras directamente conectadas.
Física	Son los circuitos y el hardware que manejan la red. Transmite secuencia de datos binarios como señales análogas, usando amplitud o modulación de frecuencia de señales eléctricas, señales de luz o señales electromagnéticas.

2.2.2 Sistemas de control en procesos técnicos

Otro ejemplo de computación cooperativa es un sistema distribuido de control de procesos técnicos. La estructura básica de los sistemas técnicos controlados por sistemas de cómputo es mostrada en la siguiente figura 2.2.2.1.

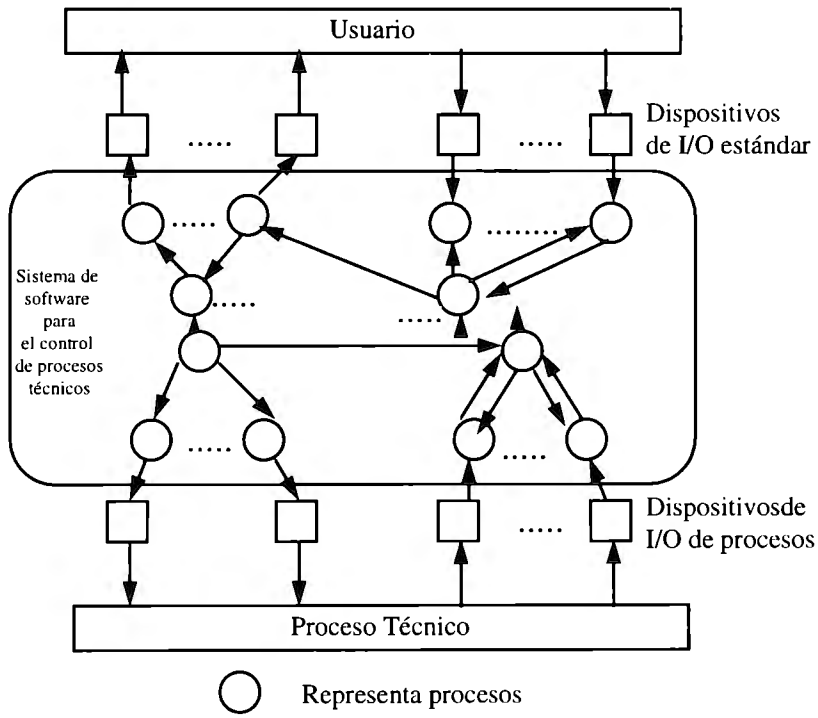


Figura 2.2.2.1. Estructura del sistema de control de procesos

La comunicación entre los sistemas de cómputo y los sistemas técnicos tiene requerimientos de tiempo real, así la comunicación con el usuario es más o menos orientada al dialogo con menos énfasis sobre condiciones de tiempo. Veremos las relaciones entre sistemas técnicos y sistemas de tiempo real.

Un sistema técnico consiste de diversas unidades funcionales mutuamente independientes con una comunicación vía interfases apropiadas con el sistema computacional. Por supuesto, el programa en tiempo real debe reaccionar simultáneamente a múltiples entradas. Esto implica la estructuración de un sistema de control de procesos que maneje un contador correspondiente al número de procesos. Cada proceso maneja un cierto grupo de señales.

Los requerimientos básicos para un sistema de control de procesos es la capacidad de seguir los cambios del sistema técnico tan rápido como sea posible. La información en el sistema de control de procesos debe estar estrechamente ligada con el estado del sistema técnico. Una forma fácil de hacerlo es diseñar un proceso para cada interfase. El sistema de procesos puede correr sobre un sistema centralizado o puede ser distribuido sobre diversos sistemas computacionales.

2.2.3 Intercambio electrónico de datos (Electronic Data Interchange - EDI)

El intercambio electrónico de datos (Electronic Data Interchange - EDI) es el intercambio de datos entre computadoras, entre compañias o internamente en las mismas, basado en ciertos estándares [DIGIT, 1990].

Los datos pueden ser estructurados o no estructurados. El intercambio de datos no estructurados permite usar estándares de comunicación específicos aunque los datos contenidos no estén en un formato estructurado. El más importante es el intercambio de datos estructurados. Algunos ejemplos de éstos son:

- **Intercambio de datos de negocio**: es usado principalmente para automatizar los procesos del negocio. Ejemplos de este tipo de intercambio incluye solicitudes de cotizaciones, ordenes de comprar, confirmación de ordenes de compra, etc.

- **Transferencia electrónica de fondos:** pagos de facturas, puntos electrónicos de venta (EPOS), etc., son algunos ejemplos.
- **Intercambio de datos técnicos:** el aprovechamiento de las comunicaciones puede jugar un gran papel en determinar los sucesos de un proyecto. Hay una gran cantidad de negocios que necesitan comunicación entre sus estaciones de trabajo de diseño asistido por computador (CAD) y las estaciones de trabajo de importantes vendedores.

2.2.4 Groupware

En las organizaciones; las personas trabajan persiguiendo un objetivo común. La interacción formal entre los miembros está establecida por estructuras y procedimientos. Adicionalmente existen interacciones informales muy importantes. Ambas interacciones se pueden soportar con computadoras. El trabajo cooperativo soportado por computadora (Computer Supported Cooperative Work - CSCW) trata con el estudio y desarrollo de sistemas computacionales llamados **groupware**, cuyo propósito es facilitar las interacciones formales e informales [Engelbart, Lehtman, 1988].

Los proyectos CSCW se pueden clasificar en cuatro tipos [Engelbart, Lehtman, 1988]:

1. Grupos geográficamente no distribuidos que requiere acceso común en tiempo real. Ejemplos: software para presentaciones, sistemas de decisión en grupo.
2. Grupos geográficamente distribuidos que requieren acceso común en tiempo real. Ejemplos: video conferencias, pantallas compartidas.
3. Colaboración asíncrona entre personas geográficamente distribuidas. Ejemplos: notas de conferencias, ediciones conjuntas.
4. Colaboración asíncrona entre personas geográficamente no distribuidas. Ejemplos: administración de proyectos, administración de horarios del personal.

El trabajo cooperativo (groupware) requiere de una red de computadoras. Por ésto, los sistemas de trabajo cooperativo son sistemas distribuidos. Los sistemas de software para trabajo cooperativo son combinaciones de redes y cómputo cooperativo.

2.2.5 Combinación de redes de cómputo y cómputo cooperativo

SOFTWARE DE EJECUCIÓN SOBRE UN SISTEMA DISTRIBUIDO	
COMPUTACIÓN EN RED	**CÓMPUTO COOPERATIVO**
ASPECTOS GENERALES	ASPECTOS GENERALES
SD implementados sobre: **Redes de Área Local (LAN)** o **Redes de Área Amplia (WAN)** Topologías más comunes de organización: **Bus, anillo, estrella, árbol y completa**.	**Conjunto de procesos corriendo sobre diversos nodos de procesamiento** La comunicación basada en **protocolos** de acuerdo al modelo OSI.
Uso del **paradigma cliente/servidor** Programas servidores administrando los datos y servicios. Clientes compartiendo los datos y servicios.	**Sistema distribuido de control de procesos técnicos** Comunicación en tiempo real
Uso de: **llamados a procedimientos remotos (RPC - Remote Procedure Calls)**.	**Intercambio electrónico de datos (Electronic Data Interchange - EDI)** Intercambio de datos entre compañías basado en estándares
Ambiente de computación distribuida (DCE). Conjunto integrado de herramientas que soportan computación en red en un ambiente de computación heterogéneo	**Groupware** Trabajo cooperativo y relaciones formales e informales basado en sistemas computacionales

Tabla 2.2. Aspectos generales de las redes de cómputo y cómputo cooperativo

La computación cooperativa puede ser combinada con sistemas cliente-servidor. Los procesos en un sistema distribuido pueden tener acceso a servidores. Desde el punto de vista de un sistema

cliente/servidor los procesos de un sistema cooperativo pueden ser considerados como procesos cliente.

La tabla 2.2 resume algunos de los principales aspectos de estos sistemas y da algunos ejemplos de los mismos.

3 ASPECTOS GENERALES SOBRE INGENIERÍA DE SOFTWARE

Diferentes tipos de programas necesitan diferentes ambientes de Ingeniería de Software, con métodos, herramientas, y técnicas de administración de proyectos, que dependen de las propiedades básicas del problema a solucionar. El objetivo de la Ingeniería de Software es proporcionar un ambiente que ayude a todos los aspectos del desarrollo de software y para todos los tipos de programas y programadores.

La implementación de un sistema es un proceso complejo que necesita la creación de un modelo intelectual para planear y controlar el proyecto. Por esto es de gran importancia contar con una metodología que suministre un esquema para la construcción paso a paso de tales sistemas. En general, un método de desarrollo identifica actividades a ser llevadas a cabo. Durante el desarrollo del sistema se producen diferentes descripciones del mismo y cada una de ellas puede reflejar diferentes niveles de abstracción.

Los sistemas distribuidos presentan diferentes problemas de diseño a los presentados por los sistemas centralizados. Un ambiente de Ingeniería de Software para el desarrollo de programas distribuidos deberá soportar procesos, su distribución en varias computadoras, y la comunicación y sincronización de los procesos.

A través de la evolución de la Ingeniería del Software, han surgido métodos para desarrollar sistemas concurrentes y sistemas distribuidos de software. En los sesentas, durante la construcción de los sistemas operativos, fueron propuestos nuevos métodos para la implementación de sistemas concurrentes [Hansen, 1973].

Dijkstra [Dijkstra, 1965][Dijkstra, 1968a] analizó problemas de comunicación y sincronización en programas concurrentes. Posteriormente, muchos conceptos de comunicación y sincronización para sistemas concurrentes se publicaron [Dijkstra, 1965], [Hoare, 1974], [Campbell, Habermman, 1974], [Hansen, 1978], [Hoare, 1978]. Paralelamente al desarrollo de estos conceptos, que permiten desarrollar lenguajes de programación para sistemas distribuidos, varios intentos se hicieron para describir sistemas concurrentes a un nivel más abstracto [Petri, 1962], [Keramidis, 1982], [Peterson, 1981], [Reisig, 1982], [Alford, 1977], [Alford, 1985a],

[Milner, 1980], [SDL, 1984], [ESTELLE,], [LOTOS]. Estos conceptos se han usado para definir los requerimientos de los sistemas concurrentes y son importantes para la implementación de sistemas operativos, sistemas de control de procesos, sistemas de comunicación, y sistemas de información en línea.

3.1 Modelo del ciclo de vida

La secuencia de pasos realizados desde el momento en que se concibe la idea de un sistema hasta que éste es realizado, es llamado el **Ciclo de Vida del Sistema**. Existen diferentes enfoques o ciclos que pueden ser aplicados, su diferencia está marcada en el nombre que se le da a ciertas actividades o fases. Algunos de estos ciclos de vida son : **ciclo clásico, ciclo de prototipos, el ciclo RAD, el ciclo incremental, ciclo de vida en espiral, ciclo de código reusable y el modelo de desarrollo concurrente**[Pressman, 1997].

3.1.1 Modelo del ciclo de vida clásico

La figura 3.1.1.1 ilustra el paradigma del ciclo de vida clásico para la ingeniería del software. Algunas veces llamado "modelo secuencial y lineal" ó "modelo en cascada", el paradigma del ciclo de vida clásico exige un enfoque sistemático y secuencial del desarrollo del software que comienza en el nivel del sistema y progresa a través del análisis, diseño, codificación, prueba y mantenimiento; actividades que se explican a continuación.

Paradigmas del ciclo de vida

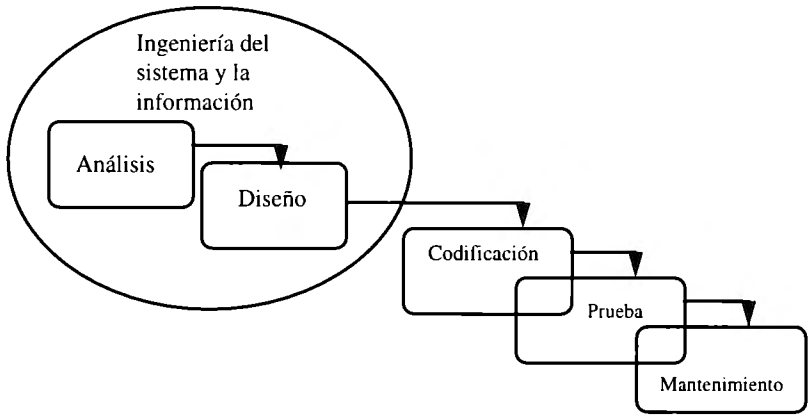


Figura 3.1.1.1 Ciclo de vida clásico

- **Ingeniería de información y del sistema:** Debido a que el software es siempre parte de un sistema mas grande, el trabajo comienza estableciendo los requerimientos de todos los elementos del sistema y luego asigna algún subconjunto de estos requisitos al software. Este planteamiento del sistema es esencial cuando el software debe interrelacionarse con otros elementos, tales como hardware, personas y bases de datos. La ingeniería del sistema y el análisis del mismo abarcan los requerimientos globales (a nivel del sistema) con una pequeña cantidad de análisis y de diseño a un nivel superior. la ingeniería de la información abarca los requerimientos globales a un nivel estratégico del negocio y al nivel de áreas del negocio.
- **Análisis de los requerimientos del software:** El proceso de recopilación de los requerimientos se centra e intensifica especialmente sobre el software. Para comprender la naturaleza de los programas que hay que construir, el ingeniero de software ("analista") debe comprender el ámbito de la información del software así como su función, el comportamiento, el rendimiento y las interfases requeridas. Los requerimientos, tanto del sistema como del software, se documentan y se revisan con el cliente.
- **Diseño:** El diseño del software es realmente un proceso de múltiples pasos que se enfoca sobre cuatro atributos distintos del programa: la estructura de los datos, la arquitectura del software, las representaciones de la interfase y el detalle de los procedimientos (algoritmos). El proceso de diseño traduce los requerimientos a una representación a nivel del software que

se establece para obtener la calidad requerida antes de que comience la codificación. Al igual que los requerimientos, el diseño se documenta y forma parte de la configuración del software.

- **Codificación:** El diseño debe traducirse en una forma legible para la máquina. El paso de la codificación realiza esta tarea. Si el diseño se realiza de manera detallada, la codificación puede realizarse mecánicamente.
- **Prueba:** Una vez que se ha generado el código, comienza la prueba del programa. La prueba se centra en la lógica interna del software, asegurando que todas las sentencias se han probado, y en las funciones externas, realizando pruebas que aseguren la no existencia de errores y que la entrada definida produce los resultados que realmente se requieren.
- **Mantenimiento:** El software, indudablemente, sufrirá cambios después de que se entregue al cliente. Los cambios ocurrirán debido a que se han encontrado errores, a que el software deba adaptarse a cambios del entorno externo (por ejemplo, un cambio solicitado debido a que se tiene un nuevo sistema operativo o dispositivo periférico), o debido a que el cliente requiere ampliaciones funcionales o del rendimiento. El mantenimiento del software aplica cada uno de los pasos precedentes del ciclo de vida al programa existente en vez de crear uno nuevo.

El ciclo de vida clásico es el paradigma más antiguo y más ampliamente usado en la ingeniería el software. Sin embargo, con el paso de unos cuantos años, se han producido críticas al paradigma, incluso por seguidores activos, que cuestionan su aplicabilidad a todas las situaciones.

3.1.2 Modelo del ciclo de vida de prototipos

Normalmente un cliente define un conjunto de objetivos generales para el software, pero no identifica los requisitos detallados de entrada, proceso o salida. En otros casos, el programador puede no estar seguro de la eficiencia de un algoritmo, de la adaptabilidad de un sistema operativo o de la forma en que debe realizarse la interacción hombre-máquina. En éstas y muchas otras situaciones, puede ser mejor método de ingeniería del software la construcción de un prototipo.

La construcción de prototipos es un proceso que facilita al programador la creación de un modelo del software a construir. El modelo tomará una de las tres formas siguientes: (1) un prototipo en papel o un modelo basado en PC que describa la interacción hombre-máquina, de forma que facilite al usuario la comprensión de cómo se producirá la interacción; (2) un prototipo que implemente algunos subconjuntos de la función requerida del programa deseado, o (3) un programa existente que ejecute parte o toda la función deseada, pero que tenga otras características que deban ser mejoradas en el nuevo trabajo de desarrollo.

La figura 3.1.2.1 muestra la secuencia de sucesos del paradigma de construcción de prototipos. Como en todos los métodos de desarrollo de software, la construcción de prototipos comienza con la recolección de los requerimientos. El técnico y el cliente se reúnen y definen los objetivos globales para el software, identifican todos los requerimientos conocidos, visualizan las áreas en donde será necesaria una mayor definición. Luego se produce un "diseño rápido". El diseño rápido conduce a la construcción de un prototipo. El prototipo es evaluado por el cliente/usuario y se utiliza para refinar los requerimientos del software a desarrollar. Se produce un proceso interactivo en el que el prototipo es "afinado" para satisfacer las necesidades del cliente, al mismo tiempo que facilita al que lo desarrolla una mejor comprensión de lo que hay que hacer.

Figura 3.1.2.1 Ciclo de vida de prototipos

3.1.3 El modelo RAD

El modelo de Desarrollo de aplicaciones en forma rápida (Rapid Application Develoment - RAD) es un modelo lineal de desarrollo de software que se fundamenta en un ciclo de desarrollo extremadamente corto. El modelo RAD es una variante del modelo clásico secuencial usando una construcción basada en componentes, es decir, la reutilización de software para desarrollar otras aplicaciones. En la figura 3.1.3.1 se muestran cada una de las fases que componen el modelo de desarrollo RAD y son las siguientes:

- **Modelado del negocio**: el flujo de información junto con las funciones del negocio se modelan de tal forma que responda a las siguientes preguntas: ¿Qué información manejan los procesos?, ¿qué información es generada?, ¿quién genera la información?, ¿a dónde va la información? y ¿quién procesa la información?.
- **Modelado de los datos**: los flujos de información definidos en la fase de modelado del negocio se refinan en un conjunto de objetos de datos necesarios para soportar el sistema. Entonces se identifican las características (llamadas atributos) de cada uno de los objetos y se definen las relaciones entre estos objetos.
- **Modelado de los procesos**: los objetos de datos definidos en la fase de modelado de los datos son transformados para obtener el flujo de información necesario para implantar una función. Se crean descripciones de los procesos para permitir adicionar, modificar, borrar o recuperar un objeto de datos.
- **Generación de la aplicación**: RAD asume el uso de técnicas de cuarta generación en lugar del uso de lenguajes de programación de la tercera generación, el método RAD se fundamenta en el reuso (cuando es posible) de componentes (programas) ya existentes o crea componentes reusables cuando es necesario. En todo se usan herramientas automatizadas para facilitar la construcción del software.
- **Evaluación y producción**: como el método RAD se basa en el reuso, muchos de los componentes de programación ya han sido evaluados por lo que el tiempo de evaluación se reduce. Sin embargo, los componentes nuevos deben ser evaluados al igual que todas las interfases.

Paradigmas del ciclo de vida

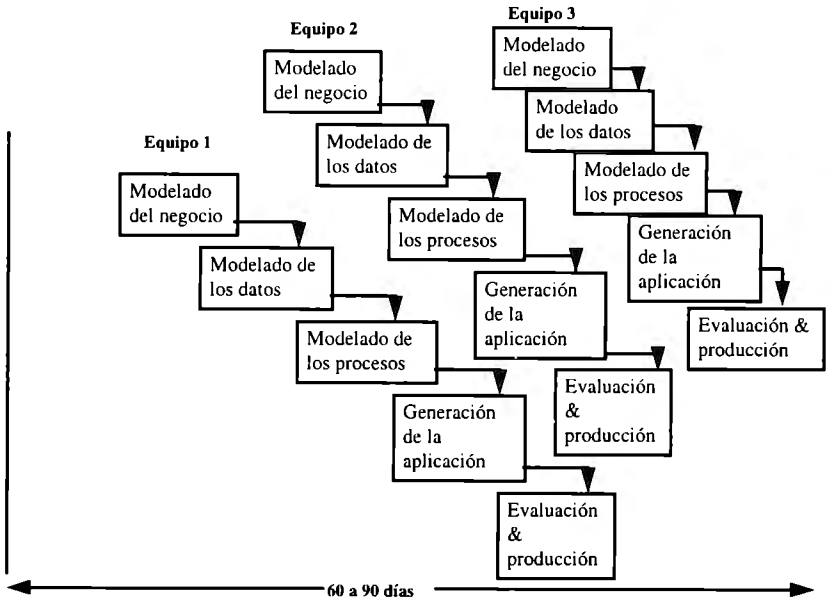


Figura 3.1.3.1 El modelo RAD

Si una aplicación puede ser modularizada en tal forma que cada función sea completada en menos de 3 meses, entonces es una aplicación candidata para desarrollarse bajo el modelo RAD y cada una de las funciones principales pueden ser programadas por un grupo de trabajo diferente para después integrarlas en una única aplicación.

3.1.4 Los modelos evolutivos para el desarrollo de software

3.1.4.1 El modelo incremental

Este modelo combina el modelo del ciclo de vida clásico (aplicado en forma repetitiva) con la filosofía iterativa del modelo de prototipos. La figura 3.1.4.1.1 muestra como el modelo incremental aplica una secuencia lineal de una manera escalonada cómo un progreso en el calendario de tiempo. Cada secuencia lineal produce un "incremento" del software o versiones más completas del mismo. También debe anotarse que para cualquier incremento se puede involucrar el paradigma de prototipos.

Cuando se usa el modelo incremental, el primer incremento es a menudo el producto base o debe dar como resultado un producto que contiene los requerimientos básicos, pero igualmente muchas características complementarias (algunas conocidas, otras desconocidas) permanecen aún sin formar parte del producto. Como un resultado de usar y/o evaluar el producto, se desarrolla un plan para realizar el siguiente incremento. Este proceso se sigue hasta que se obtiene el producto completo.

Paradigmas del ciclo de vida

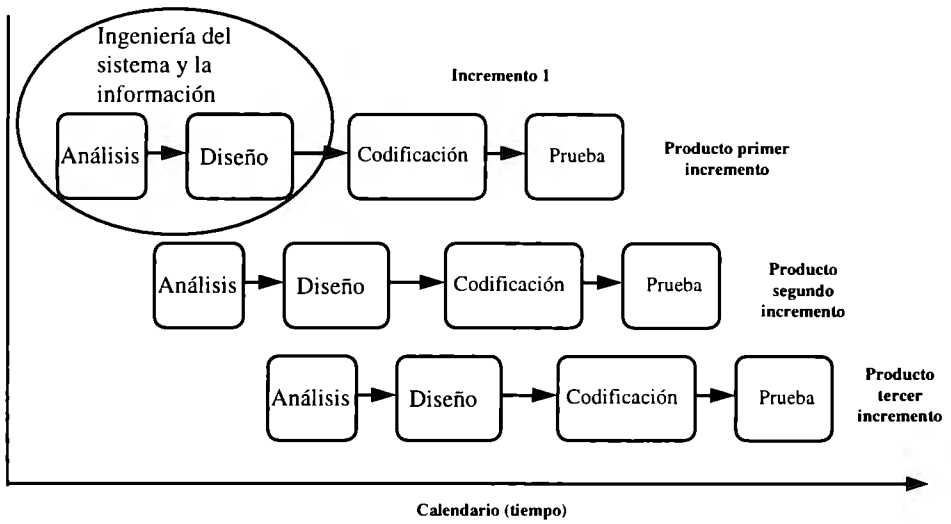


Figura 3.1.4.1.1 El modelo incremental

3.1.4.2 Modelo del ciclo de vida en espiral

El modelo en espiral [Boehm, 1988], es un modelo evolutivo para el desarrollo de software que une la naturaleza iterativa del modelo de prototipos con los aspectos sistemáticos y de control del modelo secuencial lineal. Permite el desarrollo rápido por medio de versiones de software en forma incremental. El modelo, representado mediante la espiral de la figura 3.1.4.2.1 es dividido en *actividades*, también llamadas *regiones de tareas* y son las siguientes:

- **Comunicación con el cliente**: son las tareas necesarias para establecer una comunicación efectiva entre el desarrollador y el cliente

- **Planeación**: son las tareas necesarias para definir recursos, lineamientos de tiempo y otra información relacionada con el proyecto
- **Análisis de riesgos**: son las tareas necesarias para evaluar riesgos, tanto técnicos como administrativos
- **Ingeniería**: son las tareas necesarias para construir una o más representaciones de la aplicación
- **Construcción y versión**: son las tareas necesarias para construir, evaluar, instalar y dar soporte al usuario (es decir, documentación y entrenamiento)
- **Evaluación del cliente**: son las tareas necesarias para obtener la retroalimentación del cliente basada en la evaluación de las representaciones del software creadas durante la etapa de ingeniería e implantadas durante la etapa de instalación.

Paradigmas del ciclo de vida

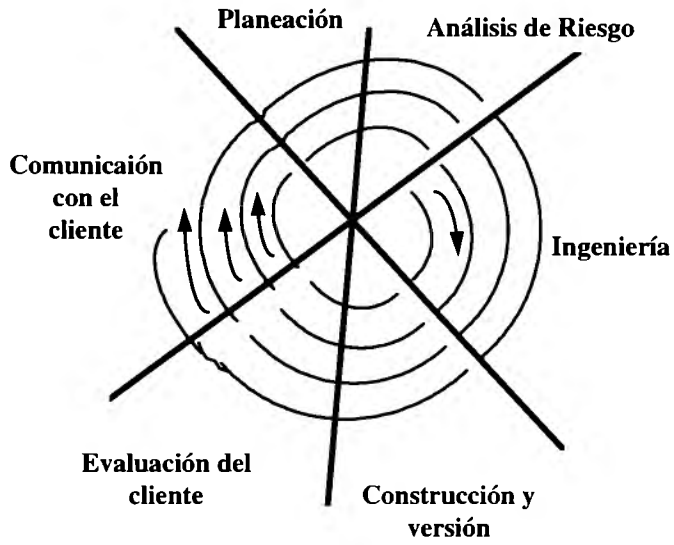


Figura 3.1.4.2.1 Ciclo de vida en espiral

El paradigma del modelo en espiral para la ingeniería del software es actualmente el enfoque más realista para el desarrollo de software y de sistemas a gran escala. Lo anterior se debe a que utiliza un enfoque evolutivo para la ingeniería del software, permitiendo al desarrollador y al cliente entender y reaccionar a los riesgos en cada nivel evolutivo.

3.1.4.3 Modelo basado en el ensamblaje de componentes

Las tecnologías orientadas a objetos son la base para un proceso de desarrollo basado en componentes. El paradigma orientado a objetos enfatiza la creación de clases que encapsulan los datos y los algoritmos que se usan para manipular dichos datos. Si las clases orientadas a objetos han sido diseñadas e implementadas adecuadamente, pueden ser reusadas para diferentes aplicaciones y arquitecturas de sistemas basados en la computadora.

El modelo de ensamblaje de componentes (figura 3.1.4.3.1) incorpora muchas de las características del modelo en espiral. Este es evolutivo por naturaleza, y demanda un ambiente iterativo para la creación del software. El modelo de ensamblaje de componentes crea aplicaciones a partir de componentes de software (algunas veces llamados "clases").

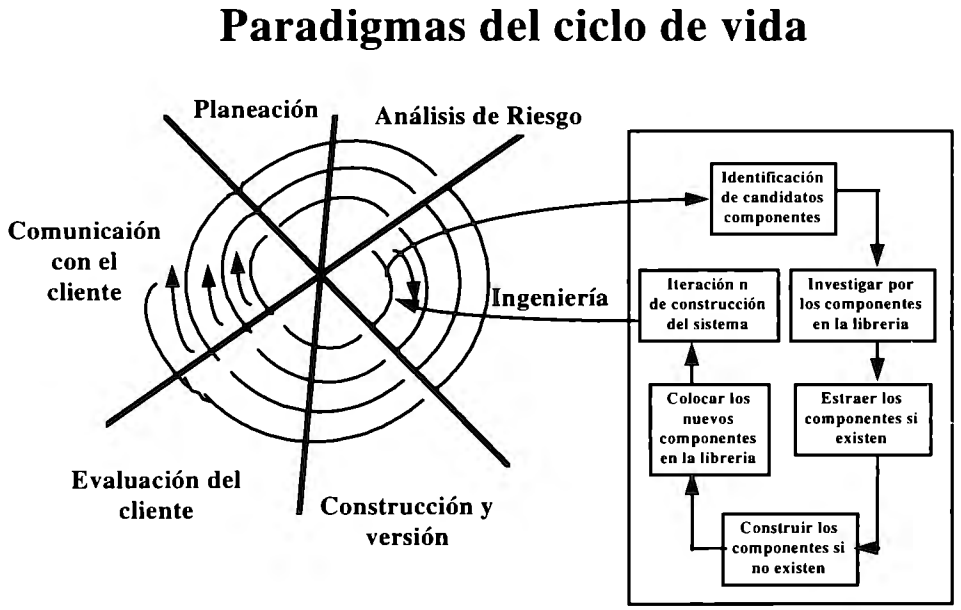


Figura 3.1.4.3.1 Modelo de ensamblaje de componentes

La actividad de ingeniería comienza con la identificación de las clases candidatas, lo que se hace examinando los datos que van a manipularse en la aplicación y los algoritmos que serán aplicados para la manipulación. Los respectivos datos y algoritmos son empaquetados en una clase.

Las clases (llamadas componentes) creadas en proyectos de ingeniería de software anteriores son almacenadas en *librerías de clases* o repositorios. Una vez que las clases son identificadas, la librería de clases es investigada para determinar si estas clases ya existen. Si es así, son extraídas de la librería y reusadas. Si una clase candidata no existe en la librería, se hace ingeniería usando métodos orientados a objetos. Entonces se compone la primera iteración de la aplicación a construir, usando las clases obtenidas de la librería y las nuevas clases construidas para cubrir las necesidades de la aplicación. El proceso entonces retorna a la espiral y finalmente regresa a la iteración durante las fases subsecuentes a través de la actividad de ingeniería.

3.1.4.4 Modelo de desarrollo concurrente

El modelo de desarrollo concurrente, algunas veces llamado ingeniería concurrente puede ser representado como una serie de actividades técnicas principales, tareas, y sus estados asociados. Por ejemplo la actividad de ingeniería definida por el modelo en espiral (sección 3.1.4.2), está basado en la utilización de las siguientes tareas: prototipado y/o modelado del análisis, especificación de requerimientos y diseño.

La figura 3.1.4.4.1 muestra el esquema de una actividad dentro del modelo concurrente. La actividad de análisis puede estar en uno de los estados mostrados en la figura en cualquier momento. Similarmente, otras actividades (por ejemplo, diseño o comunicación con el cliente) se pueden representar de manera análoga. Todas las actividades existen concurrentemente pero están en estados diferentes. Por ejemplo, inicialmente en un proyecto la actividad de comunicación con el cliente ha completado su primera iteración y existe en un estado llamado **espera de cambios**. La actividad de análisis (que está en el estado **ninguno,** mientras se completa la comunicación con el cliente) hace una transición al estado **bajo desarrollo**. Si el cliente indica cambios entonces la actividad de análisis se mueve de su estado **bajo desarrollo** al estado en **espera de cambios**.

Paradigmas del ciclo de vida

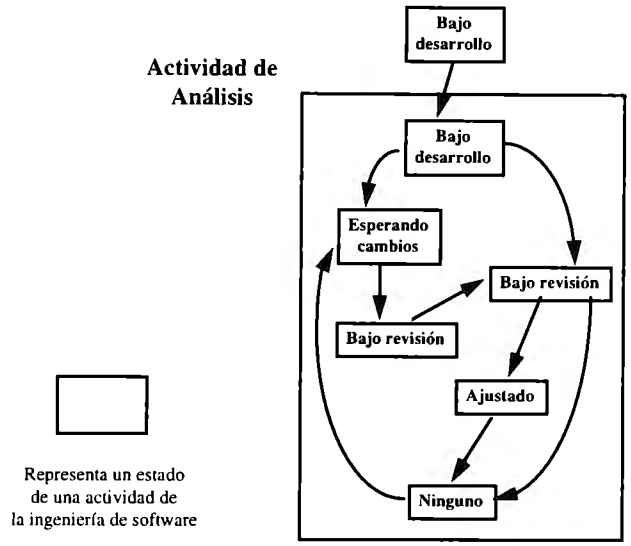


Figura 3.1.4.4.1 Un elemento del modelo concurrente

El modelo concurrente define una serie de eventos que podrían disparar transiciones de un estado a otro en cada una de las actividades de la ingeniería de software. Por ejemplo, si durante las primeras etapas del diseño, se descubre una inconsistencia en el modelo de análisis, se genera un *corrección en el modelo de análisis* que mueve la actividad de análisis del estado **ninguno** al estado en **espera de cambios**.

3.1.5 Combinación de paradigmas

Frecuentemente, se describen los paradigmas de la ingeniería del software, tratados en las secciones anteriores, como métodos alternativos para la ingeniería del software en lugar de como métodos complementarios. En muchos casos, los paradigmas pueden y deben combinarse de forma que puedan utilizarse las ventajas de cada uno en un único proyecto. El paradigma del modelo en espiral lo hace directamente, combinando la creación de prototipos y algunos elementos del ciclo de vida clásico, en un enfoque evolutivo para la ingeniería del software. Pero ninguno de los paradigmas puede servir como base en la cual se integren los demás.

La Fig. 3.1.5.1 muestra como pueden combinarse los tres paradigmas mencionados durante un trabajo de desarrollo de software.

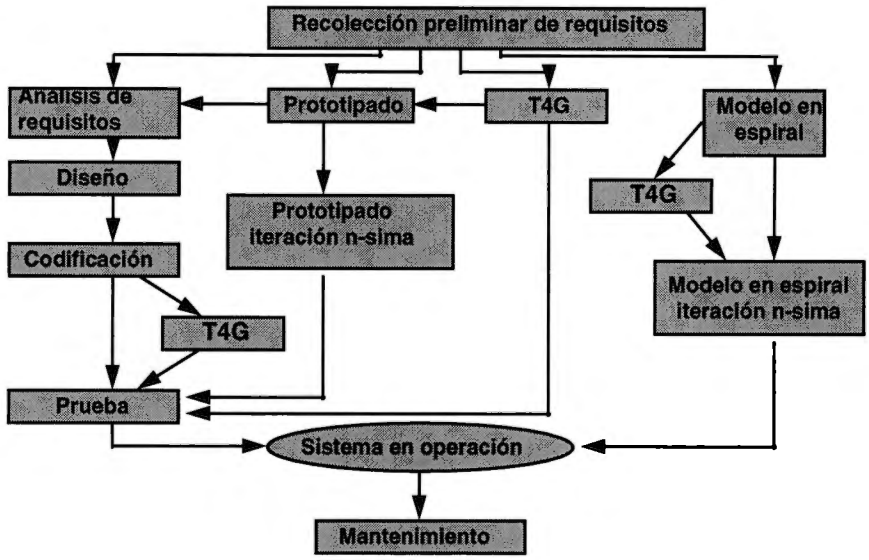


Figura 3.1.4.1 Combinación de paradigmas

El término "técnicas de cuarta generación" (T4G) abarca un amplio espectro de herramientas de software que tienen algo en común: todas facilitan, al que desarrolla el software, la especificación de algunas características del software a alto nivel. Luego, la herramienta genera automáticamente el código fuente basándose en la especificación del técnico.

Es importante recalcar que cualquiera que sea el ciclo de vida aplicado, todos ellos involucran una fase muy importante denominada **Fase de Definición de requerimientos.**

Existen dos tipos de requerimientos [Gracia-Catalin, 1985]:

- **Requerimientos funcionales**: describen un modelo de la solución del sistema. Incluye modelado de los estados internos relevantes y el comportamiento del sistema solución y su ambiente.
- **Requerimientos no funcionales:** describe la parte técnica, política, y comercial dentro de la cual la solución del sistema debe funcionar. Ejemplo: la elección de la base de datos a usar,

hardware, sistema operativo o lenguaje de programación a utilizar, ¿con qué sistemas existentes deberá cooperar. Técnicamente abarca consideraciones como portabilidad, interfase, rendimiento y disponibilidad. Del lado comercial establece que debe desarrollarse bajo un costo máximo dado.

La **especificación de requerimientos** es una herramienta valiosa para establecer en forma más clara las características que el sistema requiere. Por medio de esta especificación de requerimientos se protege a los analistas y diseñadores ya que se indica claramente lo que se espera del sistema y ayuda a los desarrolladores en su trabajo.

3.2 Diseño del software

Un aspecto complejo en el desarrollo de software, es que no siempre es muy clara la diferencia entre especificación y el diseño. Básicamente, durante el **proceso de diseño** del software se halla una solución en términos de software que está de acuerdo con los requerimientos definidos. El diseño envuelve la identificación de los componentes (subsistemas) y la descripción de las funciones de estos componentes. Se puede decir que hay tres tipos de actividades de diseño.

- **Diseño externo**: comportamiento externo, interfases, reportes, repositorio de datos, fuentes de datos.
- **Diseño arquitectural**: descomposición en partes funcionales y objetos de datos internos.
- **Diseño detallado**: estructura interna de los componentes hallados en el diseño arquitectural.

Se tienen tres ambientes para el diseño arquitectural y detallado:

- **Análisis funcional y de datos**: se tienen los datos que representan información y el conjunto de funciones que los manipulan. Algunas de las características de este análisis son:

- Puede empezar o bien con las funciones o bien con los datos
- Para identificar las funciones se utiliza normalmente un ambiente top-down
- En el análisis de datos, el punto de inicio son los objetos de datos del sistema y su estructura. La estructura de los datos define la estructura del programa.
- Después se definen las funciones que manipulan los datos.

- **Análisis orientado a objetos**: Basado en objetos: combinación de datos y funciones. Algunas características de este análisis son:

  - Un objeto describe los posibles valores y la estructura de un objeto de datos junto con las operaciones que pueden ser ejecutadas sobre este.
  - Una descripción que aplica a uno o mas objetos similares es llamada una clase de objeto.
  - Una nueva clase hereda cada propiedad de la clase a partir de la cual fue generada.
  - El diseño orientado a objetos sigue los siguientes pasos: se identifican los objetos y clases objeto de la solución, se definen las operaciones para cada objeto y clases objeto, se define la visibilidad de cada objeto para con los otros objetos, se define la interfase de cada objeto, y finalmente, cada objeto es implementado.

- **Diseño orientado a procesos**: descomposición del sistema en procesos.

La tabla 3.2.1 resume los tres tipos de actividades de diseño:

DISEÑO	
Diseño Externo	comportamiento externo, interfases, reportes, repositorio de datos, fuentes de datos.
Diseño Arquitectural	descomposición en partes funcionales y objetos de datos internos
Diseño Detallado	estructura interna de los componentes hallados en el diseño arquitectural.
Análisis funcional y de datos	datos representando información y el conjunto de funciones que los manipulan
Análisis orientado a objetos	Combinación de datos y funciones
Análisis orientado a procesos	Descomposición del sistema en procesos

Tabla 3.2.1. Resumen de las actividades básicas de diseño

3.3 Métodos

En ambientes de Ingeniería de Software, **los métodos** son requeridos para definir, describir, abstraer, modificar, refinar y documentar un producto de software. Para cada fase del ciclo de vida se usan métodos específicos. Los métodos usados en una fase del ciclo de vida deben permitir una fácil transición a los métodos usados en la siguiente fase. Una cadena de tales métodos es llamada una metodología.

En conclusión los métodos están basados en ciertos conceptos fundamentales y en lenguajes derivados de esos conceptos (figura 3.3.1). Finalmente, un método define un conjunto de procedimientos que dan la dirección y orden para obtener los resultados requeridos.

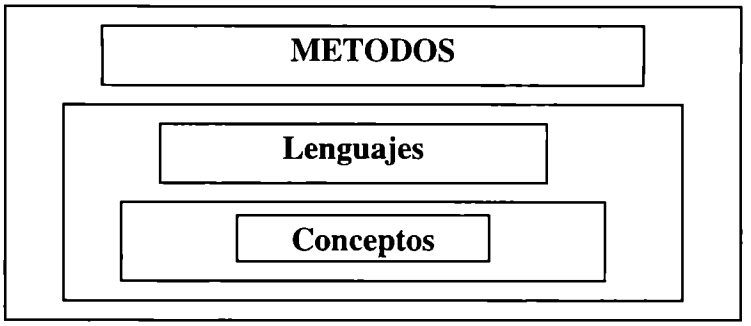


Figura 3.3.1. Fundamentos de un método.

En conclusión los métodos están basados en ciertos conceptos fundamentales y en lenguajes derivados de esos conceptos (figura 3.3.1). Finalmente, un método define un conjunto de procedimientos que dan la dirección y orden para obtener los resultados requeridos.

Varios métodos han sido desarrollados para cada una de las actividades (especificación de requerimientos, implementación, integración, y evaluación) en el desarrollo de software. Los conceptos básicos que cubren estos métodos, especialmente para la especificación de requerimientos y el diseño son descritos a continuación. [Jones, 1979], [Bergland, 1981], [Zave, 1982], [Balzert, 1989].

3.3.1 Orientado a funciones

Su objetivo es describir las funciones a ser ejecutadas. Normalmente se usa una técnica en lenguaje natural para describir los requerimientos del sistema. El principio "divide y conquista" es la base de este método. Las funciones de alto nivel se descomponen en otras de más bajo nivel, hasta que el programa es finalmente descrito. Es un ambiente de desarrollo top-down para la descripción y solución del problema. La siguiente figura 3.3.1.1 ilustra este principio:

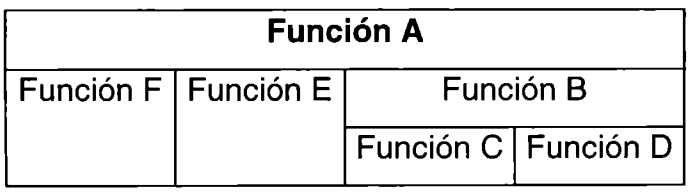


Figura 3.3.1.1. El principio de "divide y conquista".

En este método orientado a las funciones, se pueden distinguir dos tipos de abstracción [Liskov, Berzins, 1986]:

- **Abstracción procedural:** las entradas de datos se mapean para determinar su salida. Las abstracciones procedurales se implementan como procedimientos o subrutinas. La descripción de una abstracción procedural consiste de:

  - **Especificación de la interfase**, nombre del procedimiento, nombre del módulo, y el nombre y tipo de las fuentes y destinos de las entradas y salidas
  - **Especificación del comportamiento,** la cual puede ser descrita **informalmente** (usando lenguaje natural o diagramas) o **formalmente** (métodos usando lógica matemática). La especificación del comportamiento puede hacerse de dos formas:

    - **Especificaciones de entrada/salida:** describe el comportamiento implícitamente (describe los valores de los datos de entrada y salida antes y después de la ejecución del procedimiento)
    - **Especificaciones operacionales:** describe el comportamiento explícitamente (la transformación de la entrada en la salida es especificada por un programa que computa la función deseada)

- **Abstracción de datos:** consiste de una familia o tipo de dato abstracto de tipos de datos relacionados. Un tipo de dato abstracto es un conjunto de objetos y un conjunto finito de operaciones relacionadas para dichos objetos. Dos métodos de especificación formal son usados:

  - **Especificación axiomática:** define el comportamiento de un tipo de dato abstracto por medio de un conjunto de axiomas (los cuales describen las relaciones entre las operaciones de un tipo de dato abstracto)
  - **Métodos de modelo abstracto:** representa los objetos de un tipo de dato abstracto con otros tipos de datos definidos anteriormente.

3.3.1.1 Especificación funcional informal

Una forma muy común de representar la descomposición funcional de un sistema es usando un diagrama jerárquico (Figura 3.3.1.1.1 Diagrama de Descomposición Funcional - DDF). Cada nodo en el diagrama representa diferentes abstracciones procedurales. Su comportamiento es normalmente descrito usando una técnica informal orientada a la operación. Un lenguaje orientado a la operación para describir abstracciones procedurales es llamado un **lenguaje de diseño de programas** (Program Design Language - PDL), el cual incorpora construcciones procedurales básicas como secuencias, selección, e iteración combinadas con frases en lenguaje natural.

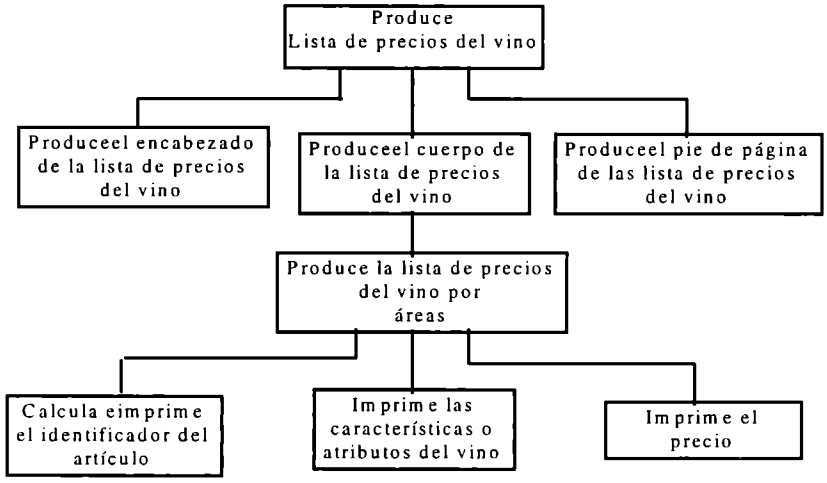


Figura 3.3.1.1.1. Diagrama de descomposición funcional (DDF).

3.3.1.2 Predicados de transformación

Es de la clase de especificaciones orientada a entradas y salidas. Su esencia es que la semántica de una instrucción está definida por transformaciones de estado. Una instrucción transforma un programa de un estado a otro. Un estado asocia los identificadores de un programa con valores. El conjunto de estados para los cuales una instrucción puede aplicarse está definido por precondiciones. La ejecución de una instrucción requiere que la precondición sea cierta. Los posibles estados después de la ejecución de la instrucción son descritos por postcondiciones -

después de la ejecución de la instrucción la postcondición es verdadera. La semántica de una instrucción está definida por las correspondientes precondiciones y postcondiciones. Si P es la precondición, S la instrucción y R la postcondición, su relación se escribe de la siguiente forma:

$$\{P\}\ S\ \{R\}$$

3.3.1.3 Especificación algebraica

En especificaciones algebraicas [Guttag, 1979] los tipos de datos abstractos se modelan como estructuras algebraicas. Una estructura algebraica consiste de un conjunto de objetos con su correspondiente conjunto de operaciones que tienen unas reglas definidas. La especificación de un tipo de dato abstracto consiste de la especificación de una interfase y la especificación de un comportamiento. La especificación de la interfase nos muestra la sintaxis y el tipo de información a evaluar: nombres, dominios y rangos de las operaciones asociadas con éste tipo. El comportamiento de las operaciones está definido por axiomas que especifican las relaciones entre las operaciones de un tipo de dato abstracto con otros y con el estado de estos tipos de datos abstractos.

3.3.2 Orientados a la estructura de datos

La estructura de los datos de entrada y salida forma la base de los métodos orientados a la estructura de datos. Un programa es aquel que transforma los datos de entrada en datos de salida. Dicho programa es derivado de las estructuras de datos.

Diferentes lenguajes y métodos han sido desarrollados con la idea de describir la estructura, atributos, uso y relaciones de los objetos de datos. Ejemplos son **Jackson Structured Programming (JSP)** [Cameron, 1989], **Warrier/Orr** [Warrier, 1974], [Orr, 1977], [Orr, 1981] y la técnica **Entity Relationship** [Chen, 1976].

3.3.3 Orientados al flujo de datos

Los más importantes métodos de análisis y diseño orientado al flujo de datos se describen en [DeMarco, 1979] y [Gane, Sarson 1979]. La descripción de un flujo de datos muestra el flujo de información a través de un sistema. El sistema acepta datos de entrada de su medio ambiente, transforma la información recibida y produce datos de salida. El ambiente del sistema de software es denominado su contexto. Los diagramas usados para este método se denominan diagramas de flujo de datos (DFD).

Algunas extensiones a los DFD permiten el desarrollo de sistemas en tiempo real. En [Ward, Mellor, 1985] y [Hatley, Pirbhai, 1987] son descritos métodos basados en flujos de datos para soportar desarrollo de sistemas distribuidos.

3.3.4 Orientados al flujo de control

La especificación de flujo de control describe el orden en el que diferentes acciones dentro de un sistema son ejecutadas y cómo éste reacciona a estímulos provenientes de su entorno. Esencialmente una especificación orientada al flujo de control define todas las secuencias de acción de un programa. Cada acción cambia el estado de un programa, es decir, los valores de sus variables. En cada estado un conjunto de acciones sucesoras es permitido.

Pueden distinguirse los siguientes sistemas de flujo de control:

- **Control simple:** son descritos por programas secuenciales. Algunas técnicas para describir éstos son las **máquinas de estado finitos** y algunas técnicas de descripción derivadas tales como las **máquinas de transición de estados**. Otra técnica a más bajo nivel son los diagramas **Nassi/Shneiderman** y los diagramas de **flujo de control**; éstos son principalmente usados para implantación [Pressman, 1987], [Marco, Buxton, 1987].

- **Control múltiple:** son descritos por programas en paralelo. Ejemplos de técnicas para éstos son **comunicación de procesos asincrónicos** y redes **Petri** [Reisig, 1982].

3.3.5 Orientados a objetos

En el desarrollo orientado a objetos los programadores representan el mundo real con abstracciones llamadas **objetos** [Booch, 1986]. Los objetos que poseen las mismas características pertenecen a una misma **clase objeto**. Los objetos son instancias de una clase, es decir son creados a partir de una clase. El desarrollo orientado a objetos se basa en los siguientes conceptos:

- ***Abstracción y encapsulamiento***: la abstracción se enfoca sobre como es visto un objeto desde el exterior, es decir como su comportamiento es visto por otros objetos y no como se implanta dicho comportamiento. Los detalles de la implantación son incluidos o encapsulados en el objeto, es decir los datos y las operaciones que se pueden ejecutar sobre los mismos se incorporan en el objeto.
- ***Herencia***: es una forma de lograr diferentes niveles de abstracción. En sistemas orientados a objetos existe una jerarquía de clases, aquellas que están en un nivel inferior son *subclases* de las clases del nivel superior (superclases). Las subclases heredan las estructuras de datos y operaciones de sus superclases pero pueden cambiar las operaciones heredadas o agregar nuevas estructuras de datos y operaciones.
- ***Polimorfismo***: es la habilidad que tienen diferentes objetos de responder adecuadamente a una operación que tiene un mismo nombre o identificador para todos. Por ejemplo, si se llama la operación dibujar sobre el objeto que hace líneas entonces, se dibujará una línea, pero si se llama la operación dibujar sobre el objeto que hace triángulos entonces se dibujará un triángulo.

La programación orientada a objetos ha llegado a ser una de las mas importantes para producir mejor software y también más reusable. Por esta razón, se verán a continuación algunos aspectos de este paradigma en relación con los sistemas distribuidos y concurrentes.

3.3.5.1 Sistemas concurrentes orientados a objetos

La concurrencia no es una de las propiedades normalmente asociadas a la programación orientada a objetos. Normalmente la invocación de una operación por medio de un mensaje tiene

semánticas similares a la de un llamado a procedimiento. El iniciador de la invocación de una operación espera hasta que el método asociado ha retornado una respuesta. Todos los cómputos son secuenciales. Sin embargo la estructura de los programas orientados a objetos implica la introducción de concurrencia.

Tres métodos pueden ser usados para introducir concurrencia [Barth, Welsch, 1988], [Wegner, 1987].

1. En general los objetos son entidades secuenciales. La concurrencia se introduce por medio de una clase especial llamada 'proceso'. Cada objeto de la clase proceso es ejecutada justamente por un hilo de control. Es posible que varios procesos invoquen operaciones o la operación de un objeto en forma concurrente. Este tipo de concurrencia es usada en SmallTalk .
2. En general, los objetos asociados con un proceso pueden ser ejecutados en paralelo. Tal objeto es llamado un actor. Toda entidad incluyendo procedimientos y datos es uniformemente representada como actores.
3. Más de un hilo de control puede ser ejecutado sobre un objeto al mismo tiempo. Cada operación de invocación puede resultar en la creación de un nuevo hilo de control, el cual realiza la operación invocada. Además de estos procesos creados dinámicamente pueden haber procesos ejecutando tareas en segundo plano (background).

En programas concurrentes orientados a objetos, los procesos deberán cooperar como normalmente lo hacen los sistemas concurrentes. Ésto se hace usando comunicaciones y métodos de sincronización.

3.3.5.2 Sistemas distribuidos orientados a objetos

Cada proceso en un sistema distribuido tiene su propio espacio de direcciones, no pudiendo tener acceso directo a recursos fuera de este espacio de direcciones [Wegner, 1987]. Como en programación concurrente, solamente objetos de una clase de procesos pueden ser ejecutados en paralelo. Los objetos que corren en un sistema distribuido utilizan los mismos conceptos de

comunicación y sincronización que los objetos concurrentes. Sin embargo la distribución de objetos presenta nuevos problemas.

En sistemas orientados totalmente a objetos, los parámetros de los mensajes también son objetos. Si la operación de un objeto localizado en otro lugar físico es invocada, los objetos y las definiciones de la clase relevante serán movidos de un sistema a otro.

Mover un objeto con su clase que lo define, significa que su contexto cambia. Sobre el nuevo computador (host) esta puede ser una clase padre con el mismo nombre, pero diferente protocolo. Esto puede significar que algunas operaciones no estén suficientemente definidas en la clase padre o si están definidas pueden tener un significado totalmente diferente.

En [Wegner, 1987] establece que la herencia, la cual es considerada típica en programación orientada a objetos es inconsistente con distribución. Normalmente el paradigma orientado a objetos es combinado con conceptos "clásicos" de programación concurrente.

3.3.6 Resumen de los diferentes conceptos

La tabla 3.3.2, resume cada uno de los conceptos básicos enunciados anteriormente y bajo los cuales se deben orientar los métodos.

MÉTODOS	
Orientado a las funciones	Su objetivo son las funciones a ser ejecutadas
Especificación funcional informal	Una forma muy común de representar descomposición funcional es el uso de un diagrama jerárquico (Diagrama de Descomposición Funcional - DDF)
Transformación de predicados	Su esencia es que la semántica de una instrucción está definida por transformaciones de estados. Una instrucción transforma un programa de un estado a otro.
Especificación algebraica	Una estructura algebraica es un conjunto de objetos con sus operaciones que tienen una reglas definidas. Este conjunto de reglas desde los axiomas de un tipo de dato abstracto.
Orientado a la estructura de datos	Lenguajes y métodos que han sido desarrollados con la idea de describir la estructura, atributos, uso y relaciones de los objetos de datos
Programación Estructurada de Jackson (JSP)	Es un lenguaje gráfico que permite la descripción de estructuras de datos y control. Cosiste de tres elementos básicos: secuencias, opciones e iteraciones.
Warrier/Orr	Basado en los diagramas de Warrier y permite la descripción jerárquica de estructuras de datos.
Diagrama Entidad-Relación	Permite describir la estructura de una base de datos usada por diferentes programas.
Orientado al flujo de datos	La descripción de un flujo de datos muestra el flujo de información a través de un sistema. Los diagramas usados para este método se denominan diagramas de flujo de datos (DFD).
Orientado al flujo de control	La especificación de flujo de control describe el orden en el que diferentes acciones dentro de un sistema son ejecutadas y como este reacciona a estímulos provenientes de su entorno
Control simple	Máquinas de estado finitos, máquinas de transición de estados, diagramas Nassi/Shneiderman, diagramas de flujo de control
Control Múltiple	Comunicación de procesos asincrónicos y redes Petri
Orientado a objetos	Representación del mundo con abstracciones llamadas objetos. Objetos con las mismas características pertenece a una clase. Método basado en abstracción, encapsulamiento, herencia y polimorfismo.
Sistemas concurrentes orientados a objetos	La concurrencia se introduce con la clase especial 'proceso'. Cada objeto de la clase proceso es ejecutado por un hilo de control. Los objetos pueden ser ejecutados en paralelo. Cada objeto está asociado con un proceso. Más de un hilo de control puede ser ejecutado sobre un objeto al mismo tiempo.
Sistemas distribuidos orientados a objetos	Objetos en paralelo tienen los mismos conceptos de comunicación y sincronización que los objetos concurrentes. Sin embargo la distribución de objetos presenta nuevos problemas

Tabla 3.3.2. Conceptos básicos cubiertos por los métodos.

4. MÉTODOS PARA EL DESARROLLO DE SISTEMAS DISTRIBUIDOS

En esta sección se presentan y muestran brevemente algunos métodos de especificación de sistemas. El propósito es mostrar varios estilos, más que describir los métodos en detalle, para determinar cuáles de ellas podrían adaptarse al área de los sistemas distribuidos.

Una *especificación* en general es un documento que puede servir como base para la ingeniería de hardware, la ingeniería de software, y la ingeniería en general. En él se describen las funciones y el alcance del sistema y los lineamientos que debe seguir la administración de su desarrollo. La especificación modela cada elemento componente del sistema, y también describe la información (datos o control) que conforman las entradas y salidas del sistema.

Los lenguajes de especificación o de programación concurrente pueden clasificarse según varios aspectos [Fleischmann, 1994]. Las características más importantes de los lenguajes de especificación y programación concurrente son:

- **La descomposición del sistema y la estructura de los procesos**: una de las principales características de todo lenguaje de especificación o programación es la posibilidad de descomponer un sistemas en un conjunto de sistemas más pequeños. Éstos incluyen formas de especificar las relaciones e interacción entre los diferentes subsistemas. Los conceptos de descomposición en programación estándar son las funciones, los tipos de datos y los objetos, los cuales pueden ser combinados en módulos.

  En programas cooperativos, el principal concepto de descomposición es el *proceso*. Los procesos cooperan para lograr un objetivo común. En un programa el número de procesos en paralelo puede ser estático o dinámico.

En una estructura de procesos estáticos, el número máximo de procesos así como su tipo e identificación son conocidos en el momento de la compilación. Esto no excluye que esos procesos sean creados o eliminados al tiempo de ejecución.

En una estructura de procesos dinámicos el número de procesos y la identificación de los mismos no es conocida en el momento de la compilación. Los procesos se crean durante la ejecución. El número de posibles procesos está restringido por los recursos disponibles, especialmente memoria.

- **Las características de comunicación**: los procesos concurrentes deben intercambiar información para coordinar sus actividades. Ellos pueden intercambiar información en forma directa o indirecta.

  En el **intercambio indirecto** de información, los procesos usan objetos. Las operaciones definidas sobre un objeto compartido pueden ser ejecutadas por diversos procesos. Los objetos compartidos en sistemas distribuidos pueden ser implementados con llamados a procedimientos remotos (RPC).

  En el **intercambio directo** de información no se usan objetos intermedios, los procesos se comunican por el paso explícito de mensajes.

- **Las características de sincronización** : Cuando los procesos están cooperando no solamente deben comunicarse sino también sincronizarse unos con otros. Para comunicarse un proceso con otro, un proceso envía un mensaje (una secuencia de ítems de datos) y otro proceso en el destino recibe el mensaje. esta actividad involucra la comunicación de datos desde el proceso que está enviando al proceso que lo está recibiendo y por lo tanto se necesita la sincronización de los dos procesos. Esto significa que los dos eventos "envío de un mensaje" y "recibir el mensaje" deben ocurrir en este orden. La sincronización puede ser vista como un conjunto de restricciones sobre el orden de los eventos, donde los eventos son operaciones sobre objetos compartidos o mensajes de operaciones.

De acuerdo a si la comunicación es directa o indirecta, los conceptos de sincronización pueden ser clasificados en: **control directo de los procesos, conceptos de sincronización para comunicación indirecta** y **conceptos de sincronización para comunicación directa**.

- **La descripción del comportamiento de los procesos**: Cada proceso tiene un conjunto de variables locales o posiblemente compartidas. Los valores de esas variables definen el estado de un proceso. Las instrucciones de un programa secuencial definen las transiciones, las cuales cambian el estado de los procesos.

  La ejecución de un programa en paralelo en un mismo procesador puede ser considerado como el intercalamiento de las secuencias de acciones atómicas de cada uno de los procesos componentes. Dos acciones que no se influyan una con la otra son ejecutadas concurrentemente y por lo tanto en cualquier orden. El posible orden está restringido por las definiciones de los diferentes procesos y las restricciones definidas por los métodos de sincronización usados. Por ejemplo, una operación de recepción no puede ser ejecutada antes de una operación de envío.

  Diversas técnicas han sido sugeridas para describir las secuencias permitidas de un proceso, es decir, la definición de un proceso objeto. De acuerdo a [Schneider, Andrews, 1986], los diferentes ambientes de especificación se basan en : **las secuencias de estados permisibles** o **el estado inicial y la secuencia de acciones atómicas permisibles**.

  Dada una secuencia de estados, la correspondiente secuencia de acción (atómica) puede ser construida y viceversa. Además el estado permisible o la secuencia de acción puede ser descrita **explícita** o **implícitamente**.

- **La asignación de tareas**: Durante el análisis y el diseño de un sistema, una tarea es descompuesta en partes más pequeñas. La asignación de tareas significa la asignación de procesos que han sido identificados durante las actividades de análisis y diseño, a

los nodos de procesamiento de un sistema distribuido o sobre el diseño de un sistema distribuido.

La comunicación entre procesos en un mismo procesador (intraprocesador) es menos costosa que la comunicación entre procesos de diferentes procesadores (interprocesador). Dos objetivos de la asignación de tareas son la minimización de los costos de comunicación y la optimización del balance de la carga, es decir, la asignación de aproximadamente la misma carga a cada procesador. Sin embargo estos dos objetivos pueden entrar generar conflictos. Para minimizar los costos de comunicación se podría asignar todos los procesos a un mismo procesador. Lo anterior podría llevar a un sistema completamente desbalanceado. Por otro lado un buen balance de la carga causa costos de comunicación entre los diferentes procesadores.

Dependiendo del periodo de tiempo sobre el cual es válido hacer una asignación se puede distinguir dos tipos de asignación de tareas, **estática** y **dinámica.** Asignación estática significa que la asignación de una tarea es válida durante todo el tiempo de ejecución. Cuando la asignación de la tarea es solo válida para un cierto periodo de tiempo, se usa la asignación dinámica de tareas. Las tareas pueden ser transferidas desde un nodo altamente cargado a otro no tan altamente cargado de acuerdo al actual balance de la carga del sistema.

La asignación de tareas es influenciada por diversos factores, por ejemplo, los métodos de comunicación y sincronización usados o si el número de procesos es estático o dinámico. Obviamente la asignación de procesos es más difícil si el número de procesos es variable. La creación de un nuevo proceso significa que se requiere una decisión respecto a qué procesador deberá ser asignado. Esto requiere que se deba conocer el balance de carga actual y futuro así como el costo actual y futuro de la comunicación entre los procesos.

La comunicación entre procesos es influenciada por el método de comunicación usado. Mecanismos de comunicación basados en memoria compartida pueden implementarse

eficientemente y son adecuados para comunicación dentro de un único procesador, pero pueden ser muy costosos para comunicación entre diferentes procesadores.

A continuación se describen algunos de los métodos de especificación existentes en ingeniería de software, buscando, como se enunció anteriormente, determinar cuáles de ellos podrían adaptarse al área de los sistemas distribuidos.

4.1 SADT (Técnica de Análisis y Diseño Estructurado)

Desarrollado por SofTech entre 1972 y 1975, es uno de los lenguajes gráficos más conocidos para expresar especificaciones. Cubre los requerimientos de análisis, el diseño y la documentación de especificaciones; suministra una buena comunicación entre analistas, desarrolladores y usuarios.

4.1.1 El método

El método **SADT** se enfoca sobre el flujo de datos e implica un refinamiento paso a paso de los así llamados Diagramas-SADT, los cuales están jerárquicamente ordenados. Los diagramas pueden ser de dos tipos:

- **Los actigramas** identifican a las funciones como elementos centrales de la descripción y suministra datos, esto es entrada o salida para las funciones.
- **Los datagramas** identifica a los datos como elementos centrales de la descripción y suministra funciones, esto es, entrada o salida para los datos.

La redundancia con estos diagramas, hace posible probar consistencia, es decir, se puede revisar si cada función en un actigrama está involucrada en algún datagrama y viceversa.

4.1.2 El lenguaje

SADT es un lenguaje de especificación gráfico que permite al usuario describir el sistema en términos de actividades y datos. Como se vió anteriormente hay actigramas (diagrama de actividad) que consisten de actividades y datos, donde las actividades se representan por cajas y los datos por flechas (figura 4.1.2.1). En un actigrama el lado izquierdo de la caja es usado para mostrar datos de entrada que serán transformados por la actividad. El lado derecho de la caja muestra los datos de salida, los cuales han sido transformados por la actividad. SADT también describe mecanismos de control y de soporte. La parte superior de la caja es usada para mostrar datos de control que sirven para restringir la operación de una actividad. El propósito principal de estos flujos de control es mostrar datos de entrada que no son transformados en una salida sino que son usados para modificar el comportamiento de una actividad. La parte inferior de la caja es usada para mostrar un mecanismo de soporte de la actividad, por ejemplo identificar el departamento, sección o inclusive el individuo que es responsable de la actividad ó que recursos físicos son necesarios para realizar la misma.

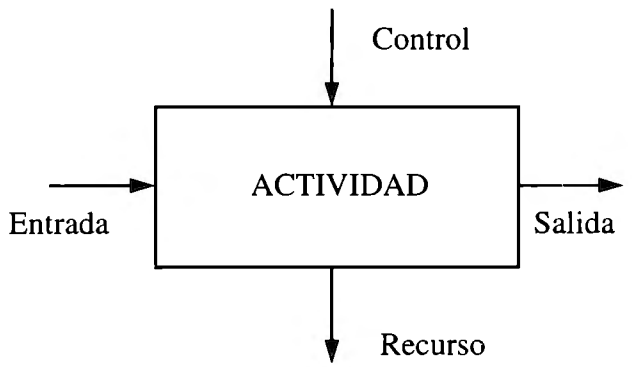


Figura 4.1.2.1 Típica caja SADT para Actigramas

Por otro lado hay datagramas, donde las cajas representan datos, mientras las flechas representan actividades. Para controlar la complejidad, el lenguaje restringe el número de cajas por diagrama SADT a siete.

4.2 Análisis Estructurado (SA)

SA fue desarrollado por [DeMarco en 1978]. Aparte del hecho de que el nombre es muy similar a SADT, solamente es común para ambos los flujos de datos como el principio central : SA está complementado con el Diseño Estructurado (SD), una técnica de diseño.

4.2.1 El método

El método permite al usuario modelar un sistema con **diagramas de flujos de datos (DFDs)** que contienen datos y procesos transformando los datos. Las llamadas **miniespecificaciones** son usadas para describir procesos en más detalle. Para refinar la estructura de datos, un **diccionario de datos (DD)** es obtenido. El comportamiento dinámico del sistema no puede ser expresado en la notación SA; por lo tanto, **diagramas en tiempo-real (RTDs)** son usados para este propósito. SA propone una descomposición paso a paso de los DFDs de tal manera que cada uno de los procesos en el DFD padre se descompone en diversos DFDs hijos.

SA propone básicamente dos pasos. El primero es desarrollar un diagrama llamado **diagrama de contexto**, para mostrar como el sistema está conectado con su ambiente. Aquí el usuario define la interfase en términos de fuentes y destinos del ambiente, procesos, flujos de datos y archivos.

En el segundo paso, el usuario divide y refina el sistema tanto como sea posible, esto es, cada proceso de un DFD es descrito en más y más detalle hasta cuando el nivel de un proceso atómico es alcanzado. Entonces se escriben **miniespecificaciones** demostrando la estructura algorítmica de estos procesos atómicos. También, se crea un **diccionario de datos** con la estructura de los datos. Existen también convenciones sobre la forma de nombrar los procesos, flujos de datos y archivos.

4.2.2 El lenguaje

Las fuentes y destinos pertenecientes al ambiente del sistema origen son mostrados como **cajas** en el diagrama de flujo de datos. Otros símbolos son **círculos** que representan procesos, **flechas** que representan flujos de datos y **barras** que representan archivos.

Las miniespecificaciones son escritas en pseudo-código, los datos descritos en el diccionario de datos están escritos en notación BNF.

4.3 Lenguaje de Descripción de Problemas/Analizador de Descripción de problemas (PSL/PSA)

PSL/PSA (Lenguaje de Descripción de Problemas/Analizador de Descripción de Problemas) fue el primer sistema basado en herramientas para especificación semi-formal. PSL inicialmente soporta análisis de requerimientos y documentación. Casi todos los sistemas actuales de Ingeniería de Software son variaciones de PSL/PSA.

4.3.1 El método

Se basa en cubrir conceptos como: ciclo de vida, desarrollo paso a paso, validación permanente. Algunos de sus métodos son: entrar la información inmediatamente, permitir descripciones informales, revisar exactitud, consistencia y ambigüedad, concentrarse en la información necesaria para una especificación; utiliza lenguajes semiformales, y diversas representaciones para las especificaciones (gráficas, tablas, etc.).

4.3.2 El lenguaje

PSL está basado en el modelo entidad-relación. Este modelo fue originalmente usado como un modelo de base de datos dividiendo el mundo a ser descrito en entidades y relaciones entre estas entidades.

A diferencia de SADT y SA, PSL es un lenguaje lineal (textual). PSL da algunas 30 clases de entidades y 75 relaciones para los usuarios. Los más importantes son:

Clases de entidades	**Descripción**
Entidades del mundo real	Objetos fuera del sistema origen
Procesos	Actividades
Entradas	Dato de entrada
Conjunto	Conjunto o elementos de datos

Relaciones	**Ejemplos/Descripción**
Generador	<process> GENERATES <data>
Receptores	<process> RECEIVES <data>
Actualizador	<process> UPDATE <data>
Estructurador	color CONSISTS amarillo, rojo, verde, azul Describe estructuras de datos

4.4 Metodología de Ingeniería de Requisitos del software (SREM)

SREM (Metodología de Ingeniería de Requisitos del Software) está basado en PSL/PSA. Soporta las primeras fases (análisis, definición, verificación y validación de requerimientos) del proceso de desarrollo de software. Es poderoso para el desarrollo de sistemas en tiempo real.

4.4.1 El método

Tiene dos importantes características no encontradas en la mayoría de los otros métodos o lenguajes para especificación. En primer lugar, permite el desarrollo paso a paso de especificaciones comenzando con descripciones informales, de las cuales se obtiene una especificación formal. En segundo lugar, los datos en la ejecución (estimados o requeridos) del sistema origen pueden ser formalmente incluidos en la especificación .

El método se aplica en siete pasos:

1. **Definición del núcleo:** identifica la interfase entre el sistema y el ambiente, y describe los flujos de datos y las unidades de procesamiento de datos dentro del sistema.
2. **Establecer la línea de base:** la primera descripción del sistema usando o bien el formalismo gráfico R-Net (R-Net significa requerimientos-red, una red estímulo-respuesta) o el lenguaje lineal RSL (requirements statement language).
3. **Definir datos:** para cada ALPHA (un componente activo que realiza una acción, un requerimiento de procesamiento, etc), define todos los datos de entrada y salida; completa y mejora la especificación RSL desarrollada; implementa procedimientos Pascal para ALPHAs.
4. **Agregar información del proyecto y establecer seguimiento:** agrega información de administración, esto es lineamientos, fechas claves, puntos claves, necesidades de herramientas etc.
5. **Simular funcionalidad:** prueba sintácticamente exactitud y simula comportamiento dinámico.
6. **Identifica requerimientos de ejecución:** define rutas de ejecución a evaluar, evalúa requerimientos de ejecución; cada camino deberá ser construido a tiempo de respuesta y exactitud.
7. **Demostrar viabilidad:** probar que el diseño actual es poderoso como para una realización

4.4.2 El lenguaje

SREM ofrece al usuario dos tipos de descripción, un lenguaje gráfico (R-Nets) (tabla 4.4.2.1) y un lenguaje textual (RSL).

Elementos: son tipos estándar definiendo características de cada objeto. Por ejemplo, MESSAGE, DATA y FILE, son tipos estándar usados para describir datos; esto es ALPHAs estándar para procesos. Los elementos representan sustantivos en el lenguaje.

Relaciones: expresa enlaces lógicos entre elementos, esto es <data> INPUT TO <alpha>. Ellos representan los verbos en el lenguaje.

Atributos: usados para completar la descripción de elementos, esto es <data> INITIAL VALUE <value>. Representan adjetivos en el lenguaje.

Estructuras: usados para definir la secuencia de pasos en los procedimientos y representan R-Nets, SUBNETs y VALIDATION-PATHs en términos de instrucciones RSL.

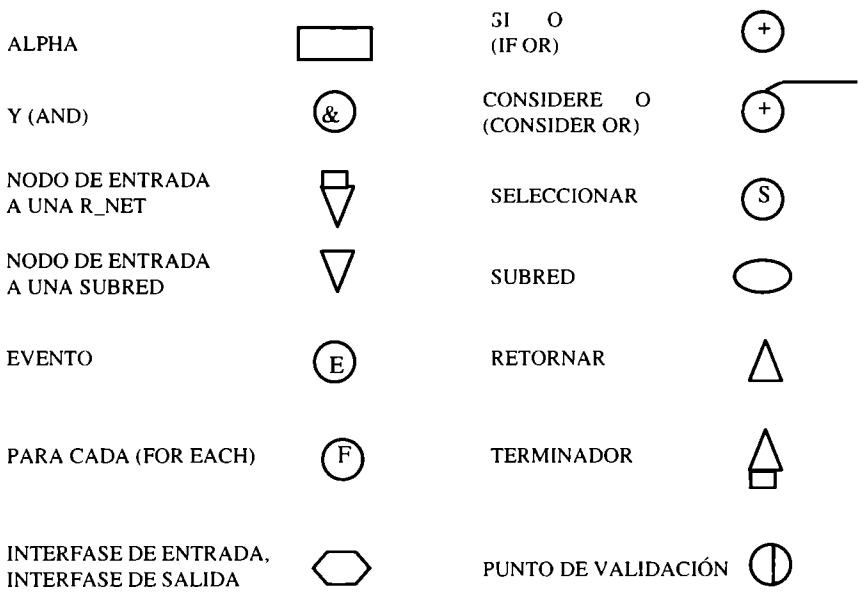


Tabla 4.4.2.1 Tipos y símbolos en SREM

4.4.3 Extensión a SREM

SREM ha sido extendido para solucionar los problemas de especificación del sistema a nivel de requerimientos y para definir un diseño distribuido [Alford, 1977], [Alford, 1985a]. Básicamente se extendió para definir requerimientos del sistema y asignarlos a unidades de procesamiento de datos. Esta extensión es conocida como Metodología de Ingeniería de Requerimientos del Sistema (System Requirements Engineering Methodology - SYSREM).

4.4.3.1 Conceptos de descomposición en SREM

En SYSREM un sistema es considerado como la transformación de entradas en salidas. La descomposición funcional es soportada en SYSREM. Un sistema es considerado como una jerarquía de funciones. Cada función transforma una entrada en una salida durante un periodo de tiempo determinado. Algunas funciones de bajo nivel operan concurrentemente y otras operan en una secuencia particular. Líneas sólidas en el gráfico representan el control del sistema. Líneas punteadas muestran la entrada y salida de las diferentes funciones.

4.4.3.2 Conceptos de comunicación y sincronización en SREM

La comunicación en SREM está basada en una combinación de las formas de control directo de procesos y el intercambio de mensajes. El operador "&" es comparable a las operaciones fork (permite a un proceso activar otros procesos y continuar la ejecución en paralelo con los mismos) y join (para sincronizar el proceso que esta llamando y el proceso que ha sido llamado, es decir detiene el proceso que llamo hasta que el proceso llamado ha finalizado). El control directo a los procesos está estrechamente relacionado con el sistema que se está estructurando. Puesto que en SREM la descomposición del sistema ya muestra la sincronización de los procesos. Las funciones en SREM, es decir, los procesos, reciben la entrada desde el ambiente o de otras funciones y producen una salida.

4.4.3.3 Descripción del comportamiento de los procesos en SREM

Como ya sabemos en SREM el sistema se descompone en funciones. Estas funciones se abren a su vez hacia un nivel inferior hasta un punto en que acepten mensajes discretos. En ese momento pueden abrirse aun más en un nivel de estimulo - respuesta. Las funciones aceptan mensajes y realizan la respectiva transición de estado. Esto significa que en SREM las especificaciones de funciones están basadas en un modelo de máquina de estados. Para cubrir el problema de representar un espacio de estados tan extenso se usa la notación llamada R-Nets. Una función puede ser especificada por varios R-Nets. Cada R-Net especifica la transformación de un mensaje único de entrada y su correspondiente estado en un mensaje de salida y su estado actualizado. Solamente una R-Net está activa en un instante determinado para una función.

4.4.3.4 Asignación de tareas en SREM

La descomposición del sistema permitida por SREM, permite al mismo tiempo la localización de tareas. Las funciones identificadas durante la descomposición del sistema son asignadas a procesadores. Es decir, en SYSREM la asignación de las tareas se hace mucho antes del desarrollo del sistema.

4.5 Lenguaje de especificación del ordenamiento temporal (Language Of Temporal Ordering Specification - LOTOS)

Utilizado para la especificación de sistemas distribuidos, LOTOS es una técnica de especificación formal desarrollada para definir estándares formales independientes de la implementación de servicios y protocolos del modelo estandarizado por la ISO de la OSI. Es usado para modelar el orden en el que los eventos de un sistema se llevan a cabo.

LOTOS se divide en 2 partes específicas, la primera ofrece un modelo de comportamiento derivado de las álgebras de proceso, principalmente de CCS (Calculus of Communicating Systems) así como de CSP (Communicating Sequential Processes) [De Alba, 1996]. La segunda

parte permite a los especificadores describir tipos de datos abstractos y valores. Esta parte está basada en el lenguaje de tipos abstractos ACT ONE.

Cada una de las partes que constituyen el lenguaje LOTOS tienen una función específica. El primer componente describe la parte de control, mientras que el segundo describe la parte de datos.

4.5.1 Conceptos de descomposición en LOTOS

En LOTOS [LOTOS] un único proceso puede ser descompuesto en subprocesos interactuantes. Estos procesos pueden ser aún más refinados en subprocesos. Una especificación en LOTOS es una jerarquía de definiciones de procesos.

En LOTOS un proceso no puede correr en paralelo con sus subprocesos. Los procesos pueden ejecutarse en paralelo; esto significa que en LOTOS las acciones de diferentes procesos son intercaladas.

Un proceso A puede activar otro proceso B, pero A debe satisfactoriamente completarse antes de que el proceso B pueda iniciar. El símbolo 'exit' es usado para marcar el lugar de terminación satisfactoria en un proceso. Los valores de las variables pueden ser pasados de un proceso activante a otro proceso activo.

Los procesos crean instancias de los subprocesos cuando el nombre de un subproceso aparece en la descripción de comportamiento del proceso que lo está creando. Cuando un proceso es creado por otro, se puede enviar valores de variables desde el proceso que lo está creando al que ha sido creado.

Un proceso puede desactivar a otro proceso. El proceso A puede ser interrumpido por el inicio del proceso B. El proceso A no resume la ejecución después de la interrupción; solamente el proceso B es continuado.

4.5.2 Conceptos de comunicación y sincronización en LOTOS

En LOTOS, un proceso se comunica con su ambiente por medio de interacciones. La forma atómica de una interacción es un evento. Ésta es una unidad de comunicación sincronizada que puede existir entre dos procesos. Los eventos en LOTOS pueden ser estructurados. Un evento estructurado consiste de un nombre de puerto identificando el punto de interacción y una lista finita de atributos.

Cuando un proceso crea un subproceso, se pueden pasar valores desde el proceso al subproceso creado; similarmente se pueden pasar valores desde un proceso activo a otro proceso activo. Los valores pasados pueden ser guardados en variables, las cuales pueden ser consideradas como compartidas. Los procesos que tienen una relación de creación o de activación no corren en paralelo; por esto, el acceso a estas variables no necesita ser sincronizado.

4.5.3 Descripción del comportamiento en LOTOS

En LOTOS, un proceso se ve como una caja negra que se comunica con su ambiente. El comportamiento observable es expresado por expresiones de comportamiento. Estas expresiones definen formalmente el orden en el que ocurren los eventos. Las expresiones de comportamiento están dentro de la definición del proceso. El formato de una definición de proceso es:

process <process_identifier><parameter_list> :=
<behavior_expression>
endproc

El identificador del proceso es el nombre por el que se referencia un proceso en la expresión de comportamiento de otro proceso. Si el nombre de un proceso A está dentro de la expresión de comportamiento de otro proceso B, entonces una instancia del proceso A puede ser creada por el proceso B.

Un ejemplo sobre expresiones de comportamiento podría ser la expresión a!(3+5);B, donde a!(3+5) es un evento y B es una expresión de comportamiento ya existente, y se interpreta como: primero ocurre el evento a!(3+5) y entonces ocurre la expresión de comportamiento B. Otro ejemplo sería una expresión de comportamiento alternativa como la siguiente: (B1 () B2) que indica que un proceso puede comportarse como B1 o B2.

4.6 SDL

[SDL, 1984] El **Lenguaje de Descripción y especificación (SDL)** es un lenguaje de especificación gráfico, formal y orientado a objetos. SDL fue desarrollado en los años setentas por la CCITT para diseñar el comportamiento interno y externo de los sistemas de conmutación telefónica. El SDL esta basado en máquinas de estado finito y puede ser usado para especificar las R-nets o secuencias de estimulo respuesta. La principal característica de SDL es que permite describir la estructura, el comportamiento y los datos de sistemas en tiempo real y distribuidos.

4.6.1 Conceptos de descomposición en SDL

En SDL un sistema puede ser considerado desde un punto de vista estático o dinámico. Los bloques representan la estructura estática de un sistema, mientras los procesos representan el comportamiento del sistema. Cada bloque puede contener uno o más procesos.

Un sistema esta compuesto de un número de bloques conectado por canales. Un canal es una ruta de transporte unidireccional de señales. Un definición del canal contiene una lista de todas las señales que puede ser enviadas a través de éste.

Un bloque puede ser dividido en subbloques y canales. Si un bloque contiene subbloques, éstos son también conectados por canales. Los canales que llegan a un bloque pueden separarse dentro de subcanales dentro de este. Los subcanales están conectados por subbloques. La lista de señales de todos los subcanales derivados desde un mismo canal deben ser disjuntas y la unión de

todos los subcanales deberá coincidir con la lista de señales del canal desde el cual están derivados.

Cada bloque puede contener subbloques o uno o más procesos. Las definiciones de los subbloques deben incluir por lo menos las definiciones de los subprocesos resultantes de la división de los procesos en el bloque para el cual ellos comenzaron.

Los procesos pueden ser creados y terminados dinámicamente. Un proceso puede ser creado por otro proceso o éste puede ya existir desde el tiempo de inicialización del sistema. Los procesos corren en paralelo sin restricciones especiales, no como en Estelle donde un proceso hijo puede solamente correr cuando su proceso padre está bloqueado.

4.6.2 Conceptos de comunicación y sincronización en SDL

En SDL la comunicación entre procesos es por señales, es decir, mensajes. Las señales en SDL tienen nombres, es decir, el identificador de la señal, y puede contener parámetros, es decir, los parámetros de la señal.

Cuando un proceso envía una señal, los valores de los parámetros de la señal son identificados con los valores de las variables locales y la señal es transportada hacia el proceso destino. El proceso que ejecuta una acción de envío nunca es bloqueado. Las señales se almacenan en forma FIFO en una cola externa para los procesos a los cuales ellas son dirigidas, hasta que el proceso está listo para recibir la señal, es decir se usa la comunicación asincrónica. Una señal es eliminada cuando el proceso destino recibe la señal. Cuando la señal es recibida, los valores de los parámetros del mensaje son copiados en la variables locales del proceso que está recibiendo.

En SDL las señales no son persistentes. Si la señal no es esperada en un estado particular ésta es eliminada.

En SDL hay otras formas diferentes a las de señales para intercambiar datos entre los procesos. Un proceso puede leer los datos de otro proceso si estos procesos pertenecen al mismo bloque y los datos han sido declarados como compartidos (share value).

Un proceso puede accesar datos de otro proceso que este en otro bloque, sólo si los datos han sido declarados como "exportable", o sea, globales. El acceso a valores compartidos y exportables no puede ser sincronizado.

4.6.3 Descripción del comportamiento de los procesos en SDL

En SDL el comportamiento de un proceso se puede representar gráficamente o en forma de texto. La representación gráfica es más fácil de entender. La figura 4.6.3.1 muestra los símbolos que se usan para describir el comportamiento de los procesos.

Los símbolos son conectados por arcos. La conexión entre símbolos tiene algunas restricciones, por ejemplo el símbolo guardar puede solamente estar conectado a un símbolo de estado. Los parámetros formales de señales, definición de variables, etc., no se muestran en la gráfica del proceso sino usando un lenguaje llamado SDL/P. En esencia el comportamiento de los procesos es mostrado con secuencias de estímulo-respuesta

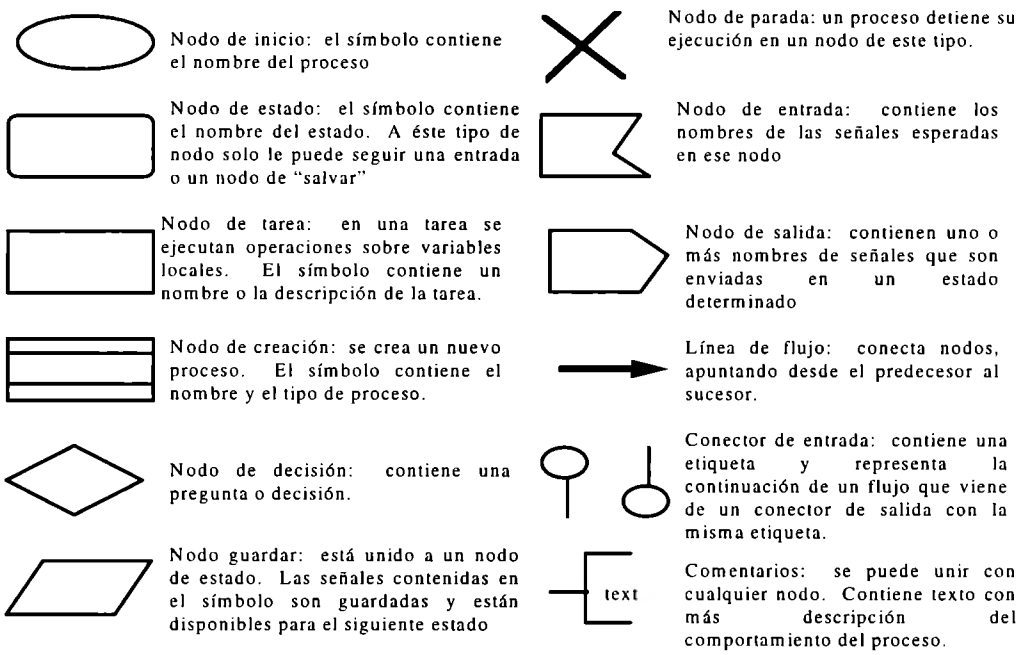


Figura 4.6.3.1 Bloques básicos de construcción en SDL

4.7 ESTELLE

4.7.1 Conceptos de descomposición en Estelle

En Estelle [ESTELLE], [Linn, 1986] una especificación puede estar autocontenida en un módulo o ser dividida en un conjunto de módulos anidados. Un módulo definido dentro del alcance de otro módulo es solamente visible dentro del alcance del módulo padre. Un módulo anidado dentro de otro módulo es llamado un módulo hijo. En Estelle hay dos tipos de módulos: **procesos** y **actividades**.

Los procesos pueden correr en paralelo con otros procesos del mismo nivel jerárquico. La ejecución de actividades de un mismo nivel debe ser intercalada. Los módulos no pueden correr en paralelo con su módulo padre y un módulo padre tiene preferencia sobre sus hijos. Ésto

significa que cuando un módulo ejecuta transacciones, todos sus módulos hijos son bloqueados. El itinerario de los módulos de un mismo nivel es no determinista.

4.7.2 Conceptos de comunicación y sincronización en Estelle

En Estelle, los módulos se comunican vía mensajes, llamados interacciones. Los mensajes pueden ser directamente intercambiados entre el módulo padre y su hijo y entre procesos en el mismo nivel de jerarquía que tengan el mismo padre.

Los procesos que desean interactuar unos con otros deben ser conectados explícitamente, por medio de operaciones como "connect" y "desconnect". Estas operaciones se ejecutan en módulos ordenados en forma jerárquica; un módulo puede ser conectado a su hijo más inmediato y a otros descendientes.

Estelle sólo permite intercambio de mensajes en forma asíncrona. Una cola FIFO infinita es usada para transferir mensajes entre módulos.

4.7.3 Descripción del comportamiento de los procesos en Estelle

Estelle usa un modelo de máquina de transición de estado finito para modelar la dependencia de las salidas con respecto a las entradas. Iniciando desde un estado inicial definido, los módulos ejecutan transiciones de un estado a otro. Cada transición está definida por:

- el estado principal actual (cláusula "from")
- la interacción de la entrada (cláusula "when")
- el predicado que está activando (cláusula "provided")
- la condición de espera (cláusula "delay")
- la prioridad (cláusula "priority")

- una lista de acciones a ser ejecutadas incluyendo la generación de la interacción de la salida, la cláusula "Begin/End"
- el estado principal siguiente, la cláusula "to"

Todas las cláusulas son opcionales. Sintácticamente, una transición o un conjunto de transiciones se indica por medio de la palabra **trans**. Por ejemplo:

trans
priority expression
from current_major_state
to next_major_state
provided predicate
when received message
begin code **end**

Si varias transacciones ocurren para un mismo módulo, se escoge la transacción con la más alta prioridad. Transacciones con la cláusula "when" son llamadas transacciones de entrada y transacciones sin esta cláusula son llamadas transacciones espontáneas. Las transacciones espontáneas pueden tener una cláusula de tiempo (delay), lo cual no es posible en transacciones de entrada. Una transacción con una cláusula 'delay', es decir 'delay (d1,d2)', puede activarse cuando la condición que la esta activando se ha cumplido continuamente para la unidad de tiempo d1 y deberá ser activada en las siguientes unidades de tiempo d2.

4.8 Relación entre los métodos y los conceptos descritos

La tabla (tabla 4.1), resume cual es la relación existente entre los conceptos básicos bajo los cuales se debe orientar un método o metodología (tabla 3.2) y los conceptos de clasificación descritos arriba. Algunos de los conceptos listados en la tabla ya hacen referencia a algunos de los métodos usados para el desarrollo de sistemas concurrentes y distribuidos descritos arriba.

Concepto	Orientado a las funciones	Orientado a la estructura de datos	Orientado al flujo de datos	Orientado al flujo de control	Orientado a objetos
Conceptos de descomposición en SREM.	X				
Conceptos de descomposición en SDL	X				
Conceptos de descomposición en ESTELLE	X				
Intercambio Indirecto de información		X			
Llamado a procedimientos remotos		X			
Sincronización			X		
Especificación del comportamiento de los procesos. Marco general.				X	
Especificación explícita orientada al estado del comportamiento de un proceso				X	
Especificación explícita orientada a la transición del comportamiento de un proceso				X	
Asignación de tareas					X

Tabla 4.1. Conceptos básicos Vs Orientación del método

En la tabla 4.2 se muestra en forma global si los métodos referenciados aplican o no para cada una de las anteriores características.

Métodos	Métodos de especificación para sistemas concurrentes	Métodos de especificación para descomposición del sistema y estructura de los procesos	Métodos de especificación para comunicación y sincronización	Métodos de especificación del comportamiento de los procesos	Métodos de especificación de la asignación de tareas
SADT		A			
SA		A			
PSL/PSA		A			
SREM (SYSREM)	A	A	A	A	A
SDL	A	A	A	A	NA
Estelle	A	A	A	A	A
LOTOS	A	A	A	A	A
Mascot	A	A	A	A	NA
SDRTS	A	A	A	A	NA
SRTD	A	A	A	A	NA
DARTS	A	NAC	A	A	NA
PAISLey	A	A	A	A	NA

A=Aplica, NA= no aplica, NAC= no aplica completamente

Tabla 4.2. Aplicabilidad de los métodos

La tablas 4.3 a, b, c, d y e, indican en forma detallada los mecanismos que usa cada uno de los métodos para cubrir las características descritas al inicio de este capítulo, o sea : la descomposición del sistema y la estructura de los procesos, la comunicación, la sincronización, la descripción del comportamiento de los procesos y la asignación de tareas.

	SREM	SDL	LOTOS	Estelle	Mascot	SDRTS	SRDT	DARTS	PAISLey
Permite jerarquización	X	X	X	X	X	X	X	X	
Por medio de:									
Funciones	1								
Tipos de datos									
Objetos									
Bloques		1							
Subbloques		2							
Procesos		3	1	2					1
Subprocesos			2						
Actividades				2	3				
Transformaciones						1	1	1	
Módulos (combinación de funciones, tipos de datos y objetos)				1					
Tareas								1	
Sistemas					1				
Subsistemas					2				
Módulos (llamados raíces)					4				
Elementos de descomposición enlazados por:									
Entradas y salidas	X								
Canales		X							
Flujos de datos						X	X		
Flujos de control							X		
IDA					X				
Permite mostrar:									
Paralelismo o concurrencia	X	X	X	X	X	X		X	X
Secuencialidad	X							X	X

Tabla 4.3 (a). Mecanismos para descomposición y estructura de los procesos

Los números (1, 2, 3 ....) indican una jerarquía en la forma de descomponer

	SREM	SDL	LOTOS	ESTELLE	Mascot	SDRTS	SRDT	DARTS	PAISLey
Por intercambio Indirecto de información									
Objetos compartidos (o sea, estructuras de datos y las operaciones necesarias)		X	X	X	X	X	X	X	
Por comunicación Directa									
Paso de mensajes	X					X	X	X	X
Paso de mensajes llamados señales		X							
Paso de mensajes llamados interacciones			X	X					
Forma:									
Asíncrona		X		X				X	X
Síncrona			X		X	X	X	X	X

Tabla 4.3 (b). Mecanismos para la comunicación

	SREM	SDL	LOTOS	ESTELLE	Mascot	SDRTS	SRDT	DARTS	PALSLey
Sincronización por Control Directo de los Procesos	X					X	X		
Sincronización por comunicación indirecta									
Sincronización orientada a los procesos									
Sincronización orientada a los recursos					X				
Sincronización por comunicación directa									
Paso de mensajes en forma asíncrona		X	X	X				X	X
Paso de mensajes restringido en forma asíncrona									
Paso de mensajes en forma						X	X	X	

Tabla 4.3 (c). Mecanismos para la sincronización

	SREM	SDL	LOTOS	ESTELLE	Mascot	SDRTS	SRDT	DARTS	PAISLey
Especificación explícita del comportamiento									
Secuencia de estados	X			X	X	X	X	X	X
Secuencia de acciones		X	X						
Especificación implícita del comportamiento									
Lógica Temporal									

Tabla 4.3 (d). Mecanismos para el comportamiento de los procesos

	SREM	SDL	LOTOS	ESTELLE	Mascot	SDRTS	SRDT	DARTS	PAISLey
ASIGNACIÓN DE TAREAS	X						X	X	

Tabla 4.3 (e). Mecanismos para la asignación de tareas

Finalmente la tabla 4.3 (f) muestra si cada uno de los métodos tiene realmente definida una metodología para la aplicación de sus técnicas.

	SREM	SDL	LOTOS	ESTELLE	Mascot	SDRTS	SRDT	DARTS	PAISLey
USO DE METODOLOGÍA	X				X	X	X	X	

Tabla 4.3 (f). Aplica o no una metodología

5. INGENIERÍA DE SOFTWARE PARA EL DISEÑO DE SISTEMAS DISTRIBUIDOS: RETOS Y OPORTUNIDADES

Diferentes características sobre el desarrollo de software han sido tratadas hasta este momento. También se ha hecho una introducción a la ingeniería de software para sistemas distribuidos. En la siguiente sección se enunciarán una serie de conclusiones respecto a las diferentes técnicas y métodos enunciados. Algunas de estas conclusiones son de carácter general para establecer de esta manera los conceptos fundamentales a tomar en cuenta y otras están específicamente relacionadas con los sistemas distribuidos.

Como puede verse, hay muchas formas de analizar un problema de desarrollo de software. La primera ventaja de aplicar un ambiente más formal al análisis de un problema es que se simplifique esta tarea. No se busca simplificar la parte fácil de este trabajo sino por el contrario la parte difícil del mismo, que podría resumirse en conceptos como organización de la información, relacionar diferentes perspectivas de las personas, sortear y resolver conflictos y obviamente el diseño interno del software.

Algunos de los métodos son una combinación de lenguajes informales (descripciones en lenguaje natural, diagramas, tablas, etc) y algún lenguaje para realizar una especificación más formal. Por otro lado algunos de ellos usan solamente un lenguaje para especificación formal. Lo anterior trae como consecuencia que en algún momento dichos métodos puedan ser entendidos o no por un usuario común. Se debe entonces buscar un equilibrio entre los diferentes lenguajes utilizados, para poder tener un método orientado al usuario y que a la vez facilite la labor de los desarrolladores.

Es bueno tener en cuenta que debe existir la posibilidad de usar diferentes ciclos de vida para el desarrollo del software dependiendo del problema a solucionar. Igualmente, poder escoger dentro de ese ciclo, el método que mejor se adapte a cada una de las etapas y tareas allí incluidas.

Respecto a las fases de análisis y diseño es importante dejar en claro lo siguiente:

- Deben determinarse primero que nada los **requerimientos del software**: ésto incluye un análisis del problema que debe concluir con una **especificación completa** del comportamiento externo deseado del sistema de software a desarrollar; también llamado descripción funcional, requerimientos funcionales, y especificaciones por otros
- El **análisis del problema** se basa inicialmente en un **proceso de descomposición**: descomposición del problema en subproblemas con el objetivo de entenderlo en forma completa.
- Existe un **diseño preliminar** que también es inicialmente una **descomposición de procesos**: descomposición de componentes de software en otros componentes de software más pequeños con el objetivo de generar un diseño que satisfaga los requerimientos.
- El **diseño detallado** define y documenta los algoritmos para cada módulo en el árbol de diseño; es llamado también **diseño del programa** por otros.

La construcción de software para sistemas distribuidos y paralelos es notoriamente difícil y costoso. Los retos adicionales generados por el **dinamismo** y la **distribución física** de los componentes y su comportamiento puede colocar a los métodos actuales de la ingeniería de software y sus herramientas más allá de sus límites.

En términos generales, la distribución y la concurrencia afectan todas las fases del ciclo de vida de desarrollo de software. La tabla 5.1 muestra algunas de las áreas, que necesitan especial atención cuando se construye un amplio rango de sistemas distribuidos y paralelos [Jelly, Gorton, 1997].

FASES DEL CICLO DE VIDA	DESAFÍOS
Análisis y requerimientos	Requerimientos de distribución física Requerimientos de tolerancia de fallas Requerimientos de ejecución y escalabilidad Identificación del paralelismo del dominio del problema
Diseño	Identificación del paralelismo del dominio de la solución Comunicaciones entre componentes Movilidad de componentes Sincronización de componentes Acceso compartido a componentes Creación o destrucción de procesos e hilos Técnicas para el manejo de excepciones distribuidas Validación de diseño

Tabla 5.1

Los métodos de Ingeniería de Software tanto para programas no distribuidos como para programas distribuidos pueden ser combinados con la idea de poder usar todas sus ventajas. La combinación de procesos en paralelo que se comunican en forma asincrónica, es decir, conceptos orientados al uso de hilos y objetos, hace más fácil el diseño y la implementación de sistemas de software distribuido, incrementando la eficiencia de los procesos de la Ingeniería de Software.

Con la idea de describir el software distribuido, todos los aspectos enunciados hasta ahora en este documento deben ser combinados para obtener una técnica de especificación, tomando en cuenta aspectos fundamentales como los siguientes:

- Las descripciones del flujo de control son muy importantes, especialmente para aplicaciones en tiempo real.
- Se han desarrollado muchos lenguajes para describir los sistemas de procesamiento en paralelo.
- Un método de diseño de aplicaciones distribuidas debe soportar 5 actividades principales:

    1. La descomposición del sistema y la estructura de los procesos
    2. Las características de comunicación
    3. Las características de sincronización
    4. La descripción del comportamiento de los procesos
    5. La asignación de tareas

Como puede verse en las tablas 4.2 y 4.3 de la sección 4 de este documento, no se podría decir que existe un método o tecnología particular que solucione todos los problemas. Y el camino que se visualiza en este trabajo y que de hecho se aplica así, es la combinación de los mejores aspectos de cada uno de estos métodos para poder diseñar en una mejor forma un sistema distribuido.

Una de las características más importantes que debe tener cualquier método para el diseño de los sistemas distribuidos, es dar la posibilidad de un análisis estructurado para la descomposición del sistema en subsistemas y procesos concurrentes.

Una técnica de Ingeniería de Software para sistemas distribuidos debe cubrir :

- Los principales métodos de comunicación para procesos, tales como, mensajes entre procesos, datos y objetos compartidos, y llamados a procedimientos remotos.

- Los métodos relacionados con la comunicación, tales como, intercambio de mensajes asíncrono y síncrono, métodos de sincronización orientados a los recursos, por ejemplo monitores.
- Las diferentes formas para describir el comportamiento de los procesos, tales como especificación explícita del comportamiento usando estados y transiciones, descripción implícita del comportamiento usando predicados de comportamiento y lógica temporal.

Métodos de análisis y diseño orientados a los objetos, ya que ellos permiten la producción de software reusable y de mejor calidad. El software orientado a objetos es más fácil de mantener debido a que los componentes de su estructura están inherentemente separados. Esto permite que los cambios necesarios al software sea menos frustrante para los ingenieros de software y para los clientes. Además los sistemas orientados a objetos son fácilmente adaptables y escalables (es decir, un sistema extenso puede ser creado reusando y ensamblando otros subsistemas).

La mayoría de los métodos se enfocan a obtener un modelo de procesos y dejan por fuera la obtención del modelo de datos. También debe existir el método que nos permita definir la configuración de nuestro sistema en cuanto a la topología que se implantaría.

Otro aspecto que dificulta el entendimiento de los métodos es la diferencia en la terminología usada. Esto hace que aunque dos métodos diferentes manejen el concepto de "procesos" tienda a entenderse diferente ya que el uno lo llama como tal, mientras otro asigna un nombre diferente para referirse a procesos. Es decir, sería conveniente estandarizar terminología para hacer más fácil el entendimiento y aplicación del método.

No todos los métodos tienen una metodología como tal. Es decir, el método dice cuales son sus mecanismos y convenciones, pero en la mayoría de los casos, no describe paso a paso lo que debería hacerse para obtener cada uno de los modelos y sus respectivas especificaciones. Una metodología más completa para el análisis y diseño de los sistemas distribuidos debe responder a

una infinidad de preguntas tales como, ¿Cuáles son las funciones principales del sistema? ¿Qué proceso es el necesario para proveer las funciones? ¿Qué estímulo activa las funciones? ¿Qué tan rápido deberá responder el sistema?. El ambiente deberá considerar las diferentes opciones de diseño disponibles. ¿Qué nodo de intercomunicación se usaría? ¿Cuál es la mejor arquitectura para el problema? ¿Cómo deben ser distribuidas y asignadas las tareas? ¿Cómo debe ser distribuida la base de datos? ¿Cuál es el ambiente de control del sistema? y muchas otras que los métodos enunciados acá contestan solo parcialmente.

Finalmente, es importante tender hacia un diseño de aplicaciones distribuidas bajo el **paradigma top-down** en lugar del **paradigma bottom-up**. Una metodología bajo el paradigma **top-down** soportada por un ambiente computacional que se adapte al mismo, ayuda en el desarrollo de aplicaciones para sistemas distribuidos y paralelos que requieren generalmente una buena experiencia en el área de programación. Para entender mejor el porqué de lo anterior, es importante enunciar los problemas relacionados con el ambiente **bottom-up** e ilustrar así los escenarios típicos de estos paradigmas.

El paradigma **bottom-up** se basa en la composición, con la cual los desarrolladores combinan componentes existentes en diferentes configuraciones para lograr varios objetivos, y posteriormente se agregan clientes que hacen llamados sobre dichos componentes. Este tipo de composición es típico en arquitecturas tales como CORBA (Commom Object Request Broker Architecture) o DCOM (Distributed Component Object Model), conjunto de herramientas (Toolkits) tales como ACE (Advance Computing Environment), y middleware tales como MPI (Message Passing Interfase). El principal problema del ambiente **bottom-up** es la disponibilidad de los componentes, más bien que los requerimientos fundamentales de la aplicación, que a menudo motivan los diseños. Más aún usando una composición por defecto típicamente se conduce al desarrollador a tener fallas en la caracterización de las propiedades globales del sistema. También, el diseño del sistema distribuido es típicamente vulnerable a errores de algoritmos fundamentales. Estos errores son difíciles de detectar y no se corrigen fácilmente (adicionar otro componente no soluciona el problema).

Se propone el diseño top-down y el desarrollo de aplicaciones de software distribuido, soportado por herramientas que ofrezcan asistencia en visualización, simulación, y depuración para cada paso en el proceso de diseño. La ventaja clave de este ambiente es el uso de *abstracciones* (en los estados iniciales del desarrollo) que reúnen la funcionalidad esencial que la aplicación requiere para comunicación, control distribuido, u otros servicios complejos. En resumen los requerimientos del sistema deben ser expresados en un amplio rango de propiedades que el proceso de desarrollo deberá respetar. Por supuesto, se enfatiza descomposición y refinamiento más bien que composición.

Se usa el concepto de *abstracción* para hacer representaciones a alto nivel de un servicio o componente dentro de una aplicación distribuida. La *abstracción* captura una visión del cliente, es decir lo que hace el servicio y no como lo hace.

Es importante distinguir entre una *abstracción* y sus implementaciones. La *abstracción* envuelve un alto nivel de descripción del servicio como una única entidad, expresando solamente los requerimientos esenciales (por ejemplo, que un mensaje debería ser liberado solamente bajo ciertas circunstancias). Una implementación suministra este servicio, usando un conjunto de procesos, que se comunican a través de una red de acuerdo a un protocolo particular. Por ejemplo, la *abstracción atómica* de una culminación o finalización (commit) puede ser implementada con procesos que ejecuten "culminación a dos fases" (two-phase commit). Implementaciones alternativas podrían usar "culminación a tres fases" (three-phase commit). Sin embargo, las aplicaciones podrían usar el servicio de culminación atómica (atomic-commit) sin tener conocimiento del protocolo en que se suministra éste.

EL DESARROLLO DE APLICACIONES DISTRIBUIDAS	
PARADIGMA BOTTOM-UP	**PARADIGMA TOP-DOWN**
Actualmente el desarrollo para aplicaciones distribuidas se basan en el paradigma **bottom-up**.	La idea es tender hacia un ambiente de diseño del tipo **top-down.**
Se fundamenta en: 1. Combinación en diferentes configuraciones de componentes ya existentes. 2. Agregar clientes que hacen llamadas sobre dichos componentes	Se fundamenta en: 1. Hacer **abstracciones**: representación abstracta a un alto nivel de un servicio o componente dentro de una aplicación distribuida. 2. Desarrollar la implementación que suministra este servicio.
Pero los principales problemas en este ambiente bottom-up son: – Se motiva más al diseño que a la especificación de los requerimientos – Al usar estos componentes se falla en caracterizar las propiedades globales del sistema – Y también el diseño del sistema es más vulnerable a errores ya que estos son difíciles de detectar y la solución no está en agregar otros componentes	En resumen permite que las necesidades requeridas por una aplicación puedan ser expresadas como un amplio rango de propiedades que el proceso de desarrollo deberá respetar. Este propósito requiere una colección de herramientas que permitan explorar el diseño inicial en el ciclo de desarrollo.

Tabla 5.2

Nuestro propósito requiere una colección de herramientas que permitan la exploración de un diseño inicial en su desarrollo. Los diseñadores pueden explorar las opciones de abstracción, trabajando con abstracciones, en lugar de especificar, implementaciones. En la exploración, el diseñador construirá intuiciones acerca del comportamiento del diseño antes de invertir esfuerzos en una codificación detallada. Solamente después de escoger un diseño general e identificar las

abstracciones fundamentales el diseñador considerará la implementación distribuida (posiblemente usando otras, abstracciones más simples).

En la tabla 5.2 se resume las características principales enunciadas en los párrafos anteriores para cada uno de los paradigmas.

En definitiva el diseño de cualquier sistema y en particular el de un sistema distribuido es difícil de realizar. Contar con una metodología que de respuestas a todos y cada uno de los aspectos que involucra un sistema distribuido es bastante complicado.

En la sección 3 de este documento, se plantea lo siguiente: "Diferentes tipos de programas necesitan diferentes ambientes de Ingeniería de Software, particulares ciclos de vida, métodos, herramientas, y técnicas, dependiendo de ciertas propiedades básicas del problema a solucionar y la solución que se quiere. El deseo de la Ingeniería de Software es desarrollar un ambiente que se adapte a todos los aspectos del desarrollo de software y para todos los tipos de programas y programadores". Como puede verse cada uno de los métodos descritos en este trabajo no pueden cumplir con dicha característica. Es necesario entonces hacer una abstracción a más alto nivel para crear un esquema que de lineamientos más generales en el diseño de sistemas distribuidos, y de esta manera generar una descomposición y refinamiento en la metodología que indique donde aplicar entonces cada uno de los métodos. A continuación se analiza una de las alternativas más prometedora en este sentido.

6. PATRONES DE SOFTWARE

El tema de esta sección es **patrones (patterns)**, un patrón según el diccionario **"es una forma creada, o un original, o un modelo aceptado para producir otro igual".** En la ingeniería de software, lo que un patrón busca básicamente es describir soluciones satisfactorias a problemas comunes en el desarrollo de software. Cada patrón describe un problema que ocurre una y otra vez en un ambiente, así como un esqueleto de solución de ese problema, de tal forma que se puede usar esa solución muchas veces sin tener que volver a crear todo el esquema de la solución nuevamente. En resumen un patrón:

- Es una solución recurrente a un problema estándar
- Es un paso hacia libros de bolsillo (manuales) para ingenieros de software
- Cuando patrones relacionados son ligados unos con otros, entonces forman un lenguaje que suministra un proceso para la resolución ordenada de problemas de desarrollo de software
- Ambos, patrones y lenguajes de patrón ayudan a los desarrolladores a comunicar su conocimiento. En general ayudan a las personas a comunicarse mejor.
- Debe balancear o controlar un conjunto de fuerzas opuestas en el contexto en que se están aplicando

La esencia de todos los patrones de diseño es la dupla formada por el **problema y su solución**. Todas las partes adicionales del esqueleto agregan detalles importantes y relaciones para la descripción del patrón (figura 6.1).

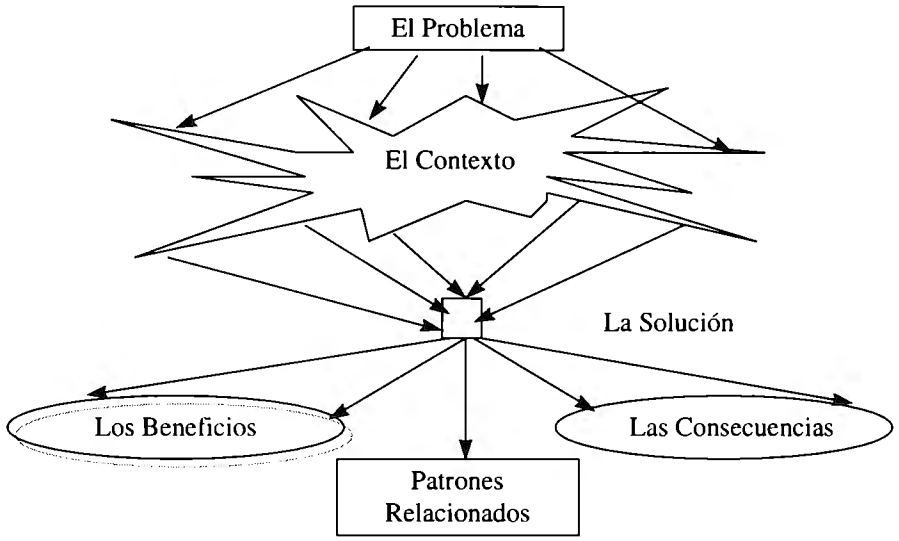


Figura 6.1 Los patrones de diseño unen un problema y una solución que resuelve fuerzas en formas que conduscan a beneficios, consecuencias y otros patrones

En general, un patrón tiene 4 elementos esenciales [GoF, 1995]:

1. **El nombre del patrón** es el título que se usa para describir un problema de diseño, sus soluciones y consecuencias en una o dos palabras. Nombrar un patrón inmediatamente incrementa nuestro vocabulario de diseño. Esto permite diseñar a un alto nivel de abstracción. Tener un vocabulario para patrones nos permite hablar de ellos con nuestros colegas, en nuestra documentación y aun con nosotros mismos. Hace más fácil pensar acerca del diseño, y para comunicar ellos y sus ventajas y desventajas a otros. Encontrar buenos nombres es una de las tareas más difíciles.

2. El **problema** describe cuando aplicar el patrón. Explica el problema y su contexto. Debe describir problemas de diseño específicos tales como, como representar algoritmos como objetos. También debe describir las estructuras de las clases u objetos que son sintomáticamente de un diseño inflexible. Algunas veces el problema incluirá una lista de condiciones que deben ser conocidas antes de tener sentido para aplicar el patrón.

3. La **solución** describe los elementos que hacen óptimo el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño concreto particular o

una implementación, porque un patrón es como una plantilla que puede ser aplicada en diferentes situaciones. En lugar de eso, el patrón provee una descripción abstracta de un problema de diseño y como un conjunto general de elementos (por ejemplo, clases y objetos) lo resuelve.

4. Las **consecuencias** son los resultados, y las ventajas y desventajas de aplicar el patrón. Aunque las consecuencias son normalmente obviadas cuando describimos decisiones de diseño, ellas son críticas para evaluar alternativas de diseño y para entender el costo-beneficio de aplicar el patrón.

6.1 Categorías de los patrones [buschmann, 1996]

Los patrones cubren varios rangos o escalas de abstracción. Algunos patrones ayudan a estructurar un sistema de software en subsistemas. Otros patrones soportan el refinamiento de subsistemas y componentes, o de las relaciones entre ellos. Algunos ayudan en la implementación de los aspectos de un diseño particular en un lenguaje de programación específico. Por eso se pueden clasificar los patrones básicamente en tres categorías:

- **Patrones arquitecturales**. Expresan un esquema de organización estructural fundamental para sistemas de software. Suministran un conjunto de subsistemas predefinidos, especifican sus responsabilidades, e incluye reglas y lineamientos para organizar las relaciones entre ellos.
- **Patrones de diseño.** Suministran un esquema para refinar los subsistemas o componentes de un sistema de software, o las relaciones entre ellos. Estos describen una estructura recurrente de componentes que se están comunicando y que solucionan un problema general de diseño dentro de un contexto particular.
- **Idiomas.** Es un patrón de más bajo nivel, específico para lenguajes de programación. Un idioma describe como implementar aspectos particulares de los componentes o de las relaciones entre ellos usando las características del lenguaje dado.

Las **relaciones entre patrones son necesarias.** Un patrón resuelve un problema particular, pero su aplicación puede generar nuevos problemas. Algunos de estos pueden ser solucionados por otros patrones. Componentes únicos o relaciones dentro de un patrón particular pueden por supuesto describirse con patrones más pequeños, todos ellos integrados por el patrón más grande en el cual ellos están contenidos. Por eso para usar los patrones efectivamente, es necesario organizarlos dentro de lo que se llama **sistemas patrón**. Un **sistema patrón** describe los patrones en forma uniforme, los clasifica, y lo más importante, muestra como están relacionados unos con otros.

6.2 Integración con el desarrollo de software

La integración de los diferentes tipos de patrones con el desarrollo de software se da básicamente de la siguiente manera: los **patrones arquitecturales** pueden ser usados en las fases de análisis y diseño preliminar, los **patrones de diseño** durante toda la fase de diseño, y los **idiomas** durante la fase de implementación.

Ahora bien, veamos cuales podrían ser las ventajas y desventajas de usar patrones de software [Schmidt, Fayad, Johnson, 1996]:

Ventajas:

- **Los patrones de diseño pueden ser usados reactivamente.** Los patrones de diseño pueden ser usados como una herramienta de documentación para clasificar fragmentos de un diseño, haciendo fácil para un equipo incluir los nuevos desarrolladores. Por ejemplo muchas de las filosofías de los equipos de diseño están capturadas explícitamente más que las estrictamente salidas del grupo original.
- **Los patrones de diseño pueden usarse proactivamente.** Los patrones son más que una herramienta de documentación. Más que eso, ellos pueden ser usados para construir diseños robustos entendiendo bien los pros y los contras de cada una de las partes componentes de

cada nivel de diseño. Usar los patrones de diseño de esta manera, requiere que el diseñador ejecute adecuadamente la aplicación del patrón, así como la habilidad de abstraer la esencia del problema a diseñar de todo el conjunto de detalles del mismo.

- **Los patrones de diseño pueden llevar a tomar una decisión orientada a ganar-ganar.** Muy a menudo un diseñador tiene que tomar una decisión entre dos cualidades altamente prioritarias. Algunos de los patrones más poderosos son aquellos que permiten a un diseñador trabajar alrededor de estas decisiones. En lugar de forzarlos a escoger entre una de las dos, estos patrones de diseño permiten dar al software ambas cualidades simultáneamente.

Desventajas:

- **Los patrones han sido sobre valorados.** Los patrones de diseño tienen su lugar, y son muy importantes, especialmente cuando la adaptabilidad es un objetivo valioso. Los beneficios de usar patrones de diseño se pueden perder a menos que todo el proceso de desarrollo de software sea modificado para tomar los patrones de diseño en cuenta. Por ejemplo cada código generado deberá ser revisado para asegurar que refleja las restricciones de los patrones de diseño apropiados, cada modificación deberá asegurarse de no romper con las restricciones de los patrones de diseño, y la adaptabilidad de los patrones de diseño del sistema deberán ser explícitamente evaluada.
- **Algunos patrones de diseño son innecesariamente difíciles de aprender.** Algunos patrones de diseño son difíciles de entender por parte de los diseñadores orientados a objetos (OO), los patrones son inherentemente no muy claros. Desafortunadamente en otros casos el patrón no es claro, aún más estos tienen un nombre o descripción que no es obvia. Esto es probablemente el resultado inevitable del incremento en personas que están proponiendo patrones de diseño y la frecuencia con que estos los proponen. Para los profesionales que usan los patrones de diseño, estos son un poderoso punto de partida ya que contienen una amplia pero simplificada instrucción del propósito del patrón. Una vez entendido este se puede generar discusiones y descripciones que usualmente acompañan a un patrón de diseño cuando es publicado.
- **Las clasificaciones de los patrones de diseño no son todavía poderosas para los practicantes.** Los patrones de diseño de varias formas, tales como, propósito, alcance y jurisdicción. Estas clasificaciones son poderosas para quienes ya entendieron los patrones de

diseño, especialmente para determinar si un nuevo patrón de diseño es redundante con otro ya existente. Sin embargo, algunas de las categorías no son poderosas para los profesionales del software mientras están aprendiendo los patrones.

6.3 Patrones de diseño para sistemas distribuidos

Los primeros patrones de software fueron escritos por desarrolladores orientados a objetos y se enfocaron sobre diseño y programación orientada a objetos [GoF, 1995] o sobre modelado orientado a objetos [Coad, 1992]. Además de los patrones orientados a objetos, una nueva tendencia son los patrones que se enfocan sobre eficiencia, confiabilidad (reliable), concurrencia escalable, paralelismo y programación distribuida. Algunos de los trabajos realizados en esta tendencia serán descritos muy brevemente a continuación.

6.3.1 [Sane 98] Una completa referencia sobre patrones para sistemas distribuidos

Esta es una investigación sobre patrones de diseño para sistemas concurrentes, paralelos y distribuidos (CPD). Esta investigación está basada en proyectos en las áreas de telecomunicaciones distribuidas a gran escala y de imágenes electrónicas en medicina. En la figura 6.3.1.1 se muestran todos los patrones y información general sobre los mismos se resume en los párrafos siguientes.

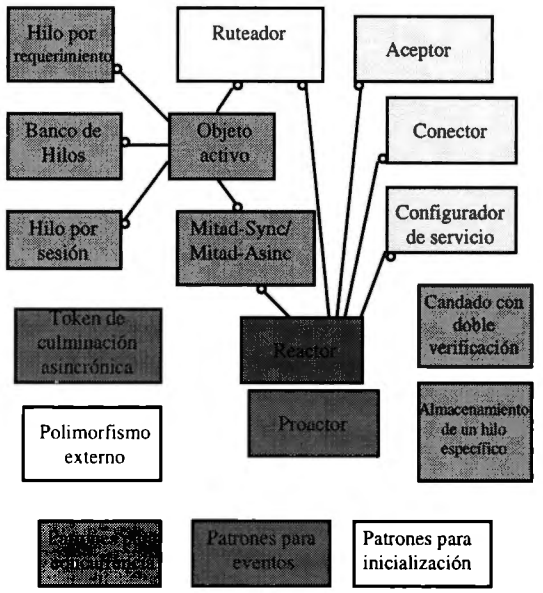


Figura 6.3.1.1. Patrones para sistemas concurrentes, paralelos y distribuidos.

- **Patrones para el manejo de eventos :**

  - **"Reactor" :** un patrón del comportamiento de un objeto para demultiplexaje de eventos y manejo de eventos. Soporta el demultiplexaje y la ejecución de manejadores de múltiples eventos que son inicializados por la ocurrencia de eventos sincrónicos. Este patrón simplifica las aplicaciones que manejan eventos integrando el demultiplexaje sincrónico de eventos y la ejecución de sus correspondientes manejadores de evento.
  - **Una ficha de culminación asincrónica (Asynchronous Completion Token) :** un patrón del comportamiento de un objeto para hacer más eficiente el manejo de eventos asincrónicos. Permite a las aplicaciones hacer más eficiente el estado asociado con la culminación de las operaciones asincrónicas.
  - **"Proactor" :** un patrón del comportamiento de un objeto que soporta el demultiplexaje y la ejecución de múltiples manejadores de eventos, los cuales son accionados por la culminación de eventos asincrónicos. Permite simplificar el

desarrollo de aplicaciones asíncronas integrando el demultiplexaje de eventos culminados y la ejecución de sus correspondientes manejadores de eventos.

- **Patrones para concurrencia :**

  - **Objeto Activo (Object Active):** un patrón del comportamiento de un objeto para programación concurrente. Este patrón desconecta el método de ejecución a partir del método de invocación y simplifica el acceso sincronizado a un recurso compartido por métodos llamados por diferentes hilos de control.
  - **Mitad síncrono/Mitad asíncrono (Half-Sync/Half-Async) :** un patrón arquitectural para hacer más eficiente y bien estructurada la entrada y salida concurrente. Este patrón desconecta entradas y salidas sincrónicas a partir de entradas y salidas asincrónicas en un sistema para simplificar los esfuerzos de programación concurrente sin degradar la eficiencia de la ejecución.
  - **Almacenamiento de un hilo especifico (Thread-Specific Storage) :** un patrón del comportamiento de un objeto para acceso por cada hilo. Permite a múltiples hilos usar un punto de acceso lógico para recuperar hilos de datos locales sin sobrecargar por cada acceso.
  - **Candado con doble verificación (Double-Checked Locking):** es un patrón para optimizar y hacer más eficiente la inicialización y el acceso a objetos por medio de hilos más seguros. Permite inicialización atómica, sin tener en cuenta el orden de inicialización y elimina subsecuentes bloqueos por sobrecarga.

- **Patrones para inicialización**

  - **Configurador de servicios (Service Configurator):** Un patrón para configuración dinámica de servicios. Este patrón desconecta la implementación de servicios desde el momento en que ellos son configurados. Permite incrementar la flexibilidad y extendibilidad de aplicaciones activando sus servicios y configurándolos en cualquier momento. Este patrón es muy usado en ambientes de aplicaciones (es decir, para configurar applets de Java en visualizadores www), sistemas operativos (es decir, para configurar manejadores de dispositivos).

- **Aceptar-Conectar (Acceptor-Connector):** un patrón para crear objetos para conectar e inicializar servicios de comunicación. La intención del patrón es desconectar (1) conexiones activas y pasivas establecidas y la inicialización de servicios a partir de (2) el procesamiento de dos puntos finales de un servicio ejecutado una vez que ellos son conectados e inicializados. Algunos ejemplos que usan este patrón son Web browsers y Web servers, Object Request Brokers y servidores que suministran servicios como login remoto y transferencia de archivos.

- **Otros patrones:**

  - **Ruteador (Router):** es un patrón para comunicar Gateways. Desconecta múltiples recursos de entrada a partir de múltiples recursos de salida para rutear los datos correctamente sin bloquear un gateway.
  - **Polimorfismo externo (External Polymorphism):** un patrón estructural para extender tipos de datos concretos de forma transparente. Este patrón permite clases que no están relacionadas por herencia y/o no tienen métodos virtuales para manejar el polimorfismo.

Otra investigación en la cual se han aplicado los patrones es la que surgió con la idea de desarrollar e implementar una memoria virtual distribuida y que se ha fundamentado en nuevas arquitecturas de software, **interesantes patrones de diseño**, un nuevo modelo de memoria distribuida y un ambiente teórico para razonamiento sobre la lógica del conocimiento. Algunos de los patrones usados en esta investigación son:

- Inspector Desprendible/corte removible (Detachable Inspector/Removable cout)
- Intercambiador de Recursos (Resource Exchanger)
- Composición de Mensajes (Composite Messages)

6.3.2 Un patrón esencial de diseño para tolerancia a fallas en sistemas distribuidos de estado compartido (An Essential Design Pattern for Fault-Tolerant Distribute State Sharing) [Islam, Devarakonda 96]

Debido a que los desarrolladores de programas distribuidos están enfocados all rendimiento y la tolerancia a fallas, los patrones de diseño distribuido deben tener presentes estos aspectos en sus soluciones. Uno de tales patrones de diseño OO es el **Distribuidor Recuperable (Recoverable Distributor)**. Es un patrón de diseño esencial para tolerancia a fallas y en programas distribuidos de estado compartido. Este hace énfasis en el rendimiento, detección de fallas y en la recuperación de los programas.

El patrón **Distribuidor Recuperable** crea vistas locales o estados locales, del estado global en un sistema distribuido; manteniendo consistencia entre el estado local y el estado global; detecta fallas en el procesador; y recupera el estado global en el evento de fallas en el procesador. Este patrón permite la combinación y unión de esquemas para la consistencia de datos y la tolerancia de tal manera que dichos esquemas puedan ser independientemente adaptados a los requerimientos de las aplicaciones encontradas mientras mantiene las interacciones esenciales.

Un programador puede desarrollar programas de estado compartido sin permitir vistas locales o sin suministrar tolerancia a fallas. Si el diseño no incorpora vistas locales, el acceso al estado global puede incurrir en una sobrecarga en las comunicaciones y la aplicación puede tener un rendimiento pobre en un sistema distribuido.

Como otra alternativa, un programa con estado compartido puede ser escrito usando un conjunto de protocolos específicos para consistencia de los datos y tolerancia a fallas. Este patrón puede ser usado cuando un programa distribuido consiste de objetos distribuidos físicamente que comparten datos. El patrón es aplicable cuando es importante el acceso eficiente a datos compartidos y es necesaria una operación continua aún si hay constantes fallas parciales. Es también aplicable cuando existe la necesidad de construir protocolos extendibles para consistencia en los datos y tolerancia a fallas.

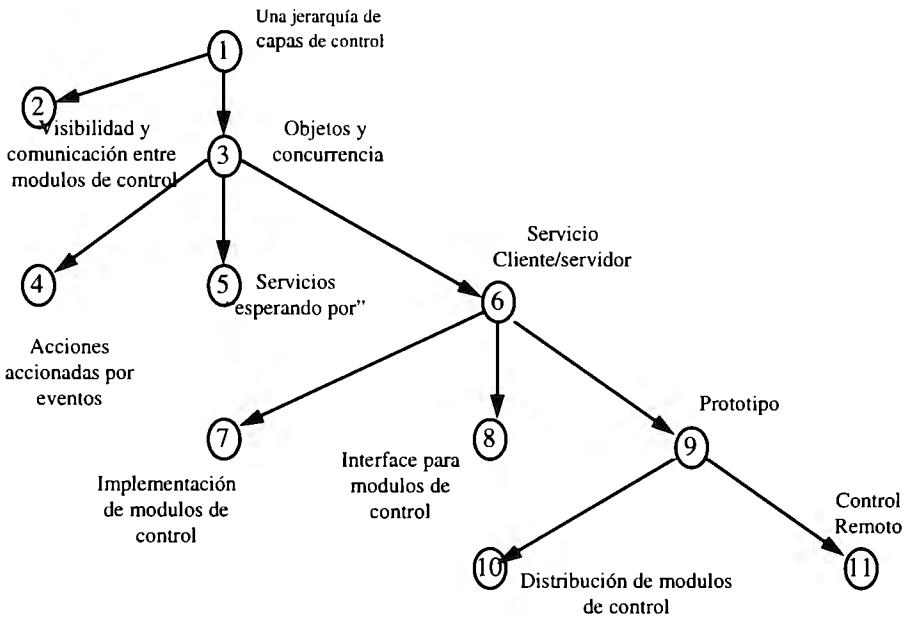
6.3.3 El lenguaje patrón: Seleccionando primitivas de bloqueo para programación paralela (Selecting Locking Primitives for Parallel Programming) [McKenney 96]

Este es un lenguaje patrón que ayuda a seleccionar primitivas de sincronización para programas en paralelo, dejando de lado primitivas que interactúan con el diseño de bloqueo del programa dado.

La razón básica para paralelizar un programa es lograr un mejor rendimiento. Sin embargo las primitivas de sincronización usadas por programas paralelos pueden estar sujetos a problemas de latencia en la memoria, pueden consumir excesiva memoria y el acceso puede resultar no equitativo o en esperas infinitas.

6.3.4 [Aarsten, Brugali, Menga] Diseñando Sistemas concurrentes y de control distribuido. El lenguaje patrón G++ (Designing Concurrent and Distributed Control Systems. El lenguaje patrón G++)

El lenguaje patrón G++ ataca el problema de diseñar software extenso para sistemas de control, compuesto de capas de módulos de control concurrente instalados sobre una arquitectura computacional distribuida, siguiendo un proceso de desarrollo evolutivo que produce la implementación final a partir de un prototipo del diseño lógico.



El lenguaje patrón G++

Figura 6.3.4.1

Los patrones en este lenguaje son estructurados en un árbol como se muestra en la figura 6.3.4.1, cada circulo denota un patrón y un punto de decisión en el diseño y los arcos, que enlazan los patrones, representan la secuencia temporal de decisiones. El proceso de desarrollo se hace siguiendo el gráfico desde la raíz hasta las hojas.

El lenguaje aplica la filosofía de "divide y conquista", una fase de generación del prototipo es obligatoria antes de intentar integrar módulos de control con el proceso físico. La adopción de un ambiente evolutivo, permite llegar suavemente desde un prototipo a la implementación física.

7. APLICACIÓN DE LOS PATRONES DE DISEÑO EN CASOS REALES

Es importante aplicar los patrones de diseño en casos reales que nos permitan entender y evaluar mejor las ventajas y desventajas que presentan. Para dicha aplicación se han escogido los siguientes sistemas: **Sistemas de Manufactura Flexible (FMS)** y **Sistemas de Bases de Datos Distribuidas**. Para el primero de ellos se aplicará el **"Lenguaje Patrón G++"** ya que éste es presentado como una metodología para el desarrollo de sistemas de información concurrentes y distribuidos. Además, su dominio de aplicación es en arquitecturas de control cliente-servidor, un esquema muy utilizado en FMS. En el caso de los Sistemas de Bases de Datos Distribuidas se aplicará el patrón "**Usando la replicación para distribución: patrones para la actualización eficiente"** ya que como su nombre lo dice ataca el problema de cómo actualizar los datos replicados en una base de datos distribuida (una de las características principales de estos sistemas), para mantener así la integridad de todas las copias de los datos en forma consistente.

Inicialmente se establecerán las principales características de los dos sistemas a los cuales se aplicarán los patrones de diseño. El objetivo es describir los conceptos básicos de dichos sistemas obteniendo un mejor conocimiento del contexto en que éstos se encuentran y al mismo tiempo estableciendo así las bases del sistema a diseñar.

El primero de los dos sistemas tomados para la aplicación de los conceptos es el **Sistema de Manufactura Flexible (FMS).** Entre las características deseables de una FMS podemos mencionar principalmente:

- Capacidad de recuperarse de fallas
- Modularidad, la cual permite reemplazar partes del sistema sin afectar su funcionamiento (flexibilidad en las partes físicas); y la flexibilidad en la planificación de tareas (flexibilidad de los métodos de programación).

El controlador de una Celda Flexible de Manufactura (CFM) es un sistema distribuido ya que puede modelarse como un sistema de múltiples agentes computacionales que cooperan para resolver un problema común. Uno de sus problema principales es el de coordinar a los agentes a

través de un sistema de planificación y ejecución. Al operar en ambientes de operación "crítica" (en tiempo y dinero) también debe ser un sistema tolerante a fallas en tiempo real.

El diseño de este tipo de sistemas cubre una gran variedad de áreas desde la arquitectura del controlador de la celda, las tecnologías de comunicaciones de redes reconfigurables, interfases con robots, software de comunicaciones, etc, que lo convierten en un buen candidato de aplicación de la metodología de análisis y diseño de sistemas distribuidos.

El segundo sistema son las **Bases de Datos Distribuidas**. Dichos sistemas consisten de una colección de datos interrelacionados y un conjunto de programas para utilizar esos datos. El objetivo primordial de un sistema Manejador de Bases de Datos (SMBD) es proporcionar un entorno que sea a la vez conveniente y eficiente para ser utilizado al extraer y almacenar información de la base de datos [Korth, Silberschatz 1993].

Algunos de los objetivos de un SMBD son: **evitar redundancia e inconsistencias** en los datos, **facilitar el acceso** a ellos, **relacionarlos** fácilmente, permitir **acceso concurrente** a los datos, **asegurar la integridad** de los datos y crear un **ambiente seguro.**

Al igual que el primer caso de aplicación, los sistemas de Bases de Datos distribuidas requieren de un análisis y diseño adecuado que permita la mejor administración de los procesos y datos que lo componen, así como cubrir de la forma más óptima posible los objetivos enunciados en el párrafo anterior.

Antes de aplicar cada uno de los patrones escogidos se hace una presentación de los mismos que incluye su **contexto**, su **solución** y finalmente la **aplicación** de dicha solución en el sistema ejemplo escogido.

7.1 Sistemas de manufactura flexibles (FMS)

[Greenwood, 1988] Un sistema de manufactura flexible, a través de la combinación adecuada de control computarizado, comunicaciones, procesos de manufactura y equipos relacionados,

permite a una organización orientada a la producción responder rápida y económicamente, de una manera integrada, a cambios significativos en su ambiente operativo. Tales sistemas típicamente están conformados por equipos para proceso (por ejemplo, máquinas herramienta, estaciones de ensamblaje, robots, etc), equipos para manejo de materiales (por ejemplo, robots, transportadores, vehículos guiados automáticamente, etc), un sistema de comunicaciones y un sistema sofisticado de control computarizado.

La tabla 7.1.1 muestra la comparación entre un sistema tradicional de manufactura y un ambiente de manufactura flexible.

AMBIENTE TRADICIONAL	FMS
1) Subdivide los trabajos en varias operaciones simples	1) Subdivide los trabajos en pocas operaciones
2) Completa operaciones en lote secuencialmente	2) Traslapa operaciones en lote cada vez que es posible
3) Completa operaciones rápidamente	3) Completa operaciones consistentemente y rápidamente
4) Operaciones individuales automatizadas	4) Secuencia de operaciones completamente automatizadas

Tabla 7.1.1. Ambiente tradicional de manufactura Vs Ambiente de manufactura flexible.

Todo sistema de manufactura flexible involucra similares elementos funcionales, que son esencialmente:

- **Estaciones de trabajo**: en las cuales las piezas son procesadas o alteradas de alguna forma.
- **Unidades de almacenamiento**: donde las piezas son solamente almacenadas y por lo tanto no modificadas, tales como buffers de almacenamiento, ASRS, etc.
- **Sistemas de transporte**, que son usados para transferir partes entre estaciones de trabajo.

Los principales elementos de un sistema de control de una FMS son:

- El computador central (host), cuya tarea es coordinar las actividades de los diferentes equipos (estaciones de trabajo).
- Una red de comunicaciones, cuya tarea es enlazar el computador central (host) a las estaciones de trabajo.
- Finalmente, unidades de control responsables por las actividades de las estaciones de trabajo, por ejemplo, una unidad de control numérico sobre una máquina herramienta o robot o posiblemente un controlador lógico programable, etc.

Adicionalmente es necesario tener un dispositivo que actúa como interfase de los dispositivos controladores, de las máquinas herramientas; con la red de comunicaciones de la FMS. Estos dispositivos son usualmente llamados unidades interfase.

La figura 7.1.2 es un diagrama de bloques mostrando cómo cada uno de estos elementos son combinados dentro de una instalación típica.

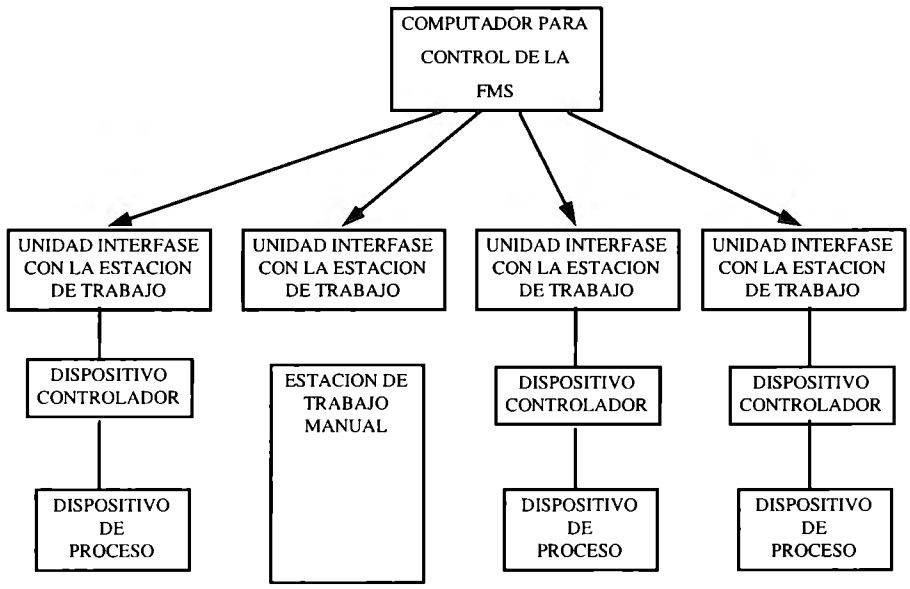


Figura 7.1.2 Elementos de control de una FMS

7.1.1 Aplicación de el Lenguaje Patrón G++ [Aarsten, Brugali, Menga] en Sistemas de Manufactura Flexible (FMS)

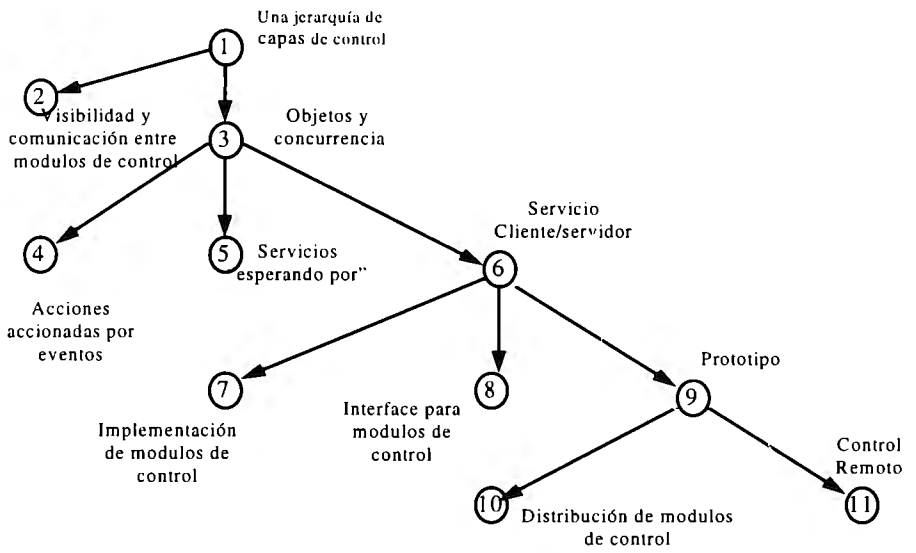
Uno de los objetivos de esta investigación es analizar la aplicabilidad de los **patrones de diseño** en casos reales; por esta razón, en el caso específico de los Sistemas de Manufactura Flexible se evaluará el patrón enunciado a continuación.

El G++ es presentado como un conjunto de *patrones de diseño*, estructurados para formar un *un lenguaje patrón.*

Siguiendo las intuiciones del arquitecto Cristopher Alexander, un diseño debe ser considerado como aquél que está formado por *"patrones"* de relaciones entre elementos, donde cada patrón es por si mismo una regla para permitir la escoger un diseño a partir del requerimiento de un problema.

Estas ideas han sido transferidas al campo del diseño orientado a objetos, donde los elementos son objetos y *"patrones"* que indican grupos de objetos cooperando enlazados por ciertas relaciones que pueden encontrarse repetidamente en la solución de una clase de problemas; ésto nos permite llamar a G++ un lenguaje patrón, ésto es, los patrones son ordenados y estructurados en tal forma que ayudan al diseñador en su decisión y la administración en el momento adecuado de los asuntos que encuentre.

El lenguaje patrón G++ ataca el problema de diseñar software extenso para sistemas de control, compuestos de capas de módulos de control concurrente instalados sobre una arquitectura computacional distribuida, siguiendo un proceso de desarrollo evolutivo que produce la implementación final a partir de un prototipo del diseño lógico.



El lenguaje patrón G++

Figura 7.1.1.1

Los patrones en este lenguaje son estructurados en un árbol como se muestra en la figura 7.1.1.1, cada círculo denota un patrón y un punto de decisión en el diseño mientras que los arcos, que enlazan los patrones, representan la secuencia temporal de decisiones. El proceso de desarrollo se hace siguiendo el gráfico desde la raíz hasta las hojas.

El lenguaje aplica la filosofía de "divide y conquista", una fase de generación del prototipo es obligatoria antes de intentar integrar módulos de control con el proceso físico. La adopción de un ambiente evolutivo, permite llegar suavemente desde un prototipo a la implementación física.

En esta aplicación de los patrones de diseño su descripción y solución se representa en la mayoría de los casos por medio de diagramas entidad-relación, donde un rectángulo con vértices redondeados representa una clase (identificadores superiores en el rectángulo) o una instancia (identificador inferior); un caracter al lado izquierdo de la etiqueta se usa para indicar qué tipo de objeto es (este lenguaje patrón tienen una clasificación para los objetos que se verá más adelante). Los datos, métodos y nombres de eventos enviados en un momento determinado son representados como un comentario al lado del ícono. Las relaciones se representan usando la notación OMT (Apéndice A) a la cual se ha adicionado el enlace de uso "USA" en el diseño lógico y de COM, indicando interconexión a través de una red de comunicación que involucra un

arrancador y un contexto remoto, en el diseño físico. Los enlaces que implican una relación de uso pueden llevar el nombre del método como el nombre del mensaje que fluye entre ellos. Las relaciones OMT de generalización/especialización del análisis, aplicadas al diseño, indican implementación de herencia y se le puede agregar al enlace la etiqueta ES_UNA.

7.1.1.1 Estructura de el FMS tomado para la aplicación

Para el análisis del lenguaje patrón G++, se tomará como referencia un Sistema de Manufactura Flexible (FMS) típico para la producción de componentes. Básicamente éste es un departamento de producción compuesto de dos celdas, una para maquinado y otra para ensamblaje. Cada celda está hecha de un conjunto de máquinas controladas numéricamente, un sistema automático de inventario para materiales y piezas terminadas, y un sistema de vehículos guiados automáticamente (AGV) para el movimiento de las piezas. La integración de todo se logra por medio de una red la cual cubre las computadoras del departamento y las celdas. Además, existen controles numéricos computarizados (CNC) para las máquinas y controladores lógicos programables (PLC) para los sistemas de inventario y AGV. La producción se presenta, en cualquier periodo de tiempo, debido a la presencia en las celdas de varios lotes pequeños de diferentes tipos de piezas en forma concurrente. Cada pieza de un lote es sometida a una secuencia de operaciones, cada una sobre una máquina diferente, según el tipo de pieza. La celda, con un despachador en tiempo real, asigna piezas a las máquinas tanto como éstas estén ociosas, y haciendo un monitoreo del piso de producción (lote de producción) determina y envía los requerimientos al sistema AGV. Existe una base de datos que contiene datos tecnológicos (operación, tipo de pieza, tipo de máquina) y de administración de los datos (orden, pieza, lote), la cual permite al departamento y las celdas controlar la producción.

7.1.1.2 Aplicación patrón 1: Capas de control en forma jerárquica

Cualquier sistema complejo organiza su funcionalidad como una arquitectura en capas jerárquicas que corresponden a módulos de control.

Contexto

Los complejos sistemas de ingeniería distribuidos, se caracterizan por la ocurrencia de múltiples actividades en forma concurrente, son difíciles de diseñar y administrar; y por otra parte, es importante poder reusar algunas de sus partes en diferentes sistemas.

Solución

Con la idea de enfrentar la complejidad, es importante organizar las funcionalidades en una arquitectura de control descentralizada y jerárquica. Cada capa de la jerarquía reúne módulos de control en clases que tienen una funcionalidad similar. Un módulo de control es una entidad caracterizada por:

- capacidad de tomar decisiones autónomas
- una serie de servicios disponibles que se ofrecen al exterior
- el control de un grupo de recursos.

Los módulos de control son objetos que deberían organizarse usando dos principios básicos de estructuración, el de inclusión o encapsulamiento (aggregate) y el de uso (acquaintance).

Toda organización define sus propios módulos de control de acuerdo a sus principios internos, que pueden ser funcionales, lógicos o simplemente dictados por tradición. Los módulos deberían cubrir funcionalidades de tal forma que puedan ser reusados para diferentes proyectos.

Organice su sistema en módulos de control, que deberán ser objetos con un cierto grado de autonomía y capaces de ofrecer servicios bien definidos para la administración de un conjunto de recursos. Dé una estructura jerárquica al conjunto de estos módulos aplicando los principios de inclusión o uso externo.

Aplicación del patrón en el FMS

En el caso del FMS los principios internos que nos permiten definir los módulos de control se indican tradicionalmente como **funciones o procesos técnicos** y en el ejemplo se pueden esquematizar como lo muestra la figura 7.1.1.2.1, en la cual las funciones aparecen relacionadas en diferentes niveles o capas de control y que por lo tanto son la base para crear la jerarquía necesaria y el control descentralizado como lo sugiere el patrón.

En la figura 7.1.1.2.2 se describe el análisis del FMS tomado como ejemplo. El análisis se hace con base en el esquema jerárquico establecido en la figura 7.1.1.2.1 y utilizando un diagrama entidad-relación en un contexto orientado a objetos en el cual las relaciones son representadas usando la notación OMT (Apéndice A).

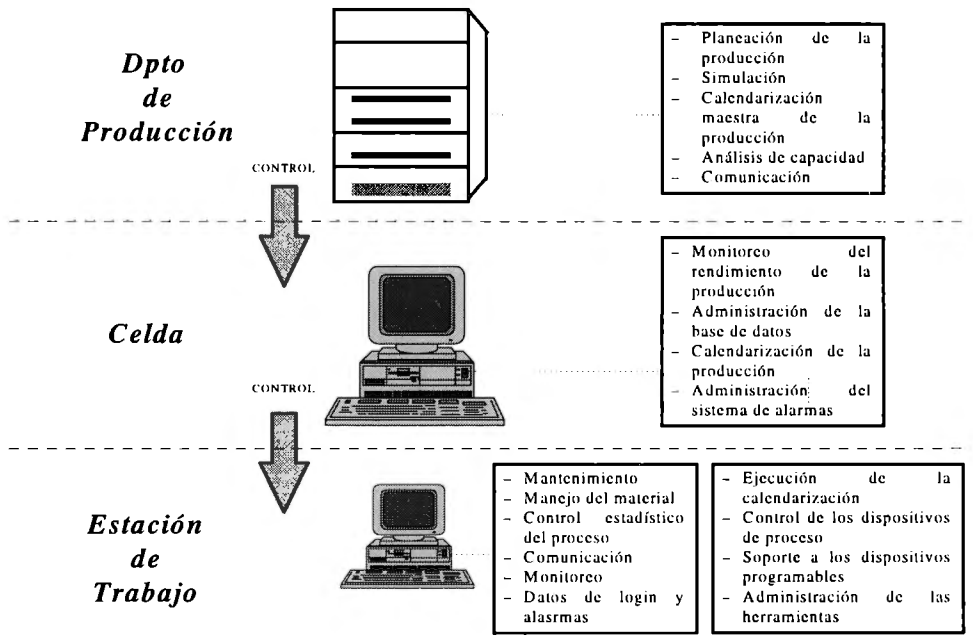


Figura 7.1.1.2.1. Jerarquía para las funciones de control

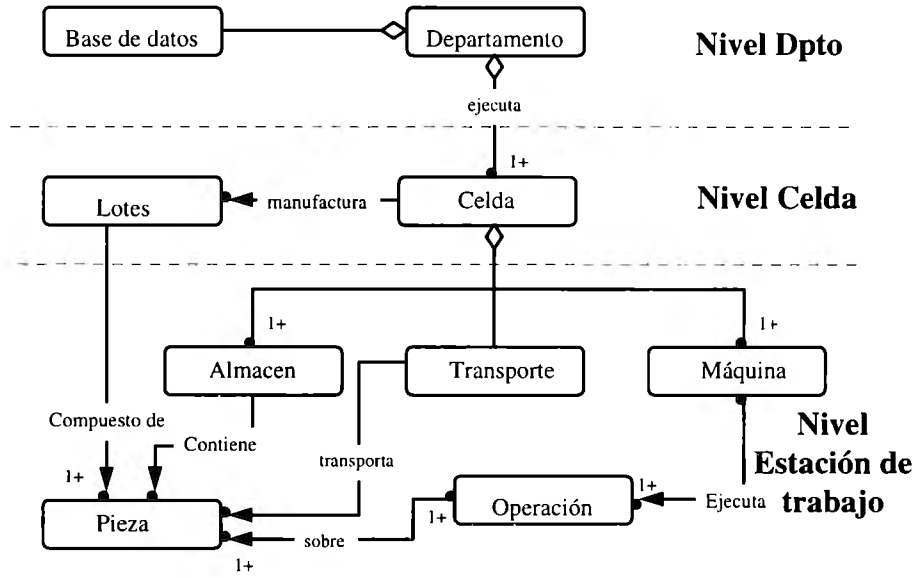


Figura 7.1.1.2.2. Modelo inicial del FMS

7.1.1.3 Aplicación patrón 2: Visibilidad y comunicación entre módulos de control

Cada módulo de la jerarquía establecida para el control y cuya función es la ejecución de los servicios, requiere servicios desde otros módulos o señales de eventos para así poder informar a otros módulos de sus estados.

Contexto

Los sistemas complejos se desarrollan en un tiempo relativamente largo. Hay dos posibilidades:

- la primera es construir módulos que serán colocados en un diseño ya existente,
- la segunda es construir "frameworks" que se toman para integrar módulos de control ya existentes.

En el primer caso, los módulos se construyen teniendo "visibilidad" del ambiente en el que operarán, mientras que el segundo caso esto no sucede. Por "visibilidad" se debe entender que

cuando se tienen dos objetos A y B, con un enlace entre los dos, y A quiere enviar un mensaje a B, B debe ser visible para A de alguna manera. Este concepto es importante porque los módulos se comunican, y la forma en que sus comunicaciones son establecidas dependen de la visibilidad que tienen entre ellos. Los módulos que se están comunicando pueden estar en forma imperativa, como en el caso de un comando enviado por un módulo a otro (es decir el módulo que envía el comando tiene visibilidad sobre quien recibe el comando), o pueden estar en forma reactiva, como en el caso del monitoreo de eventos (es decir un módulo difunde eventos para que sean captados por los módulos que hacen el monitoreo ya que quien envía no sabe exactamente a quien hacerlo por no tener visibilidad sobre estos). Las dos situaciones tienen una relación directa con los dos mecanismos básicos de comunicación entre objetos, aquí llamados **invocador/proveedor (call/provider (C/P))** y **difusión/escucha (broadcaster/listeners (B/L)**) [Aarsten, Brugali, Menga, 1996].

- El mecanismo C/P se usa cuando un objeto invoca un método de otro objeto. Este mecanismo es muy utilizado en la programación OO.
- El mecanismo B/L se logra dando a todos los objetos del "framework" la capacidad de difundir y escuchar eventos. El termino "evento" es usado acá para indicar un mensaje enviado en forma de difusión en un cierto instante de tiempo por un objeto, identificado por un nombre simbólico y con algún dato asociado con éste.

Solución

La reusabilidad en una jerarquía de capas de control se puede mejorar eliminando ciclos de visibilidad, no permitiendo que dos controles en la misma capa se comuniquen directamente, y siguiendo en su lugar las reglas siguientes:

- a dos módulos de la misma capa no se les dará derecho de comunicarse directamente,
- cuando un cliente dirige un mensaje (comando) a un servidor, ésto deberá hacerse explícitamente en forma C/P,
- cuando un servidor responda, por ejemplo enviando la información que está monitoreando, esto deberá hacerse señalando eventos a través de un mecanismo B/L.

Todos los módulos de un sistema de control son entidades que se comunican entre sí por medio de mecanismos C/P o B/L. La comunicación se da usando un mecanismo C/P a partir de la capa que contiene los módulos más nuevos (o en cualquier caso el más volátil) hacia la capa que contiene el más antiguo (o el más estable), y un mecanismo de comunicación del tipo B/L en el sentido contrario.

Aplicación del patrón en el FMS

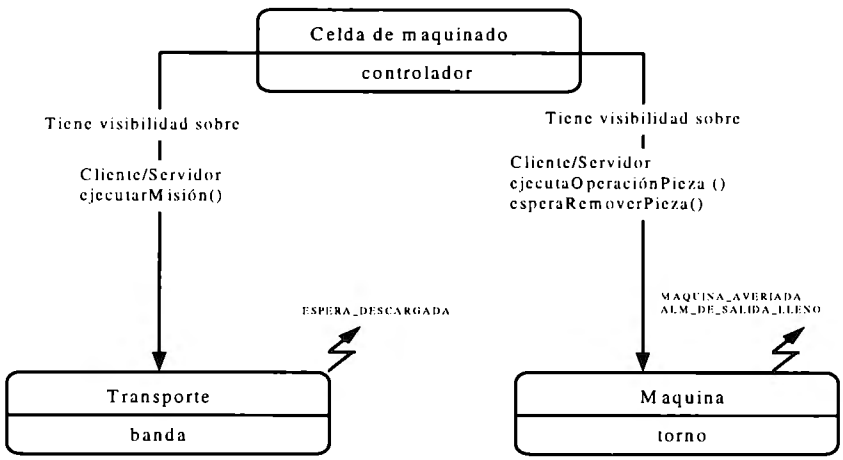


Figura 7.1.1.3.1. Ejemplo de la Visibilidad y comunicación en el FMS

En el ejemplo de FMS, la celda para maquinado, que aparece en la figura 7.1.1.3.1, solicita servicios por medio del mecanismo invocador/proveedor en forma directa a las máquinas (como el de ejecutar una operación sobre una pieza, ejecutarOperaciónPieza() ó esperar para remover una pieza, esperaRemoverPieza(), y otros más) , y al componente de transporte (como el de ejecutar una misión, ejecutarMisión()); pero la evolución de la celda de producción la determina los eventos difundidos por sus componentes utilizando el mecanismo difusión/escucha (representado en la figura por el símbolo), por ejemplo el evento ALM_DE_SALIDA_LLENO (que indica cuando se llena el almacén que guarda las piezas que van saliendo de una máquina) en el momento en que entra una pieza al almacén de salida de una máquina.

7.1.1.4 Aplicación patrón 3: Establecer categorías de objetos para la concurrencia

Los módulos de control en el sistema ejecutan servicios concurrentemente, y la concurrencia asume diferentes escalas de granulidad.

Contexto

El primer problema es que existen actividades concurrentes en un sistema complejo, y que la concurrencia asume diferentes escalas de granulidad. Las actividades concurrentes de grano fino se realizan con actividades simples que tienen una débil cohesión; la concurrencia de grano grueso se puede representar con operaciones compuestas de una secuencia ordenada de acciones que tienen una fuerte cohesión; esta concurrencia de grano grueso se representa con módulos de control operando en paralelo.

Un segundo problema es que los módulos de sistemas concurrentes complejos interactuan en varias formas, en particular:

- a través de entidades donde los servicios almacenan datos
- a través de entidades donde los servicios entran en conflicto respecto a que tanto se puede compartir un recurso limitado, o
- a través de entidades que realizan (sub)servicios desarrollados sobre el tiempo.

Solución

Use procesos concurrentes para expresar la concurrencia en su forma global. En G++, los procesos concurrentes son encapsulados en objetos activos; en este caso los procesos representan los servicios ofrecidos por un objeto en respuesta a un requerimiento de una operación.
Los **objetos pasivos** no poseen hilos de control, y dependen de hilos del cliente para la ejecución de sus funciones. Se pueden subdividir en objetos *secuenciales* y objetos *bloqueantes*. La semántica de *secuencial* se garantiza solamente en la presencia de un único hilo de control. Tales clases tienen una "S" en su etiqueta. Los objetos *bloqueantes*, designados para mantener su semántica cuando más de un hilo los accede, ofrece la primitiva *"wait"* que es indispensable para

bloquear hilos del cliente y permitir que puedan interactuar. Tales clases tienen una "B" en su etiqueta.

Por otro lado los **objetos Activos** difieren de los objetos pasivos debido a que poseen, crean e internamente administran uno o más hilos de control los cuales gobiernan la ejecución de sus servicios. Una clase **HiloDeControl** es usada para representar un servicio concurrente. Los objetos activos son los candidatos naturales para representar módulos de control; tienen una "A" en su etiqueta.

Modele las actividades concurrentes de una arquitectura de control compleja, de acuerdo a su nivel de granulidad, por acciones activadas por eventos, por procesos secuenciales (servicios) que interactuan por medio de recursos compartidos, o por módulos concurrentes que intercambian requerimientos de servicios. Más aún, para soportar servicios concurrentes, identifique los objetos ***secuenciales***, ***los objetos bloqueantes*** *y* ***los objetos activos*** *y represente los módulos de control con los objetos activos.*

Aplicación del patrón en el FMS

La figura 7.1.1.4.1 muestra el modelo del FMS usando la nomenclatura OMT. Como puede verse el FMS es una estructura jerárquica conformada por objetos. En esta estructura los módulos de control de más alto nivel tienen visibilidad sobre los módulos de control de más bajo nivel, y los siguientes módulos de control, son identificados como objetos activos (etiquetados con la letra A):

- El departamento, *departamento*
- las celdas, *celda_de_maquinado* para maquinado y *celda_de_ensamblaje* para el ensamblaje de lotes de piezas ;
- las máquinas, una colección de máquinas se identifica con el nombre *máquinas*;
- el sistema de transporte, *transporte*.

Almacenes de piezas semitrabajadas y finalizadas (llamados *almacenamiento_de_entrada* y *almacenamiento_de_salida)* se modelan como objetos bloqueantes, puesto que ellos actúan como recursos compartidos entre diferentes grupos de servicios de producción.

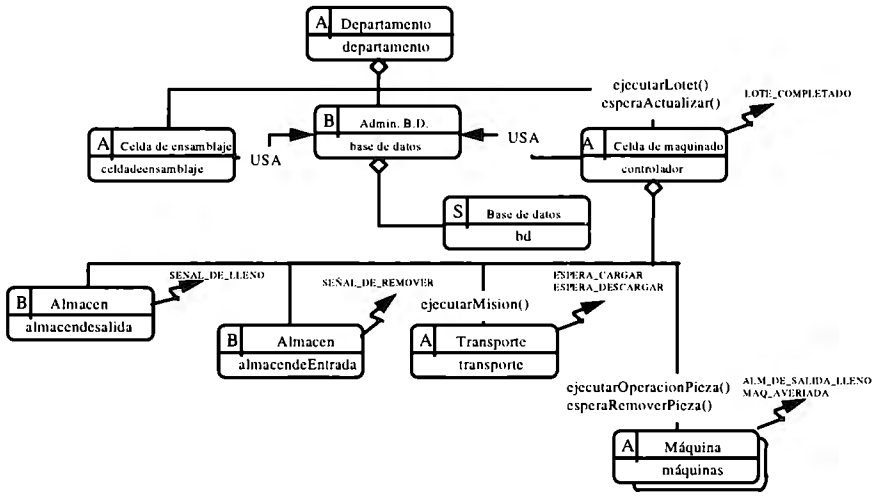


Figura 7.1.1.4.1. Posibles objetos activos, bloqueantes y secuenciales de la FMS

También en la figura puede verse el concepto de agregación o composición. Por ejemplo el departamento de producción está compuesto de la celda de ensamblaje, el administrador de la base de datos y la celda de maquinado. A su vez, la celda de maquinado está compuesta de los objetos almacén (almacenamiento_de_entrada y almacenamiento_de_salida), transporte y máquinas. Por otra parte se muestran ejemplos de los mecanismos de comunicación C/P y B/L enunciados en el patrón 2. Por ejemplo la *Celda de Maquinado* envía mediante el mecanismo C/P el mensaje *ejecutarOperaciónPieza()* y la *máquina* responde mediante el mecanismo B/L (representado en la figura por el símbolo $\lightning$) con el evento MAQ_AVERIDA (máquina averiada).

7.1.1.5 Aplicación patrón 4: Acciones disparadas por eventos

La comunicación a través de los eventos puede ser intra-servicio, generando una concurrencia de grano fino, o entre servicios resultando en un concurrencia de grano grueso.

Contexto

En el desarrollo de una aplicación, es posible encontrar requerimientos de concurrencia **intra-servicio**, o **inter-servicio**, cuando dos objetos en dos procesos concurrentes, en distintos módulos de control, desean comunicarse intercambiando eventos unos con otros.

Solución

El mecanismo B/L para emitir o escuchar eventos, que fue introducido con los módulos de control para comunicación entre el más antiguo y uno nuevo, es implementado en la *clase objeto* y es heredado por todas las clases del "framework".

Use el mecanismo B/L para modelar acciones concurrentes de grano fino para la interacción entre objetos secuenciales destinados a uno de los servicios ofrecidos por un objeto activo. Use este mismo mecanismos para la comunicación entre los módulos de control de las capas inferiores y los de las capas superiores.

Aplicación del patrón en el FMS

En buen ejemplo de requerimientos de concurrencia intraservicio seria un sistema de toma de decisiones basado en reglas para la calendarización de la producción. Por otro parte, para el caso de requerimientos de concurrencia inter-servicio, un servicio como el de *maquinar_una_pieza* genera un evento de falla que es escuchado por el servicio de *producción_de_un_lote* de la celda.

7.1.1.6 Aplicación patrón 5: Servicios "Esperando por"

Los servicios ofrecidos por un módulo de control esperan a que ocurra una condición o esperan por datos transferidos desde ó hacia un recurso compartido, antes de realizar alguna acción en concurrencia con otros servicios.

Contexto

Las dos situaciones enunciadas en el párrafo anterior indican la necesidad de entidades que puedan administrar la interacción entre los servicios.

Solución

Las entidades poderosas para este problema son los **objetos bloqueantes**. En efecto, en un mundo orientado a objetos la única forma como un servicio puede entrar en un estado de suspensión de sí mismo es enviando un mensaje (requiriendo un servicio que implica la espera de una condición) a un objeto bloqueante. El flujo será reanudado a partir del estado de espera cada vez que la condición apropiada sobre el objeto bloqueante ocurra.

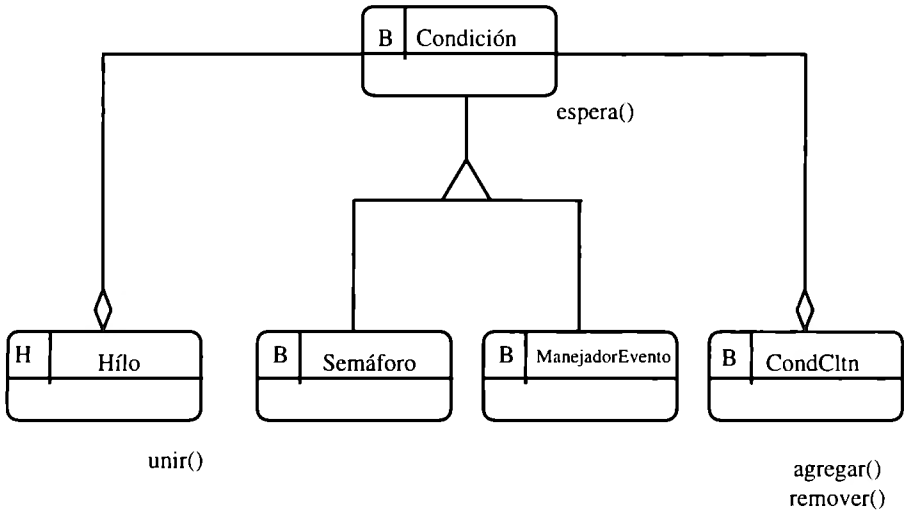


Figura 7.1.1.6.1. Objetos Bloqueantes

Condición es la clase abstracta usada por el "framework" para definir cualquier objeto bloqueante, a partir del cual implementaciones específicas tales como temporizadores (timers), semáforos, manejadores de eventos y colas compartidas (determinados almacenamientos, es decir **CondCollection**) pueden derivarse por herencia o encapsulamiento (ver figura 7.1.1.6.1).

Use los objetos bloqueantes para modelar los recursos compartidos del módulo de control cada vez que la interacción entre sus servicios concurrentes sea necesaria.

Aplicación del patrón en el FMS

En el FMS ejemplo, hay diferentes objetos bloqueantes (ver figura 7.1.1.6.2):

- el ***almacen_de_entrada*** y el ***almacen_de_salida*** que son del tipo ***CondCltn*** y que pertenecen a la clase ***Celda_de_maquinado***. Ellos actúan como un almacenamiento asignado: el servicio de la celda automáticamente se detiene si ésta trata de tomar una pieza cuando el

almacen_de_entrada está vacío o cuando se trata de colocar una pieza en el ***almacen_de_salida*** estando éste lleno.

- ***en_entrada***, ***en_trabajo*** y ***en_salida***, son también del tipo ***CondCltn*** y actúan como objetos bloqueantes de los servicios de la ***máquina***.
- la ***base_de_datos*** es modelada como un objeto bloqueante, ya que ésta es un recurso compartido por las celdas;
- la ***Celda_de_Maquinado*** tienen un *Manejador_de_Eventos* usado para bloquear sus servicios mientras espera por eventos provenientes desde el transporte, las máquinas y los almacenes. Acá el controlador de la celda usa un *Manejador_de_Eventos* para monitorear sus recursos (por ejemplo ALM_DE_SALIDA_LLENO o EN_ESPERA_ALM_DE_SALIDA_NO_LLENO) y para tomar las acciones apropiadas.

Los anteriores objetos bloqueantes pueden ser administrados por monitores (objetos que permiten la sincronización orientada a un recurso). El monitor contiene variables permanentes que almacenan su estado y un conjunto de operaciones que actúan sobre estas variables. Las variables mantienen sus valores entre cada activación del monitor, es decir son la memoria del monitor. El monitor trabaja bajo condiciones de sincronización que forzan una espera en la ejecución del procedimiento monitor. Si un procedimiento monitor es demorado por una condición de sincronización, otros procesos pueden entrar al monitor.

En un sistema de manufactura, el primer caso enunciado al principio de este patrón (esperar a que ocurra una condición) se da en la celda por el servicio de producción de un lote mientras se está esperando por el evento "ALM_DE_SALIDA_LLENO" de una máquina. Lo anterior quiere decir inicialmente que el servicio de producción de un lote asigna las piezas que vienen en el lote a la máquina adecuada para trabajar dicha pieza, y controla la entrada o salida de las mismas de la máquina que está dando el servicio. Por lo tanto, cuando una pieza está lista para salir de la máquina pero una condición indica que el almacen de salida (o sea aquel en el cual se van acomodando las piezas al salir de la máquina está lleno), el servicio de producción de un lote debe primero retirar piezas de dicho almacenamiento para luego proceder a agregar nuevamente piezas al mismo una vez que la condición de "ALM_DE_SALIDA_LLENO" no existe. En el segundo caso (esperar por datos ha ser transferidos desde ó hacia un recurso compartido) se da, por ejemplo, en el servicio de transporte bloqueado mientras se espera por el primer transporte

disponible de aquellos que no estén ocupados. En este caso el servicio de transporte es un recurso compartido en el cual las operaciones pueden ser ejecutadas por diversos procesos que pueden intercambiar información en forma indirecta, es decir existen variables que pueden ser asignadas o leidas por cualquiera de los procesos. Según lo anterior una variable booleana podría indicar si hay transporte disponible o no, mientras otra variable podría indicar cual transporte de los disponibles ha sido asignado para ejecutar el servicio.

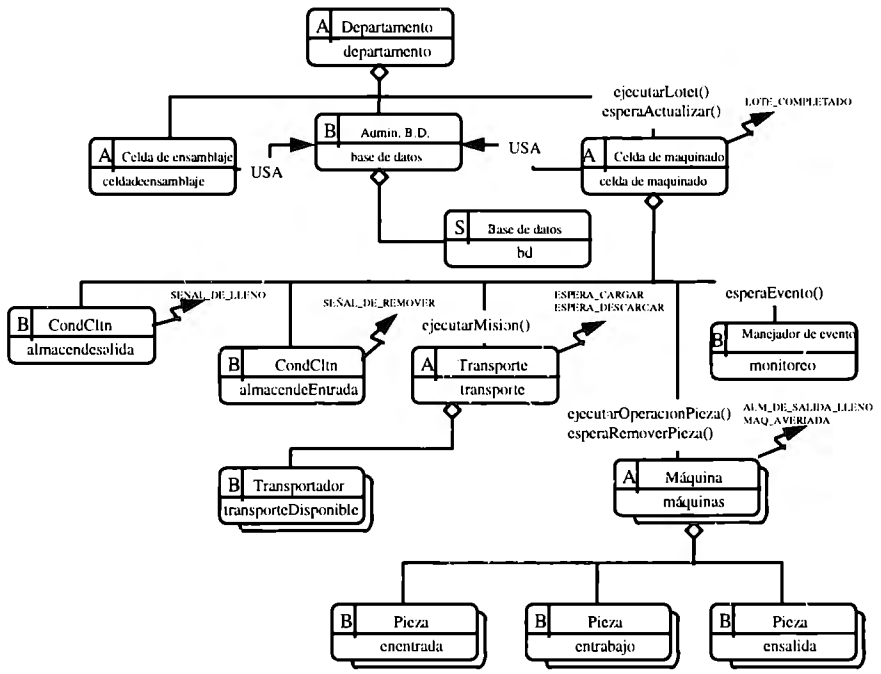


Figura 7.1.1.6.2. Diseño lógico completo del FMS

7.1.1.7 Aplicación patrón 6: El Cliente/Servidor/Servicio: implementando módulos de control

Los módulos de control son objetos activos que ofrecen múltiples servicios en forma concurrente y están presentes en diferentes niveles de la arquitectura jerárquica del sistema.

Contexto

En el diseño de módulos de control, un primer requerimiento es que haya más de un servicio en cada momento para ese cliente.

Otro requerimiento es que se desea una representación común de los módulos de control en los diferentes niveles de la jerarquía para propósitos de estandarización, así como para forzar la reusabilidad y facilitar la tarea de distribución de la aplicación. Por ésto, es necesario representar un sistema como una estructura de máquinas virtuales jerarquizadas por capas.

Finalmente, los módulos de control deberían tener la capacidad para encapsular los recursos y los servicios que manipulan.

Solución

Los anteriores requerimientos se satisfacen con el modelo Cliente/Servidor/Servicio propuesto en este patrón. Se necesitan dos clases para soportar este modelo: el ***Servicio*** y el ***Servidor*** (ver figura 7.1.1.7.1).

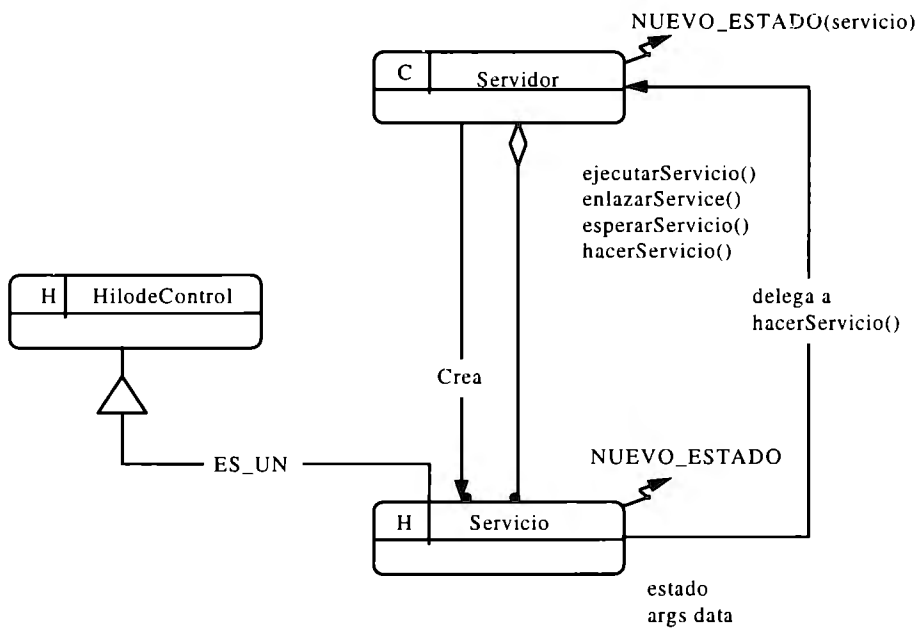


Figura 7.1.1.7.1. Cliente/Servidor/Servicio

El ***Servicio*** extiende el ***HiloDeControl*** de tal manera que éste llega a ser una máquina de estado finito que involucra la dinámica de la aplicación y tiene las siguientes características: siempre pertenece a un único servidor propietario, tiene datos internos (objetos secuenciales), mantiene un valor de estado simbólico, y difunde eventos con el nombre del estado de entrada cada vez que ocurre una transición de estado. Raramente necesita ser redefinido ya que delega la ejecución de su dinámica a un método de comportamiento propio del servidor propietario.
El ***Servidor*** es una clase abstracta que define la implementación común de los objetos activos, y que ha sido redefinida con la idea de construir servidores concretos.

Éste contiene una ***colección de objetos de servicio***: cada servicio ejecuta las operaciones del servidor en forma concurrente, mientras comparte el mismo conjunto de recursos. Un servicio del servidor puede, a su vez, usar otro (sub)servidor y en este caso actúa como un cliente para ese servidor.

Para ser accesado, el ***Servidor*** ofrece tres métodos públicos: ***ejecutarServicio(), esperarServicio(), enlazarservicio()*** que representan requerimientos de servicio asíncronos, síncronos, y síncrono diferido, respectivamente. Ellos, aceptan como argumentos, objetos secuenciales solamente por valor, objetos bloqueantes y pasivos son aceptados por referencia también.

Cada vez que uno de estos tres métodos públicos es llamado, un nuevo ***Servicio*** es creado, y éste es agregado a la lista de servicios activos; y ejecuta el método privado ***hacerServicio()*** de su ***Servidor***. El método *hacerServicio()* ha sido redefinido en la clase concreta derivada.

Más aún el ***Servidor*** retransmite al exterior los eventos relacionados con el servicio de tal manera que su estado puede ser monitoreado por otros objetos.

Inherentemente cree ***Servidor y redefinalo como Servidor::hacerServicio()*** *para obtener un objeto activo único con múltiples hilos concurrentes de control, el cual comparte los componentes del* ***servidor*** *así como recursos comunes entre ellos.*

Aplicación del patrón en el FMS

Un ejemplo claro en un FMS del requerimiento de poder suministrar más de un servicio en cada momento a un cliente, es que en una celda se puede estar administrando más de un lote de piezas. Es decir a la celda están llegando lotes de diferentes tipos de piezas a las cuales les corresponden servicios diferentes en máquinas diferentes. Al llegar estos lotes de piezas deben ser distribuidos dependiendo del tipo de pieza a la máquina que puede ofrecerle los servicios necesarios para ese tipo de pieza. De esta manera el controlador se convierte en el cliente que solicita los servicios a las máquinas que en este caso son los servidores de los mismos. Y de esta manera se establece una relación entre el controlador y las máquinas del tipo Cliente/Servidor/Servicio.

El patrón ***Cliente/Servidor/Servicio*** es fundamental ya que éste es la estructura esencial para la solución de problemas concurrentes. Más aún, puesto que los módulos de control en el modelo de referencia FMS de la figura 7.1.1.2.1 son objetos activos, el patrón aplica en un sistema de control FMS.

7.1.1.8 Aplicación patrón 7: Implementación de Módulos de Control de "Múltiples Tipos de Servicios"

Los módulos de control administran diferentes grupos de recursos compartidos y posiblemente a menudo ofrecen diferentes tipos de servicios.

Contexto

Los módulos de control pueden necesitar ofrecer múltiples tipos de servicios, y no únicamente múltiples servicios concurrentes del mismo tipo.

La clase ***servidor*** ofrece un objeto activo con una interfase "estándar", representada por los métodos *ejecutarServicio(), esperarServicio(), y enlazarServicio()*, y ejecuta múltiples instancias del mismo "servicio".

Solución

Los módulos de control de "Múltiples Tipos de Servicios" pueden obtenerse por herencia a partir de un *servidor* (ver figura 7.1.1.8.1) y usando los siguientes lineamientos:

- Deben identificarse los recursos que necesita el módulo de control. Los recursos que son compartidos por los servicios de un objeto activo pueden ser solamente objetos bloqueantes (Condición o CondCltn) u otros objetos activos (servidor). No hay objetos secuenciales dentro de *servidor* , ya que ellos están convenientemente encapsulados dentro de los objetos bloqueantes o vinculado a los servicios.
- Los diferentes tipos de servicios ofrecidos pueden ser identificados y su comportamiento deberá ser especificado en términos de máquinas de estado finito extendidas por medio de objetos secuenciales vinculados a los servicios, e implementados por métodos de comportamiento privado.
- El método *servidor::hacerServicio()* debe redefinirse con la idea de lograr un esquema de alternativas (switch case) el cual es controlado por un nombre del parámetro, así como para acceder los diferentes tipos de servicio.

La razón práctica para este comportamiento es que en la librería G++ es más fácil para el trío *esperar-enlaza-ejecutarServicio()* crear siempre la ejecución *hacerServicio()*.

La representación de todos los comportamientos del servicio en el "framework" con el único método *hacerServicio()* del *servidor* es una opción de diseño alternativo.

En lugar de *servicio/cliente/servidor* use un patrón que será llamado **Ambiente Delegado**: los comportamientos del servicio son métodos que ofrece el objeto pero pueden ser ejecutados concurrentemente gracias al contexto delegado a partir de las instancias de la clase *servicio*. Esta solución tienen dos ventajas ya que ofrece un ambiente de ejecución protegido para cada instancia del servicio, y al mismo tiempo permite a todos los servicios dentro de un servidor compartir los recursos servidores sin problemas de visibilidad .

Módulos de control son implementados a partir de ***servidor****, pero encapsulando o referenciando recursos compartidos, los cuales serán objetos bloqueantes u objetos activos, y especificando el comportamiento de los métodos de sus servicios en términos de máquinas de estado finito extendidas.*

Aplicación del patrón en el FMS

En la ***celda_de_maquinado*** se pueden identificar los servicios *"lotedeProducción()" e "instalarMáquina()"* los cuales pueden ser implementados por métodos de comportamiento privado, por ejemplo *hacerLotedeProducción() y hacerInstalacióndeMáquina().*

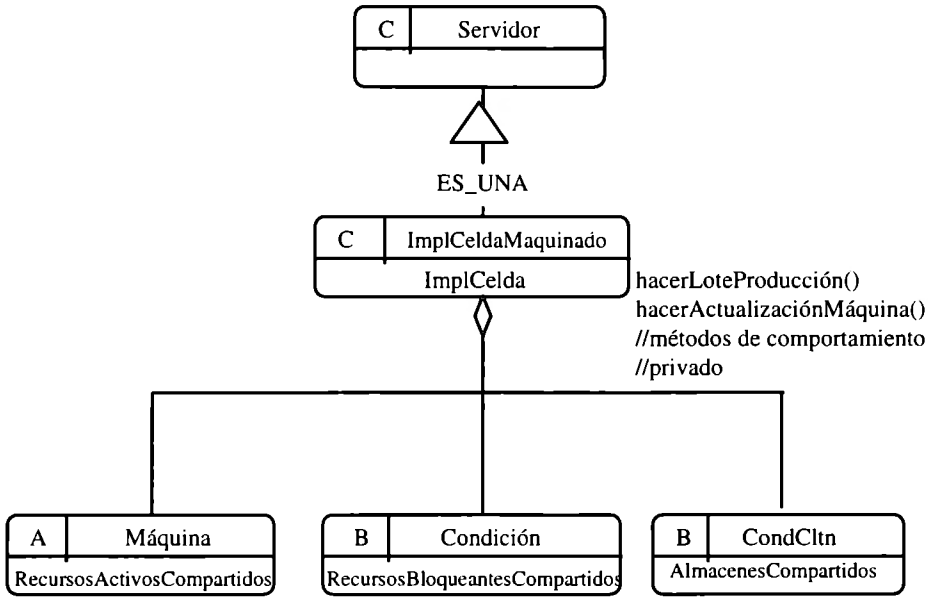


Figura 7.1.1.8.1. Implementación de los Módulos de Control

En la figura 7.1.1.8.1 se puede apreciar cómo los módulos de control de "Múltiples Tipos de Servicios" pueden obtenerse por herencia a partir de un *servidor.* En dicha figura también se han aplicado los lineamientos enunciados en la solución, como el de la herencia obtenida por el objeto *ImplCeldaMaquinado* apartir del objeto servidor. Otro lineamiento aplicado es la identificación de los recursos (como las *máquinas* y aquellos que operan bajo el esquema de condiciones tales como los *almacenes*). Además estos recursos son identificados como activos y bloqueantes tal como lo plantea la solución.

En el diseño de la *celda_de_maquinado* del FMS, definido por la clase *implementaciondelaCeldadeMaquinado*. Esta hereda a partir *servidor* y sus miembros de datos (ver figura 7.1.1.6.2) son el sistema AGV de *transporte*, instancia de la clase *transporte*, la colección de máquinas *máquinas*, almacenes de piezas semitrabajadas y terminadas (*almacen_de_entrada* y *almacen_de_salida*, de la clase *CondCltn*).
El servicio de ruteo de piezas entre máquinas y almacenes está expresado en el método *hacerLotedeProducción()*; más que un servicio puede ser activado en cierto momento. Otros servicios toman en consideración instalación y mantenimiento de máquinas.

7.1.1.9 Aplicación patrón 8: La interfase para módulos de control

Los módulos de control ofrecen diferentes tipos de operaciones a sus clientes.

Contexto

Las clases derivadas por herencia a partir de *servidor* en el patrón 8 tienen la propiedad de ser objetos activos y también encapsulan recursos para el módulo de control que representan.

Solución

Use un objeto **interfase** para accesar cada **servidor.** La **interfase** ofrece las operaciones del servidor a los clientes.

Cuando un nuevo objeto activo es concebido y los servicios ofrecidos por este son identificados, una nueva clase **interfase** es generada, asignando a ésta las siguientes características para cada tipo de servicio:

- un enlace al objeto de implementación;

- una operación interfase, la cual delega uno-a-uno, de acuerdo a las tres semánticas de sincronización de la especificación, sus funcionalidades para que sean tres métodos estándar de **servidor**

En ambos casos el objeto **interfase** puede ser generado automáticamente por un ambiente CASE a partir de la especificación del objeto activo. Para objetos activos, aún en aquellos que no serán distribuidos como se describe en los dos patrones siguientes, esta separación a menudo ofrece reusabilidad.

Los módulos de control-clientes tienen que referirse a sus módulos de control-servidores por medio de objetos ***interfase*** *los cuales están enlazados a sus* **implementaciones** *a través de una relación de herencia o uso.*

Aplicación del patrón en el FMS

En las figura 7.1.1.9.1 y 7.1.1.9.2 se puede ver como se usa un objeto **interfase** para accesar cada **servidor**, determinando de esta manera que la interfase será quien ofrezca las operaciones del servidor a sus clientes.

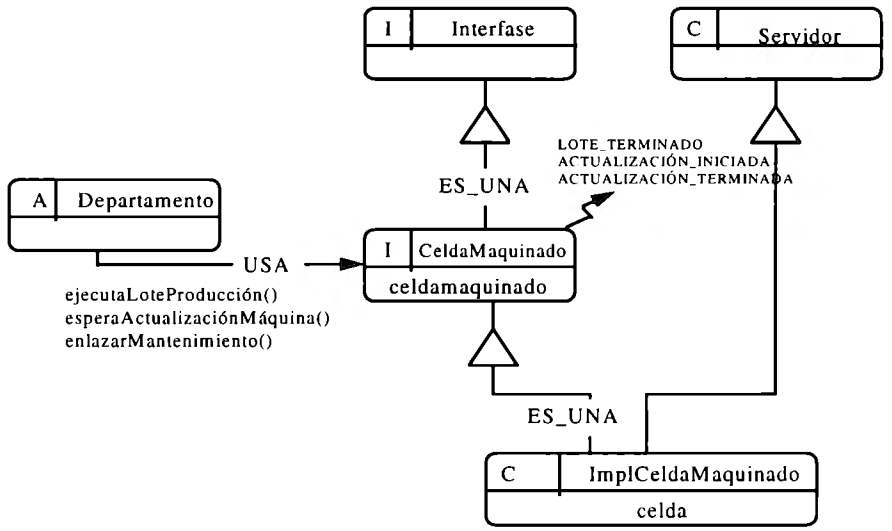


Figura 7.1.1.9.1. Interfase e implementación enlazados por herencia

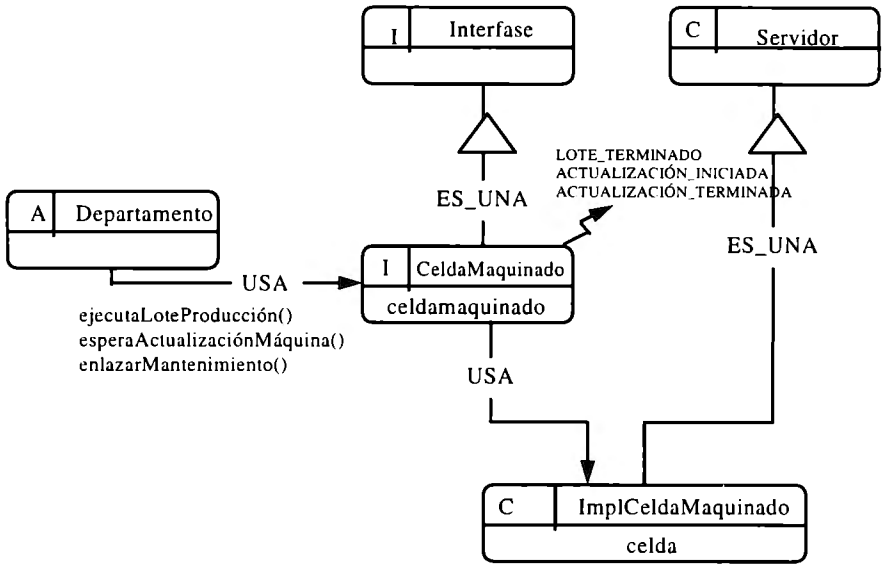


Figura 7.1.1.9.2. Interfase e implementación enlazados por USA

En el FMS podemos concebir un nuevo objeto e igualmente identificar el servicio ofrecido, por ejemplo *celda_de_maquinado* puede realizar *"lotedeProducción"*, *"actualizaciónMáquina"* y *"mantenimiento"*, entonces una nueva clase **interfase** es generada, teniendo en cuenta el enlace al objeto de implementación y la operación interfase para las tres semánticas de sincronización de la especificación. Por ejemplo *ejecutarLotedeProducción()* es asíncrono y llama a *ejecutarServicio()*, *esperaInstalaciónMáquina()* es síncrono y llama a *esperaServicio()*, *enlazarMantenimiento()* es síncrono diferido y llama a *enlazarServicio()*.

Las figuras 7.1.1.9.1 y 7.1.1.9.2 presentan dos arquitecturas de solución alternativas para este patrón, donde interfase e implementación son enlazadas por relaciones de herencia y de "uso", respectivamente.

7.1.1.10 Aplicación patrón 9: Prototipo y realidad

Cualquier aplicación compleja requiere prototipos y simulación de los diferentes elementos a ser integrados antes de derivar una implementación.

Contexto

La necesidad de prototipos y simulación está particularmente presente en sistemas de control y en otros tipos de aplicaciones que son interfaces para controladores de hardware o son por su naturaleza distribuidas.

Solución

Los siguientes patrones consideran la transición del modelo lógico al modelo físico mediante la sustitución de los prototipos por su contraparte física.

Para cualquier objeto que necesite ser simulado, mantenga dos versiones coexistentes:

- el *" prototipo simulado o emulado"*, el cual simula o emula el comportamiento del objeto;
- el *"objeto real"*, el cual involucra el objeto físico.

El *"objeto real"* deberá ser un manejador de dispositivo (hardware), una encapsulación de una funcionalidad externa como una base de datos relacional, o un sustituto para un objeto externo (de arranque o iniciación) como se describe en el siguiente patrón.

Cuando se pase de la simulación a la realidad, reemplace el objeto simulado con el objeto (por ejemplo manejador de dispositivo) el cual nos lleva a la realidad.

La consistencia en esta transición es garantizada o bien generando por herencia las dos implementaciones desde una clase común básica que define su interfase, o por la aplicación del patrón 8 sustituyendo el objeto de implementación deseado en la relación de uso del objeto interfase.

La evolución desde un diseño lógico a uno físico y desde el prototipo a la implementación se logra manteniendo dos representaciones de la misma entidad (el "prototipo" y la "realidad") y transformado el prototipo a la realidad física, lo cual se hace explotando el polimorfismo o la relación de uso.

7.1.1.11 Aplicación patrón 10: Distribución de los módulos de control

Los módulos de control usualmente residen sobre computadoras remotas o dispositivos periféricos, interconectados a través de una red de comunicación común. Estos módulos definen la arquitectura física que deberá obtenerse para el sistema distribuido final.

Contexto

Los sistemas distribuidos complejos comparten la necesidad de simular discutida en el patrón anterior. Cuando se va de una simulación no distribuida a la realidad distribuida, los objetos que son movidos a nodos remotos no son gran parte del programa original; ellos llegarán a ser programas independientes

Como en el patrón anterior, el resto del sistema no deberá ser afectado por el movimiento de algunos objetos a un nodo remoto. Además, el movimiento de los objetos a nodos remotos deberá ser relativamente fácil con la idea de que el sistema no cambie con la arquitectura física, esto es, se desea explotar un ambiente evolutivo.

Solución

Cree objetos de grano grueso, en G++ llamados *contextos remotos*, uno para cada nodo que contenga objetos remotos. El contexto remoto contiene uno o más objetos (típicamente activos), junto con los servicios que éstos necesitan.

El contexto remoto escucha los requerimientos que están llegando por la red de comunicación desde otros nodos, desempaquetando estos requerimientos y reenviando ellos al objeto adecuado. Si los objetos en un contexto remoto hacen requerimientos a otros objetos que residen en otros nodos, el programa instalable deberá también contener los objetos **de arranque o iniciadores**, como se explicó anteriormente.

En el sistema original, reemplace los objetos que han sido movidos a otros nodos con objetos **arrancadores** (stub) los cuales encapsulan la interfase de distribución y comunicación. Estos

arrancadores (stubs) tienen la misma interfase del objeto que reemplazan, y reenvían cada requerimiento sobre la red o enlace de comunicación al objeto remoto que ellos representan. Como en el patrón anterior, la consistencia se asegura o bien por herencia de polimorfísmo, o por delegación.

Cada **Arrancador** y **ContextoRemoto** encapsula una instancia de la clase **Red**. Mientras las primeras dos clases raramente se modifican, la **Red** podría necesitar adaptación a una plataforma de comunicación particular. Note que una herramienta CASE suficientemente inteligente podría automatizar la transición a un sistema distribuido generando los arrancadores necesarios y los contextos remotos, y reemplazando los objetos de simulación original con los arrancadores.

Desarrolle el prototipo para una aplicación distribuida usando ***arrancadores*** *para crear implementaciones alternativas de módulos remotos de control a ser sustituidos en el modelo, y* ***ContextosRemotos*** *que encapsulen esos módulos remotos de control y suministre un ambiente de contexto para ellos.*

Estas dos clases son un par ofrecido por el "framework" y son raramente modificados, ya que ellos delegan la implementación de los protocolos de bajo nivel a un objeto ***Red****, el cual envuelve los manejadores de comunicación (por ejemplo el ORB), y será suministrado para cada diferente implementación.*

Aplicación del patrón en el FMS

En la figura 7.1.1.11.1 la aplicación del patrón en el FMS se hace por medio de herencia de polimorfismo, es decir el objeto *CeldaMaquinado* que en este caso es el prototipo y el objeto *ArranCeldaMaquinado (o iniciador)* están enlazados por una relación de herencia en la que el objeto *ArranCeldaMaquinado* hereda su comportamiento (métodos) de *CeldaMaquinado* y *Arrancador*, objetos donde está definido dicho comportamiento.

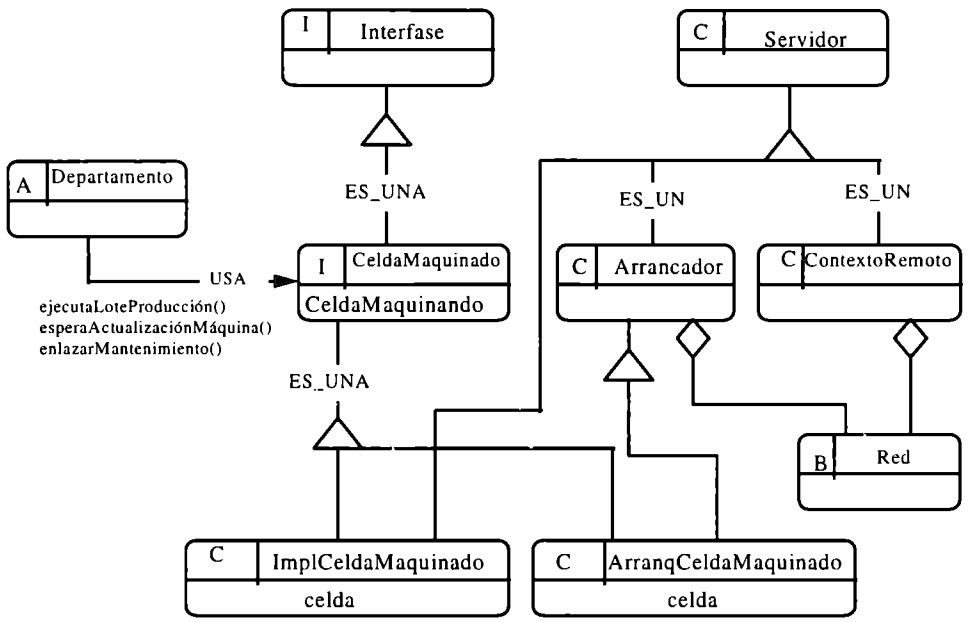


Figura 7.1.1.11.1. Prototipo e Iniciador enlazado por herencia

En el caso de la figura 7.1.1.11.2 la aplicación del patrón en el FMS se hace por medio de una relación de uso, es decir el objeto *CeldaMaquinado* delega la ejecución de los servicios a los *ArranCeldaMaquinado* objetos *ImpiCeldaMaquinado* y *ArranCeldaMaquinado*. En la figura 7.1.1.11.2 el prototipo (*CeldaMaquinado)* y el iniciador (arrancador) están enlazados por una relación de uso, en la que prototipo u objeto interfase básicamente delega la ejecución de los servicios al iniciador de la celda de maquinado (*ArranqCeldaMaquinado*).

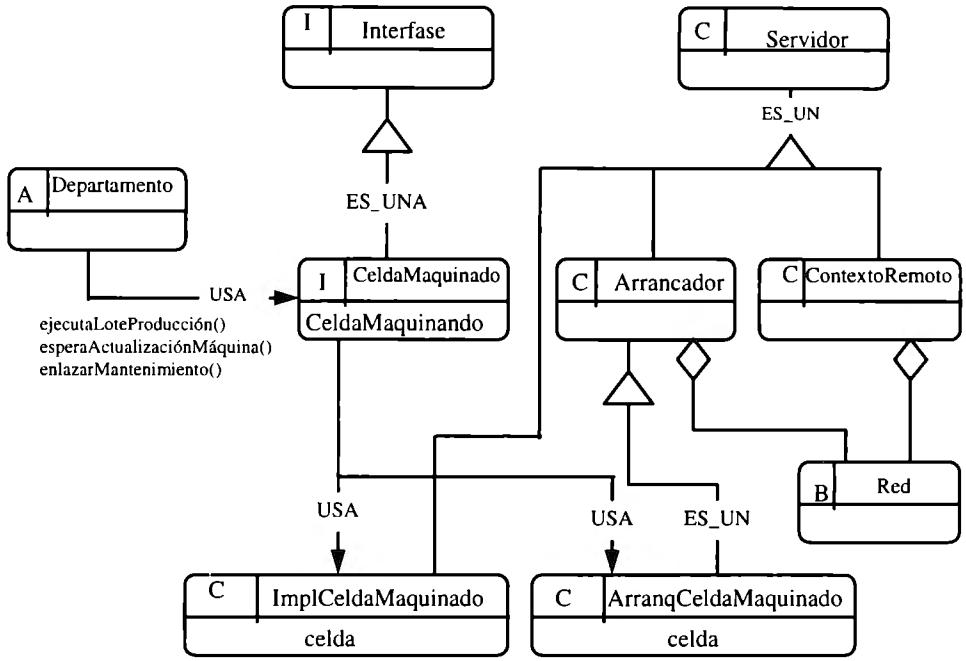


Figura 7.1.1.11.2. Prototipo e iniciador enlazados por Uso

Distribución del departamento y una celda

Asumimos que los módulos de control del **Departamento** y la **celda_de_maquinado** son instalados sobre computadoras respectivamente interconectadas a través de un backbone de red en una amplia fábrica y que las celdas periféricas, en particular la colección de máquinas, están conectadas a la computadora de la celda a través de LAN's en la planta del departamento.

Enfocándonos sobre la distribución de la celda, la figura 7.1.1.11.3 muestra como en el controlador departamental, el **arrancador_de_celda_de_maquinado** sustituye el **Implementador_de_celda_de_maquinado**, el prototipo del módulo de control de la celda. El **arrancador_de_celda_de_maquinado** se comunica con su correspondiente **ContextoRemoto**, el cual encapsula la **Celda_de_maquinado**. A su vez el **Implementador _de_celda_de_maquinado** contiene una colección de implementaciones de aquellas **máquinas**, también, ha sido sustituido por sus respectivos **arrancadores.**

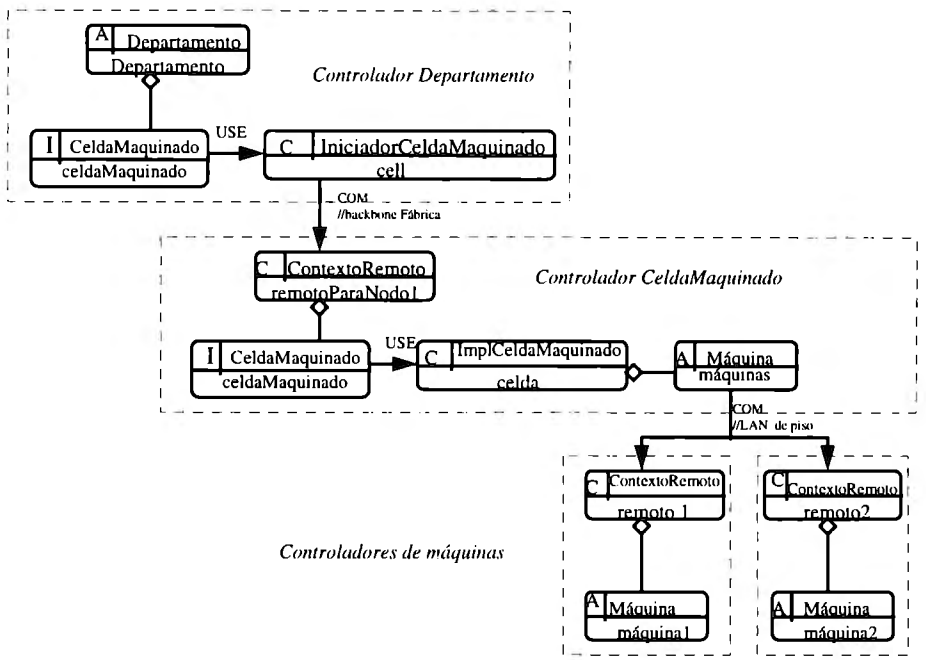


Figura 7.1.1.11.3. Distribución Física

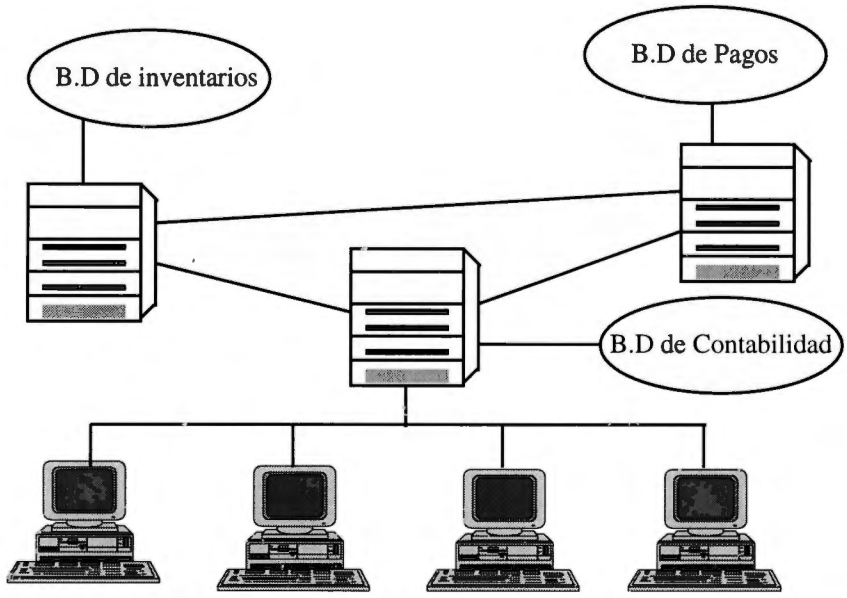
7.2 Un ambiente distribuido: alternativas para la distribución de datos

Debido a algunas desventajas de los sistemas centralizados, las organizaciones se están inclinando por los sistemas distribuidos cliente-servidor para cubrir mejor sus demandas. Dos de las más usadas tecnologías de distribución de datos son las **bases de datos distribuidas** y **las bases de datos replicadas** [SUN Microsysterns, 1994].

7.2.1 Bases de datos distribuidas

Un sistema administrador de bases de datos distribuidas es una reunión de bases de datos independientes, localizadas físicamente en diferentes servidores sobre una red local o de área amplia que aparece ante los usuarios y programas como una sola base de datos lógica (ver figura 7.2.1.1)

Un sistema distribuido soluciona la limitación sobre el número de usuarios que un sistema mainframe puede soportar, dividiendo una gran base de datos en componentes discretos más pequeños. Estos componentes se pueden colocar sobre servidores con la capacidad de procesamiento y almacenamiento adecuados. Se puede localizar cada componente en forma más cerrada con respecto a los usuarios que más frecuentemente necesitan de ellos.



DEPARTAMENTO DE CONTABILIDAD

Figura 7.2.1.1 B.D distribuidas localizadas independientemente en diferentes sitios

Adicionalmente, cuando los datos son divididos en múltiples bases de datos, copias del mismo ítem de datos pueden existir en múltiples sitios. Por supuesto, el sistema distribuido necesita asegurar la integridad de los datos manteniendo todas las copias de los datos en forma consistente. Uno de los métodos actualmente usados para mantener la consistencia a través de múltiples servidores de bases de datos en un ambiente de bases de datos distribuido es el protocolo **a dos fases**. Como el nombre lo indica, este trabaja en dos fases: la **fase 1** bloquea todas las copias de los datos y la **fase 2** o bien actualiza todas las copias (ver figura 7.2.1.2), o si hubo un error en la actualización, recupera todas las copias a su estado original. Así, el protocolo de culminación a dos fases (two-phase commmit) requiere que todos los sitios estén disponibles antes de que cualquier transacción pueda ser completada.

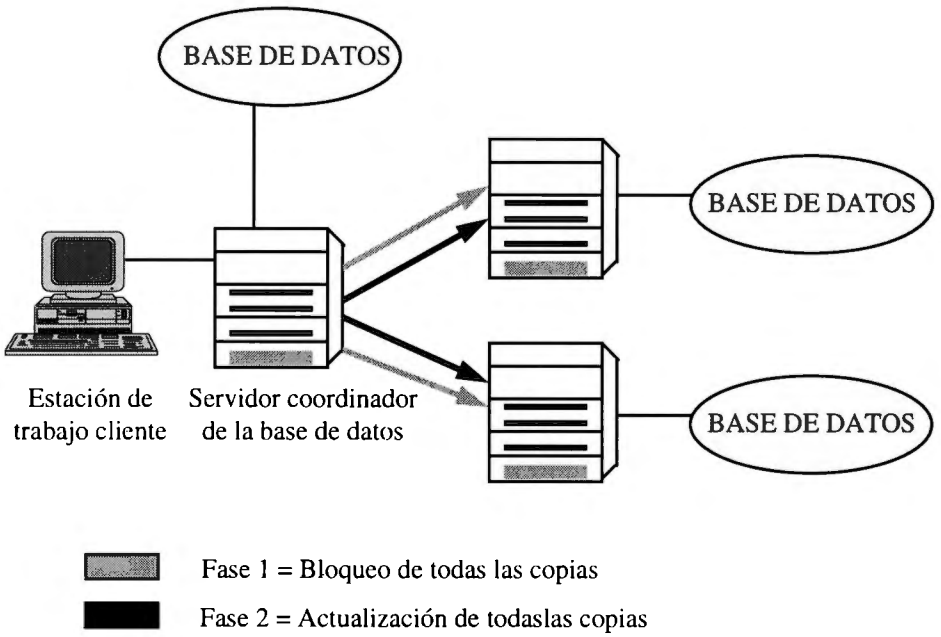


Figura 7.2.1.2. Protocolo de culminación a dos fases

7.2.2 Bases de datos replicadas

La replicación hace copias de datos cada vez que éstos son usados (figura 7.2.2.1) y estas copias son entonces distribuidas a sitios remotos. La replicación es la primera opción práctica para bases de datos distribuidas.

Hay dos tipos de bases de datos replicadas: bases de datos replicadas que actualizan todas las copias de la base de datos al mismo tiempo (esto se llama, **replicación sincronizada**) y aquellas que permiten a algunas de las copias contener diferentes datos por un periodo de tiempo corto antes de que todas sean actualizadas (esto es llamado, **replicación asincrónica).**

La principal diferencia entre los dos tipos de bases replicadas es su habilidad para superar los problemas cuando las actualizaciones de los datos son sobre una WAN.

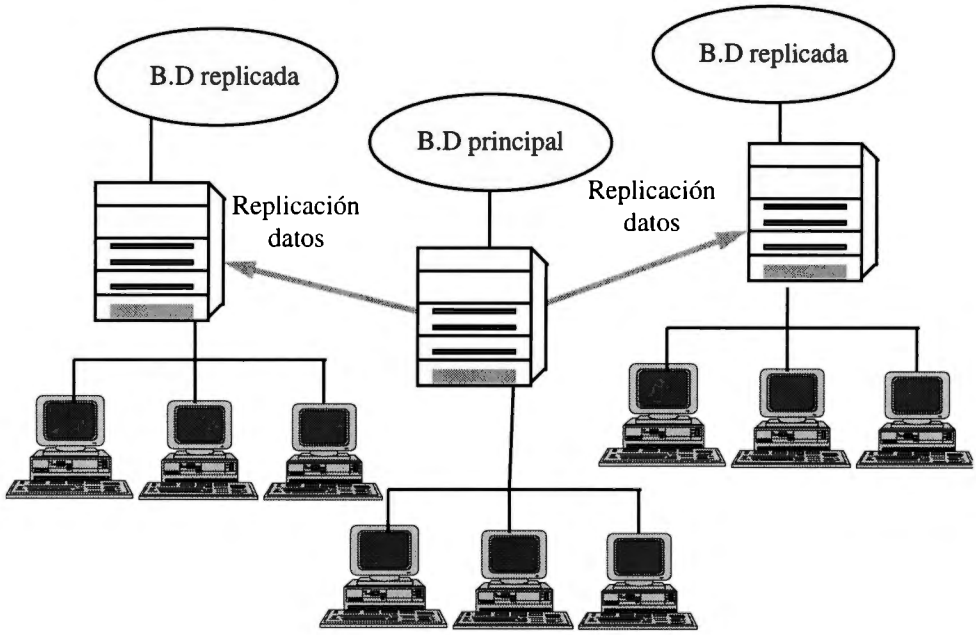


Figura 7.2.2.1 En una B.D replicada, una copia de datos es obtenida de sitios remotos

Si una aplicación requiere que todos los datos estén sincronizados, entonces usa el mismo protocolo de culminación a dos fases (two-phase commit) descrito anteriormente, para obtener todas las copias replicadas durante la fase de bloqueo. La misma situación se presenta cuando se usa el protocolo de culminación a dos fases (two-phase commit) con replicas como con bases de datos distribuidas, incluyendo esperas de acceso a los datos, el ambiente todo o nada para las actualizaciones, y el bloqueo de todos los datos participantes si el nodo que está coordinando la culminación a dos fases (two-phase commit) falla.

7.2.3 Aplicación del patrón "Usando replicación para distribución: patrones para una actualización eficiente" en Bases de Datos Distribuidas

Al igual que para los Sistemas de Manufactura Flexible es de interés evaluar la aplicación de los patrones de diseño en el caso de bases de datos distribuidas. El patrón arriba mencionado será analizado y evaluado al respecto.

En bases de datos distribuidas la replicación es un aspecto importante para lograr un buen rendimiento, una alta disponibilidad y una buena tolerancia a fallas. Uno de los aspectos

directamente relacionados con la replicación es la actualización de los datos replicados. El patrón **"Usando la replicación para distribución: patrones para la actualización eficiente" [OOPSLA'95]** trata sobre la necesidad que tienen las aplicaciones de replicar un objeto a través de diversos procesos y por consiguiente soluciona el problema de cómo enviar las actualizaciones desde el objeto maestro a las réplicas.

7.2.3.1 Aplicación del patrón: particionando campos

Contexto

En un sistema distribuido se escoge una de dos opciones para administrar cambios a objetos distribuidos. La primera es tener una copia maestra de cada objeto, y que los otros procesos cuenten solamente con interfaces sustitutas. La figura 7.2.3.1.1 muestra dicha opción. El sitio 1 contiene la copia maestra del objeto (C.M) y los otros sitios cuentan con una interfas para el objeto maestro. Un proceso P en dichos sitios puede hacer un requerimiento a una interfas que se encarga de reenviar dicho requerimiento en forma de consulta al sitio donde esta la copia maestra. Las actualizaciones sólo se hacen en la copia maestra.

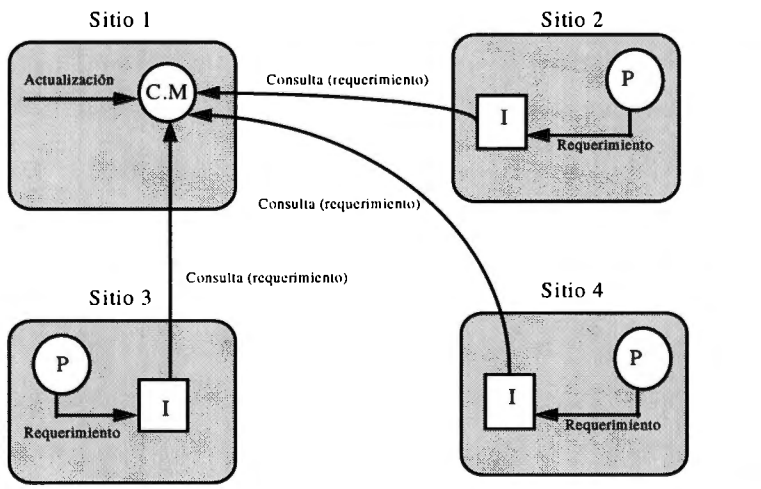


Figura 7.2.3.1.1. Copia maestra del objeto e interfaces sustitutas

La segunda opción consiste en que otros procesos contengan réplicas de la copia maestra, y cualquier actualización sobre la copia maestra cause que las réplicas deban ser actualizadas (figura 7.2.3.1.2). El segundo patrón es más apropiado en casos donde los objetos son accesados frecuentemente, pero no sufren modificaciones tan frecuentemente y nunca son actualizados por los procesos que contienen réplicas.

En la figura 7.2.3.1.2, el sitio 1 contiene la copia maestra del objeto (C.M) y los otros sitios contienen replicas de la copia maestra (R.C.M) asociadas a una interfas (de solo lectura) que procesa los requerimientos de los procesos (P). También se muestra en la gráfica que una actualización a la copia maestra es enviada a las réplicas.

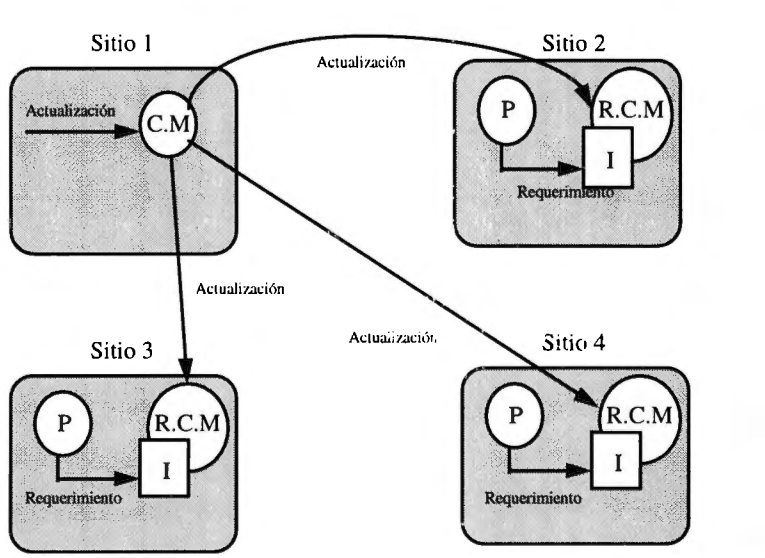


Figura 7.2.3.1.2. Copia maestra y replicas de la copia maestra

Sin embargo, hay un problema sobre cómo enviar las actualizaciones desde el objeto maestro a las replicas. Las dos opciones presentan sus propios problemas:

- Se podría usar un mecanismo de encadenamiento para enviar el objeto entero cada vez. En este mecanismo de encadenamiento se tiene una cadena de items de datos, producidos por quien los envía y consumidos por quien los recibe. Los ítems de datos enviados son incluidos

en una cola de almacenamiento hasta que el receptor este listo para recibirlos. El receptor deberá esperar cuando no haya ítems de datos disponibles y quien envía deberá esperar si la cola de almacenamiento esta llena. Muchas librerías y "frameworks" soportan ésto a través de alguna forma de empaquetamiento o protocolo de encadenamiento. Sin embargo hay una sobrecarga en el ancho de banda de la red, puesto que típicamente la mayoría de los atributos del objeto no serán modificados por una única actualización.

- Se podrían usar llamadas remotas a funciones simples actualizadoras sobre cada réplica para cambiar cada atributo. Sin embargo ésto es difícil de optimizar usando técnicas de difusión (multicast) en redes; esto también requiere definir para la réplica un conjunto extra de funciones actualizadoras además de aquellas para el maestro. Esto significa que las réplicas no pueden ser de la misma clase que la maestra, creando un problema de mantenimiento así como un diseño no muy atractivo.

Debido a lo anterior es necesario ver un ambiente alternativo.

La solución

La solución es dividir una actualización en actualizaciones parciales y llevar a cabo cada una de ellas. Para hacer esto, se particionan los datos en cada objeto dentro de ítems separados, llamados "campos". Cada campo está etiquetado con un nombre único dentro del contexto de la clase. Cada campo es atómico, esto es, los datos asociados con este tienen una única representación externa (por ejemplo, texto ASCII, XDR, o dato binario). Así cuando la versión maestra de un objeto cambia, las actualizaciones son distribuidas a las réplicas como uno o más pares campo-valor.

De esta forma cada objeto distribuido necesita una interfase en términos de estos campos. La representación interna de los datos no es cambiada; internamente se escoge la representación que es más conveniente o compatible. Sin embargo hay funciones adicionales para cada objeto distribuido que leen y escriben sus datos en términos de la representación campo-valor.

¿Quién genera estos pares campo-valor?. Este debería ser el rol de la versión maestra del objeto. Cualquier actualización a la versión maestra deberá generar una colección de pares campo-valor a

distribuir a los objetos replicados. Una forma apropiada para generar estos pares es usar los identificadores del campo como un parámetro a un mecanismo de dependencia (como por ejemplo, el patrón OBSERVADOR, que define una dependencia uno-a-muchos entre objetos de tal manera que cuando un objeto cambia su estado, todos los objetos dependientes de este son notificados y actualizados automáticamente). El mecanismo de distribución recibe notificación de cada campo cambiado, interroga al objeto por la representación del nuevo valor de cada campo, y entonces despacha la combinación campo-valor a las replicas.

Así la responsabilidad de cada objeto replicado es:

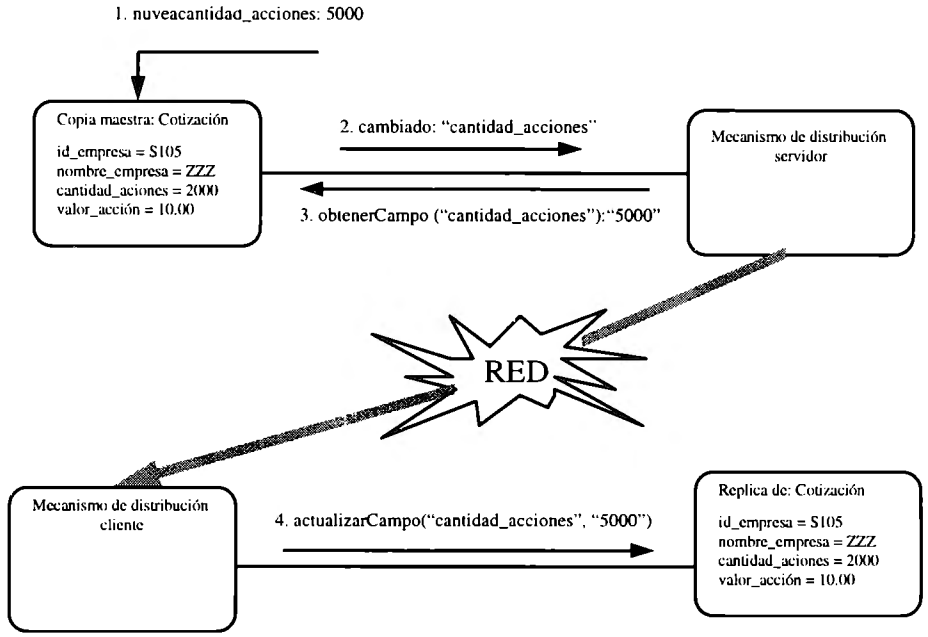
- suministrar notificación de cada campo cambiado
- suministrar un "consultor" (usado en la instancia maestra) para que dado el nombre de un campo suministre su contenido. El "consultor" es una función que permite leer pero no modificar el objeto.
- suministrar un "actualizador" (usado en las instancias replicas) para actualizar los datos dado un par campo-valor. El "actualizador" es una función que permite modificar el objeto.

Aplicación en Bases de Datos Distribuidas

Una base de datos distribuida puede perfectamente cumplir una función como la propuesta en este patrón, es decir en dicha base de datos existirán objectos (tablas) cuyos datos requieren ser consultados en tiempo real y en forma periódica por aplicaciones clientes. Además los datos (campos) de dichos objetos (tablas) solo pueden ser modificados por el nodo que tiene la copia maestra de los objetos ya que pueden contener información cuyo carácter es importante, por ejemplo para la toma de una decisión, y por lo tanto no debe existir la posibilidad de que dicha información sea modificada por cualquiera de los nodos que disponen de las replicas.

La figura 7.2.3.1.3 presenta un ejemplo del mecanismo de distribución de los datos en una base de datos con objetos replicados según lo enunciado por el patrón. En dicho ejemplo se simula una aplicación de la bolsa de valores en la que un objeto (tabla) llamado Cotizaciones, representa los movimientos en la bolsa de las acciones de una empresa, los datos manejados por dicha tabla son id_empresa, nombre_empresa, cantidad_acciones (número de acciones que están a la venta) y

valor_acción. La copia maestra de dicha tabla se encuentra en la Bolsa de Valores y replicas de dicha tabla en nodos que son usados por los agentes financieros distribuidos geográficamente para realizar la compra y venta de dichas acciones. En este ejemplo se pueden enunciar dos transacciones, una podría ser el aumento del número de acciones a la venta o una variación en el precio de las acciones y otra podría ser cambios tanto en el número de acciones como en el precio al mismo tiempo. En el primer caso un solo par campo-valor será generado para actualizar el número de acciones puestas a la venta; mientras en el segundo caso se generarán dos pares campo-valor, uno para actualizar la cantidad de acciones y otro para actualizar el valor de la acción.



7.2.3.1.3 Mecanismo para la distribución de los datos

Cuando se presenta un cambio en Cantidad_Acciones, el **sujeto** que en este caso es la copia maestra informa a su(s) **observador(es)** que en este caso es el mecanismo de distribución del campo, enviando el mensaje **cambiado: "Cantidad_acciones"**, mensaje en el cual el nombre del campo ya ha sido convertido a una representación externa (por ejemplo ASCII) para que pueda viajar por la red. Una vez que el **observador** ha sido informado del cambio, envía el mensaje **obtenerCampo("Cantidad_Accciones")** para el cual hay un valor de retorno que representa el nuevo valor correspondiente a la cantidad de acciones que igualmente ha sido convertido a una

cadena de representación externa mostrada como "5000", o sea, **obtenerCampo("Cantidad_Acciones"): "5000"**. Entonces el mecanismo de distribución envía el mensaje **actualizarCampo("Cantidad_Acciones", "5000")** a los objetos replicas para que actualicen el valor del campo cambiado.

7.2.3.2 Aplicación del patrón: Criterio de interés

La replicación funciona muy bien cuando las aplicaciones clientes pueden decir exactamente cuales objetos (por ejemplo, instrumentos financieros, transacciones comerciales) desean acceder. Sin embargo lo más común es que las aplicaciones no conocen estos objetos con anticipación. En su lugar ellos especifican un criterio para seleccionar los objetos: todos los instrumentos sobre Intercambio Mercantil en Chicago, o todas las transacciones financieras que usan Swedish Krona.

Sin embargo en muchas aplicaciones, los objetos seleccionados en esta forma no forman un conjunto estático. En su lugar, nuevos objetos son continuamente creados dentro del sistema. Así, este criterio define no solamente una colección de objetos existentes, sino también define cuales objetos nuevos recibidos por el sistema, deberán ser agregados a esta colección. ¿Cómo se implementa esto?

En esta discusión se distingue el proceso **'servidor'** , el cual contiene la versión maestra de los objetos, y procesos **'cliente'**, que contienen las replicas. Además, un proceso puede ser tanto cliente como servidor, por conveniencia se ignora esta posibilidad, y en la siguiente discusión se define un cliente que tienen un único criterio para selección.

La solución

La forma más simple para implementar esto es como sigue. La aplicación servidora mantiene una lista de todos los objetos conocidos por ella; cada nuevo objeto o actualización a un objeto existente es distribuida a todos los procesos clientes existentes. Los clientes toman el objeto recibido o sus campos a actualizar (ver el patrón anterior). Si éste es un objeto ellos determinan

si este satisface su criterio, si es una actualización, aplican éste solamente si el correspondiente objeto está ya en su colección.

Sin embargo ésto hace pesado el tráfico y procesamiento en redes: cada cliente deberá recibir todas las actualizaciones y todos los nuevos objetos en el sistema.

En su lugar, la aplicación servidora puede mantener una copia de cada criterio del cliente. Para hacer esto, se necesita una forma para convertir un criterio de selección a una forma transportable por la red.

Así, las responsabilidades del cliente son:

- Notificar al servidor de su criterio para selección de objetos.
- Recibir los objetos y actualizaciones que satisfacen el criterio y procesarlos adecuadamente.

Las responsabilidades del servidor son:

- Recibir y almacenar el criterio para cada colección cliente.
- Aplicar este criterio a cada objeto creado o actualizado y pasar la versión inicial y actualizaciones al cliente solamente si ellas igualan el criterio.

Aplicación en Bases de Datos Distribuidas

Retomando el ejemplo utilizado en la aplicación del patrón Particionando Campos, en la Bolsa de Valores estará el proceso servidor, es decir el que contiene la versión maestra de los objetos (por ejemplo Cotizaciones, IndicesFinancieros, etc). Los procesos clientes serán los que existen en los nodos de los agentes financieros, quienes tienen diferentes criterios de selección según el ramo de las finanzas que sea de su interés (por ejemplo computación, petróleo, etc). De acuerdo a lo anterior a un proceso cliente le puede interesar, por ejemplo Cotizaciones de las empresas en el área de la computación y/o IndicesFinancieros de los metales preciosos, ó a un proceso cliente interesarle solamente Cotizaciones y a otro proceso cliente los IndicesFinancieros.

En la Bolsa de Valores existe entonces una lista de los objetos (instrumentos financieros) usados en el mercado financiero (Cotizaciones, IndicesFinancieros, etc), nuevos instrumentos financieros serán agregados a dicha lista y las actualizaciones realizadas a los objetos maestros existentes. Por otro lado, los nuevos objetos o las actualizaciones a los ya existentes serán distribuidas a todos los procesos clientes (es decir, a los agentes financieros) que contienen las réplicas. Antes de la distribución de objetos o actualizaciones el proceso servidor aplicará los criterios de todos los clientes a los objetos o actualizaciones y distribuirá, dichos objetos o actualizaciones (pares campo-valor) a los procesos clientes correspondientes.

La mecánica será la misma que se mostró en el patrón Particionando Campos, es decir los criterios deberán ser convertidos a una representación externa que pueda viajar por la red para que los criterios puedan ser enviados por los procesos clientes al proceso servidor. La otra parte relacionada con la replicación de nuevos objetos o actualizaciones se realiza en la forma como lo específica el patrón Particionando Campos.

7.3 Conclusiones

Como se muestra en los dos ejemplos, Sistemas de Manufactura Flexible (FMS) y Bases de Datos Distribuidas, la aplicación de patrones de diseño es perfectamente viable.

En el primer ejemplo, se puede ver cómo una serie de patrones pueden ser encadenados formando una metodología ya que cada uno de los patrones componentes de dicho lenguaje corresponden a pasos que deben ser realizados en una secuencia determinada. Este aspecto es importante ya que, a la vez que creamos la estructura necesaria de la solución, se crea el proceso de desarrollo del sistema. Por otra parte puede verse cómo en dichos patrones está capturado el conocimiento que los expertos aplican en la solución de problemas que se presentan en forma recurrente. Es decir podríamos aplicar el mismo lenguaje patrón a otro sistema de manufactura flexible en el que los componentes pudieran tener otro tipo de características, por ejemplo un sistema donde existan robots como mecanismos de transporte. Por otra parte puede verse que el dominio de aplicación de este patrón está más orientado a la arquitectura cliente-servidor. El paradigma utilizado en el patrón, es el de programación orientada a objetos y toda su estructura está basada en un concepto

jerárquico que se logra por medio de la herencia y niveles de jerarquía. El aspecto de tolerancia a fallas no es muy tratado en este lenguaje patrón, sin embargo existen algunos aspectos del diseño logrado como los mecanismos de difusión/escucha que permiten a los módulos de control estar haciendo un monitoreo constante de cada uno de los dispositivos a su cargo. En este caso puede verse que el lenguaje patrón podría crecer para ir resolviendo más problemas relacionados con los sistemas distribuidos, es decir se puede agregar y encadenar un patrón que plantee una solución más específica a un problema como el de tolerancia a fallas.

En el segundo ejemplo de aplicación, se tomó un patrón más específico para un problema más específico y no tan generalizado como en el primer ejemplo. Ésto demuestra que la aplicación de los patrones puede tomarse como un catálogo del cual puede tomar no solo uno sino varios patrones y ver cual de ellos se adapte de una forma más adecuada o cubre mejor los requerimientos del problema a solucionar. Un patrón particular puede igualmente servir como punto de partida para la solución que se busca, aunque el mismo no sea la solución más adecuada. Es decir, a partir del patrón se pueden hacer más abstracciones y adicionarle al mismo más elementos que estén de acuerdo con el dominio de aplicación que se busca.

La aplicación de un patrón permite documentar el sistema (ya que se puede evaluar si la solución del patrón satisface completamente los requerimientos del sistema que se está diseñando) incluyendo en un documento el patrón y la información que el especialista considere necesaria para aclarar dicha aplicación. Ahora bien los patrones de diseño pueden ser en determinado momento difíciles de entender dependiendo del conocimiento que el experto tenga de los paradigmas bajo los cuales está desarrollado el patrón. Por ejemplo no siempre todas las personas entenderán fácilmente el paradigma orientado a objetos en que se fundamente dicho patrón. Sin embargo la idea de la solución planteada sí puede ser un valioso punto de partida.

La tabla 7.3.1 es un resumen de los patrones aplicados para el caso del Sistema de Manufactura Flexible y la tabla 7.3.2 es un resumen de los patrones aplicados para el caso de el Sistema de Bases de Datos Distribuidas. Dichas tablas relacionan cada patrón indicando cual es la intención que tiene el mismo, y describen un ejemplo real al aplicarlo en el respectivo sistema.

PATRÓN: Lenguaje patrón G++
ENFOQUE: Diseño orientado a objetos de sistemas de información concurrentes y distribuidos
APLICACIÓN EN: Sistemas de Manufactura Flexible (FMS)

PATRÓN	INTENCIÓN	EJEMPLO EN LA APLICACIÓN
Una jerarquía de capas de control	Cualquier sistema complejo organiza su funcionalidad como una arquitectura en capas jerárquicas que corresponden a módulos de control	Las jerarquías definidas fueron: Nivel departamento, nivel celda y nivel estación de trabajo
Visibilidad y comunicación entre módulos de control	Cada módulo en la jerarquía requiere servicios de otros módulos que son requeridos usando el mecanismo de comunicación C/P, o señales de eventos usando el mecanismo B/L	La comunicación de la celda de maquinado con los transportes y las máquinas
Objetos y Concurrencia	Los módulos ejecutan servicios concurrentemente y la concurrencia sume escalas de granulidad. Lo anterior se modela con objetos activos, bloqueantes y secuenciales	Un ejemplo de objeto activo es la celda de maquinado, un ejemplo de objeto bloqueante es el almacén.
Acciones disparadas por eventos	La comunicación a través de los eventos puede ser intra-servicio o inter-servicios. Use el mecanismo B/L para ésta comunicación.	Por ejemplo el servicio de maquinar una pieza envía eventos de falla al servicio de producción de un lote

Tabla 7.3.1. Patrones aplicados en el Sistema de Manufactura Flexible

PATRÓN: Lenguaje patrón G++
ENFOQUE: Diseño orientado a objetos de sistemas de información concurrentes y distribuidos
APLICACIÓN EN: Sistemas de Manufactura Flexible (FMS)

PATRÓN	INTENCIÓN	EJEMPLO EN LA APLICACIÓN
Servicios "esperando por"	Los servicios ofrecidos por un módulo de control esperan a que ocurra una condición o esperan por datos transferidos desde ó hacia un recurso compartido, antes de realizar alguna acción en concurrencia con otros servicios. Use para esto los objetos bloqueantes.	Por ejemplo el almacén de entrada es un objeto bloqueante que informa constantemente de su estado.
Cliente/Servidor/Servicio: implementando módulos de control	Los módulos de control son objetos activos que ofrecen múltiples servicios en forma concurrente.	Por ejemplo el departamento es un objeto activo que solicita servicios a otro objeto activo como lo es la celda de maquinado.
Implementación de módulos de control de "múltiples tipos de servicios"	Los módulos de control administran diferentes grupos de recursos compartidos y posiblemente a menudo ofrecen diferentes tipos de servicio. Definir los módulos como servidores encapsulando en ellos los recursos compartidos que serán objetos bloqueantes o activos	Por ejemplo la implementación del servidor de la celda de maquinado encapsula las máquinas, y almacenes.
La interfase para módulos de control	Los módulos de control ofrecen diferentes tipos de operaciones a sus clientes. Use un objeto interfase para accesar cada servidor.	Por ejemplo el departamento accesa al servidor de la celda de maquinado a través de la interfase llamada celda de maquinado.
Prototipo y realidad	Cualquier aplicación compleja requiere prototipos y simulación de los diferentes elementos a ser integrados antes de derivar una aplicación. Debe obtenerse dos versiones del objeto, "el objeto simulado" y el "objeto real".	La interfase creada para la celda de maquinado (o sea el prototipo) pasa a ser real con la implementación del objeto celda de maquinado.

Continuación tabla 7.3.1. Patrones aplicados en el Sistema de Manufactura Flexible

PATRÓN: Lenguaje patrón G++
ENFOQUE: Diseño orientado a objetos de sistemas de información concurrentes y distribuidos
APLICACIÓN EN: Sistemas de Manufactura Flexible (FMS)

PATRÓN	INTENSIÓN	EJEMPLO EN LA APLICACIÓN
Distribución de los módulos de control	Los módulos de control usualmente residen sobre computadoras remotas. Estos módulos definen la arquitectura física. Para estos módulos remotos cree objetos llamados contextos remotos.	Por ejemplo a nivel de los controladores de las máquinas existe un objeto "contexto remoto" que encapsula una máquina determinada.

Continuación tabla 7.3.1. Patrones aplicados en el Sistema de Manufactura Flexible

PATRÓN: Usando replicación para distribución: patrones para una actualización eficiente.
ENFOQUE: : Patrones para una actualización eficiente
APLICACIÓN EN: Sistemas de Bases de Datos Distribuidas

PATRÓN	INTENCIÓN	EJEMPLO EN LA APLICACIÓN
Particionando campos	Optimizar el proceso de actualización de las replicas de un objeto en el sistema. Para esto se divide una actualización en actualizaciones parciales en forma de Campo-Valor.	Enviar primero la actualización de una cantidad de acciones y luego enviar la actualización de su precio..
Criterio de interés	Recibir nuevos objetos o actualizaciones a objetos y poder determinar si ese objeto o esa actualización le compete al respectivo nodo que recibe los objetos o las actualizaciones. Mantener en el nodo criterios de evaluación que filtren los objetos o las actualizaciones.	Puede existir una lista de los objetos (instrumentos financieros) usados en el mercado financiero. Nuevos instrumentos financieros serán agregados a dicha lista y las actualizaciones realizadas a los objetos maestros existentes.

Continuación tabla 7.3.1. Patrones aplicados en el Sistema de Bases de Datos Distribuidas

Finalmente el problema que puede existir aún con los patrones de diseño es que la problemática a solucionar es muy amplia; pero a medida que dichos patrones puedan ser mejor enlazados y clasificados los usuarios de los mismos podrán obtener un beneficio más claro.

8. METODOLOGÍA PARA ANÁLISIS Y DISEÑO DE SISTEMAS DISTRIBUIDOS

Este capítulo describe como los diferentes conceptos de ingeniería de software y de sistemas distribuidos descritos en los capítulos anteriores pueden ser combinados para construir un ambiente de ingeniería de software para sistemas distribuidos.

Lo que se presenta en este capítulo es un **esquema metodológico** cuyo objetivo es establecer lineamientos generales a tener en cuenta para el análisis y diseño de sistemas distribuidos y, en algunos casos, profundizar en ciertos lineamientos considerados importantes. Para efectos de simplificación el término **metodología** será utilizado en el resto de este capítulo para hacer referencia al concepto ya explicado de esquema metodológico. La metodología que se propone a continuación se basa en el concepto de patrones de diseño cuyo marco conceptual y aplicación se abordaron en los capítulos 6 y 7 respectivamente. De dichos patrones se usa una versión simplificada en la que se utilizan dos elementos: el contexto o problemática y la solución planteada para la misma. Como ya se explicó, lo que se busca con la metodología propuesta es dar un esquema y enunciar los patrones que deben formar dicho esquema y que se consideran fundamentales para el desarrollo de una aplicación distribuida de acuerdo a los aspectos analizados en este trabajo de tesis. Debido a que la esencia de todos los patrones de diseño es la *dupla* formada por el **problema y su solución**, la metodología propuesta se basará igualmente en un paradigma **problema-solución.** Se busca que lo planteado pueda ser utilizado en forma recurrente para los diferentes tipos de sistemas distribuidos. También es importante que la metodología dé la facilidad de incorporar nuevos patrones de diseño para así lograr un ambiente de ingeniería de software más abierto que permita solucionar los diferentes problemas presentados a la hora de diseñar un sistema distribuido.

Otro aspecto fundamental en el que se basa la metodología, es el principio de "divide y conquista" de tal manera que las parejas **problema-solución** estarán enlazadas para permitir ir de un análisis general hacia uno particular o en otras palabras al estilo top-down y no bottom-up. Esto permitirá que los planteamientos de una pareja problema-solución puedan ser refinados en sucesivas parejas problema-solución que den como resultado final un completo análisis y diseño del sistema.

Para lograr lo anterior se llevará a cabo una categorización de los patrones y se creará un sistema de patrones, el cual inherentemente nos proporciona la metodología necesaria ya que cada patrón componente del sistema corresponderá a pasos ligados que deben ser realizados para el análisis y diseño de un sistema distribuido.

La figura 8.1 muestra en forma de árbol cada uno de los patrones que se definieron como parte de la metodología planteada.

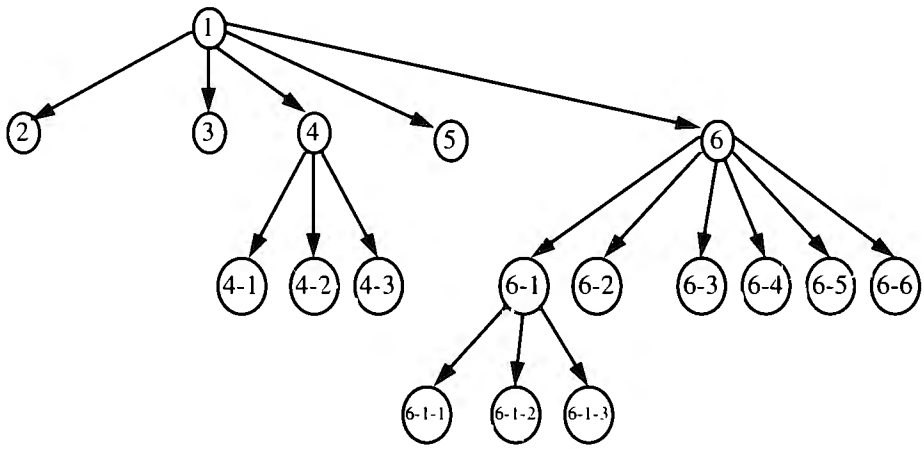


Figura 8.1. Patrones aplicados para obtener el esquema metodológico

Patrón 1: Creación de una metodología para el análisis y diseño de un sistema de software

Patrón 2: Determinación del ciclo de vida para el desarrollo de un sistema distribuido

Patrón 3: Determinación de la arquitectura de un sistema distribuido

Patrón 4: Aspectos básico para el esquema metodológico

Patrón 4-1: Separación de asuntos en sistemas distribuidos

Patrón 4-2: Características de los "lenguajes"

Patrón 4-3: Objetos distribuidos

Patrón 5: Determinar las etapas del enfoque metodológico

Patrón 6: Las etapas del enfoque metodológico

Patrón 6-1: Etapa de análisis y modelado no distribuido

Patrón 6-1-1: Modelo de datos

Patrón 6-1-2: Modelo de procesos

Patrón 6-1-3: Modelo de objetos

Patrón 6-2: Modelo de la distribución lógica

Patrón 6-3: Diseño del sistema

Patrón 6-4: Modelo de la distribución física

Patrón 6-5: Necesidades de implementación

Patrón 6-6: Integración

La parte inicial de este capítulo (sección 8.1) presenta el patrón tomado en este trabajo como la abstracción más general o punto de partida para la solución de la problemática del análisis y diseño de sistemas distribuidos.

Como se enunció en el capítulo 6, los primeros patrones de software fueron escritos por desarrolladores orientados a objetos y se enfocaron sobre diseño y programación orientada a objetos o sobre modelado orientado a objetos. Posteriormente surgió una nueva tendencia con patrones que se enfocan sobre sistemas concurrentes, paralelos y distribuidos; ejemplos de esta tendencia son el patrón **"Lenguaje Patrón G++"** y el patrón **"Usando replicación para distribución: patrones para una actualización eficiente en Bases de datos distribuidas"** analizados en el capítulo 7. La tendencia a trabajar con patrones sigue creciendo y su aplicación ha demostrado ser satisfactoria en diferentes ámbitos. Por lo anterior, el esquema metodológico presentado a continuación, se fundamenta en el concepto de patrones de diseño como una alternativa valida para optimizar el proceso de analizar y diseñar sistemas de software para sistemas distribuidos.

8.1 Patrón 1: Creación de una metodología para el análisis y diseño de un sistema de software

Contexto

La obtención de una metodología para el análisis o diseño de sistemas de software es un proceso complejo ya que hay múltiples aspectos, tales como estilo, ambiente, especificación, descomposición, recursos y muchos otros que deben ser tenidos en cuenta. Antes de formular o definir una metodología particular, es conveniente determinar los bloques generales necesarios para la creación de la misma. Esto permitirá establecer lineamientos a tener en cuenta para el desarrollo de la metodología y al mismo tiempo clarificar aspectos que hagan de la normativa un esquema soportado en los fundamentos básicos de lo que es una metodología.

Solución

En el presente trabajo se han determinado cinco bloques básicos a desarrollar en la creación de una metodología. La figura 8.1.1 muestra dichos bloques y una descripción del objetivo buscado en cada uno de ellos es explicado a continuación.

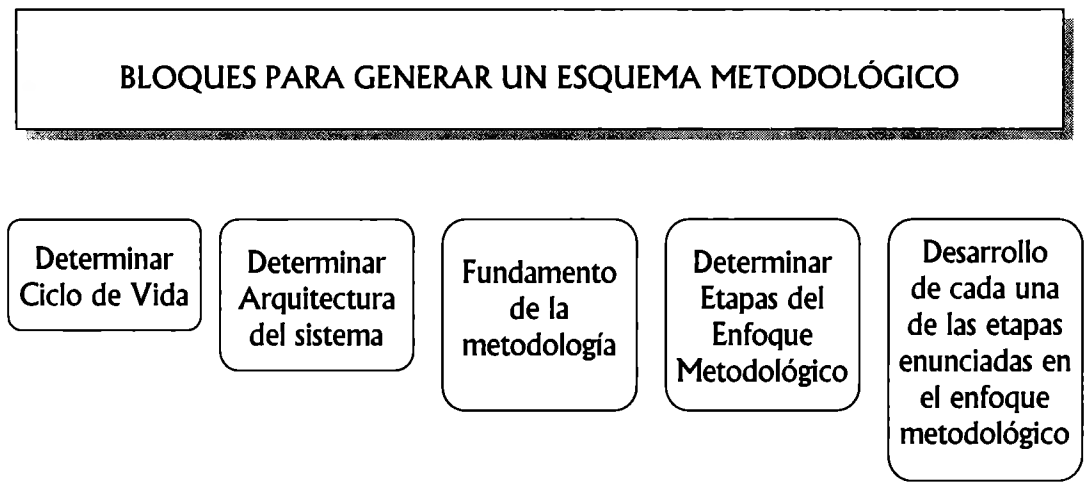


Figura 8.1.1. Bloques del esquema metodológico

- **Bloque: Determinar un Ciclo de vida**. El primer aspecto a solucionar es la determinación del tipo de Ciclo de Vida de desarrollo del software que se utilizará. Como se puede ver éste es una variación del ambiente problema-solución y nos da un esquema sistémico y ordenado para la solución de problemas de sistemas.
- **Bloque: Determinar la arquitectura del sistema**. En este contexto, la arquitectura del sistema planteada se refiere a los modelos más generales que se deben obtener en el desarrollo de una aplicación. Para obtener los modelos se realizan varias actividades que dan como resultado diferentes productos (por ejemplo un diccionario de datos, un diagrama de objetos, etc.) que conforman un modelo específico (por ejemplo un modelo de datos).

Bloque: Fundamento de la metodología a plantear. Antes de plantear cada una de las etapas de la metodología se debe definir en forma general el concepto en que se fundamenta dicha metodología en relación con el sistema específico al cual se quiere aplicar. Esto permite enfocar mejor los esfuerzos para la determinación de las etapas y actividades que las componen.

- **Bloque: Determinar etapas del enfoque metodológico**. Lo que se busca en este punto es establecer y describir cada una de las etapas que se aplicarán en el desarrollo del sistema y que permitirán obtener cada uno de los modelos enunciados en el bloque anterior.
- **Bloque: Desarrollo de cada una de las etapas enunciadas en el enfoque metodológico**. Una vez determinadas las etapas para el enfoque metodológico cada una de ellas debe ser desarrollada indicando los modelos, técnicas, herramientas, etc; que deben usarse para obtener los productos allí requeridos.

8.2 Patrón 2: Determinación del ciclo de vida para el desarrollo de un sistema distribuido

Contexto

Como se vió en el análisis de los patrones en el capítulo 7, un ambiente del tipo **problema-solución** para un sistema consiste en generar e implementar soluciones correctas que toman la forma de un nuevo sistema o de un sistema mejorado. Como resultado de esta investigación se puede ver que el desarrollo de un sistema de software se puede hacer utilizando una variación al

ambiente **problema-solución** que podemos llamar "Ciclo de vida de desarrollo del sistema", que es un ambiente sistemático y ordenado para solucionar problemas.

Solución

Para el caso de un sistema distribuido y debido a la complejidad que se presenta en el desarrollo del mismo es conveniente poder tener evaluaciones continuas del sistema que se está desarrollando antes de su implementación definitiva. Para lograr este objetivo se recomienda utilizar un ciclo de vida incremental en el cual se involucre el paradigma de prototipos en cualquier incremento del ciclo de vida. La figura 8.2.1 muestra el ciclo de vida propuesto, en el que se incluyen cada uno de los pasos generales y flujos que componen el ciclo de vida.

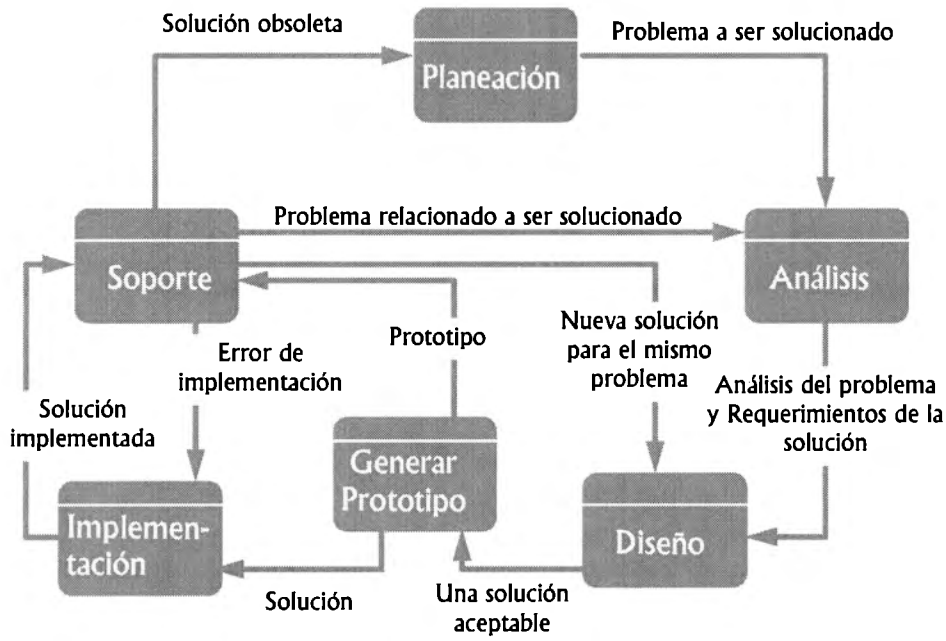


Figura 8.2.1. Ciclo de vida para el desarrollo de un sistema distribuido

Cada rectángulo representa una etapa del ciclo de vida y las flechas denotan los flujos que entran y salen de cada una de ellas. Como puede verse, el ciclo permite el planteamiento de un problema, la planeación de su solución y la ejecución de dicho plan. El análisis y diseño permite producir una solución aceptable para la cual se genera un prototipo, que pasa a una evaluación en la etapa de soporte de la cual pueden nacer refinamientos al prototipo. Dichos refinamientos

volverán a la etapa de diseño para generar una nueva versión del prototipo. Obtenida la solución definitiva se hace la etapa de implementación que igualmente es evaluada en la etapa de soporte.

Una vez aceptada la solución y su implementación pueden surgir nuevos requerimientos o la aplicación puede llegar a ser obsoleta; dependiendo de ésto puede ser necesario entrar nuevamente a planeación o por el contrario pasar a una etapa de análisis nuevamente y en cualquiera de los dos casos cubrir nuevamente el ciclo completo.

8.3 Patrón 3: Determinación de la arquitectura de un sistema distribuido

El punto de partida para el desarrollo del esquema metodológico será definir la arquitectura en la cual se basará toda la metodología propuesta.

Contexto

Es importante en el desarrollo de cualquier sistema definir una **arquitectura del sistema** . La **arquitectura del sistema** debe suministrar un esquema dentro del cual personas con diferentes perspectivas puedan organizar y ver los bloques fundamentales de construcción del mismo.

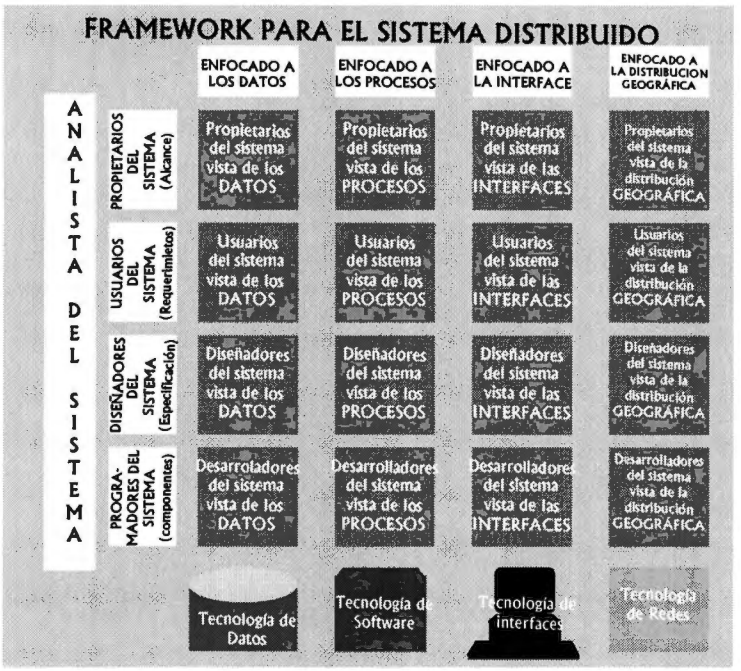
Solución

Involucrar en **la arquitectura del sistema,** personas que jueguen los siguientes roles: **propietarios del sistema, usuarios del sistema, diseñadores del sistema** y **desarrolladores del sistema**. También se deben involucrar en dicha **arquitectura del sistema** las siguientes tecnologías: **tecnología de datos, tecnología de software, tecnología de interfaces, tecnología de redes**.

Otro elemento en la **arquitectura del** sistema es el enfoque sobre el cual se trabajará. Para este caso se deben utilizar los siguientes enfoques:

- **Enfoque de DATOS**: éste se genera creando un grupo de datos (entidades y atributos) usados para crear información fundamental para el sistema distribuido.
- **Enfoque de PROCESOS**: son las actividades que permiten llevar a cabo la misión del sistema distribuido.
- **Enfoque de INTERFACES**: es la forma como el sistema interactua con las personas y otros sistemas.
- **Enfoque GEOGRÁFICO**: determina donde deben capturarse los datos y donde deben ser almacenados, donde deben distribuirse y ejecutarse los procesos, y finalmente donde deben estar y también ejecutarse las interfaces del sistema distribuido.

La figura 8.3.1 muestra la relación existente entre todos estos elementos componentes de la **arquitectura del sistema**.



8.3.1. Bloques de construcción del sistema distribuido

Como lo muestra la figura 8.3.1 cada enfoque está representado por su propia columna. La intersección de cada perspectiva (fila) y cada enfoque (columna) define un bloque básico de construcción (celda) del sistema. Dependiendo de quien sea la persona (propietario, usuario, diseñadores, desarrolladores) y sobre el enfoque que desee (datos, procesos, interfaces, o redes), se tiene una vista de la arquitectura del sistema. Por ejemplo, un diseñador de la base de datos

tiene una vista del esquema desde el punto de vista de la base de datos mientras que un desarrollador tienen una visión de los programas de la aplicación.

La anterior matriz (figura 8.3.1) es el esquema básico del esquema metodológico propuesto aquí. Sin embargo, debe aclararse que las partes relacionadas con los propietarios y desarrolladores del sistema distribuido están fuera del alcance de esta tesis pero son enunciados ya que se pretende plantear un esquema general que permita involucrar los diferentes actores en un sistema distribuido. Por lo tanto, los aspectos tratados de aquí en adelante son relevantes a la parte de los usuarios del sistema y de los diseñadores del sistema, es decir la relación directa con las etapas de análisis y diseño enunciados en el patrón "Ciclo de vida para el desarrollo de un sistema distribuido".

8.4 Patrón 4: Aspectos básico para el esquema metodológico

Como ya se enunció, el proceso de desarrollo propuesto es iterativo e incremental y además permite la verificación parcial de resultados. Sin embargo, eso no quiere decir que el trabajo de desarrollo sea inherentemente menos complejo. Los tres patrones que se presentan a continuación buscan establecer lineamientos a tener en cuenta en el planteamiento de un enfoque metodológico. Estos lineamientos están directamente relacionados con los **sistemas distribuidos** y con los **"lenguajes"** para especificación y programación concurrentes. En este contexto el término "lenguajes" se refiere a técnicas o métodos usados en un esquema metodológico.

8.4.1 Patrón 4-1: Separación de asuntos en sistemas distribuidos

Contexto

En [Couloris, 1994] se describen las características claves que debe tener un sistema distribuido. Estas son **repartición de recursos, sistema abierto, concurrencia, escalabilidad, tolerancia a fallas y transparencia**. La transparencia juega un papel muy importante para alcanzar el objetivo buscado en este trabajo. La transparencia es definida como el ocultamiento de la separación de los componentes en un sistema distribuido para el usuario y para el programador de

la aplicación, de tal forma que el sistema es percibido como un todo más que como una colección de componentes independientes. Las implicaciones de la transparencia influyen en el diseño del sistema de software.

La separación de componentes es una propiedad inherente de los sistemas distribuidos. Dicha separación trae como consecuencia la necesidad de comunicación y de administración explícita del sistema y la aplicación de técnicas de integración. La separación permite la ejecución paralela de programas en forma más real, la detección de fallas en componentes y la recuperación de fallas sin interrumpir todo el sistema, el uso del aislamiento y control de canales de comunicación como un método para buscar seguridad y políticas de protección, así como el crecimiento continuo o la reducción del sistema a través de la adición o sustracción de componentes. Existen ocho formas de transparencia las cuales suministran un resumen sobre la motivación y objetivos de un sistema distribuido. Tomando en cuenta que el término "objeto de información" se usa para denotar las entidades para las cuales la distribución de transparencia es aplicada, se procede a explicar cada una de las formas de transparencia:

- La **transparencia de acceso** permite a objetos de información local y remotos ser accedidos usando las mismas operaciones.
- La **transparencia de localización** permite a los objetos de información ser accesados sin conocimiento de su localización.
- La **transparencia de concurrencia** permite a varios procesos operar concurrentemente usando objetos compartidos de información sin interferirse entre ellos.
- La **transparencia de replicación** permite a múltiples instancias de objetos de información ser usados para incrementar confiabilidad y rendimiento sin conocimiento de las replicas por parte de usuarios o programas.
- La **transparencia a fallas** permite el ocultamiento de fallas, permitiendo a usuarios y programas completar sus tareas a pesar de la falla de los componentes de hardware o software.
- La **transparencia de migración** permite los movimientos de objetos de información dentro de un sistema sin afectar la operación de los usuarios o programas de aplicación.
- La **transparencia en el rendimiento** permite al sistema ser reconfigurado para mejorar rendimiento tanto como la carga varíe.

- La **transparencia de crecimiento** permite al sistema y aplicaciones expandirse en escala sin cambiar la estructura del sistema o los algoritmos de la aplicación.

Las dos transparencias más importantes son la transparencia de acceso y la transparencia de localización; su presencia o ausencia afectan fuertemente la utilización de recursos compartidos. Estas dos transparencias son algunas veces referenciadas como una sola unidad llamada ***transparencia de red***.

Solución

Como se puede ver en el contexto anterior hay varios **asuntos** a tratar a la hora de diseñar un sistema distribuido. Para poder cubrir todos estos asuntos se propone hacer el desarrollo de la aplicación distribuida separando dichos asuntos. Es decir en nuestro ciclo de vida se debe desarrollar la aplicación paso a paso, de tal manera que se trate cada asunto pero solamente uno de ellos sea manejado a la vez para cada etapa del ciclo de desarrollo. Por ejemplo, si hablamos del asunto "tolerancia a fallas" entonces aplicamos todas las etapas del ciclo de vida a este solo asunto como lo muestra la figura 8.3.1.

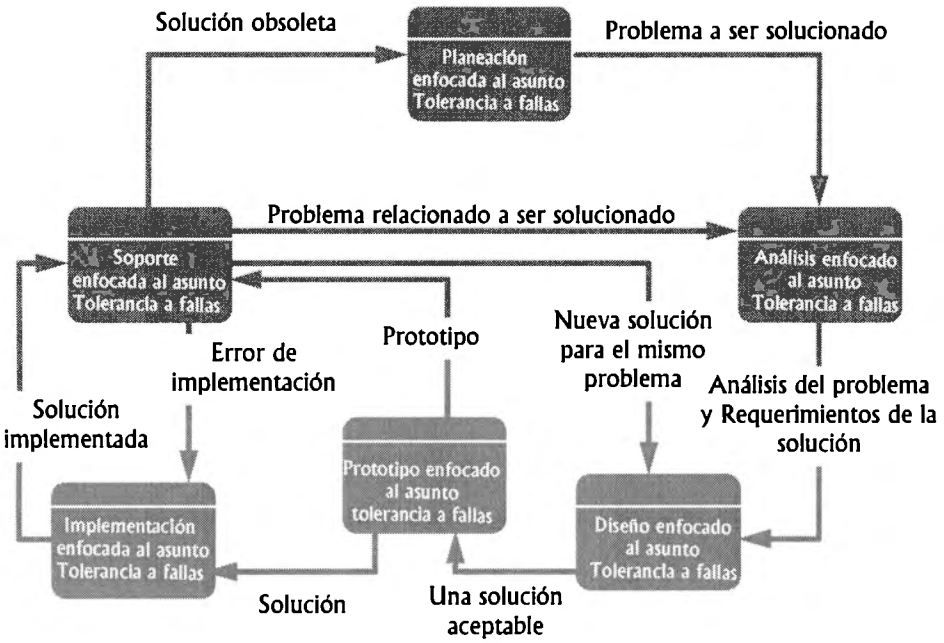


Figura 8.3.1. Ciclo de vida aplicado a un solo asunto

Para determinar los asuntos a tener en cuenta en el análisis y diseño de sistemas distribuidos se tomó como base la característica de transparencia ya que como se enunció anteriormente ésta suministra un resumen sobre la motivación y objetivos de un sistema distribuido. Se determinaron seis asuntos para cubrir cada uno de los aspectos enunciados por la característica de transparencia y otros que son importantes para el desarrollo software distribuido. Dichos asuntos son los siguientes:

- **Nombramiento**
- **Concurrencia**
- **Distribución de los datos**
- **Tolerancia a fallas**
- **Escalabilidad/Crecimiento**
- **Rendimiento/Desempeño**

8.4.2 Patrón 4-2: Características de los "lenguajes"

Contexto

Con la idea de describir los sistemas de software distribuido los conceptos básicos relacionados con los "lenguajes" (el término "lenguaje" es usado aquí para referirse a las técnicas y métodos) que se usan en el desarrollo de un sistema en general, deben ser analizados y por consiguiente dar como resultado lineamientos o características que los mismos deben cubrir en el sistema específico de aplicación. Estos lineamientos deben ser enunciados y tomados en cuenta en algunos de los "lenguajes" que se usen para modelar el sistema.

Solución

Para el caso de los sistemas distribuidos, este debe incluir "lenguajes" que le permitan cubrir las siguientes características:

1. La descomposición del sistema y la estructura de los procesos

2. Las características de comunicación
3. Las características de sincronización
4. La descripción del comportamiento de los procesos
5. La asignación de tareas

8.4.3 Patrón 4-3: Objetos distribuidos

Contexto

La distribución de los objetos podría verse desde dos puntos de vista diferentes:

- Se puede tener un sistema donde ciertamente tanto datos como métodos (código) puedan ser distribuidos en conjunto en la localización donde han sido asignados, es decir se distribuye el objeto como una sola unidad.
- Por otro lado podría darse el caso en que dentro de una clase, alguna instancia puede estar en un disco local y otras a través de la red. Dentro de una única instancia, alguna variable (atributo) pueda estar en un nodo de la red y otra en otro nodo; sobre una misma máquina alguna variable puede estar en un archivo y otra en otro. De la misma manera los métodos podrían ser similarmente distribuidos. Y nuevamente, se podría tener múltiples copias de una instancia o de ciertos atributos o métodos, sobre diferentes máquinas y diferentes sitios.

Solución

Para poder enfrentar las dos situaciones enunciadas anteriormente se debe realizar un análisis y diseño orientado tanto a datos y procesos como también a objetos. Es decir obtenga como productos modelos en los que datos y procesos estén separados y por otra parte productos en los que datos y procesos estén encapsulados, es decir un paradigma orientado a objetos. Esto permitirá sincronizar mejor cada uno de los modelos obtenidos y lograr así un diseño más consistente y robusto.

8.5 Patrón 5: Determinar las etapas del enfoque metodológico

Contexto

Una vez determinada la arquitectura que se tendrá para el desarrollo del sistema distribuido, es necesario establecer cuales son las etapas a cubrir para la obtención de los productos relacionados con cada una de los usuarios y enfoques mostrados en el patrón 3.

Solución

Las etapas que formarán parte del proceso de desarrollo de aplicaciones distribuidas son: **análisis y modelo no distribuido, modelo de la distribución lógica, diseño del sistema, modelo de la distribución física, necesidades de la implementación y finalmente integración**.

- **Análisis y modelado no distribuido**. Durante esta fase se definen los requerimientos del usuario y se modela el sistema ignorando asuntos de distribución.
- **Modelo de la distribución lógica**. Un sistema de computación distribuida está definido como un sistema de múltiples procesadores autónomos que cooperan solamente enviando mensajes sobre un canal de comunicación. Nótese que esta definición no distingue entre componentes físicamente separados y módulos autónomos que lógicamente se están comunicando vía mensajes. La etapa de distribución lógica diseña la aplicación como un conjunto de mundos distribuidos lógicamente que pueden comunicarse solamente a través de mensajes.
- **Diseño del sistema**. La fase de análisis se enfoca principalmente sobre la lógica, aspectos independientes de la implementación de un sistema (los requerimientos), el diseño de sistemas trata con los aspectos físicos o dependientes de la implementación del sistema (especificaciones técnicas del sistema).
- **Modelo de la distribución física**. La distribución lógica debe dar como resultado final una distribución física. Las abstracciones deberán ser implementadas usando mecanismos concretos de la plataforma.

- Necesidades de implementación**.** En esta etapa se tendrán en cuenta cada uno de los asuntos enunciados en el patrón 4. Dichos asuntos deben ser incluidos una vez que se ha cumplido con

las etapas anteriores para de esta manera determinar la influencia que los mismos pueden tener en los modelos obtenidos hasta el momento.

- **Integración.** En esta etapa se busca integrar cada uno de los productos obtenidos al tratar cada uno de los asuntos sugeridos para el análisis y diseño de sistemas distribuidos. Las soluciones suministradas por cada uno de los asuntos podría de alguna manera afectar las soluciones dadas por los otros. El resultado de esta integración sería una solución consistente para la aplicación distribuida.

8.6 Patrón 6: Las etapas del enfoque metodológico

Contexto

Como ya se estableció anteriormente, el ciclo de vida está compuesto de una serie de etapas las cuales deben dar como resultado productos finales según los requerimientos del sistema. Por esta razón, es necesario definir y describir cada una de las etapas del ciclo de vida. Indicando las actividades que conlleva y los productos que deben ser obtenidos

8.6.1 Patrón 6-1: Etapa de análisis y modelado no distribuido

Contexto

En esta etapa lo que se busca es establecer los requerimientos del sistema y la descomposición del mismo en las diferentes piezas que lo componen para estudiar cómo estos componentes interactuan y trabajan. Subsecuentemente lo que se busca es hacer una síntesis del sistema, es decir reemsamblar las piezas componentes del sistema para formar todo el sistema con la idea de obtener un sistema mejorado.

Por otra parte, con el fin de tener una visión del sistema a desarrollar es conveniente antes de involucrar los aspectos relacionados con los sistemas distribuidos, obtener un modelo del sistema

actual sin tomar en cuenta las características de distribución. Esto permitirá identificar en forma inicial todos los diferentes agentes externos, datos y procesos que componen o interactuan con el sistema.

Solución

1. Utilizar las siguientes estrategias para el análisis y solución del problema:

    - Un análisis moderno estructurado
    - Un análisis con datos y procesos separados
    - Un análisis orientado a objetos

2. Obtener los siguientes modelos:

    - Un modelo de datos utilizando diagramas Entidad-Relación (E/R, ver apéndice B)
    - Un modelo de procesos utilizando diagramas de flujo de datos (DFD, ver apéndice C)
    - Un modelo de objetos utilizando casos de uso (ver apéndice D)

8.6.1.1 Patrón 6-1-1: Modelo de datos

Contexto

Es necesario en cualquier sistema modelar los datos como una técnica más para definir los requerimientos del sistema y sentar las bases para la creación de una base de datos. Para modelarlos se recomienda utilizar una técnica que permita organizar y documentar los datos del sistema

Solución

Hay diversas notaciones para el modelado de los datos. El modelo que se debe usar en esta metodología es el **diagrama entidad relación (E/R)** (ver apéndice B) ya que éste muestra los

datos en términos de las entidades y relaciones establecidas por los datos. Hay diferentes notaciones para los modelos para los diagramas E/R, en esta metodología se ha adoptado la notación de Martín (ver apéndice B).

8.6.1.2 Patrón 6-1-2: Modelo de procesos

Contexto

Es necesario organizar y documentar la estructura y flujo de los datos a través de los procesos de un sistema. También es importante establecer la lógica, políticas y procedimientos a ser implementados por los procesos del sistema.

Solución

Hay diferentes técnicas para modelar los procesos tales como diagramas llamados cartas estructuradas, diagramas de flujo, o tablas de decisión. En esta metodología se plantea usar el **diagrama de flujo de datos**.

8.6.1.3 Patrón 6-1-3: Modelo de objetos

Contexto

En algunos sistemas los datos y los procesos no se modelan de forma independiente, por lo tanto se hace necesario utilizar otra técnica que nos permite unir (encapsular) los datos y los procesos en una sola unidad (objeto). Esta técnica debe permitir estudiar los objetos existentes y determinar si ellos pueden ser reusados o adaptados a nuevos usos. Por otra parte también debe permitir crear nuevos objetos o modificar objetos que serán combinados con objetos ya existentes dentro de una aplicación.

Solución

Para construir este modelo de objetos, es decir identificar los objetos que componen el sistema y las relaciones que existen entre ellos, se recomienda usar la técnica de **casos de uso** [[Jacobson, 1992], [Textel, Williams, 1997].

El proceso para obtener el modelo de objetos en esta etapa de análisis es el siguiente:

1. **Encontrar e identificar los objetos del sistema**. Para llevar a cabo esta actividad se debe hacer lo siguiente:

    - **Paso 1. Obtener una lista inicial de los objetos**. Para obtener una lista inicial de los posibles objetos, se debe conseguir un documento donde los usuarios narren o describan el sistema que ellos desean y que debe ser diseñado. Después es necesario subrayar cada uno de los sustantivos o cláusula sustantiva (sujeto) y ponerlos en una tabla.
    - **Paso 2. Incrementar o corroborar la lista inicial de objetos**. Realizar una segunda iteración para corroborar los objetos ya encontrados en la primera iteración de búsqueda o para encontrar e identificar nuevos objetos y agregarlos a dicha lista Para hacer ésto se recomienda utilizar la técnica llamada **Casos de Uso** (ver apéndice D). El modelado con casos de uso permite identificar y modelar los eventos del sistema, quien los inicia y como el sistema responde a ellos. Los pasos involucrados en el modelado con casos de uso son los siguientes:

    Actividad 1. Identificar los actores y los casos de uso. Para hacer esto se recomienda utilizar el **diagrama de contexto** del sistema obtenido en el modelado de los procesos por medio de la técnica de **diagramas de flujo de datos (DFDs**, ver apéndice C). El diagrama de contexto ilustra las partes externas que interactuan con el sistema, las cuales suministran las entradas para el sistema y reciben del mismo sus salidas. En resumen el diagrama de contexto identifica las fronteras y el alcance del sistema. De este diagrama deben tomarse las partes externas que suministran las entradas al sistema y considerarlas cada una como un ***actor*** (el término *actor* usado en casos de uso es

diferente al utilizado por el paradigma de programación orientada a objetos, ver apéndice E). La secuencia de eventos con los cuales el sistema responde a la entrada de las partes externas se considera un ***caso de uso***.

Actividad 2. Construir un modelo del caso de uso. Con todos los casos de uso y actores identificados, la funcionalidad del sistema está definida. Ahora se debe hacer un **diagrama modelo de los casos de uso** (ver apéndice D), que representa el alcance y la fronteras en términos de los casos de uso y los actores. En este diagrama se representan las relaciones entre actores y los casos de uso que a su vez deben ser agrupados en subsistemas a los cuales pertenecen.

Actividad 3. Documentar el curso de los eventos en el caso de uso. Para cada caso de uso se debe documentar el curso normal de los eventos del caso de uso, es decir una descripción con el actor que inicia el caso de uso y finalizando con el evento del sistema. Los elementos para dicha descripción pueden ser:

- El nombre del caso de uso
- El nombre del actor que lo inicia
- Una descripción muy general del caso de uso
- El curso normal del evento describiendo los principales pasos del caso de uso, desde su inicio hasta el fin de su interacción con el actor
- Una condición previa describiendo el estado del sistema antes de ejecutar el caso de uso
- Y finalmente una condición posterior que describa el estado del sistema una vez que se ejecute el caso de uso.

Actividad 4. Identificar las dependencias en el caso de uso. Algunos casos de uso pueden ser dependientes de otros, es decir puede haber un estado resultado de un caso de uso que a su vez es la condición previa para otro caso de uso. Para esto debe utilizarse un **diagrama de dependencia del caso de uso** (ver apéndice D).

Actividad 5. Documentar el curso alterno de los eventos en el caso de uso. Es necesario modelar cursos alternos o condiciones de excepción en los casos de uso. Los cursos alternos son desviaciones o ramificaciones del curso normal del evento. Los cursos alternos deben ser documentados como otro curso del caso de uso y se puede hacer incluyendo un elemento más de documentación del caso de uso que describa los cursos alternos.

Actividad 6. Encontrar los objetos potenciales. Debe tomarse cada caso de uso y encontrar los sustantivos que corresponden a entidades o eventos del sistema. Cada sustantivo debe ser agregado a la lista inicial de objetos identificados o verificar que ya exista en esta lista inicial.

Actividad 7. Seleccionar los objetos propuestos. De la lista de objetos identificados, elimine los sustantivos que representen lo siguiente: *sinónimos, sustantivos que estén fuera del alcance del sistema, sustantivos que son realmente acciones o atributos, sustantivos que corresponden a roles externos.*

2. **Organizar los objetos e identificar sus relaciones.** Una vez definidos los objetos del sistema, deben ser organizados y se deben documentar las relaciones principales entre los objetos. El modelo de asociación de objetos es usado para mostrar gráficamente los objetos y sus relaciones. En este modelo también se debe incluir la multiplicidad, relaciones de Generalización/Especialización y relaciones de agregación. Debe usarse la notación OMT para la construcción del diagrama. Los pasos envueltos en esta actividad son:

   - **Paso 1. Identificar las asociaciones y la multiplicidad.** En este paso se deben identificar las relaciones o asociaciones que existen entre los objetos o clases. Una vez identificadas las relaciones se debe determinar la multiplicidad o cardinalidad de esas asociaciones.
   - **Paso 2. Identificar relaciones de Generalización/Especialización** (ver apéndice A).
   - **Paso 3. Identificar las relaciones de agregación o composición** (ver apéndice A).
   - **Paso 4. Preparar el modelo de asociación de objetos.** Este modelo debe reflejar las asociaciones y la multiplicidad que se encontraron en el paso 1, las relaciones de

Generalización/Especialización encontradas en el paso 2, y las relaciones de agregación o composición encontradas en el paso 3.

8.6.2 Patrón 6-2: Modelo de la distribución lógica

Contexto

La etapa de distribución lógica diseña la aplicación como un conjunto de entidades distribuidas lógicamente, que pueden comunicarse a través de mensajes. Es necesario determinar cómo el esquema global de la aplicación es la composición de diversos esquemas locales. Se busca entonces documentar las localizaciones lógicas del sistema independientemente de la parte física. Consecuentemente a los modelos lógicos de objetos (datos, procesos e interfaces) les seguirá un modelo físico que describe el diseño del sistema y la solución distribuida.

Solución

Usar un modelo lógico para determinar cómo los objetos (datos, procesos e interfaces) deben ser distribuidos en cada una de las localizaciones determinadas. Para lograr lo anterior se debe considerar lo siguiente:

- Utilizar una técnica de **modelado de la red** que permita documentar la estructura geográfica de un sistema en términos de sus localizaciones. La técnica definida aquí es un **diagrama de conectividad de las localizaciones** (ver apéndice E). Dicha técnica permite crear un modelo en términos de la localización de los objetos (datos, procesos, interfaces y usuarios) y las interconexiones necesarias entre estas localizaciones.
- Usar una técnica de descomposición de las localizaciones que consisten de otras localizaciones y grupos. Dicha técnica es llamada **diagrama de descomposición de las localizaciones** (ver apéndice E) y muestra la descomposición geográfica de las localizaciones a ser incluidas en el sistema.
- Sincronizar los modelos del sistema. Los modelos de red, datos, interfaces y procesos son diferentes vista del mismo sistema, que se encuentran interrelacionadas entre ellas. Es necesario sincronizar dichas vistas para asegurar que estén completas y consistentes con las

especificaciones de todo el sistema. Los siguientes son las sincronizaciones básicas que habría que hacer:

- Sincronización del modelo de datos y del modelo de procesos
- Sincronización del modelo de red y del modelo de datos
- Sincronización del modelo de datos y del modelo de interfaces
- Sincronización del modelo de red y el modelo de procesos

Se puede hacer la sincronización por medio de una **matriz CRUD para la localización de datos** en la cual las filas indican entidades (y posiblemente atributos); las columnas indican localizaciones, y las celdas (intersección de filas y columnas) muestran el nivel de acceso que puede ser C=Crear, R= lectura o uso, U=actualizar o modificar y D=eliminar o desactivar.

El proceso para la construcción de los modelos lógicos de red es el siguiente:

1. **Construir un diagrama de descomposición de las localizaciones**. Lo primero que debe definirse son sus localizaciones. Para esto se debe tener en cuenta lo siguiente:

   - Pensar en los lugares donde usuarios directos e indirectos pudieran estar localizados.
   - Utilizar los agentes externos de sus DFDs para obtener posibles localizaciones externas.
   - No olvidar agregar a la lista localizaciones móviles o en movimiento.
   - Agrupar las localizaciones obtenidas en el diagrama de descomposición, tomando las localizaciones similares que están en un mismo nivel o dentro de la misma rama del árbol.

2. **Diagrama de conectividad de las localizaciones**. Para lograr este diagrama se siguen los siguientes pasos:

   - Obtener un primer diagrama de conectividad que contenga las localizaciones externas y las localizaciones que tienen sublocalizaciones.

- Hacer una explosión del primer diagrama de conectividad obtenido. Es decir abrir una localización en sus sublocalizaciones. Hacer dicha explosión tomando el diagrama de conectividad del sistema y desplazándose hacia abajo para mantener de esta manera consistencia entre los diagramas.

8.6.3 Patrón 6-3: Diseño del sistema

En el análisis se identificaron los objetos y casos de uso basados en condiciones ideales e independientes de cualquier solución de hardware o software. Durante esta etapa se refinarán estos objetos y casos de uso para reflejar el ambiente actual de la solución propuesta. Las actividades a realizar son las siguientes:

1. **Refinar el modelo de casos de uso para reflejar el ambiente de implementación**. En esta iteración del modelado de casos de uso, los casos de uso serán refinados para incluir los detalles de como el actor (usuario) interactua con el sistema y como el sistema responde a esos estímulos para procesar el evento. La manera como el usuario accede el sistema (por medio de un menú, ventana, botón, lector de barras, impresora, etc) deberán describirse en detalle.

   El refinar los casos de uso, puede surgir la necesidad de adicionar nuevos casos de uso. Un caso de uso puede tener una funcionalidad compleja que esta compuesta de varios pasos que son difíciles de entender. Para simplificar el caso de uso y hacer este más fácil de entender, se pueden extraer los pasos más complejos dentro de su propio caso de uso. Este tipo de caso de uso es llamado una **extensión a un caso de uso** en el sentido que extiende la funcionalidad del caso de uso original. También se pueden encontrar casos de uso que realizan pasos de idéntica funcionalidad. En esta situación se extraen los pasos comunes en un caso de uso llamado **un caso de uso abstracto**.

   Los pasos a seguir en esta primera actividad son:

   - **Paso 1. Transformar los casos de uso del análisis a casos de uso del diseño**. En este paso se refinan cada uno de los casos de uso para reflejar los aspectos físicos del

ambiente de implementación el nuevo sistema. Por ejemplo en donde se especifica el actor del caso de uso se puede agregar cual es el usuario real del sistema, en la descripción del curso de los eventos en el caso de uso se puede describir las ventanas que se muestran al usuario y el contenido (nombres de los campos) de dichas ventanas, también puedo agregar la dirección física de una impresora.

- **Paso 2. Actualizar el diagrama modelo de casos de uso y otra documentación para reflejar cualquier nuevo caso de uso**. Después que todos los casos de uso del análisis han sido transformados a casos de uso del diseño, es posible que nuevos casos de uso, dependencias entre casos de uso, o nuevos actores, se hayan encontrado. Actualice el diagrama de casos de uso, el diagrama de dependencias entre los casos de uso de acuerdo a la nueva información encontrada en el paso 1.

2. **Modelado de las interacciones y comportamientos que soportan el escenario de los casos de uso**. En esta actividad se identifican y establecen categorías para los objetos del diseño que fueron establecidos para cada caso de uso. Además se identifican las interacciones del objeto, sus responsabilidades y sus comportamientos. Los pasos en esta actividad son:

   - **Paso 1. Identificar y clasificar los objetos para el diseño de los casos de uso**. Existen tres categorías de objetos de diseño, interfase, control y entidad. En este paso se deben examinar cada uno de los casos de uso del diseño e identificar y clasificar los tipos de objetos requeridos por la lógica del caso de uso.
   - **Paso 2. Identificar los atributos del objeto**. Durante el análisis y el diseño se pueden encontrar los atributos del objeto. En este paso se debe examinar cada caso de uso para ver si hay atributos adicionales y actualizar los diagramas de asociación del objeto con los atributos.
   - **Paso 3. Modelar las interacciones del objeto para un caso de uso**. Una vez identificados y categorizados los objetos del diseño involucrados en un caso de uso, es necesario modelar estos objetos y sus interacciones por medio de un **diagrama para modelar los objetos ideales** (ver apéndice D).
   - **Paso 4. Identificar los comportamientos y responsabilidades del objeto**. En este paso se debe examinar la descripción del caso de uso para identificar los verbos que significan acciones. Estos verbos son posibles comportamientos (métodos) que son

necesarios para completar el escenario de los casos de uso. Una vez determinados estos comportamientos se debe establecer si se realizan en forma manual o automatizada. Si son automatizados deben asociarse con el tipo de objeto adecuado que tendrá la responsabilidad de ejecutar el método. Una tercera tarea es documentar estos comportamientos y las colaboraciones entre los objetos por medio de la **tarjeta de responsabilidades y colaboración entre las clases (CRC)** (ver apéndice A).

- **Paso 5. Modelo detallado de las interacciones del objeto para un caso de uso**. Una vez determinado el comportamiento de los objetos y sus responsabilidades, se debe crear un modelo detallado que muestre la interacción entre los objetos para lograr la funcionalidad especificada en cada caso de uso tenido en cuenta en el diseño. El modelo aquí usado es el **diagrama de interacción de objetos** (ver apéndice A).

3. **Actualizar el modelo de objetos para reflejar el ambiente de implementación**. Una vez diseñados los objetos y las interacciones requeridas por ellos, se puede refinar el modelo de objetos para incluir los comportamientos o métodos de implementación que los objetos necesitan.

8.6.4 Patrón 6-4: Modelo de la distribución física

Contexto

Una vez modelados los requerimientos durante la fase de análisis, se debe modelar durante el diseño del sistema los requerimientos y la arquitectura tecnológica. Durante el análisis se obtuvieron los modelos lógicos de los objetos esenciales para los requerimientos del sistema y fueron identificados y modelados por medio de casos de uso. Ahora se debe especificar como los objetos lógicos serán físicamente implementados.

Solución

Lo que aquí se plantea es un diseño general de alto nivel que pueda servir como una arquitectura de la aplicación para el sistema. Para la obtención del modelo de la distribución física tome como base los modelos obtenidos en la fase de análisis:

- El *modelo lógico de los datos,* el *modelo lógico de los procesos* ó, el *modelo lógico de los objetos* (datos y procesos encapsulados en un objeto) y,
- el *modelo lógico de la red.*

Dados estos modelos se deben distribuir los objetos (datos, procesos ó datos y procesos encapsulados) para crear un diseño general, que normalmente esta restringido por aspectos tales como **estándares de la arquitectura** que predeterminan la escogencia de sistemas administradores de bases de datos, **tecnología y topología de la red**, **interfaces del usuario**, **métodos de procesamiento**, y finalmente **factibilidad a nivel de costos** representados en dinero y rendimiento del sistema.

Las siguientes son las actividades que deben realizarse para la obtención del modelo físico:

1. Se debe obtener un diagrama físico llamado **diagrama de objetos con la topología de la red**, en el cual se asignan procesadores físicos (clientes y servidores) y dispositivos (por ejemplo máquinas y robots) a una red. Después debe establecerse, la conectividad entre clientes y servidores, y desde donde los usuarios interactuarán con los procesadores. Se puede dibujar un solo diagrama físico que abarque todos los objetos o un conjunto de diagramas físicos de los objetos. En este diagrama, debe indicarse lo siguiente:

    - Los servidores y sus localizaciones físicas
    - Los clientes y sus localizaciones físicas
    - Especificaciones del procesador
    - Protocolos de transporte

2. Distribuir los objetos (datos, procesos o datos y procesos encapsulados) teniendo en cuenta aspectos tales como la localización de objetos específicos en diferentes servidores, o la localización de subconjuntos de objetos en diferentes servidores, o replicación (duplicados) de objetos específicos o subconjuntos de objetos en diferentes servidores. Por otra parte considerar si los procesos van asignados a un procesador servidor ó a un procesador cliente.

8.6.5 Patrón 6-5: Necesidades de implementación

Contexto

En esta etapa ya tenemos modelado nuestro sistema desde el punto de vista del análisis y el diseño. Es importante ahora orientar los esfuerzos hacia las características que debe tener un sistema distribuido. En el patrón 4 se hizo una generalización de los asuntos más importantes a tener en cuenta para el análisis y diseño de sistemas distribuidos partiendo de la característica de transparencia de dichos sistemas. La idea en esta etapa es ver cómo cada uno de estos asuntos afecta los modelos obtenidos. Por consiguiente cada asunto puede afectar los modelos obtenidos para un asunto anterior, lo cual quiere decir que existe una relación muy fuerte entre cada uno de ellos a la hora de hacer el análisis y diseño del sistema distribuido.

Por otra parte es necesario definir como se puede atacar cada uno de los asuntos definiendo un esquema general para todos.

Solución

Cada asunto puede trabajarse en tres niveles: **modelos**, **políticas**, y **mecanismos**. Los modelos describen las expectativas del usuario acerca de la aplicación o comportamiento del sistema. Las políticas definen algoritmos para soportar los modelos de la aplicación. Los mecanismos definen la funcionalidad que es usada por las políticas para implementar sus algoritmos.

En el siguiente ejemplo se muestran los tres niveles enunciados. Supóngase que el asunto que se va a desarrollar es la concurrencia. En este asunto se podría hablar de generación y control de

acceso. Éste último se toma para el ejemplo y para los tres niveles enunciados se tiene lo siguiente:

- **Modelos**: un posible modelo para el control de la concurrencia es el modelo serial que establece que en el acceso concurrente a los recursos no debe haber conflicto, lo cual significa que los efectos de la ejecución de actividades en forma concurrente debe equivaler a una ejecución secuencial.
- **Políticas**: el anterior modelo es soportado por políticas para el control de la concurrencia, dichas políticas tienen dos diferentes perspectivas: *pesimista* cuando la aplicación espera tener una alta interacción, y *optimista*, cuando el nivel de interacción se supone bajo. Por otro lado las políticas se consideran *dinámicas* en las cuales el orden de serialización es determinado por el orden en el cual los objetos son accesados, y *estáticas*, en las cuales el orden de serialización está basado en un orden total predefinido.
- **Mecanismos abstractos**: cada una de las políticas puede usar mecanismos abstractos para generar la concurrencia, por ejemplo la *Actividad* la cual abstrae mecanismos de *hilo*, y para control, por ejemplo *exclusión mutua* que abstrae mecanismos de *mutex*.

En esta etapa toma relevancia el concepto de los patrones de diseño ya que como hemos visto los patrones de diseño son soluciones orientadas a trabajar con estos niveles de modelos, políticas y mecanismos de abstracción. Además en la actualidad los mecanismos abstractos son un particular tipo de patrón, en el sentido de que ellos están compuestos de diversos mecanismos.

8.6.6 Patrón 6-6: Integración

Contexto

Durante todo el análisis y diseño del sistema se obtienen productos que deben ser analizados en conjunto para de esta manera determinar si los mismos son coherentes o no y si definitivamente están cubriendo todos los requerimientos del sistema.

Solución

Se debe tomar todos los modelos obtenidos y realizar una verificación entre ellos. Es decir se puede establecer por ejemplo si un flujo de datos encontrado por medio de un diagrama de flujo de datos es definitivamente consistente con las entidades que contienen los datos en un diagrama entidad relación. Igualmente se puede establecer si los procesos encontrados en un diagrama de flujo de datos realmente han sido asignados y encapsulados en un objeto adecuado. Por otro lado también puede suceder el hecho de que haya inconsistencias entre los modelos, por ejemplo algo encontrado en uno que no aparece en el otro, lo cual también debe ser aclarado. En resumen la idea fundamental aquí es establecer relaciones entre todos los modelos y generar documentación que contenga los resultados obtenidos de dicho análisis para de esta manera tomar la decisión de si se pasa definitivamente a la etapa de desarrollo o por el contrario es necesario hacer una nueva iteración sobre las etapas de análisis y diseño del sistema, de tal manera que queden cubiertos todos los requerimientos del mismo.

8.7 Conclusiones

Las metodologías existentes para el desarrollo de sistemas de software distribuido, normalmente están enfocadas sobre problemas muy específicos con relación a cada uno de los asuntos más relevantes en un sistema distribuido. Cada una de estas metodologías tiene un aporte importante a la hora de desarrollar un sistema de software distribuido, sin embargo si se desea analizar y diseñar un sistema distribuido en el cual alguna característica está fuera del alcance de dichas metodologías su aplicabilidad queda supeditada a un problema determinado y no para enfrentar en términos generales el diseño de cualquier sistema distribuido. Ahora bien, lograr un esquema completamente general que aplique para cualquier tipo de problema no es tarea fácil. De acuerdo a lo anterior, la única forma de crear una metodología como la enunciada anteriormente, es empezar con una abstracción al más alto nivel posible con respecto al proceso que se debería seguir para el análisis y diseño de los sistemas de software distribuido. La obtención de dicha abstracción permite que posteriormente se puedan ir tomando cada uno de los elementos enunciados en la misma y en niveles inferiores generar nuevos procesos que aporten al esquema general. Por otra parte se busca que los elementos componentes de todo el esquema puedan ser

reutilizados para problemas que se presentan en forma recurrente y es aquí donde los patrones de diseño tienen una gran importancia ya que la filosofía de los mismos está enfocada a plantear soluciones que puedan ser reusadas.

Lo propuesto en el presente trabajo de tesis es precisamente un esquema metodológico generalizado que se enfoca sobre los aspectos más generales de un sistema distribuido enunciando entonces lineamientos básicos a tener en cuenta para el análisis y diseño de sistemas de software distribuido. Por esta razón hay gran diferencia entre cada una de las metodologías existentes y el esquema aquí presentado ya que muchas de esas metodologías pueden ser posteriormente integradas a este esquema como componentes de la misma. El estado ideal al que se debe llegar es obtener un "libro de bolsillo" como lo tienen otras disciplinas en el área de las ingenierías, que describa soluciones satisfactorias para problemas conocidos.

El presente trabajo es importante ya que es bien reconocido que los programadores expertos no piensan con respecto a los programas en términos de elementos de lenguajes de programación de bajo nivel, sino en abstracciones en un nivel muy superior. Entonces al trabajar con patrones las personas están sistemáticamente documentando abstracciones diferentes a las de algoritmos y estructuras de datos. Es importante como se hizo en este trabajo basado en patrones no concentrarse inicialmente sobre formalismos del desarrollo para expresar los patrones o en herramientas para usar ellos. En su lugar los esfuerzos se encaminaron en documentar los patrones claves para ser usados por los analistas y diseñadores de sistemas en general.

La adaptación del esquema propuesto a los sistemas distribuidos se da debido a que los elementos del patrón (contexto y solución) están directamente enfocados sobre aspectos relacionados con el análisis y diseño de sistemas de software distribuidos. Además, el tratamiento que se da con la **separación de asuntos**, permite que los requerimientos de un sistema distribuido cualquiera puedan ser cubiertos y solucionados por el esquema con la aplicación de patrones ya existentes o en su defecto con la creación de uno nuevo a raíz del planteamiento de una nueva solución para el requerimiento en cuestión.

9. CONCLUSIONES

El desarrollo de cualquier sistema de software es una tarea compleja debido a la gran variedad de aspectos que hay que tener en cuenta. Es definitivo que el desarrollo de un sistema de software, requiere de una metodología que permita, en forma ordenada, aplicar diferentes técnicas para modelar cada uno de los aspectos característicos de un sistema determinado.

Antes de formular o definir una metodología en particular, es conveniente revisar los conceptos generales necesarios para la creación de la misma. Esto permite establecer lineamientos a tener en cuenta para el desarrollo de la metodología y al mismo tiempo clarificar aspectos que hagan de la normativa un esquema soportado en los fundamentos básicos de lo que es una metodología.

En la creación de una metodología es importante el **estilo**, debido a que éste repercute en diferentes aspectos de la misma. El estilo de una metodología puede tener una gran variedad de definiciones. Se debe tener en cuenta lo siguiente para este aspecto:

- Es tentador definir en forma exacta el trabajo de desarrollo del sistema, en términos de las tareas involucradas y su secuencia, pero algunas veces esto es indeseable dadas las características tanto de los sistemas, como de los equipos de personas del proyecto. Por esta razón, en el presente trabajo de tesis se consideró conveniente plantear lineamientos más generales que permitan posteriores investigaciones relacionadas con cada uno de los lineamientos, obteniendo al final un producto que realmente permita el mejor desarrollo de un sistema de software. Como se vió en lel capítulo 7, la aplicación de los patrones, está más orientada a solucionar un problema específico. En cambio, en el esquema propuesto aquí los lineamientos son más generales con relación a los aspectos de análisis y diseño de cualquier sistema de software distribuido.
- Otro aspecto importante es considerar es el punto de inicio formal del desarrollo del sistema. Puede definirse como una sentencia de entradas y salidas del sistema; como un análisis de entidades o eventos en el ambiente del sistema; como un análisis o modelo de los objetos de datos internos que corresponden a estas entidades y eventos; o como una sentencia de las funciones del sistema. En el esquema metodológico propuesto se determinó conveniente que la metodología no considerara solo uno de estos puntos de vista como el correcto, para que

no restrinja las posibilidades de aceptación. Debido a esto se utilizaron diferentes técnicas que permiten modelar el sistema, ya sea a través de flujos de datos, de eventos o funciones del sistema, contrario a lo propuesto por los patrones aplicados en el capítulo 7.

- Otro aspecto sobre el cual una metodología debe ser clara es el esquema conceptual o modelo. Aunque una metodología plantee aspectos en forma general su lineamiento debe indicar en forma clara las características de las técnicas, herramientas y en general de los recursos que pueden encajar adecuadamente en la misma. Es decir hay aspectos en los que la metodología debe ser autoritaria y otros en donde debe ser liberal.

Como también se vió en el capítulo 7, la aplicación de los patrones permite modelar un sistema, sin embargo no existen en ellos mecanismos de verificación en el caso de haber obviado algún aspecto importante a la hora de hacer los modelos. Por el contrario, en el esquema metodológico presentado en este trabajo de tesis se cubren los anteriores aspectos, ya que trata de ser flexible en los paradigmas escogidos y los combina con el objetivo de que cada técnica aplicada para cualquiera de estos paradigmas ayude a verificar los productos que se van obteniendo. Por ejemplo, en el esquema propuesto, una técnica utilizada para identificar y modelar los procesos del sistema es el diagrama de flujo de datos (DFDs). Posteriormente se puede aplicar la técnica de casos de uso que también me permite identificar y modelar los procesos del sistema. Cada una de las técnicas se aplica independientemente pero en forma secuencial, con el objetivo de que los procesos obtenidos con la primera técnica puedan ser corroborados o complementados con la aplicación de la segunda. Esto permite que aspectos que se hayan escapado al aplicar una de las técnicas puedan ser detectados al aplicar otra o, en el peor de los casos, verificar que nuestro primer modelo está correcto.

Otro aspecto importante en la creación de una metodología es el **ambiente** y se debe considerar que pueden haber varios y que ellos son complementarios. Un ambiente puede ser aquél en donde los especialistas piensan acerca del problema e investigan una solución; otro en el que el desarrollador debe descubrir con mucho detalle información acerca del dominio del problema en el cual trabaja, y construir un modelo de éste; uno más en donde observaciones y conjeturas son sometidas a análisis para reducir información o reducir errores; y por último un ambiente en el que el término prototipo puede ser usado para permitir que el diseño y la implementación se puedan realizar rápidamente, tal vez con poco conocimiento sobre la eficiencia y poca evaluación

operacional. Este ambiente tiene una estrecha participación del usuario así como el natural aprendizaje del proceso de desarrollo. Refleja las distinciones mínimas entre especialistas y no especialistas y entre desarrollo y operación.

Los patrones aplicados en el capítulo 7 solamente se orientan en modelar el sistema a través de diagramas entidad-relación para objetos. En cambio en el esquema metodológico presentado aquí se dan además de estas herramientas, muchas otras que permiten obtener un mejor detalle tanto del dominio del problema como de los modelos encontrados. En resumen el esquema metodológico permite la presencia de todos los ambientes en su esquema conceptual, y permite su correcta interacción durante el desarrollo del sistema.

El concepto de **especificación** es importante en el desarrollo de sistemas. La especificación es una descripción de **qué** hace un objeto, lo opuesto a un diseño, que es la descripción de **cómo** éste lo hace. Muchas metodologías ven la especificación como una actividad que ocurre al inicio del "ciclo de vida" y no vuelve a ocurrir. Esta actividad no ve solamente al sistema como un todo que necesita ser especificado; sino que lo descompone en subsistemas, y dentro de componentes tales como bases de datos, interfaces, programas y módulos de programación. Entonces cada una de las necesidades debe ser especificada en un punto apropiado del tiempo durante el desarrollo del sistema.

Para cada especificación de un objeto le debe corresponder un diseño del objeto. Excepto para el nivel más bajo de la descomposición, un diseño es expresado como un conjunto de especificaciones para objetos del siguiente nivel de detalle. Especificación y diseño pueden así ser vistas como actividades que van de la mano, alternando a través del proceso de desarrollo. En este aspecto el esquema metodológico propuesto sugiere técnicas de especificación informales, así como otras más formales al indicar en el patrón 4-2 del esquema metodológico las características que deberían cubrir los lenguajes para especificar un sistema de software distribuido.

La descomposición (o términos como "top-down", "jerarquía", "refinamiento", "estructurado") es un ambiente necesario y deseable a través del desarrollo del sistema. En el caso de los sistemas distribuidos a pesar de que existen herramientas para desarrollo "bottom-up" es conveniente

analizar y diseñar el sistema en forma de descomposición para tener una mejor visión de cada uno de los componentes que serán enlazados en el desarrollo al estilo bottom-up.

Fundamentalmente lo que se busca con la aplicación del esquema metodológico presentado, es que las descripciones de la información importante de las aplicaciones del sistema se desarrollen en forma clara, completa y acordada. Para ser clara la descripción debe estar en términos que tanto usuarios como los desarrolladores la entiendan. Los diagramas deberán ser usados donde ayuden a explicar los conceptos, y descripciones en lenguaje estructurado y lógica estructurada deberán emplearse donde se requiera más precisión. En este sentido como más adelante se concluye, se recomienda utilizar patrones orientados a lenguajes de programación que permitan especificar u obtener descripciones más formales.

Una aplicación de software administrada adecuadamente generará un conjunto de requerimientos del usuario durante su desarrollo y mantenimiento que posteriormente deben ser documentados a medida que ocurran los cambios. De esta forma, El personal administrador y desarrollador puede cambiar pero el sistema mantiene una historia completa de lo que ha ocurrido durante su tiempo de vida. Los requerimientos del sistema son una definición a más alto nivel que la dada por los manuales de mantenimiento del programador, ya que los requerimientos están soportando las discusiones del usuario respecto a los cambios que deberían implementarse, la forma como ellos afectarían el resto del sistema y como deberían ser actualizadas las versiones liberadas.

Actualmente se busca mucho que cualquier metodología esté bastante orientada al usuario para mejorar la comunicación con los especialistas en el desarrollo. Sin embargo, en este trabajo se ha encontrado que ésta es una tarea difícil de lograr y más en el caso de sistemas de software distribuido debido a las características de los mismos. Si se piensa por ejemplo en un ambiente de análisis de requerimientos centrado en el usuario, las descripciones de la aplicación deberán estar orientadas hacia los usuarios más bien que hacia los desarrolladores. Los resultados del análisis del sistema deben ser escritos en términos de fácil entendimiento para los usuarios y deberá indicar claramente cómo los usuarios interactuan con el sistema automatizado. Sin embargo, este principio es difícil de lograr ya que la mayoría de los paradigmas aplicados en el desarrollo de las aplicaciones, involucran técnicas que requieren un conocimiento especializado. Algunas técnicas como los diagramas de flujo de datos (DFDs) son más fáciles de entender para

el usuario ya que los elementos (agentes externos, flujos, procesos y almacenamientos) en los que se basa la técnica y la forma en que se relacionan está muy apegada con el esquema de trabajo diario del usuario. En el caso de objetos tal vez los modelos más generales sean fácilmente entendibles por el usuario, pero a medida que se adentra en los mismos para obtener más detalles, la notación y las abstracciones se hacen muy complejas para un usuario sin conocimiento en el área de desarrollo de sistemas de software. Por otra parte los requerimientos funcionales (instrucciones de los servicios que la aplicación deberá suministrar) y los requerimientos de datos (descripción de la información manejada por la aplicación) son igualmente importantes en la descripción de la aplicación, y deberán ser desarrollados en paralelo. Estos lineamientos siguen la idea de combinar descomposición funcional y análisis de flujo de datos con el análisis para soportar una base de datos integrada. La idea es que, mediante sucesivos pasos, se obtengan diferentes niveles de detalle tanto en requerimientos funcionales como de datos.

Por todo lo anterior, lo propuesto aquí permite que el usuario pueda de alguna manera no estar solamente limitado a un paradigma como el de objetos (propuesto por la mayoría de los patrones aplicados en el capítulo 7) sino que por el contrario tenga alrededor de este otras técnicas que le permitan entender y establecer mejor una relación con el modelo de objetos.

En el caso de sistemas distribuidos la complejidad de sus características hace que cualquier metodología obtenida siga estando más orientada a los especialistas que al usuario. Inclusive los conceptos claves en un sistema distribuido, sin relacionarlos directamente con aspectos de análisis y diseño del sistema de software, no dejan de ser complejos de entender para un usuario del sistema. Es importante entonces, como lo hace el esquema propuesto, enunciar un concepto más general antes de aplicar una técnica, ya que de esta manera se puede entender mejor cúal es el objetivo buscado al aplicar un paradigma determinado.

Se busca que una metodología pueda aplicarse en un momento determinado a una gran variedad de aplicaciones, pero es necesario algunas veces, técnicas adicionales para situaciones especiales, por esta razón es importante dar el ambiente modelo para cada paso en el análisis de sistemas, y comparar este ambiente con otras técnicas disponibles. En resumen, algunos de los sistemas tienen soluciones que atacan directamente un problema específico (como en el caso de los patrones aplicados en el capítulo 7) pero que no tienen un marco más general del cual partir para

ir llegando a una metodología completa. Por esta razón el producto final de esta tesis es un esquema metodológico en lugar de una metodología cerrada, de tal manera que la incorporación de nuevos elementos se haga menos compleja.

Ahora bien, el concepto de patrones de diseño aplicado en esta tesis permite documentar mejor una solución propuesta. Debe resaltarse que los patrones de diseño en casos como el de los sistemas distribuidos, no son fáciles de entender, sin embargo estos patrones son muy valiosos ya que además del concepto de reusabilidad involucrado en ellos, también son un buen y definitivo punto de partida para la solución de un problema o para el planteamiento de adiciones al patrón que permitan enriquecerlo a medida que se aplica a diferentes problemas.

En el esquema metodológico propuesto para el análisis y diseño de sistemas distribuidos se trabaja en una de sus etapas con el concepto de **separación de asuntos**, aspecto que el lenguaje patrón G++ aplicado en el capítulo 7 sólo aplica parcialmente. De esta manera se busca que el análisis y el diseño sean realmente más completos. Este enfoque permite que la complejidad del sistema pueda tratarse bajo un paradigma de "divide y conquista" ya que, como se describió en esta tesis, un sistema distribuido tiene una gran variedad de aspectos que hacen más complejo su análisis y diseño. La separación de asuntos permite en primer lugar establecer prioridades del sistema con respecto a dichos asuntos, así como también, determinar si alguno de ellos no es relevante para el sistema. En resumen, se puede hacer una planeación estratégica de los asuntos, es decir priorizar cada uno de los asuntos y atacarlos según su orden de importancia. Por otro lado también se puede llegar a desechar uno de estos asuntos si definitivamente el sistema no requiere del mismo.

En el tratamiento de los asuntos, los patrones de diseño toman una gran relevancia ya que en ellos están involucrados **métodos**, **políticas**, y **mecanismos de abstracción** que permiten plantear mejor una solución a un asunto determinado así como a un problema más específico dentro de dicho asunto.

Dichos patrones cubren varios rangos o escalas de abstracción. Algunos patrones ayudan a estructurar un sistema de software en subsistemas. Otros patrones soportan el refinamiento de subsistemas y componentes, o de las relaciones entre ellos. Algunos ayudan en la

implementación de los aspectos de un diseño particular en un lenguaje de programación específico. Por eso, es importante tener en cuenta la clasificación dada a los patrones en el capitulo 6, sección 6.1, ya que es importante en el momento de aplicar el esquema metodológico, la integración de los diferentes tipos de patrones con el desarrollo de software. Los **patrones arquitecturales** pueden ser usados en las fases de análisis y diseño preliminar, los **patrones de diseño** durante toda la fase de diseño, y los de **idiomas** durante la fase de implementación. Con lo anterior, el esquema metodológico planteado en esta tesis, no sólo se basa en el paradigma de patrones sino que además está dando lineamientos para la solución de un problema, ya que permite por medio de la anterior clasificación orientarse mejor en la determinación de si un patrón tienen cabida en la etapa del ciclo de vida que se está desarrollando.

En el capítulo 6, sección 6.1, también se enunció que es necesario establecer relaciones entre los patrones ya que un patrón resuelve un problema particular, pero su aplicación puede generar nuevos problemas. Algunos de estos pueden ser solucionados por otros patrones. Componentes únicos o relaciones dentro de un patrón particular pueden por supuesto describirse con patrones más pequeños, todos ellos integrados por el patrón más grande en el cual ellos están contenidos. Por eso para usar los patrones efectivamente, el esquema metodológico propuesto en este trabajo de tesis organizó los patrones dentro de lo que se llama un **sistemas patrón** (figura 8.1), con el fin de describirlos en forma uniforme, clasificarlos, y lo más importante, mostrar como están relacionados unos con otros.

Finalmente, a medida que los patrones de diseño puedan ser clasificados y ligados en forma adecuada y el catálogo de los mismos crezca, las herramientas para el desarrollo de software podrían trabajar bajo el concepto de una librería de patrones similar a lo aplicado hoy en día en el paradigma orientado a objetos. Este esquema sería fundamentalmente para el desarrollo de sistemas de software como es el caso de los sistemas distribuidos ya que las herramientas para el desarrollo y la metodología relacionada con ellas estarían internamente trabajando con una base fundamentada en el concepto de patrones.

Los trabajos futuros con relación a esta tesis se deben enfocar en tomar el esquema planteado y empezar a agregar elementos (patrones) para cada uno de los lineamientos. Posteriormente,

habría que pensar en crear una herramienta del modelo conceptual obtenido como metodología para ayudar al análisis y diseño de sistemas distribuidos.

10. BIBLIOGRAFIA

[Aarsten, Brugali, Menga, 1996] Aarsten A., Brigali D., Menga G., "Diseñando sistemas concurrentes y de control distribuido", Commun. ACM 39, Vol 10, Octubre 1996.

[Alford, 1977] Alford M. W. "A Requirements Engineering Methodology for Real Time Processing Requirements", IEEE Trans., Software Engineering, Enero 1977.

[Alford, 1985a] Alford M. W. "SREM at the Age of Eight: The Distributed Computing Design System", IEEE Computer, Abril 1985.

[Balzert, 1989] Balzert H. "CASE Systeme und Werkzeuge", Bibliographisches Institut, 1982.

[Barth,Welsch, 1988] Barth G., Welsch C., "Objekorientierte Programmierung. Informationstechnik it, Junio 1988.

[Bergland, 1981] Bergland G. D. "A Guided Tour of Program Design Methodologies Computer", Octubre 1981. También contenido en Oman P. W., Lewis T. G. (De.) "Milestones in Software Engineering", IEEE Press, 1990.

[Boehm, 1988] Boehm B. "A Spiral Model for Software Development and Enhancement", Computer, Vol. 21, No 5, Mayo de 1988.

[Booch, 1986] Booch G. "Object Oriented Development", IEEE Transactions on Software Engineering, Febrero 1986.

[Buschmann, 1996] Buschmann F. "A system of patterns: Pattern - Oriented Software Architecture", Wiley, 1996.

[Cameron, 1986] Cameron J.R., "Overview of JSD". IEEE Transactions of Software Engineering, Febrero 1986.

[Campbell, Habermman, 1974] Campbell R. H., Habermman A. H.; "The Specification of process Synchronisation by Path Expressions". Lecture Notes in Computer Science Vol 16, Springer Verlag, 1974.

[Chen, 1976] Chen P. P. "The Entity Relationship Model - Towards a Unifying View of Data". ACM Transactions on Data Base Systems, Marzo 1976.

[Coad, 1992] Coad P., "Object-Oriented Patterns". Communications of the ACM, V 35, N 9, Septiembre 1992.

[Coulouris, Dollimore, Kindberg 1994] Coulouris G. ; Dollimore J.; Kindberg T. "Distributed Systems, Concepts and Design". Addison Wesley, 1994.

[De Alba, 1996] De Alba M. R. "Especificación formal del manejo y administración de una memoria compartida distribuida: Administrador de memoria externo", Tesis de Maestría en Ciencias Computacionales, ITESM-CEM, 1996.

[DeMarco, 1979] DeMarco T. "Structured Analysis and System Specification". Prentice Hall, 1979.

[DIGIT, 1990] Digital Equipment Corporation, "The Digital Cohesion Environment for CASE", 1990.

[Dijkstra, 1965] Dijkstra E. W. "Solution of a Problem in Concurrent Programming". Communications of the ACM, 1965.

[Dijkstra, 1968a] Dijkstra E. W. "Cooperating Sequential, in F. Genuys (De.) Programing Languages. Academic Press, 1968.

[Engelbart, Lehtman, 1994] Engelbart D., Lehtman H., "Working Together". BYTE, Diciembre 1988.

[ESTELLE] "International Standard Organization, Information Processing Systems - Open System Interconnection - ESTELLE - A Formal Description Technique based on an extended state transition model". ISO Draft Proposal 9074.

[Flamenco, 1996] Flamengo F. J. "Manejo de fragmentación y replicación transparente en bases de datos relacionales distribuidas". Tesis de Maestría en Ciencias Computacionales, ITESM-CEM, 1996.

[Fleischmann, 1994] Fleischmann A. "Distributed Systems: Diseño e implementación de software". Springer-Verlag, 1994.

[Fontaine Charles, 1988] Fontaine C. "User-Centered Requirements Analysis". Prentice Hall, 1988.

[Fowler, 1997] Fowler M. "UML Distilled", Addison-Wesley , 1997.

[Fowler, 1997] Fowler M. "Analysis Patterns: Reusable Object Models", Addison-Wesley , 1997.

[Franky, 1988] Franky M. C.. "JOYCE+. Modelo, Lenguaje y Metodología de Programación de Sistemas Distribuidos sobre Redes de Comunicación". Conferencia Latinoamericana de Informática (CLEI), 1988.

[Gane, Sarson 1979] Gane C., Sarson T. "Structured System Analysis: Tools and Techniques". Prentice Hall , 1979.

[GoF, 1995] Gang of Four. Gamma E., Helm R., Johnson R., Vlissides John., "Design Patterns: Elements of Reusable Object-Oriented Sofware", Addison-Wesley , 1995.

[Gracia-Catalin, 1985] Gracia-Catalin R., "A Taxonomy of Current Issues in Requirements Engineering", IEEE Computer, Abril 1985.

[Greenwood, 1988] Greenwood N. R., "Implementing Flexible Manufacturing Systems", Macmillan Education, 1988.

[Hansen-Brinch, 1973] Brinch-Hansen P., "Operating System Principles", Prentice Hall, 1973.

[Hansen-Brinch, 1978] Brinch-Hansen P., "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, Noviembre 1978.

[Hatley, Pirbhai, 1987] Hatley D. J., Pirbhai Y. A. "Strategies for Real-Time Sytem Specification". Dorset House Publishing, 1987.

[Hoare, 1974] Hoare C. A. R., "Monitors: An Operating System Structuring Concept", Commun. ACM 17, Octubre 10, 1974.

[Hoare, 1978] Hoare C. A. R., "Communicating Sequential Process", Commun. ACM 21, Agosto 8, 1978.

[Islam, Dvarakonda 1996] Islam N., Dvarakonda M. "An Essential Design Pattern for Fault-Tolerant Distributed State Sharing", Commun. ACM 39, Vol 10, Octubre 1996.

[ITESM, UT] ITESM; UT; "Tecnologías de Comunicación avanzadas en Robótica y Celdas Flexibles de Manufactura". http://research.cem.itesm.mx/jesus/reportes/proynsf.html (27 Feb. 1997).

[Jacobson, 1992] Jacobson I., "Object-Oriented Software Engineering", Addison Wesley, 1992.

[Jelly, Gorton, 1997] Jelly Y. E., Gorton Y. "Software engineering for parallel & distributed systems", Revista IEEE Concurrency, Julio-Septiembre, 1997.

[Jones, 1979] Jones C. "A survey of programming Design and Specification Techniques", IEEE Proceedings, Specification of Reliable Software. También contenido en Oman P. W., Lewis T. G. (De.) "Milestones in Software Engineering", IEEE Press, 1990.

[Keramidis, 1982] Keramidis S., "Eine Methode zur Spezifikation und korrekten Implementierung von asynchronen Systemen", Arbeitsberichte des IMMD, Univ. Erlangen, 1982.

[Liskov, Berzins, 1986] Liskov B. H., Berzins V. "An Appraisel of Program Specifications, en N. Gehani, A. D. McGettric (De.) Software Specification Techniques. Addison Wesley Publishing, 1986.

[LOTOS] International Standard Organization. "Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique based on the temporal Ordering of observation behavior". ISO Draft Proposal 8807.

[McKenney 1996] McKenney P. "Selecting Locking Primitives for Parallel Programming", Commun. ACM 39, Vol 10, Octubre 1996.

[Manfred] Manfred B. "A Design Methodology for Distributed Systems". Institut für Informatik, Technische Universität München.

[Manfred] Manfred B. "The Design of Distributed Systems - An Introduction to FOCUS". Institut für Informatik, Technische Universität München.

[Marco, Buxton, 1987] Marco A., Buxton J. "The Craft of Software Engineering". Addison Wesley, 1987.

[Milner, 1980] Milner R. "A Calculus of Communicating Systems". Lecture Notes in Computer Science 92, Springer Verlag, Heidelberg 1980.

[Minton, 1996] Minton G. "Programming with CORBA". Artículo de la Unix Review, 1996.

[Mowbray, 1997] Mowbray T.J. "CORBA Design Patterns", John Wiley & Sons, 1997.

[Olle, Soland, Tully, 1987] Olle T. W.; Soland H. G.; Tully C. J.. "Information systems design methodologies: a feature analysis". North-Holland , 1987.

[OOPSLA'95] "Workshop on Design Patterns for Concurrent, Parallel, and Dsitributed Objetc-Oriented Systems". http://www.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers.html", 1995.

[Orr, 1977] Orr K. T. "Structured System Development". Yourdan Press , 1977.

[Orr, 1981] Orr K. T. "Structured Requirements Definition". Ken Orr & Associates, 1981.

[Peterson, 1981] Peterson J. -L, "Petri Net Theory and Modelling Systems". Prentice Hall, 1981.

[Petri, 1962] Petri C. A., "Kommunikation mit Automaten". Schriften des Instituts für instrumentelle Mathematik, 1962.

[Pressman, 1997] Pressman R. S.. "Software Engineering. A practitioner's Approach". Mc Graw Hill, 1994.

[Reisig, 1982] Reisig W. "Petrinetze, Eine Einführung". Springer Verlag, 1982.

[Sane 1998] Sane A. "Referencias sobre patrones para sistemas distribuidos". Http://www.agcs.com/patterns/othersources.html", 1998.

[Sanchez, 1995] Sanchez J. "Notas del curso Sistemas Distribuidos". Maestría en Ciencias Computacionales ITESM-CEM, 1995.

[Schmidt, Fayad, Johnson, 1996] Schmidt D.C., Fayad M., Johnson R. E., "Software Patterns", Revista Communications of tne ACM, Volumen 39, Numero 10, Octubre 1996.

[SDL, 1984] "SDL - Functional Specification and Description Language", CCITT Red Books Volumen VI - Fascicle VI.10 and VI.11, 1984.

[Shatz, Wang, 1989] Shatz S. M., Wang J-P., "Distributed Software Engineering". IEEE Press, 1989.

[SUN Microsystems, 1994] SUN Microsystems, "Managin a Distributed Enviroment: Alternatives for Data Distribution", SUN, 1994.

[Teichroew, Dávid, 1985] Teichroew D.; Dávid G. "System description methodologies". North-Holland, 1985.

[Texel, Williams, 1997] Texel P., Williams C., "Use Cases Combined With BOOCH/OMT/UML: process and products". Prentice Hall, 1997.

[Ward, Mellor, 1985] Ward P. T., Mellor. "Structured Development for Real-Time Systems". Prentice Hall, 1985.

[Warrier, 1974] Warrier J. P. "Logical Construction of Programs". Van Nostrad Reinhold, 1974.

[Wegner, 1987] Wegner P., "Dimensions of Object Based Language Design. OOPSALA '87 proceedings, ACM 1987.

[Zalewski, Ehrenberger, 1988] Zalewski J.; Ehrenberger W. "Hardware and Software for real time process control". North-Holland, 1988.

[Zave, 1982] Zave P., "An Operational Approach to Requirements Specifications for Embedded Systems". IEEE Transactions on Software Engineering, Mayo 1982.

APÉNDICE A. UML, OMT Y OTROS DIAGRAMAS PARA OBJETOS

El **Lenguaje deModelamiento Unificado (UML)** es el sucesor para la oleada de métodos de análisis y diseño orientados a objetos (OOA&D) que aparecieron al final de los ochentas y principios de los noventas. Este unifica los métodos de Booch, Rumbaugh (OMT), y Jacobson. El UML es llamado un lenguaje de modelamiento, no un método. La Técnica para el Modelado de Objetos (OMT) es el método propuesto por Rumbaugh en relación al paradigma orientado a objetos.

A.1 Diagrama de clases

La figura A.1a muestra la notación OMT para clases abstractas y concretas. Una clase está definida por una caja con el nombre de la clase en la parte superior de la misma. Las operaciones están bajo el nombre de la clase. Cualquier variable instancia aparece bajo las operaciones.

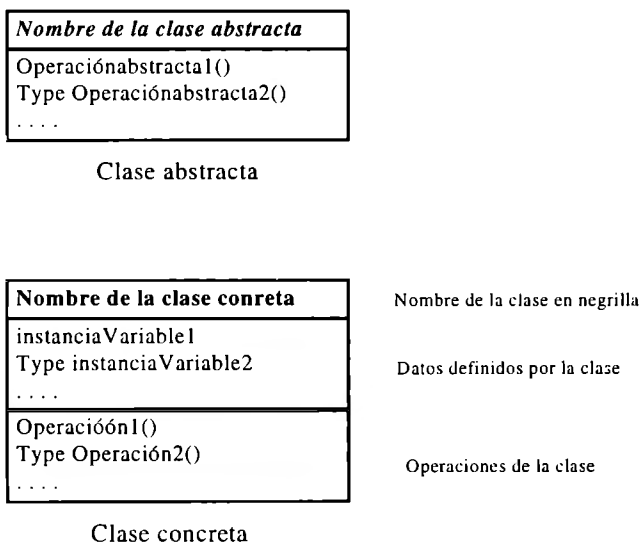
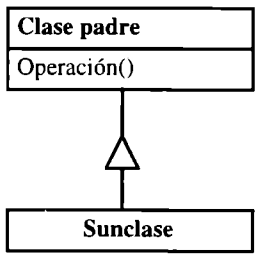


Figura A.1a. Notación para clases abtractas y clases concretas

Herencia

La notación OMT para herencia de clases (figura A.1b) es un triángulo conectando una subclase a la clase padre.

Nuevas clases se definen en terminos de clases ya existentes usando **herencia de clase**



Una subclase se indica con una linea vertical y un triangulo

Figura A.1b. Notación para la herencia de clases

Generalización/Especialización

Cuando hay atributos y comportamientos que son comunes a diferentes clases de objetos, se agrupan en su propia clase llamada una ***superclase***. Los atributos y métodos de la superclase son heredados por todas las clases objeto. La figura A.1c muestra una generalización/especialización. Donde la generalización es la clase persona y las especializaciones son las clases estudiante y profesor.

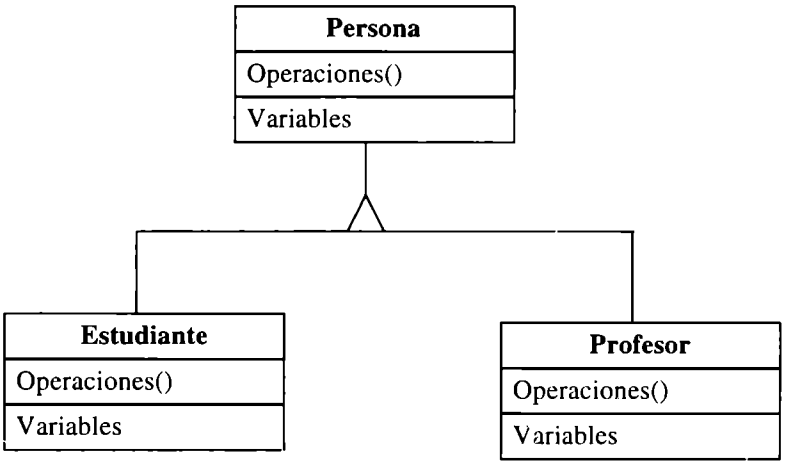
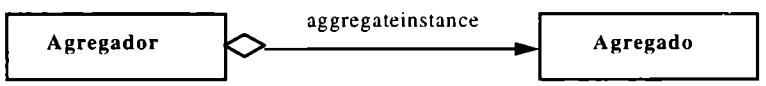


Figura A.1c. Un ejemplo de Generalización/Especialización

Agregación o composición

Un objeto que representa una "parte de" o relación de agregación o composición (figura A.1d) se representa por una línea en forma de flecha con un diamante en la base. La flecha apunta a la clase que es agregada. Una línea en forma de flecha sin el diamante denota "conocimiento". Un nombre puede aparecer cerca de la base para distinguir esta de otra referencias.



Agregación (aggregation): Un objeto posee o es responsable de otro objeto. Se dice que un objeto ha o está siendo parte de otro objeto. Su dueño y el objeto agregado tienen igual tiempo de vida.

Conocimiento(acquaintance): un objeto solamente conoce a otro objeto. También se le llama "asociación" o la relación "usando". Requieren operaciones uno del otro, pero ninguno es responsable del otro.

Figura A.1d. Notación para agregación o composición

A.2 Diagrama de objetos

Un diagrama de objetos muestra instancias solamente. Los objetos (figura A.2.1) son identificados como "a*algo*" ,donde *algo* es la clase del objeto. El símbolo para un objeto es un rectángulo redondeado con una línea separando el nombre del objeto de cualquier referencia al objeto.

Figura A.2.1. Notación para representar un objeto

Tarjeta de responsabilidades y colaboración entre las clases (CRC)

La tarjeta CRC (figura A.2.2) permite identificar todos los comportamientos asociados con un tipo de objeto y las colaboraciones entre los objetos.

Nombre de la clase	
Comportamientos y responsabilidades	**Colaboraciones**

Figura A.2.2. Estructura de una tarjeta CRC

Diagrama de interacción de objetos

Los diagramas de interacción (figura A.2.3) muestran con detalle como los objetos interactuan unos con otros a través del tiempo.

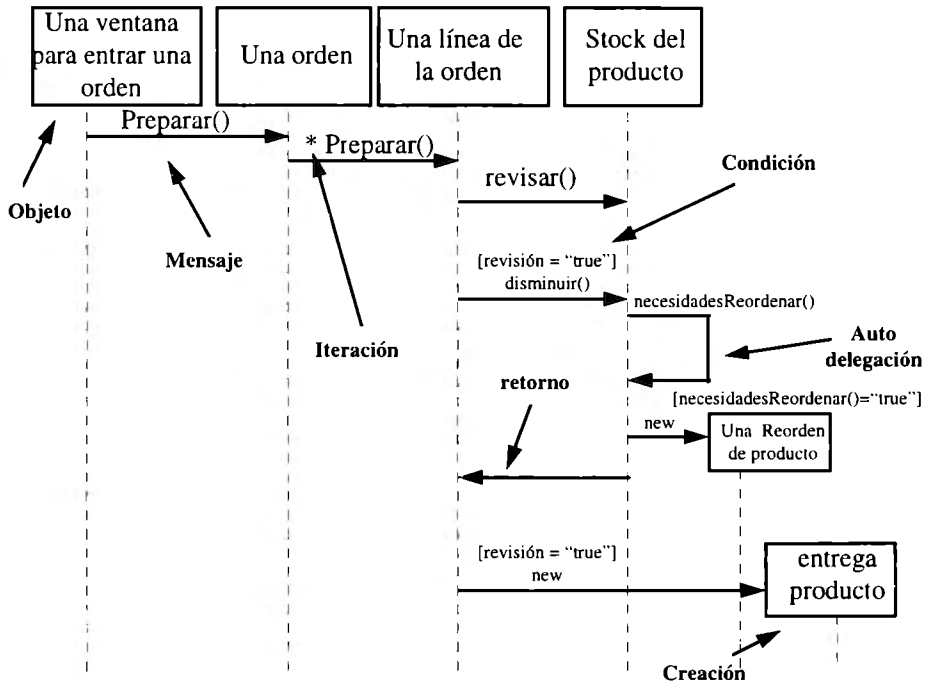


Figura A.2.3. Elementos componentes de un diagram de interacción

Una buena fuente sobre la nomenclatura de objetos, OMT y UML (Lenguaje Unificado para modelado), se encuentra en el libro [Flower, 1997], en éste se trata el lenguaje estándar para el modelado de objetos.

APÉNDICE B. DIAGRAMA ENTIDAD RELACIÓN

El diagrama **entidad-relación** describe datos en términos de entidades y relaciones entre dichas entidades.

Una **entidad** (figura B.1) es alguna cosa acerca de la cual se desea guardar datos. Es decir es una clase de personas, lugares, objetos, eventos, o conceptos acerca de los cuales se desea capturar o almacenar datos.

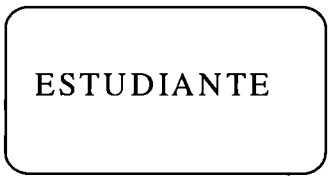


Figura B.1. Representación de una entidad

Una **instancia de una entidad** es una ocurrencia única de una entidad. Por ejemplo la entidad ESTUDIANTE puede tener múltiples instancias: María, Jorge, Carlos, etc.

Un **Atributo** es una propiedad o característica de una entidad. Una **llave** es un atributo, o un grupo de atributos, que tiene un único valor para cada instancia de la entidad.

Una **relación** es una asociación natural que existe entre una o más entidades. Las relaciones pueden representar un evento que enlaza las entidades o una afinidad lógica que existe entre las entidades.

La **cardinalidad** (figura B.2) define el número mínimo o máximo de ocurrencias de una entidad para una única ocurrencia de la entidad con que está relacionada.

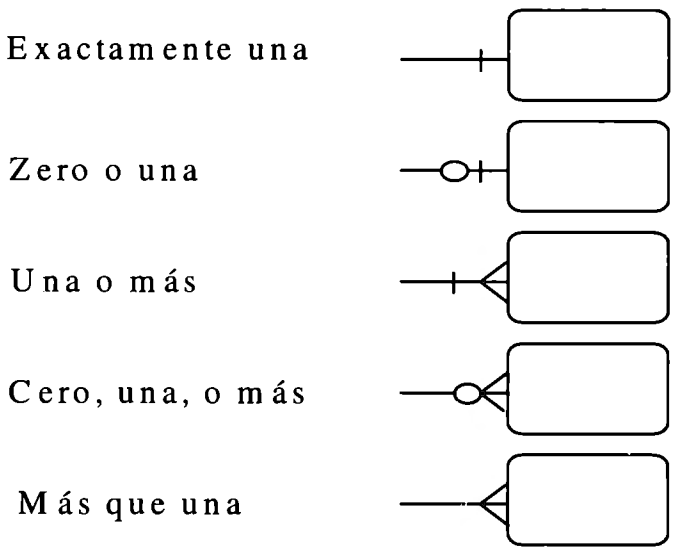


Figura B.2. Los tipos de cardinalidad

Una **entidad asociativa** es una entidad que hereda su llave primaria desde otras entidades (más de una). Cada Parte de la llave concatenada apunta a una y solamente una instancia de cada una de las entidades que están conectadas.

APÉNDICE C. DIAGRAMAS DE FLUJOS DE DATOS (DFD)

El **diagrama de flujo de datos (DFD)** es una herramienta que muestra los flujos de datos a través del sistema y el trabajo o procesamiento realizado por el sistema. Existen diferentes notaciones para DFDs según el autor, la figura C.1 muestra una forma de notación:

Figura C.1. Elementos de un diagrama de flujo de datos (DFD)

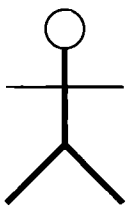
APÉNDICE D. CASOS DE USO

El **modelado de casos de uso** es el proceso de identificar y modelar eventos del sistema, quién los inicia, y cómo el sistema responde a ellos.

Un **caso de uso** es una secuencia de pasos relacionados (un escenario), tanto manual como automatizado, con el propósito de completar una tarea del sistema. La figura D.1b muestra la notación para representar un caso de uso.

Un **actor** representa cualquier cosa que interactua con el sistema para intercambiar información. Un actor es un usuario o un rol en el que podría estar un sistema externo o persona. La figura D.1a muestra la notación para un actor.

La figura D.1 muestra los símbolos usados en un caso de uso:



Símbolo para un actor

(a)

Símbolo para un caso de uso

(b)

Figura D.1. Símbolos para el modelado con casos de uso

En la figura D.2 es un ejemplo de un diagrama modelo de los casos de uso en el cual se utilizan los símbolos enunciados anteriormente:

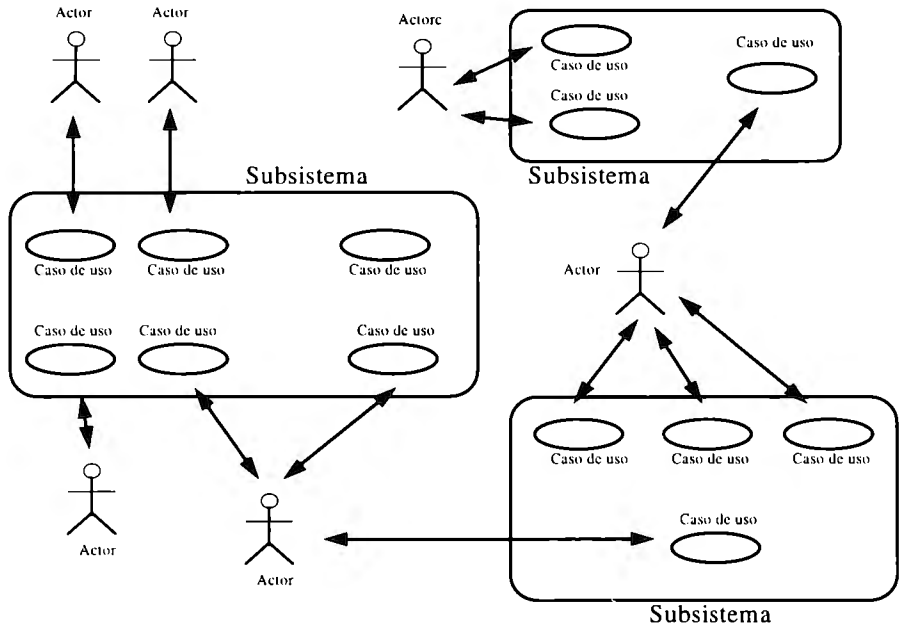


Figura D.2. Ejemplo de un diagrama modelo de los casos de uso

Un ejemplo de un **diagrama de dependencia del caso de uso** se muestra en la figura D.3.

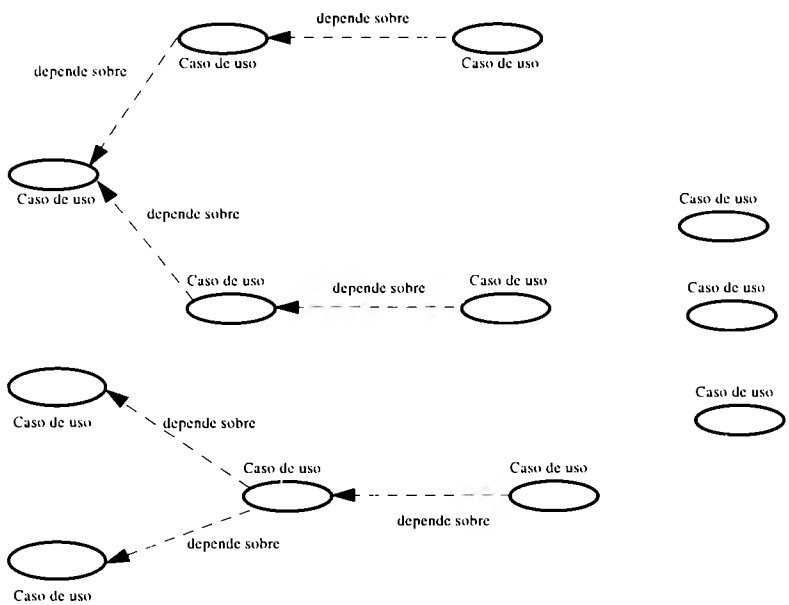


Figura D.3. Diagrama de dependencia del caso de uso

Los **objetos interfase** (figura D.4) son los objetos a través de los cuales el los usuarios se comunican con el sistema. La funcionalidad del caso de uso que describe el usuario que interactua directamente debe colocarse en objetos interfase.

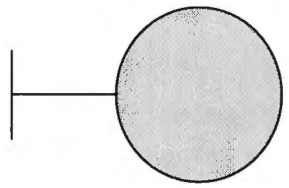


Figura D.4. Notación para un objeto interfase

Los **Objetos entidad** (figura D.5) usualmente corresponden a ítems en la vida real que contienen información, conocidos como atributos, que describen las diferentes instancias de la entidad. Estos objetos también encapsulan el comportamiento (métodos) que mantienen su información o atributos.

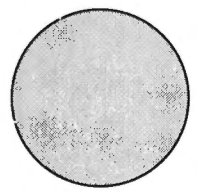


Figura D.5. Notación para un objeto entidad

Objetos de control (figura D.6). Cuando se esta distribuyendo el comportamiento y las responsabilidades entre los objetos, hay comportamientos que a menudo no residen ni en objetos interfase ni en objetos entidad. En otras palabras, el comportamiento no esta relacionado en la forma como el usuario interactua con el sistema, ni está relacionado con la forma como son manejados los datos en el sistema. Más bien su comportamiento esta relacionado con la administración de las interacciones de los objetos para soportar la funcionalidad del caso de uso. Los objetos de control sirven como "controladores de tráfico" ya que contienen la lógica de la aplicación o reglas del sistema para un evento y así administran o dirigen la interacción entre los objetos. Una regla general es que un objeto de control debería estar asociado con uno y solamente un actor.

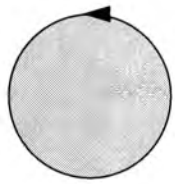
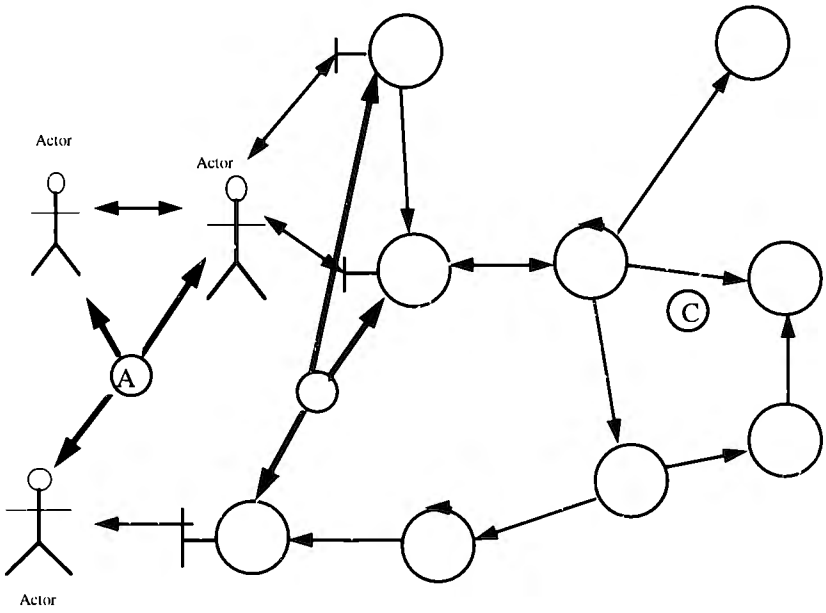


Figura D.6. Notación para un objeto de control

El **diagrama para obtener el modelo ideal de los objetos** (figura D.7) nos muestra los objetos y sus interacciones. Este diagrama incluye símbolos para representar actores, interfaces, controles y objetos entidad, y flechas que representan mensajes o comunicación entre los objetos.



D.7. Diagrama para obtener el modelo ideal de los objetos

En [Jacobson, 1992], se encuentran los fundamentos y notaciones sobre casos de uso. En [Texel, Williams, 1997] se tratan los casos de uso combinados con OMT y UML.

APÉNDICE E. DIAGRAMA DE CONECTIVIDAD DE LAS LOCALIZACIONES

Modelado de la red. Es una técnica de diagramación usada para documentar la forma de un negocio o sistema de información en términos de sus localidades.

Diagrama de conectividad de las localizaciones. Es un diagrama para modelar la red lógica que muestra la forma de un sistema en términos de la ubicación de sus usuarios, procesos, datos e interfaces y, la interconexión entre estas localizaciones. Los símbolos que pueden ser utilizados para este diagrama se muestran en la figura E.1.

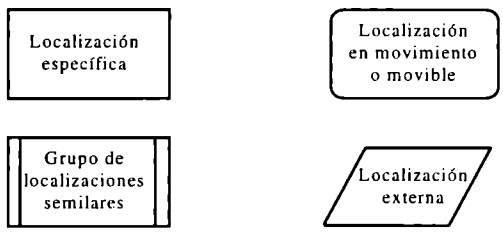


Figura E.1. Símbolos para un diagrama de conectividad de las localizaciones

Localización. es un lugar desde el cual los usuarios interactuan con el sistema o aplicación. También es cualquier lugar donde las funciones pueden ser realizadas o ejecutado el trabajo.

La localización puede significar diferentes cosas para diferentes personas. Los el administradores de un negocio y usuarios normalmente identifican **localizaciones lógicas,** donde las personas ejecutan el trabajo o funciones. Para técnicos en la información se identifican **localizaciones físicas**, donde la tecnología de computadoras y redes está localizada. Las localizaciones lógicas pueden estar:

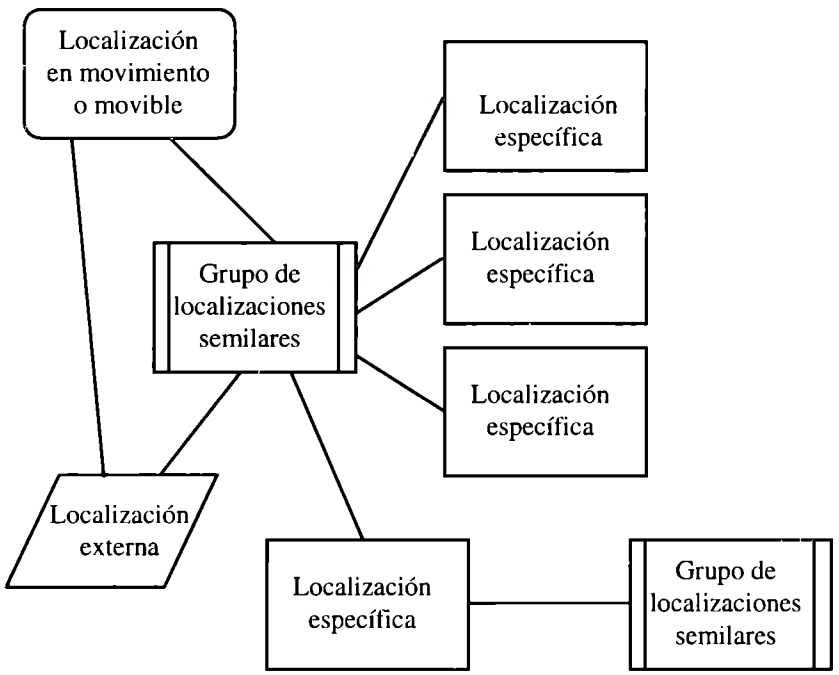
- Esparcidas a través del negocio para cualquier sistema de información dado
- En movimiento (por ejemplo un representante de ventas en viaje)

- Externas a la empresa para la cual se está construyendo el sistema. Por ejemplo los clientes pueden ser usuarios del sistema vía teléfono o Internet.

Por otro lado las localizaciones lógicas pueden representar:

- Grupos de localizaciones similares. Puede ser que se necesite mostrar un grupo de individuos que realizan la misma labor en la misma localización como un grupo de localizaciones similares.
- Organizaciones y agentes externos pero que interactuan o usan el sistema, posiblemente usuarios directos

La figura E.2 muestra un ejemplo de un **diagrama de conectividad de las localizaciones.**



E.2. Ejemplo de un diagrama de conectividad de las localizaciones

Diagrama de descomposición de las localizaciones

Con la descomposición se logra romper un sistema en susbsistemas. Cada nivel de abstracción revela tantos detalles como se deseen acerca de todo el sistema o un subconjunto de ese sistema. De esta manera el analista particiona el sistema en subconjuntos lógicos de localizaciones para involucrar comunicación, análisis y diseño. La figura E.3 es un ejemplo un diagrama de descomposición de las localizaciones.

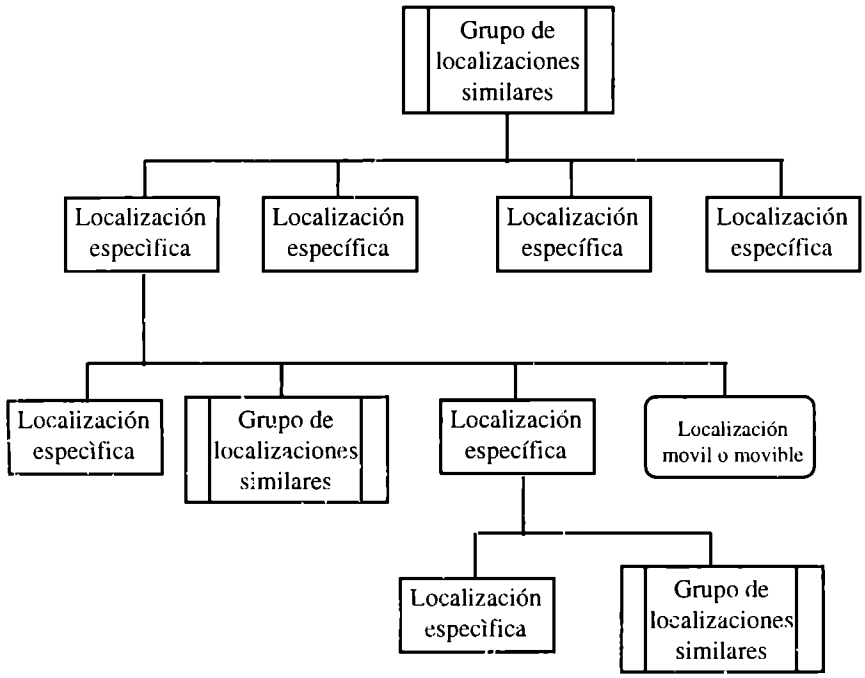


Figura E.3. Diagrama de descomposición de las localizaciones