Tecnologías Orientadas por Objetos

Juan ▪ Guillermo ▪ Lalinde ▪ P.

Las tecnologías orientadas por objetos han permeado todos los aspectos del desarrollo del software. Actualmente se tienen lenguajes de programación, metodologías de análisis y diseño, bases de datos, etc. En el presente artículo se hace una descripción de dichas tecnologías y su relación con el proceso del desarrollo del software.

INTRODUCCIÓN

Un término de moda en el mundo de la informática es la orientación por objetos. Se

---

Juan Guillermo Lalinde Pulido. Ingeniero de Sistemas, Universidad EAFIT. Matemático, Universidad Nacional de Colombia. Candidato a Doctor en Telecomunicaciones, Universidad Politécnica de Valencia. Profesor del Departamento de Informática y Sistemas.

habla de lenguajes de programación, metodologías, herramientas, componentes, etc. En estas circunstancias, es natural preguntarse por las posibilidades que brindan las tecnologías orientadas por objetos (TOO), pues su objetivo final es el desarrollo de software y éstas son sólo un medio. El primer paso a seguir es realizar una reflexión sobre su desarrollo.

El primer elemento de reflexión surge de la labor misma del desarrollo del software; no se habla de producción porque el software no se produce en el sentido clásico, sino que se desarrolla. No se desgasta por el uso, pero se deteriora a medida que se introducen modificaciones. Aunque normalmente asocian las modificaciones con la corrección de errores, no se debe perder de vista que el software se desarrolla para resolver un problema específico. Si las condiciones del problema cambian hay que modificar el software para que se adapte a las nuevas condiciones. Un ejemplo simple es una contabilidad sistematizada. Cada vez que se introducen modificaciones en la legislación hay que adaptar la contabilidad.

El primer elemento de reflexión surge de la labor misma del desarrollo del software; no se habla de producción porque el software no se produce en el sentido clásico, sino que se desarrolla. No se desgasta por el uso, pero se deteriora a medida que se introducen modificaciones. Aunque normalmente asocian las modificaciones con la corrección de errores, no se debe perder de vista que el software se desarrolla para resolver un problema específico. Si las condiciones del problema cambian hay que modificar el software para que se adapte a las nuevas condiciones.

Un segundo elemento, relacionado con el anterior, es la necesidad de que el software pueda ser adaptado fácilmente. Cuando una organización decide invertir en software, está esperando que éste se convierta en un factor de éxito que la ayude a ser competitiva en el mercado. Al desarrollar su actividad en un mundo cambiante debe ser flexible para adaptarse a un entorno cambiante. El problema del cambio de siglo es un fiel reflejo de esta situación. El software se hizo utilizando una representación de dos dígitos para el año (i.e. 1997 se representa por 97). El problema surge con el año 2000, pues de acuerdo con esta representación sería 00 y al comparar dos años, resulta que es imposible distinguir 1900 de 2000. Se estima que la inversión que hay que realizar a nivel mundial para la actualización del software existente para que maneje adecuadamente las fechas es del orden de US$66.000.000.000. Esta situación tiene como agravante el hecho de que dicha inversión no produce ningún beneficio adicional. Hay que hacerla para poder seguir funcionando.

Un tercer elemento, y no por eso el menos importante, es que el desarrollo del software se debe realizar con recursos limitados. Cuando se analiza la crisis del software se resume básicamente en la sofisticación del hardware que hace que sea más difícil explotar su potencial real, la gran demanda de nuevos programas y la dificultad para mantener los programas existentes por mal diseño y uso inadecuado de recursos. La ingeniería del software surge como respuesta a la crisis del software, y es por esta razón que debe

facilitar el proceso del desarrollo del software garantizando un buen diseño y un uso adecuado de los recursos para poder responder adecuadamente a la demanda. La adopción de una tecnología particular debe ir guiada por estos conceptos.

El desarrollo del software se puede dividir en tres grandes etapas: definición, implementación y mantenimiento. En el enfoque clásico, el punto de partida son unos requerimientos y el resultado es un sistema automático que se ajuste a ellos. La praxis ha demostrado que esto no es cierto. Hay dos supuestos implícitos en el enfoque clásico, conocido también como el modelo de la cascada, que no son ciertos. El primero tiene que ver con el punto de partida. Lo que ocurre en la práctica es que en la mayoría de los casos no se conocen completamente los requerimientos del sistema. Se tiene una idea general y, a medida que se va avanzando en el desarrollo, se ajustan los requerimientos. El segundo está relacionado con la invariabilidad de los requerimientos. Aún suponiendo que se conocieran completamente los requerimientos, éstos cambian con el tiempo y el software se debe adaptar a los nuevos requerimientos.

La ingeniería del software surge como respuesta a la crisis del software, y es por esta razón que debe facilitar el proceso del desarrollo del software garantizando un buen diseño y un uso adecuado de los recursos para poder responder adecuadamente a la demanda.

Partiendo de lo anterior, se puede afirmar que la adopción de tecnologías nuevas se justifica en la medida que ayuden a solucionar algunos de estos problemas. La adopción de las TOO sólo se justifica si permiten mejorar la calidad del software realizado y/o reducir los costos del desarrollo.

1. RESEÑA HISTÓRICA

Los conceptos fundamentales de las TOO surgieron a partir de un lenguaje de programación desarrollado en la década de los 60: Simula 67. En él se introdujeron por primera vez los conceptos básicos de esta tecnología. Aunque es un lenguaje poco conocido fuera de los círculos dedicados a la simulación, no se puede menospreciar su impacto en el desarrollo de las TOO. Es significativo el hecho de que estas ideas surgieran en ambientes de simulación, ya que todo sistema de información puede ser mirado en última instancia como una simulación de un proceso determinado. El carácter del proceso puede ser concreto o abstracto, pero siempre se puede mirar como una simulación.

El lenguaje que facilitó el desarrollo y difusión de esta tecnología en su fase inicial fue el ***Smalltalk***. Este fue desarrollado en los laboratorios de la Xerox y su finalidad básica era facilitar la interacción hombre - máquina. Uno de los resultados de los trabajos realizados en ese laboratorio es la interfaz gráfica de usuario (GUI) que involucra el uso del ratón.

Una vez conocido el potencial de los objetos, se buscó incorporar todas estas facilidades al lenguaje de moda: C. Surgieron dos variaciones. El ***Objective C*** que adiciona a C muchas características del ***Smalltalk,*** y el C++ desarrollado por Bjarne Stroustrup en los

laboratorios de AT&T Bell. El C++ era originalmente un preprocesador que generaba código en C. Actualmente el más difundido es el C++, aunque hay aplicaciones muy interesantes del ***Objective C*** tales como el ***NextStep.*** La más reciente adición a la familia de los lenguajes orientados por objetos es el Java, desarrollado por Sun Microsystems y que ha tenido gran impacto en el mundo de Internet.

En el dominio de los lenguajes orientados a objetos hay otros actores de importancia, entre los cuales se destaca el Eiffel, desarrollado por Bertrand Meyer. Este lenguaje tiene como característica importante que implementa los conceptos de programación por aserciones.

Vale la pena resaltar el hecho de que los creadores de ***Smalltalk, C++*** y Eiffel estuvieron relacionados directa o indirectamente con el Simula 67, formando parte en algunos casos del comité encargado de fijar el estándar para ese lenguaje.

Antes de terminar con la evolución de los lenguajes, vale la pena mencionar que ya está definido un estándar que involucra objetos en el lenguaje más utilizado a nivel mundial: COBOL. Dicho estándar es conocido como ***Object Oriented Cobol*** (OOCobol).

Aunque originalmente se trató de utilizar las metodologías de análisis y diseño tradicionales, poco a poco se fueron desarrollando diversas metodologías pensadas específicamente para la TOO. Algunas de ellas son Booch, OMT (Rubaugh et al), Coad/Yourdon, etc. Alrededor de las diferentes metodologías han surgido compañías que soportan el uso de la metodología mediante consultorías y herramientas CASE.

El siguiente paso en la evolución fue la aparición de bases de datos basadas en este nuevo paradigma: ***Object Database Management System*** (ODBMS). Las primeras aparecieron hacia 1986, y entre ellas se pueden destacar ***GemStone, Poet, Versant,*** etc. El objetivo principal es poder almacenar estructuras de datos muy complejas (objetos) y proporcionar un acceso rápido a ellas. Hacia 1993 surge el primer lenguaje de consulta estándar: OQL. Fue definido por el OMG ***(Object Management Group)*** y específicamente por el ODMG ***(Object Data Management Group).*** Es casual el hecho de que el estándar aparece siete años después de que surge el primer ODBMS, exactamente el mismo período de tiempo transcurrido entre la aparición del modelo relacional y el SQL. Los ODBMS han sido utilizados con éxito en muchas aplicaciones, aunque muchos sistemas que utilizan TOO confían la persistencia de los objetos a bases de datos relacionales (RDMBS). Vale la pena notar que empresas con gran trayectoria en el mundo relacional han comenzado a vincularse como observadores al ODMG.

Los conceptos de TOO tienen, como veremos más adelante, las características propias de sistemas distribuidos y de ahí surge la necesidad de distribuir las aplicaciones en varias máquinas y encontrar mecanismos que permitan que interactúen entre sí. Las ideas de la TOO van más allá de distribuir un sistema en varios equipos, y nos permiten pensar en definir la forma de integrar sistemas.

Los ODBMS han sido utilizados con éxito en muchas aplicaciones, aunque muchos sistemas que utilizan TOO confían la persistencia de los objetos a bases de datos relacionales (RDMBS). Vale la pena notar que empresas con gran trayectoria en el mundo relacional han comenzado a vincularse como observadores al ODMG.

El OMG ha cumplido un papel muy relevante como organismo normalizador que busca entre sus objetivos el poder definir la forma como deben interactuar los objetos entre sí en un ambiente distribuido. Actualmente está conformado por más de 800 socios a nivel mundial, entre los cuales se encuentra la Universidad EAFIT. El resultado de este trabajo es la especificación CORBA ***(Common Object Requester Broker)***. El OMG no es la única que ha trabajado en este sentido, y aparecen también otras alternativas como DCOM de Microsoft.

El crecimiento y la evolución que ha tenido la TOO ha sido tan grande que ya podemos hablar de normas que exigen que el sistema sea orientado a objetos. Un ejemplo de esto es la norma de la familia M.30 de la ITU-T (anteriormente CCITT), más conocida como TMN ***(Telecommunication Management Network)***. En ella se especifica como debe ser un sistema que integre la gestión de las redes de telecomunicación. Todo el sistema está formulado en términos de objetos.

En este momento éste es el panorama general del mundo de los objetos. Los sistemas operativos (SO) están siendo portados a objetos. Se crean nuevos SO como ***Taligent***. Las TOO son una realidad que está presente en nuestro trabajo diario. Podemos encontrar los rudimentos de las TOO en la definición de dispositivos en los SO o en los DLL ***(Dynamic Link Library)*** que son utilizados a diario por todo usuario que aplique Windows.

2. CONCEPTOS BÁSICOS

Una de las principales características de las TOO es que dan gran importancia a la semántica. Se puede decir que se basan en dos pilares fundamentales: los tipos de datos abstractos y la teoría de la clasificación. Los tipos de datos abstractos se caracterizan por las operaciones que pueden realizar y por la semántica. La representación interna no es relevante. Esta característica los hace ideales para formalizar el desarrollo del software. Este hecho es muy importante, sobre todo porque permite definir una filosofía de desarrollo independiente de las herramientas que se utilicen y que busca mejorar la calidad del software al reflejar en el código mismo de los programas la semántica del problema original.

La teoría de la clasificación se refiere a la forma como los seres humanos aprehenden el mundo que los rodea. Trata de explicar cómo se crean los modelos mentales que el hombre utiliza para comprender el mundo que lo rodea. Es natural que esta teoría sirva de soporte a una filosofía de trabajo que surge del mundo de la simulación, pues antes de poder realizar una simulación hay que crear un modelo mental que represente el mundo que se va a simular.

La filosofía de objetos se basa en unos conceptos claves: Objetos, métodos y herencia. La idea central es dividir el mundo en entidades independientes, llamadas objetos, que pueden realizar operaciones. La interacción entre objetos se da mediante la solicitud de servicios. Un objeto solicita a otro un servicio mediante el envío de un mensaje. El objeto que solicita el servicio se conoce como cliente, mientras que el que lo presta se llama servidor. Un objeto puede asumir ambos roles en un sistema, siendo cliente para algunos objetos y servidor para otros. Un objeto que actúa como servidor presenta a los demás objetos del sistema una interfaz que especifica claramente los servicios que presta.

Cada objeto que forma parte de un sistema debe tener un tipo específico. El objeto es, por lo tanto, una instancia de una clase o tipo de dato definido en el sistema. El mecanismo de la herencia permite especificar los tipos al implementar la abstracción mediante la relación "es un" o "es una". La herencia es la contribución más importante de la filosofía de objetos a la definición de tipos y es un paso más allá de la definición de tipos de datos abstractos. De hecho, para decir que un sistema soporta la filosofía de objetos se requiere que implemente el mecanismo de la herencia. Aquellos sistemas que permiten definir objetos y operaciones, pero que no soportan la herencia, son denominados "basados en objetos".

Una de las principales características de las TOO es que dan gran importancia a la semántica. Se puede decir que se basan en dos pilares fundamentales: los tipos de datos abstractos y la teoría de la clasificación. Los tipos de datos abstractos se caracterizan por las operaciones que pueden realizar y por la semántica. Este hecho es muy importante, sobre todo porque permite definir una filosofía de desarrollo independiente de las herramientas que se utilicen.

2.1 OBJETO Y CLASE

Es la reunión en una sola unidad de una estructura de datos y todas las operaciones que puede realizar. En un lenguaje tradicional existen tipos de datos básicos y para cada uno de ellos hay un conjunto de operaciones que son válidas. Un objeto es una definición de un tipo de dato y sus operaciones. En algunos trabajos se diferencia el concepto de clase y objeto. Se dice que una clase es la definición de un tipo de dato y sus operaciones y un objeto es una instancia de una clase. Es una relación muy similar a la que existe entre tipo de dato y variable. Esto es especialmente cierto en los lenguajes y las metodologías.

2.2 ATRIBUTO

Dado que un objeto es un conjunto de estructura de datos y operaciones válidas, a cada uno de estos componentes se les identifica con un nombre. Llamamos atributo a cada uno de los elementos de la estructura de datos definida por el objeto. Los atributos deben ser modificados únicamente por las operaciones que son válidas para el objeto, y su contenido refleja el estado del objeto en un momento dado.

2.3 MÉTODO

Es una operación válida sobre un objeto. Los métodos son los únicos elementos del sistema que requieren conocer los atributos

del objeto. Los métodos se ejecutan como respuesta a mensajes y son el punto de contacto de un objeto con el mundo exterior. Algunos autores, especialmente Coad, los llama servicios. Se dice que un método es primitivo si no puede ser llevado a cabo sin conocer los atributos del objeto. Todo objeto debe tener implementados los métodos primitivos. Adicionalmente, puede implementar métodos no primitivos.

2.4 MENSAJE

Es una solicitud que se hace a un objeto para que ejecute una operación válida (método). La forma de interactuar entre los objetos es mediante el envío de mensajes. Por esta razón todo modelo de objetos es en esencia un modelo distribuido. No importa donde está físicamente cada objeto. Si hay un mecanismo de comunicaciones que permita llevar el mensaje al objeto (i.e. Internet), el sistema debe funcionar.

2.5 HERENCIA

Con el fin de aprovechar las características comunes de los objetos se creó el concepto de herencia. La idea general es tan simple como poderosa: si se desea definir un objeto nuevo que es una implementación más específica de un objeto genérico que existe en el sistema, se puede decir que el objeto específico es hijo del objeto genérico y hereda todas sus características. Esto quiere decir que los métodos y atributos que tendrá dicho objeto son, por defecto, los mismos que el padre, y que sólo hay que preocuparse por definir métodos y atributos nuevos o redefinir los que son específicos para esa implementación en particular.

La herencia puede ser simple o compuesta. En la simple un objeto sólo puede tener un padre. En la compuesta un objeto puede tener varios padres y hereda las características de todos ellos.

2.6 JERARQUÍAS

Son relaciones de orden que se introducen en el conjunto de clases. Son básicamente de dos tipos: jerarquías de herencia, que se basan en la relación «es un(a)» y jerarquías de composición, que se basan en la relación «es parte de». Dos clases se relacionan en la jerarquía de herencia si una de ellas una especialización de la otra. Dos clases se relacionan en la jerarquía de composición si una de ellos es una característica de la otra.

Nótese que la definición de jerarquía se hace con relaciones de orden, lo que impide que se den ciclos en la misma. Dicho de otra manera, si se escoge una clase y se recorren todas sus especializaciones (atributos) recursivamente, no se puede llegar a la clase original. Es lo que formalmente se conoce como grafo dirigido acíclico.

2.7 POLIMORFISMOS

Dado que la comunicación entre objetos se realiza mediante el paso de mensajes, se puede enviar el mismo mensaje a varios objetos y cada uno de ellos puede responder de manera totalmente diferente. Para implementar los polimorfismos se recurre a la jerarquía de herencia y a dos hechos fundamentales:

- Donde pueda utilizar una instancia de una clase puedo utilizar cualquier instancia de sus especializaciones porque ellas cumplen con las mismas propiedades.

- Cada objeto mantiene su propia identidad, lo que garantiza que la respuesta a cualquier mensaje es la definida para ese objeto en particular.

2.8 ENCAPSULAMIENTO

Se refiere al hecho de que los atributos de un objeto están ocultos a los demás, y la única forma de tener acceso a ellos es a través del mecanismo de los mensajes. Llevado al terreno práctico este principio dice que si se desea tener plena libertad para modificar la estructura de datos utilizada para un objeto sin tener que modificar los objetos relacionados con él, se debe esconder esta estructura mediante la creación de una serie de operaciones que puedan ser utilizadas por los demás objetos.

2.9 ABSTRACCIÓN

Es la base de la herencia. Se refiere a diversos tipos de generalización. Cuando se mira un objeto se puede hacer desde diversos puntos de vista. Mientras más general sea, menos información se requiere, pero también hay menos operaciones válidas. La utilización de la abstracción permite definir las jerarquías de herencia y realizar un diseño por diferenciación, o sea, sólo definir en cada nivel de abstracción aquellas características que hacen ese objeto particular y distinto del nivel de abstracción superior.

2.10 PERSISTENCIA

Decimos que un objeto es persistente cuando sigue existiendo aún cuando la aplicación que lo creó termine su ejecución. La persistencia de los objetos se garantiza mediante las bases de datos.

3. MODELO DE REFERENCIA DEL OMG

El principal objetivo del OMG ha sido la definición de un marco de referencia arquitectónico para reducir la complejidad, disminuir los costos y acelerar la introducción de nuevas aplicaciones. La Arquitectura de Gestión de Objetos (***Object Management Architecture*** - OMA) es el resultado del trabajo realizado al interior del OMG y proporciona todos los elementos necesarios para la interoperabilidad entre objetos en un ambiente distribuido y soportando especificaciones detalladas de las interfaces. El OMG genera especificaciones que conducen a la industria hacia el desarrollo de componentes de software portables, reutilizables e interoperables basados en interfaces orientadas por objetos abiertas y estándares. La implementación es responsabilidad de los vendedores, usuarios finales y todos aquellos que estén desarrollando productos y proyectos para resolver un problema particular de cómputo o una aplicación comercial.

El OMG ha desarrollado una serie de lenguajes, interfaces y normas de protocolos para ambientes multivendedores, entre las cuales se destacan:

- Procesamiento de transacciones sobre sistemas distribuidos.

- Ejecución concurrente de diferentes objetos en el mismo o en diferentes sistemas en un entorno distribuido.
- Notificación de eventos, que han ocurrido en alguna parte en el sistema distribuido, a un objeto.
- Notificación de cambios ocurridos en las estructuras para asegurar que las referencias a los objetos utilicen las versiones adecuadas.
- Internacionalización.

Dado que la filosofía de objetos se fundamenta en elementos independientes que intercambian mensajes entre sí, se puede afirmar que un modelo de objetos es un modelo distribuido. Lo único que se requiere es que se defina claramente la forma de la interacción y se proporcione un medio adecuado para el intercambio de mensajes. Por esto el trabajo del OMG va encaminado a garantizar la interoperabilidad entre objetos que han sido desarrollados en ambientes diferentes.

El concepto de sistemas abiertos ha probado ser una buena estrategia para optar por una tecnología particular sin depender de uno o varios proveedores. La idea general parte de establecer un modelo de referencia que sea conocido y respetado por los productores y los usuarios. Cualquier sistema que se ajuste a lo recomendado por el modelo debe ser compatible con cualquier implementación específica del mismo. El modelo de referencia normalmente es definido al interior de un foro que involucra tanto a los proveedores como los usuarios y define aquellas características del modelo que sean estrictamente necesarias para lograr uniformidad en el mercado. El mundo de los objetos no ha sido ajeno a esta necesidad. El foro de discusión es el OMG, el cual fue fundado en 1989 por ocho empresas líderes y que actualmente cuenta con más de ochocientos afiliados.

Dado que la filosofía de objetos se fundamenta en elementos independientes que intercambian mensajes entre sí, se puede afirmar que un modelo de objetos es un modelo distribuido. Lo único que se requiere es que se defina claramente la forma de la interacción y se proporcione un medio adecuado para el intercambio de mensajes.

El modelo de referencia OMA del OMG establece una clasificación general de los objetos que son necesarios para la implementación de un sistema y describe cuál es la función de cada uno de ellos en un sistema cualquiera. Se pretende crear un marco de referencia para el desarrollo de software de manera que cada proveedor se pueda especializar en componentes y que éstos se puedan ensamblar entre sí para obtener el sistema deseado.

El modelo de referencia particiona el espacio del problema en componentes arquitectónicos prácticos de alto nivel que pueden ser enfrentados por los proponentes de la tecnología. De esta manera se obtiene una guía conceptual para ensamblar las tecnologías resultantes permitiendo simultáneamente la existencia de soluciones implementadas a partir de diferentes diseños.

El modelo de referencia identifica y caracteriza los componentes, interfaces y protocolos que componen la OMA, pero no los define en

detalle. A partir de estos elementos, la OMA puede ser vista como tres segmentos principales que constan de varios componentes críticos:

1. Orientados a la aplicación: La OMA caracteriza interfaces y Facilidades Comunes (Common Facilities) como componentes específicos de una solución que están ubicados muy cerca al usuario final.

2. Orientados a los sistemas: Los Gestores de Peticiones de Objetos ***(Object Request Brokers*** - ORB) y Servicios de Objetos ***(Object Services)*** se relacionan directamente con el "sistema" y con la infraestructura para computación distribuida y su gestión.

3. Orientados a mercados verticales: Las Interfaces de Dominio (Domain Interfaces) son específicas para un dominio o para una aplicación vertical. Si se combinan con interfaces heredadas de facilidades comunes y de servicios de objetos, se obtienen marcos de referencia críticos para aplicaciones en una gran variedad de industrias.

Todas las comunicaciones entre componentes, independiente del segmento que se mire, son manejadas por el gestor de peticiones de objetos. No es de extrañar que el ORB sea la base de la OMA y, por lo tanto, que la tecnología ORB se considere el aspecto más importante para el desarrollo e implantación de soluciones computacionales abiertas y distribuidas.

Es también muy importante notar que OMA asume que los servicios subyacentes proporcionados por el sistema operacional de la plataforma y por otros elementos de bajo nivel, tales como los recursos para computación distribuida, están disponibles y son utilizables por las implementaciones de OMA.

3.1 DEFINICIÓN DE COMPONENTES OMA

Gestor de Peticiones de Objetos: El ORB es el núcleo de comunicaciones en el estándar OMA. Proporciona una infraestructura que permite a los objetos interactuar entre sí, sin importar la plataforma ni las técnicas utilizadas para implementar los objetos. El ajustarse al estándar ORB garantiza portabilidad e interoperabilidad de objetos sobre una red de sistemas heterogéneos. El ORB estándar desarrollado por el OMG se conoce comercialmente con el nombre de CORBA™ ***(Common Object Request Broker Architecture)***.

Servicios de Objetos: Son componentes que estandarizan la gestión del ciclo de vida de los objetos. Se definen interfaces para crear objetos, controlar acceso, realizar seguimiento de objetos relocalizados y para controlar las relaciones entre los estilos de objetos (gestión de clases). También proporcionan los ambientes genéricos para que objetos individuales puedan ejecutar sus tareas. Proporcionan consistencia a las aplicaciones y ayudan a incrementar la productividad de los programadores.

Facilidades Comunes: Proporcionan un conjunto de funciones genéricas para las aplicaciones, las cuales pueden ser configuradas de acuerdo con los requerimientos

específicos de una aplicación en particular. Son todos aquellos recursos que están ubicados más cerca del usuario, tales como impresión, gestión de documentos, bases de datos y facilidades de correo electrónico. La estandarización conduce a la uniformidad en operaciones genéricas y permite ofrecer mejores opciones a los usuarios para configurar sus ambientes de trabajo.

Interfaces de Dominio: Representan áreas verticales que proporcionan funcionalidad específica para usuarios finales en dominios de aplicación particulares. Pueden combinar algunas facilidades comunes y servicios de objetos, pero se diseñan para realizar tareas particulares para usuarios dentro de cierto mercado vertical o industria.

Objetos de Aplicaciones: Aún cuando no es una actividad de estandarización actual en el OMG, las interfaces de aplicación son críticas al considerar una arquitectura de sistemas que es amplia y coherente. Representan las aplicaciones basadas en componentes que ejecutan tareas particulares para el usuario. Normalmente, una aplicación se construye a partir de un número muy grande de objetos básicos - algunos específicos a la aplicación, otros específicos al dominio, unos a partir de servicios de objetos y otros construidos a partir de las facilidades comunes. Estas aplicaciones obtienen grandes beneficios a partir de las fortalezas de un sistema robusto de desarrollo de objetos. Algunos beneficios bien conocidos de un buen desarrollo orientado a objetos de aplicaciones son: Mejor abstracción de los espacios del problema y de la solución, reutilización de componentes y una extensión más simple.

Llama la atención el hecho de que las bases de datos no se incluyan como un componente específico de OMA. Aparecen catalogadas como facilidades comunes. La razón de ser de este hecho radica en que se ha corregido un error que apareció en las bases de datos relacionales. Con la introducción del SQL se creó un lenguaje estándar para interactuar con las bases de datos que se podía utilizar bajo dos modalidades: como lenguaje independiente o integrado en un lenguaje de programación de tercera generación. Como lenguaje de programación, el SQL tiene una sintaxis y una semántica muy bien definida que es propia. En el proceso de integración con los lenguajes de tercera generación se decidió permitir la utilización de expresiones SQL, conservando su sintaxis y su semántica, en medio del programa en el lenguaje específico. Como consecuencia de esta decisión se producen dos problemas: La necesidad de que el programador domine dos lenguajes de programación diferentes, cada uno con su propia sintaxis y semántica, y la necesidad de mover información entre los dos dominios.

En particular, se tiene que la información sólo puede ser almacenada en la base de datos si está disponible en una variable de SQL. Si se desea leer información de la base de datos aparece la misma situación. Se debe leer en una variable de SQL y después se debe copiar en una variable del lenguaje de tercera generación para que pueda ser procesada. La principal consecuencia de esta problemática es que la base de datos cobra una importancia que realmente no tiene en el desarrollo del sistema de información, llegando a un punto tal que en el ciclo de vida del software se debe

dedicar un gran esfuerzo al diseño de la base de datos.

Para las bases de datos en el mundo de objetos se adopta el principio de Atkinson: "La persistencia es ortogonal al tipo". Atkinson se hizo una pregunta muy simple: Por qué no tener un solo sistema de tipos de datos que abarque al lenguaje de programación y a la base de datos, permitiendo al programador definir algunas instancias de dichos tipos como transientes y otras como persistentes?. Aunque el problema sigue existiendo al utilizar una base de datos relacional para almacenar objetos, el OMG no incluye las bases de datos como un componente específico en la OMA pues éstas son simplemente una solución tecnológica que se utiliza para permitir la persistencia de objetos. Bajo la filosofía de objetos las bases de datos no deben determinar la forma de representar la información, sino simplemente implementar la persistencia.

Bajo esta nueva filosofía, ¿qué ocurre con las consultas (queries)? Se incrementa su utilidad. La forma como se desarrolló el SQL obligó a que el concepto de consulta se relacione directamente con la información en la base de datos. Si se desea hacer una consulta sobre datos que no sean persistentes se tienen dos alternativas: se crea una tabla temporal, se coloca la información allí y luego se consulta; o se desarrollan procedimientos y funciones especiales en el lenguaje de tercera generación para realizar la consulta. Además se tiene el problema de que para realizar la consulta sólo se cuenta con los recursos del SQL, los cuales son muy limitados (i.e. no se pueden incluir criterios basados en funciones desarrolladas por el programador). Al aplicar el principio de Atkinson se le devuelve a las consultas el lugar que realmente les corresponde: son operaciones que se realizan sobre conjuntos de objetos sin importar si éstos son persistentes o no.

3.2 EL MODELO DE OBJETOS DEL OMG

Al igual que cualquier cuerpo de múltiples contribuyentes trabajando hombro a hombro para obtener un beneficio técnico común, al interior del OMG se sintió la necesidad de construir una terminología común para poder facilitar el entendimiento de sus miembros. Por tal motivo, el Modelo de Objetos (Object Model) del OMG define una semántica común para especificar las características externas visibles de los objetos de una manera estándar e independiente de la implementación. Esta semántica común caracteriza los objetos que existen en un sistema que se ajuste a las especificaciones del OMG.

El modelo de objetos del OMG se basa en unos pocos conceptos básicos:

- Objetos
- Operaciones
- Tipos y Subtipos

Un objeto puede modelar cualquier clase de entidad. Las operaciones se aplican a los objetos y permiten concluir cosas específicas acerca del objeto. Las operaciones asociadas con un objeto caracterizan colectivamente el comportamiento del objeto.

Los objetos se crean como instancias de tipos. Un tipo puede ser visto como una plantilla para la creación de objetos. Un tipo caracteriza el comportamiento de sus instancias al describir las operaciones que pueden ser aplicadas a esos objetos. Pueden existir relaciones entre tipos. Estas relaciones se conocen como supertipos/subtipos.

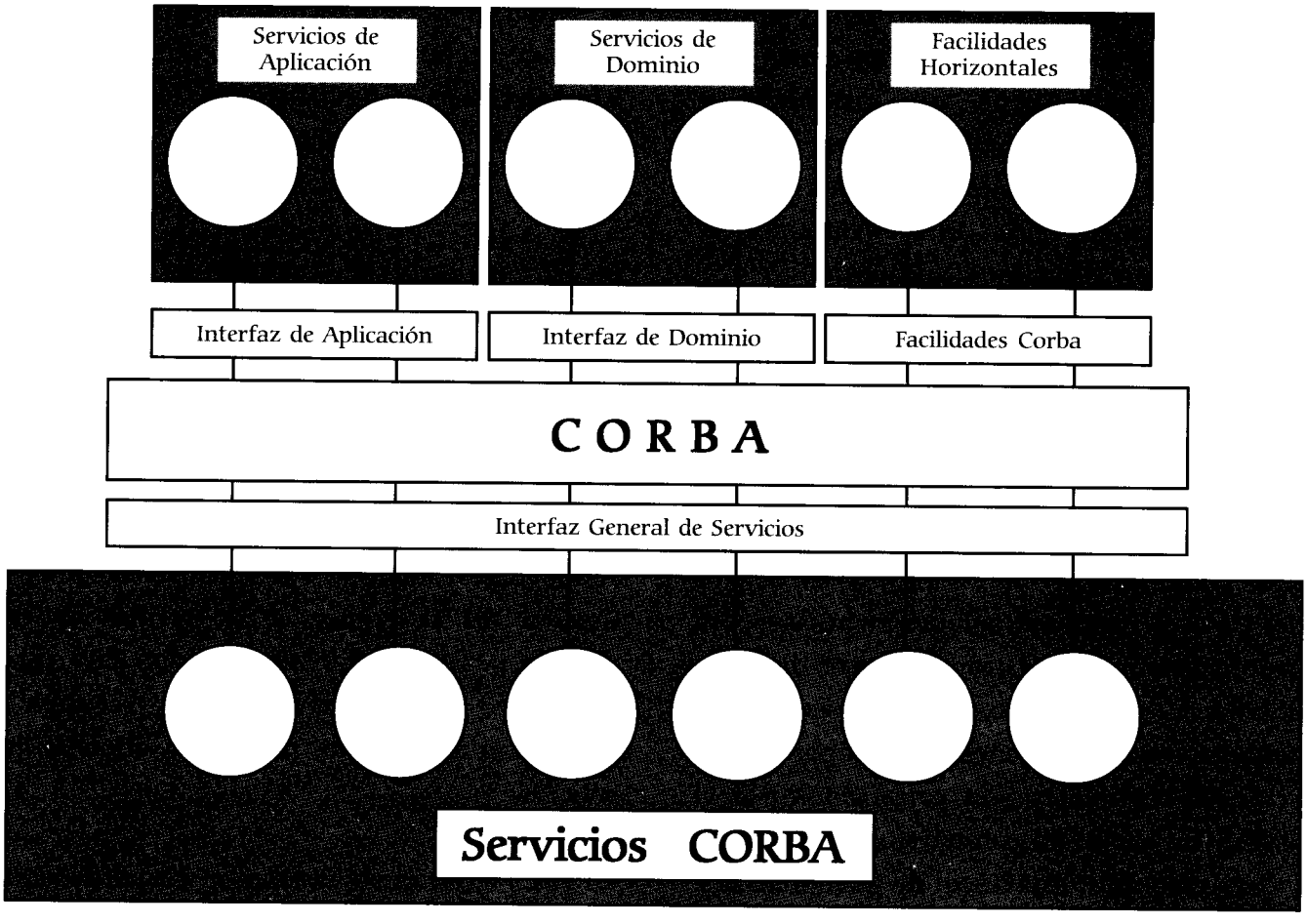
El modelo de objetos del OMG define un conjunto básico de requerimientos, basados en los conceptos anteriores, que debe ser soportado por cualquier sistema que cumpla con el estándar. Aunque el modelo central sirve como base común, el modelo de objetos del OMG permite extensiones al núcleo para permitir mayor integración entre diversos dominios tecnológicos. Estos conceptos se conocen como Componentes (Components) y Perfiles (Profiles) y son soportados por el OMA. Un perfil está compuesto por el núcleo y un conjunto específico de componentes. CORBA y ODMG93 son perfiles.

El modelo de objetos del OMG es una abstracción de alto nivel del concepto de objeto, pues debe permitir la interacción de objetos creados en lenguajes diferentes. Aunque a primera vista ésto no debería ser un problema, la dificultad surge porque la implementación del concepto en cada lenguaje de programación es particular. Los lenguajes más difundidos son el Smalltalk y el C++, e implementan el concepto de objeto de manera muy diferente.

En el ***Smalltalk*** el concepto de objeto se implementa como un descriptor que contiene toda la información necesaria para la gestión del mismo. Define, entre otras cosas, cual es su interfaz, su tipo y sus atributos. Todas las referencias al objeto son realmente apuntadores al descriptor. Cuando el objeto recibe algún mensaje, se analiza la información del descriptor y se verifica si es posible o no prestar el servicio. En caso afirmativo se pasa el control a la implementación del objeto para que atienda la solicitud.

En el C++ no existe descriptor. La posibilidad o no de prestar un servicio se determina de manera estática durante la compilación. Todas las referencias al objeto son apuntadores a la posición de memoria en la cual están almacenados los atributos y la tabla de funciones virtuales.

ARQUITECTURA DE OBJETOS GESTIONADOS



4. LENGUAJES DE PROGRAMACIÓN

Dentro de la secuencia en este texto, se sigue el orden histórico de los acontecimientos: Lenguajes, metodologías, bases de datos e integración de sistemas.

Los lenguajes más importantes han sido Smalltalk, C++ y Java. Se va a hablar un poco de cada uno de ellos. De todas formas es importante resaltar la existencia del Eiffel como una buena alternativa para desarrollo y la reciente aparición del OOCobol.

Dado que en los lenguajes orientados a objetos se diferencia entre clases y objetos de la misma manera que en los lenguajes tradicionales se hace entre tipos y variables, se utiliza la palabra clase como sinónimo de tipo y la palabra objeto como sinónimo de variable.

4.1 SMALLTALK

Fue desarrollado por Alan Kay en PARC y su influencia no puede ser sobrestimada. Basta con decir que el ***Smalltalk*** es el responsable de la aparición de las interfaces gráficas de usuario. En este lenguaje, la generalización y especialización se representa con superclases y subclases. Los atributos se denominan variables. ***Smalltalk*** viene con

muchas clases que pueden ser reutilizadas, pero hay dos que tienen mucha importancia: La clase «Objeto» que es la base de toda la jerarquía del ***Smalltalk*** y la clase «Modelo». La clase modelo es importante porque define un mecanismo de transmisión que es heredado por todas las hijas, y que proporciona mecanismos de dependencia. Normalmente se implementan como hijos de la clase modelo todos los elementos que pertenecen al dominio del problema, mientras que se utilizan otras clases para implementar los dominios de interfaz de usuario y manejo de tareas. Esto permite descomponer el problema en elementos ortogonales que pueden ser reutilizados independientemente.

Los métodos se agrupan en categorías funcionales llamadas «categorías de métodos» o «protocolos». Las más comunes a nivel de clase son inicialización, demostración de capacidades y creación. A nivel de instancia son acceso, adición de elementos, comparación, conexión, conversión, copia, exhibición, enumeración, manejo de errores, inicialización o liberación de conexiones, manejo de menús, impresión, métodos privados y pruebas.

Una de las características más importantes del ***Smalltalk*** es que la resolución de los tipos de datos se realiza durante la ejecución.

El ***Smalltalk*** utiliza el modelo MVC (model-viewer-controller) como infraestructura para el desarrollo de aplicaciones.

4.2 C++

Fue diseñado por Bjarne Stroustrup en los laboratorios de AT&T Bell. Es un lenguaje definido como un superconjunto del lenguaje C. Utiliza clases base y derivadas para implementar la especialización y generalización. Los atributos son implementados en forma de variables y los métodos se dan como funciones miembro.

En C++ se declara una clase y luego se definen los atributos y métodos. Para los atributos y métodos se define accesibilidad. Se manejan tres tipos:

- Privado: Sólo los métodos propios del objeto tienen acceso al atributo o método.

- Protegido: Tienen acceso al atributo o método cualquier método de la propia clase o de una clase hija.

- Público: Cualquier método de cualquier clase tiene acceso al método o atributo.

En el lenguaje se pueden redefinir casi todos los operadores, incluidos el operador de llamado de función. Originalmente era un lenguaje con chequeo estricto de tipos de datos, lo que lo colocaba en desventaja con respecto al ***Smalltalk.*** Este problema fue solucionado mediante la adición de templates, que son definiciones genéricas que se pueden utilizar para generar definiciones específicas para un tipo concreto, donde la única restricción que hay sobre el tipo es que soporte las operaciones requeridas por la definición.

Las adiciones más recientes al lenguaje incluyen manejo de excepciones y la capacidad de determinar en tiempo de ejecución el tipo de

un dato utilizando solamente el apuntador, llamada RTTI ***(Run Time Type Identification).***

4.3 JAVA

Fue desarrollado en Sun Microsystems. Su principal objetivo era obtener un lenguaje y sistema operativo portables para electrodomésticos. Fue desarrollado por un equipo conocido como ***FirstPerson,*** Inc. para ofrecer servicios de video bajo demanda con ***Time-Warner.*** El proyecto se le asignó **a Silicon Graphics,** así que Sun quedó con un producto sin mercado. Eventualmente, se descubrió la sinergía que se producía entre Java y el WWW y se configuró un mercado para Java.

Java es realmente una mezcla de un lenguaje de programación y un ambiente para la ejecución de programas. Puede ser utilizado para desarrollos autónomos o para desarrollar "applets" que puedan ser utilizados en el Web.

Sun colocó la definición de Java de dominio público, de manera que cualquier persona pueda desarrollar e implementar su propio ambiente Java. Hay implementaciones de la máquina virtual de Java para Macintosh, Windows 3.1, Windows 3.11, Windows 95, Windows NT, Solaris, OS/2 y Linux. El ambiente de desarrollo de Java originario se conoce como JDK, y consta de un conjunto de herramientas. Su distribución es gratuita y se puede obtener en http://www.javasoft.com. La versión para Windows de 16 bits (3.1 y 3.11) está en fase beta y la está desarrollando IBM.

El lenguaje de programación viene con un conjunto de librerías estándar que pueden ser utilizadas en el desarrollo. Se garantiza su existencia en todos los ambientes Java. Están organizados en forma de paquetes con el fin de evitar que haya conflicto de nombres.

Los creadores de Java lo definen como "un lenguaje simple, orientado por objetos, distribuido, interpretado, robusto, seguro, independiente de la arquitectura, portable, de alto rendimiento, con múltiples hilos y dinámico".

Simple: Se basó en C++. Eliminaron aquellas características del C++ que pueden ser peligrosas si no se manejan adecuadamente: Apuntadores, recarga de operadores, goto, etc. Automatiza la gestión de memoria y proporciona soporte a nivel del lenguaje para multithreading.

Orientado por Objetos: Como cualquier lenguaje moderno, es orientado por objetos. En cuanto al modelo de objetos, es más similar al de Smalltalk que al de C++. Exceptuando los tipos primitivos, todo en Java es un objeto. No soporta herencia múltiple ni clases paramétricas (templates de C++).

Las clases tienen métodos y atributos. Se puede definir su visibilidad: pública, privada o protegida. Soporta clases abstractas e interfaces. No hay variables globales. Hay variables de clase y de instancia. En los métodos se pueden utilizar variables locales.

Interpretado: Disminuye el tiempo d edicado al ciclo edición-compilación-encadenamiento-prueba. (En realidad lo que desaparece es el proceso de encadenamiento que se realiza en tiempo de ejecución). El interpretador da la

muchas clases que pueden ser reutilizadas, pero hay dos que tienen mucha importancia: La clase «Objeto» que es la base de toda la jerarquía del ***Smalltalk*** y la clase «Modelo». La clase modelo es importante porque define un mecanismo de transmisión que es heredado por todas las hijas, y que proporciona mecanismos de dependencia. Normalmente se implementan como hijos de la clase modelo todos los elementos que pertenecen al dominio del problema, mientras que se utilizan otras clases para implementar los dominios de interfaz de usuario y manejo de tareas. Esto permite descomponer el problema en elementos ortogonales que pueden ser reutilizados independientemente.

Los métodos se agrupan en categorías funcionales llamadas «categorías de métodos» o «protocolos». Las más comunes a nivel de clase son inicialización, demostración de capacidades y creación. A nivel de instancia son acceso, adición de elementos, comparación, conexión, conversión, copia, exhibición, enumeración, manejo de errores, inicialización o liberación de conexiones, manejo de menús, impresión, métodos privados y pruebas.

Una de las características más importantes del ***Smalltalk*** es que la resolución de los tipos de datos se realiza durante la ejecución.

El ***Smalltalk*** utiliza el modelo MVC (model-viewer-controller) como infraestructura para el desarrollo de aplicaciones.

4.2 C++

Fue diseñado por Bjarne Stroustrup en los laboratorios de AT&T Bell. Es un lenguaje definido como un superconjunto del lenguaje C. Utiliza clases base y derivadas para implementar la especialización y generalización. Los atributos son implementados en forma de variables y los métodos se dan como funciones miembro.

En C++ se declara una clase y luego se definen los atributos y métodos. Para los atributos y métodos se define accesibilidad. Se manejan tres tipos:

- Privado: Sólo los métodos propios del objeto tienen acceso al atributo o método.

- Protegido: Tienen acceso al atributo o método cualquier método de la propia clase o de una clase hija.

- Público: Cualquier método de cualquier clase tiene acceso al método o atributo.

En el lenguaje se pueden redefinir casi todos los operadores, incluidos el operador de llamado de función. Originalmente era un lenguaje con chequeo estricto de tipos de datos, lo que lo colocaba en desventaja con respecto al ***Smalltalk.*** Este problema fue solucionado mediante la adición de templates, que son definiciones genéricas que se pueden utilizar para generar definiciones específicas para un tipo concreto, donde la única restricción que hay sobre el tipo es que soporte las operaciones requeridas por la definición.

Las adiciones más recientes al lenguaje incluyen manejo de excepciones y la capacidad de determinar en tiempo de ejecución el tipo de

un dato utilizando solamente el apuntador, llamada RTTI ***(Run Time Type Identification).***

4.3 JAVA

Fue desarrollado en Sun Microsystems. Su principal objetivo era obtener un lenguaje y sistema operativo portables para electrodomésticos. Fue desarrollado por un equipo conocido como ***FirstPerson,*** Inc. para ofrecer servicios de video bajo demanda con ***Time-Warner.*** El proyecto se le asignó **a Silicon Graphics,** así que Sun quedó con un producto sin mercado. Eventualmente, se descubrió la sinergía que se producía entre Java y el WWW y se configuró un mercado para Java.

Java es realmente una mezcla de un lenguaje de programación y un ambiente para la ejecución de programas. Puede ser utilizado para desarrollos autónomos o para desarrollar "applets" que puedan ser utilizados en el Web.

Sun colocó la definición de Java de dominio público, de manera que cualquier persona pueda desarrollar e implementar su propio ambiente Java. Hay implementaciones de la máquina virtual de Java para Macintosh, Windows 3.1, Windows 3.11, Windows 95, Windows NT, Solaris, OS/2 y Linux. El ambiente de desarrollo de Java originario se conoce como JDK, y consta de un conjunto de herramientas. Su distribución es gratuita y se puede obtener en http://www.javasoft.com. La versión para Windows de 16 bits (3.1 y 3.11) está en fase beta y la está desarrollando IBM.

El lenguaje de programación viene con un conjunto de librerías estándar que pueden ser utilizadas en el desarrollo. Se garantiza su existencia en todos los ambientes Java. Están organizados en forma de paquetes con el fin de evitar que haya conflicto de nombres.

Los creadores de Java lo definen como "un lenguaje simple, orientado por objetos, distribuido, interpretado, robusto, seguro, independiente de la arquitectura, portable, de alto rendimiento, con múltiples hilos y dinámico".

Simple: Se basó en C++. Eliminaron aquellas características del C++ que pueden ser peligrosas si no se manejan adecuadamente: Apuntadores, recarga de operadores, goto, etc. Automatiza la gestión de memoria y proporciona soporte a nivel del lenguaje para multithreading.

Orientado por Objetos: Como cualquier lenguaje moderno, es orientado por objetos. En cuanto al modelo de objetos, es más similar al de Smalltalk que al de C++. Exceptuando los tipos primitivos, todo en Java es un objeto. No soporta herencia múltiple ni clases paramétricas (templates de C++).

Las clases tienen métodos y atributos. Se puede definir su visibilidad: pública, privada o protegida. Soporta clases abstractas e interfaces. No hay variables globales. Hay variables de clase y de instancia. En los métodos se pueden utilizar variables locales.

Interpretado: Disminuye el tiempo d edicado al ciclo edición-compilación-encadenamiento-prueba. (En realidad lo que desaparece es el proceso de encadenamiento que se realiza en tiempo de ejecución). El interpretador da la

posibilidad de ejecutar aplicaciones Java en cualquier ambiente en el cual exista.

Distribuido: En Java se pueden construir aplicaciones distribuidas mediante el desarrollo de clases para uso en aplicaciones de redes. Una aplicación puede acceder a un servidor remoto mediante el uso de la clase URL. Es importante resaltar que el poder construir aplicaciones distribuidas, no quiere decir que Java sea distribuido.

Robusto: Chequeo estricto de tipos. Los problemas de los tipos de datos se resuelven en tiempo de compilación y no se permite la conversión implícita de variables. Se simplifica el manejo de memoria mediante la eliminación de los apuntadores y la adición del mecanismo de recolección de basura (garbage collection).

Seguro: Al no permitir apuntadores, se tiene mayor control sobre el código existente en el ambiente Java. Dada la posibilidad de ser utilizado en Internet, se creó el concepto del verificador del código binario ***(byte-code).*** Éste verifica el código para garantizar que se ajusta a las normas y que es seguro de utilizar.

En particular, revisa:

- Gestión de apuntadores.
- Conversión ilegal de objetos.
- Overflow o Underflow en el stack de operadores.
- Parámetros pasados a las funciones.
- Cumplimiento de las reglas de visibilidad de atributos y operadores.
- Características - Independencia de Plataforma.

Dado que el compilador de Java genera ***byte-code*** que son interpretadas por la máquina virtual, la independencia de la plataforma se obtiene en la implementación de la máquina virtual en cada plataforma.

Portable: Los tipos de datos primitivos se definen independientemente de la máquina en que se utilice Java. Adicionalmente es portable pues el compilador está escrito en Java y la máquina virtual en C de acuerdo al estándar POSIX.

Alto Rendimiento: Aunque su rendimiento no es el que se puede obtener con C o C++, para la mayoría de las aplicaciones el rendimiento de la máquina virtual es muy bueno. En el diseño de la máquina virtual y el lenguaje, el rendimiento siempre fue uno de los factores más importantes.

Multithreads: Soporta múltiples hijos de ejecución sincronizados que se ejecutan concurrentemente. El soporte está implementado directamente en el lenguaje y en la máquina virtual y se da mediante clases.

Dinámico: Debido a que el Java es interpretado, los problemas de encadenamiento se resuelven en tiempo de ejecución. Esto es una gran ventaja pues el ambiente Java se puede extender con clases que están localizadas en servidores remotos.

5. METODOLOGÍAS

El análisis orientado a objetos (OOA) y el diseño orientado a objetos (OOD) son todavía un arte. Durante los primeros 20 años se hizo mucho énfasis en los lenguajes y la programación (OOP). Sólo durante la década

de los 90's, cuando los lenguajes alcanzaron algún nivel de madurez, la atención se desplazó hacia las técnicas de OOA/OOD. En este momento se están dando los primeros pasos a nivel teórico para formular metodologías de desarrollo.

A pesar de este hecho, existen diversas metodologías que se están utilizando activamente en el desarrollo de sistemas de información, y lo que realmente nos interesa es responder dos preguntas:

- ¿Pueden ser utilizadas en aplicaciones reales y producir sistemas viables?

- ¿Es su grado de madurez tal que puede ser definida y enseñada a los diseñadores de software?

Las evidencias disponibles hasta la fecha sugieren que la respuesta a dichas preguntas es afirmativa. Muchas compañías han reportado la aplicación exitosa de OOA/OOD para sacar productos comerciales. Se están comercializando herramientas de software que soportan las notaciones de estas metodologías. Esto sugiere que es factible hablar de OOA y OOD.

El análisis orientado a objetos (OOA) y el diseño orientado a objetos (OOD) son todavía un arte. Durante los primeros 20 años se hizo mucho énfasis en los lenguajes y la programación (OOP). Sólo durante la década de los 90's, cuando los lenguajes alcanzaron algún nivel de madurez, la atención se desplazó hacia las técnicas de OOA/OOD. En este momento se están dando los primeros pasos a nivel teórico para formular metodologías de desarrollo.

Las tendencias que se aprecian en dichas metodologías nos permiten reconocer un proceso de maduración acelerado. Las principales de ellas son:

- Aunque cada metodología utiliza sus propios términos y notación, los conceptos en los que se basan y el enfoque general son más similares que diferentes. Estas similitudes sugieren que el proceso gradual de consenso y convergencia dentro de la disciplina ha comenzado y va a comenzar a arrojar resultados sólidos.

- Virtualmente todas las metodologías sugieren una división muy tenue entre OOA y OOD, en lugar de la división rigurosa que aparece en los métodos estructurados. Este hecho es más importante de lo que parece a primera vista. En el enfoque estructurado, los límites entre análisis, diseño e implementación son muy definidos y la información generada en cada uno de ellos utiliza un «lenguaje» diferente. La aproximación OOA/OOD proporciona una aproximación unificada para la representación de la información. Al tener una representación de la información más unificada, es menos probable que la información se pierda o distorsione entre las fases del desarrollo.

- La utilización de OOA y OOD conduce a modelos que pueden ser fácilmente compartidos, entendidos y criticados por los usuarios finales. Esto incrementa la comunicación, lo cual puede conducir a mejores diseños.

- Las TOO han logrado involucrar sinergéticamente muchas tecnologías aliadas,

tales como bases de datos, inteligencia artificial, simulación, GUI, etc.

Se dará un breve vistazo a las metodologías más conocidas actualmente sin entrar a profundizar sobre ellas.

5.1 BOOCH

Divide las actividades de moderación entre vistas lógicas y físicas, y en cada una de ellas, semántica estática y dinámica. Utiliza diagramas familiares, tales como estado-transición y diagramas de tiempos, además de vistas relacionadas con el paradigma de objetos. ***Booch*** se refiere a estos modelos/diagramas/vistas colectivamente como objetos asociados.

En la vista lógica, se diferencia entre estructuras de clases y estructuras de objetos. La estructura de clases se representa con los siguientes diagramas: diagramas de clases, diagramas de especificación de clases, diagramas de categorías y diagramas estado - transición. La vista lógica de la estructura de objetos se representa con los siguientes diagramas: diagramas de objetos y diagramas de tiempo. En la vista física se utilizan diagramas de módulos, de subsistemas y de procesos.

El proceso se realiza de manera cíclica. Hace énfasis en el desarrollo incremental e iterativo de un sistema mediante el refinamiento de diferentes vistas consistentes del sistema como un todo. Este proceso de refinación consta de cuatro actividades principales:

- Identificar clases y objetos: Utilizando análisis de dominios se identifican objetos, relaciones y operaciones en el dominio de la aplicación tal y como son entendidos por un experto.

- Identificar la semántica entre éstas y objetos: Su objetivo es identificar el significado de las clases y los objetos. Se debe decidir sobre el comportamiento de una clase y sus interacciones con otras clases y objetos.

- Identificar las relaciones entre clases y objetos: Consiste en identificar las relaciones (herencia, composición, etc.) entre clases y sus características estáticas y dinámicas. Se concentra en la interfaz que presentará cada clase.

- Implementar las clases y objetos: Tiene que ver con los aspectos físicos del sistema. Las clases se asignan a módulos y/o subsistemas y se crea el modelo de procesos del sistema.

5.2 TÉCNICA DE MODELACIÓN DE OBJETOS (OMT)

Fue desarrollada por un grupo liderado por Rumbaugh. Divide el problema en tres submodelos: de objetos, dinámico y funcional. Esta división es muy similar a la que se utiliza en análisis y diseño estructurado, pero, a diferencia de ésta, se hace énfasis en el modelo de objetos y no en el funcional.

La estructura estática del sistema se representa en el modelo de objetos. Describe las clases y objetos que componen un sistema, sus relaciones y los atributos y métodos que definen cada clase/objeto.

El modelo dinámico captura las relaciones temporales presentes en el sistema. Esto incluye secuencias de eventos, control de flujo y concurrencia. OMT utiliza una combinación de escenario de aplicación, seguimiento de eventos y diagramas estado - transición anidados para capturar esta información.

La información de la transformación de los datos en el sistema se presenta en el modelo funcional. Se utilizan diagramas de flujo de datos anidados (DFD) para capturar este aspecto del sistema.

El proceso se divide en tres pasos: Análisis, diseño del sistema y diseño de objetos. Aunque cada paso tiene su propio objetivo y sus salidas, Rumbaugh hace énfasis en que el proceso es iterativo y que las fronteras entre los pasos deben ser muy transparentes.

La fase de análisis crea un modelo local del sistema. Los pasos que se siguen son:

- Escribir el problema.
- Construir el modelo de objetos del sistema.
- Crear el modelo dinámico.
- Crear el modelo funcional.
- Iterar cada modelo tantas veces como sea necesario para verificar su corrección y nivel de detalle.

El diseño del sistema es utilizado para organizar y definir la arquitectura del sistema. Las actividades durante esta fase incluyen:

- Identificar los subsistemas necesarios.
- Comprender los problemas de concurrencia.
- Asignar los subsistemas a procesadores y tareas.
- Seleccionar el mecanismo a utilizar para el almacenamiento de datos.
- Determinar los mecanismos de control de acceso a los recursos globales.
- Identificar condiciones de frontera.
- Definir criterios que guiarán las decisiones de diseño.

El diseño de objetos sirve como base para la implementación. Comienza con el modelo de análisis y adiciona los detalles necesarios para la implementación. Los pasos involucrados incluyen:

- Identificar y diseñar los algoritmos requeridos.
- Optimizar las rutas de acceso a los datos.
- Refinar la estructura de clases para incrementar el uso de la herencia.
- Decidir la estrategia de implementación para asociaciones.
- Decidir la estructura de implementación para los atributos de los objetos.
- Dividir las clases en módulos para el proceso de implementación

6. OBJETOS DISTRIBUIDOS

En estos momentos, cuando se habla permanentemente de Internet, Intranet y Extranet, los objetos distribuidos cobran mayor importancia. La computación por objetos distribuidos (DOC) es la aplicación de la tecnología de objetos a la computación distribuida. DOC, sin embargo, es más significante y compleja que lo que puede implicar la asociación con la programación orientada a objetos. DOC es una nueva arquitectura. Una nueva forma de computación. Va más allá de la simple partición de

base de datos y aplicaciones que constituye la interpretación típica de la arquitectura cliente - servidor. Permite un estilo de computación en el cual la red es el computador.

Uno de los factores más importantes para DOC es la estandarización y el OMG es la entidad que ha liderado esta actividad, produciendo el CORBA (Common Object Request Broker Architecture).

6.1 ORB (Object Request Broker)

Un ORB proporciona acceso transparente a los objetos. La interfaz de un objeto ORB está claramente separada de su implementación. Las interfaces se definen en un lenguaje declarativo (IDL) y el acceso a los objetos ORB está restringido a esta descripción externa. Los detalles acerca del lugar de existencia de los objetos y como son implementados son manejados por el ORB, manteniendo la aplicación cliente completamente aislada de futuros cambios de implementación.

Los programas clientes acceden a los objetos a través del ORB, el cual localiza los objetos y les envía los mensajes. Los clientes nunca tienen los objetos en su espacio de direccionamiento. En vez de direcciones, el cliente utiliza referencias a objetos, que pueden ser asociadas con apuntadores remotos. El ORB no proporciona ningún mecanismo de persistencia o manejo de datos.

Las referencias a objetos son creadas por medio de un adaptador de objetos (OA). Un cliente obtiene las referencias de objetos ya existentes consultando un servicio de nombres o como resultado de solicitudes a otras referencias de objetos.

Cuando un programa cliente termina de utilizar un objeto ORB, libera la referencia del objeto. Esto libera todos los recursos asociados con el acceso del cliente al objeto. El objeto puede continuar existiendo en el servidor.

6.2 CORBA

CORBA (Common Object Request Broker Architecture) es un ORB definido por el OMG. Como tal, soporta las características de los ORB, siendo sus características más sobresalientes las relacionadas con las propiedades de las interfaces.

En CORBA las interfaces se pueden relacionar unas con otras por medio de la herencia, habilitando polimorfismos. Esto hace posible extender aplicaciones. Las nuevas interfaces se pueden adicionar derivándolas de las existentes. Las operaciones de la interfaz base pueden ser utilizadas por las aplicaciones sobre los objetos nuevos adicionados al sistema. Esto se puede hacer aún con lenguajes no orientados a objetos, porque el modelo de objetos es parte del ORB mismo. También son permitidas otro tipo de relaciones.

El CORBA se basa en la arquitectura de manejo de objetos (OMA) del OMG, la cual describe un modelo de objetos en el cual cada interfaz puede tener asociada más de una implementación. Su principal objetivo es garantizar interoperabilidad entre los productos de hardware y software que proliferan en el mercado. CORBA permite a las

aplicaciones comunicarse una con otr a sin importar donde están localizadas o quien las ha creado. CORBA 1.1 fue introducido en 1991 por el OMG y contiene básicamente el Lenguaje de Definición de Interfaces ***(Interface Definition Language*** - IDL) y la Interfaz de Programación de Aplicaciones ***(Application Programming Interface*** - API) que permite la interacción de objetos cliente/servidor al interior de una implementación específica de un ORB. CORBA 2.0 fue adoptado en diciembre de 1994 y permite verdadera interoperabilidad al definir la forma como deben interactuar los ORB desarrollados por diferentes vendedores.

El ORB es el ***middleware*** que establece relaciones cliente/servidor entre objetos. Al usar un ORB, un cliente puede invocar un método en un servidor sin tener ningún conocimiento sobre éste. El ORB debe interceptar la llamada y es el responsable de encontrar un objeto que pueda implementar la solicitud, pasarle los parámetros, invocar su método y retornar los resultados. El cliente no necesita saber donde está localizado el objeto, en que lenguaje fue programado, en que sistema operativo se está ejecutando, ni ningún otro detalle del sistema que no sea parte de la interfaz del objeto. Al cumplir esta función, el ORB proporciona interoperabilidad entre aplicaciones en diferentes máquinas en ambientes distribuidos heterogéneos e interconecta suavemente múltiples sistemas de objetos.

El CORBA se basa en la arquitectura de manejo de objetos (OMA) del OMG, la cual describe un modelo de objetos en el cual cada interfaz puede tener asociada más de una implementación. Su principal objetivo es garantizar interoperabilidad entre los productos de hardware y software que proliferan en el mercado. CORBA permite a las aplicaciones comunicarse una con otra sin importar donde están localizadas o quien las ha creado.

En las aplicaciones típicas cliente/servidor, los desarrolladores usan su propio diseño o un estándar reconocido para definir el protocolo que se va a utilizar entre los dispositivos. La definición del protocolo depende del lenguaje de implementación, del transporte en la red y de muchos otros factores. Con un ORB el protocolo se define a través de las interfaces de la aplicación mediante una especificación independiente del lenguaje, el IDL. Además, el ORB permite al programador escoger el sistema operativo, el ambiente de ejecución y aún el lenguaje de programación más apropiados para cada componente en un sistema en construcción.

Otra ventaja adicional que proporciona un ORB es la integración de componentes existentes. En una solución basada en ORB, los desarrolladores simplemente modelan el componente ya existente utilizando el mismo IDL que utilizan para crear nuevos objetos, y luego escribe código que convierta entre el bus estándar y las interfaces originales.

En el mercado ya hay disponibles implementaciones completas de CORBA 2.0, tales como Orbix. Además se cuenta con el compromiso de las compañías que conforman el OMG. En particular se tiene que Hewllet-Packard incluye en su entorno de computación distribuida a CORBA además de DCE y Motif de OSF como interfaz de usuario, el DOE ***(Distributed Objects Everywhere)*** de SunSoft se basa en CORBA,

la OSF ha adoptado CORBA en su especificación del DME (Distributed Management Environment).

6.3 LENGUAJE DE DEFINICIÓN DE INTERFACES (INTERFACE DEFINITION LANGUAJE - IDL)

IDL no es un lenguaje de programación completo. Es un lenguaje para definir interfaces sin determinar cómo se implementan éstas. Proporciona las construcciones necesarias para definir tipos nuevos (incluidos tipos complejos) y las interfaces. Las interfaces se dan en términos de las operaciones que soportan y se permite la definición de relación de herencia entre interfaces. No se permiten instrucciones ejecutables, tales como asignaciones, ciclos, etc.

El IDL se define para proporcionar un lenguaje de definición de interfaces que sea independiente del lenguaje de programación en el cual va a implementar. Las interfaces se definen en el IDL y luego se convierten automáticamente en las interfaces correspondientes para el lenguaje particular. La interfaz que se define en IDL puede contener atributos y métodos. Los atributos pueden ser de sólo lectura o de lectura y escritura. Los métodos se definen especificando el tipo de dato retornado, el nombre, los parámetros, las excepciones, la semántica y el contexto. La semántica puede ser de mejor esfuerzo ***(«best effort»)*** o máximo una vez ***(«at most once»)***. En la primera no hay garantía de que se ejecute el método. En la segunda se garantiza que el método se ejecuta o se genera una excepción.

Vale la pena mencionar que el definir atributos en la interfaz da la sensación de que se está violando el principio de encapsulamiento. Esto es falso pues, como se verá en el ejemplo, lo que se define realmente son las funciones de acceso a atributos, tal y como las plantean Yourdon y Coad en su clasificación de los servicios prestados por un objeto. La siguiente es una implementación en IDL de una cuenta y su traducción a C++ (el tipo de dato float sólo se utiliza como ejemplo. En una aplicación comercial no se debe utilizar esta representación pues hay errores por efecto de los redondeos en los cálculos):

```
// IDL
interface cuenta {
  readonly attribute float balance;
  void HacerDeposito(in float f);
  void HacerRetiro (in float f);
};

// C++
class cuenta {
public:
  virtual float balance();
  virtual void HacerDeposito(float f);
  virtual void HacerRetiro(float f);
};
```

También se puede hacer uso de la herencia. Para la creación de una cuenta corriente que herede las características de la cuenta se especifica la interfaz como heredera de cuenta:

```
// IDL
interface cuentaCorriente : cuenta {
  readonly attribute float limiteSobregiro;
};
```

```
// C++
class cuentaCorriente : public virtual cuenta {
public:
  virtual float limiteSobregiro();
};
```

La forma de trabajo con CORBA consiste en especificar la interfaz con IDL y luego generar automáticamente la interfaz del lenguaje de programación que se usará para la implementación. Aunque en principio es posible el definir la interfaz con C++ (u otro lenguaje), la generalidad que maneja IDL no permite representar ciertas construcciones particulares (i.e. funciones con número variable de parámetros) lo que dificulta esta situación.

6.4 DCOM

El modelo de componentes de objeto distribuido (COM) es una propuesta de Microsoft para obtener la interacción entre componentes. Es simplemente una extensión de bajo nivel de COM. Es tecnología ActiveX. Microsoft está otorgando licencias de la tecnología DCOM a otras compañías de software para implementar en otros sistemas operativos. Software AG, tiene corriendo DCOM en un sistema operativo basado en Solaris.

6.4.1 COM

Es un estándar abierto, propuesto por Microsoft, documentado desde los más bajos niveles de los protocolos a los niveles más altos. El modelo de componentes objeto es un modelo de programación basado en objetos diseñado para promover la interoperabilidad. Permite que dos o más aplicaciones o «componentes» cooperen fácilmente, sin importar si fueron escritos por diferentes vendedores en diferentes épocas, en diferentes lenguajes de programación, o si son ejecutados sobre diferentes máquinas basadas en diferentes sistemas operativos. Para soportar las características de interoperatividad, COM define e implementa mecanismos que permiten a las aplicaciones conectarse con otras como objetos de software. Un objeto de software es un conjunto de funciones relacionadas y los estados asociados a estas funciones.

En otras palabras, COM provee las operaciones a través de las cuales un cliente de algún servicio se puede conectar a múltiples proveedores de ese servicio de una manera polimórfica. Una vez la conexión es establecida, el cliente y el objeto se comunican directamente sin tener sobrecarga.

6.4.2 ACTIVEX

Está diseñado para brindar contenido activo y las facilidades de red a aplicaciones de escritorio. Está construido sobre la tecnología OLE existente. OLE fue originalmente diseñada para proveer pequeños objetos reusables, llamados controles OLE. Estos poseen una interfaz de programación común, la cual puede ser compartida entre las aplicaciones locales o a través de la red de área local (LAN). ActiveX es el sucesor de OLE, y los controles ActiveX pueden ser enlazados a través de Internet. Los atributos de un control ActiveX puede ser manipulado por simples lenguajes de scripting como VBScript, JScript.

La seguridad de ActiveX está basada en una nueva tendencia llamada «fuente confiable», limitando el acceso para la instalación de nuevos controles activos en el computador. Los controles ActiveX pueden ser marcados digitalmente por el autor del control, y esa marca puede ser registrada con un certificado. De esta manera los controles que son certificados, pueden ser reportados a su autor en el momento en que presenten algún problema.

Cuando un control activo se va a instalar automáticamente en el computador, el browser debe mostrar una ventana de registro informando la fuente de creación. Dado el caso que se vaya a instalar un control que no ha sido certificado, el browser debe preguntar al usuario si desea instalarlo.

Los controles ActiveX se reciben en formato nativo para cada máquina, lo que los hace poco portables. Sin embargo, vale la pena tener en cuenta que, en ambientes homogéneos, el costo de tener soluciones abiertas tipo CORBA no justifica en la mayoría de los casos, pues éstas están pensadas para proporcionar interoperabilidad en ambientes heterogéneos complejos.

7. OBJETOS PERSISTENTES

Como se dijo anteriormente, la persistencia no está definida en el modelo básico de objetos del OMG. En principio la persistencia es un servicio adicional. De hecho en CORBA se especifican dos servicios: persistencia y consultas. Aunque se hará referencia a manejadores de bases de datos, vale la pena tener en cuenta que en el modelo de objetos puro no se requiere hacer diseño de la base de datos. Lo único que hay que hacer es determinar cuáles objetos van a ser persistentes y cuáles transientes (no persistentes).

7.1 BASES DE DATOS ORIENTADAS A OBJETOS

A medida que fueron evolucionando los lenguajes orientados a objetos, fue apareciendo la necesidad de poder crear objetos persistentes, que pudieran conservar la información requerida aún cuando la aplicación no se ejecutará en ese momento.

7.2 IDEAS PRINCIPALES

La primera aproximación a este problema fue tratar de utilizar la tecnología relacional. Las bases de datos relacionales están construidas con base en una teoría matemática muy rigurosa que le da bases sólidas. Tiene además como características muy importantes el hecho de que ya está establecida en el mercado y existe un estándar para la interacción con ellas: el SQL.

Esta aproximación es muy válida, y de hecho se continúa utilizando en muchos proyectos. El principal problema que aparece es que para poder almacenar los objetos en la base de datos hay necesidad de dividirlos para normalizar la base de datos, y además no se pueden almacenar en ella los métodos que forman parte integral del objeto.

Con el fin de romper este problema que planteaba la distancia semántica existente entre el modelo de objetos y el modelo relacional, se comenzó a trabajar en dos

frentes: Extender el modelo relacional para que soporte los conceptos de la TOO y generar bases de datos a partir de los conceptos mismos de la TOO.

7.3 ODBMS vs RDBMS

La primera diferencia que surge entre los ODBMS y los RDBMS es tan profunda que puede ser utilizada como ejemplo para comprender el cambio de paradigma que se dió con la TOO. Es importante saber que ambas tecnologías son muy buenas y que se pueden utilizar para almacenar objetos, sin embargo es muy importante saber que la selección adecuada de la base de datos depende del tipo de aplicación que se va a desarrollar.

El modelo relacional se basa en el álgebra relacional y por lo tanto es orientado a manejo de conjuntos. Cada tabla es un conjunto compuesto por elementos simples del mismo tipo. Las relaciones entre elementos de dos conjuntos diferentes no quedan almacenadas en la base de datos. Para reconstruir una relación entre dos conjuntos diferentes hay que recorrer ambos conjuntos y fijar una condición que permita seleccionar los elementos de ambos conjuntos que cumplen con ella y crear con ellos un nuevo tipo de elemento y un conjunto nuevo. Esta es la razón básica por la cual hay que normalizar la base de datos. Los RDBMS trabajan sobre datos simples y optimizan la búsqueda para datos simples.

Los modelos orientados a objetos se basan en elementos complejos en los cuales se deben almacenar como una unidad. Las relaciones entre objetos deben quedar almacenadas en la base de datos, y de hecho son utilizadas para recuperar los objetos. A diferencia de los RDBMS, las operaciones no se realizan con un enfoque de conjuntos, sino con un enfoque de elementos.

Se puede apreciar que ambas tecnologías no sólo son diferentes por naturaleza, sino que representan los dos extremos en cuanto a complejidad de datos. Los ODBMS son muy eficientes para procesar información muy compleja, mientras que los RDBMS son muy eficientes para manejar información simple. Un primer criterio que surge para la selección de la base de datos es la complejidad de los objetos que se manejen: Si los objetos son simples se puede utilizar un RDBMS. Si son complejos, se debe utilizar un ODBMS.

Éste no es el único criterio que se debe tener en cuenta. Se deben mirar además una serie de problemas que se derivan del hecho de tratar de almacenar objetos en un RDBMS, y que surgen como consecuencia de la distancia semántica que hay entre el modelo de datos orientado a objetos y el enfoque estructurado. Con la aparición de los objetos aparecen la herencia y la composición. Con dos ejemplos simples se puede comprender la dificultad que presentan estos dos conceptos para ser implementados en un RDBMS. La herencia implica que donde utilice un objeto, se puede utilizar cualquier objeto que herede de él. ¿Cómo se puede representar ésto en un RDBMS de una manera eficiente? La composición implica que dos objetos diferentes tengan como atributo simultáneamente dos objetos que sean iguales entre sí. ¿Cómo se almacena esta información en el RDBMS de manera que

se mantenga la integridad y cada objeto conserve su propia identidad? Nótese que el problema no es insoluble, pero la complejidad que presenta implica que el rendimiento que se puede obtener no es bueno.

7.4 ODMG-93

ODMG-93 define el estándar OQL para ODBMS. Es análogo al estándar SQL para los RDBMS. El estándar contiene un modelo de objetos, un lenguaje de definición de objetos (ODL), un lenguaje de consulta (OQL) y especificaciones para los lenguajes Smalltalk y C++.

7.4.1 Modelo de Objetos

Utiliza como base el modelo de objetos del OMG. El modelo del OMG fue diseñado para ser un denominador común para ORB (Object Request Broker), ODBMS, lenguajes de programación y otras aplicaciones. El estándar desarrolla un perfil del ODBMS manteniendo la arquitectura del OMG y adicionando componentes (i.e. relaciones) que soporten las necesidades de los ODBMS.

7.4.2 Lenguaje de Definición de Objetos

Es el equivalente al lenguaje de definición de datos. Está desarrollado utilizando el lenguaje de definición de interfaces (IDL) del OMG como base para la sintaxis.

7.4.3 Lenguaje de Consulta

Es un lenguaje no procedimental para consultar y actualizar la base de datos. Se basó en el estándar relacional SQL en todos los aspectos en los cuales fue posible, y se dieron además otras poderosas capacidades.

7.4.4 Definición para C++ y Smalltalk

Es una definición que integra al lenguaje (C++ o Smalltalk), utilizando su sintaxis, la capacidad de manipular objetos persistentes. Incluye un lenguaje de manipulación de objetos (OML), una versión del ODL que utiliza sintaxis del lenguaje y un mecanismo para invocar OQL.

Con el objetivo de poder garantizar la validez del estándar, ODMG estableció vínculos con ANSI X3H2(SQL), X3J16(C++) y X3J20 (Smalltalk).

8. REUTILIZACIÓN

La posibilidad de reutilizar el código ya desarrollado aparece como un elemento básico para la reducción de los costos y el tiempo del desarrollo del software. Desde la aparición de las TOO, la reutilización se ha presentado como una de las ventajas inmediatas que se obtienen por adoptar este paradigma. Si bien es cierto que las TOO proporcionan un ambiente de trabajo en el cual se conjugan la mayoría de los elementos necesarios para la reutilización, también es cierto que el hecho de adoptar esta tecnología no garantiza la reutilización.

Al hablar de reutilizar componentes se habla de la posibilidad de utilizar en un proyecto nuevo elementos de software previamente desarrollados, sin necesidad de realizar

ninguna modificación sobre ellos. Esto es una realidad. En el mercado se pueden adquirir componentes de software para las más variadas aplicaciones, tales como generadores de interfaces gráficas (GUI), librerías de utilitarios para comunicaciones, librerías de acceso a bases de datos, etc. Estos productos son utilizados a diario por miles de programadores en proyectos diferentes, mientras que un grupo reducido es el responsable por mantenerlos y actualizarlos: los productores del software. En la actualidad, la mayoría de los componentes están organizados como frameworks, que tratan de explotar al máximo los beneficios de la TOO. Dado que en muchas compañías se utilizan este tipo de componentes para agilizar el desarrollo, se podría pensar que en ellas es una realidad la reutilización del software, lo cual es falso.

Al desarrollar sistemas de información se están creando modelos del funcionamiento de la empresa. El problema real es qué tan adecuado es el modelo y cuánto tiempo y dinero se debe invertir en la creación del mismo. La utilización de librerías comerciales es un primer paso que tiene como finalidad el poder concentrarse en el problema real: El modelo de la empresa. De la misma manera que las librerías representan una solución reutilizable a problemas de «bajo nivel» tales como interacción con el usuario, con bases de datos, etc., se pueden plantear soluciones reutilizables a problemas de «alto nivel». Para poder dar este tipo de soluciones se deben orientar las políticas de desarrollo de software hacia la reutilización, con todas las implicaciones que ésto tiene. La reutilización no es una consecuencia de las TOO. Las TOO son sólo una herramienta que facilita el lograr la reutilización. Al analizar detalladamente los elementos de las TOO, se ve que hay un elemento básico que facilita la reutilización: Las jerarquías. La jerarquía de herencia permite realizar la programación por diferenciación. Cuando un objeto es derivado de otro, sólo hay que implementar las diferencias entre ambos. Es la respuesta de la TOO a un problema que ocurre muy a menudo: el nuevo sistema que se va a desarrollar funciona «casi» como uno ya existente. Con la TOO no hay que modificar el código existente, sino que se programa únicamente la diferencia. La jerarquía de composición permite utilizar un objeto bien definido y probado como componente del objeto nuevo, lo que evita tener que implementar nuevamente toda la funcionalidad ya capturada en éste.

A pesar de que se tienen estas facilidades, la TOO no garantiza la reutilización. Para poder heredar las características de un objeto ya existente se requiere que este objeto haya sido diseñado pensando en la reutilización. El diseñador debe capturar en el objeto todas las características necesarias para que sea realmente útil. Lo mismo ocurre con la composición. De estos hechos se concluye que las TOO proporcionan herramientas para la reutilización, pero que ésta sólo se puede alcanzar si hay un compromiso claro de la empresa en utilizar estas características y se asignan los recursos necesarios para poder crear un grupo de diseñadores que se dedique exclusivamente a diseñar objetos genéricos.

9. CONCLUSIÓN

Las TOO son una realidad y están actualmente disponibles. Se ha dado un vistazo a la situación actual de las TOO. Al

igual que todas las tecnologías, tienen pros y contras. La decisión de adoptarlas no puede ser tomada a la ligera. Aunque hay productos comerciales disponibles basados en TOO que van desde lenguajes de programación hasta ambientes integrados que incluyen bases de datos, las TOO no están completamente maduras y, adicionalmente, muchos productos utilizan el calificativo de orientado por objetos como una estrategia comercial. Un elemento clave, que nunca se debe perder de vista, es la relación costo/beneficio de la adopción de las TOO. La adopción de las TOO se justifica si permiten reducir costos y tiempo de desarrollo y/o incrementar la calidad del software. Un principio aplicable, sin llegar a ser regla general, es que las TOO se justifican cuando se desarrolla software de gran complejidad semántica.

BIBLIOGRAFÍA

Arnold, Ken y James Gosling. (1997). El lenguaje de programación Java. Madrid: Addison-Wesley/Domo.

Atwood, Thomas. (1993). ODMG93 - The Object DBMS Standard en: Object Magazine. Sept-Oct. pp. 37-44.

Atwood, Thomas. (1994). ODMG93 - The Object DMBS Standard, part 2" en Object Magazine. Enero. pp. 32 -38.

Booch, Grady. (1996). Análisis y Diseño Orientado a Objetos con Aplicaciones. 2a. Ed. Wilmington Addison-Wesley/Diaz de Santos.

Coad, Peter y Edward Yourdon. (1991). Object Oriented Analysis. 2 Ed. New Jersey: Yourdon Press.

_____ (1991). Object Oriented Design. 2 Ed. New Jersey: Yourdon Press.

Cohn, Mike et al. (1996). Java Developer's Reference. Sams net. U.S.A.

Ellis, John. (1994). Objetifying Real-Time Systems. Sigs Books. U.S.A.

Jacobson, Ivar. et al. (1992). Object Oriented Software Engineering - A Use Case Driven Approach. Wilmigton Addison-Wesley/ACM Press.

Lalinde, Juan G. (1996). Arquitectura de Gestión de Objetos del OMG en: Informática al Día. AUC. Abril-Mayo 96. No. 104, pp. 16-24.

Martin, James y James J. Odell. (1994). Análisis y Diseño Orientado a Objetos. México: Prentice Hall.

OMG. (1995). The Common Object Request Broker: Architecture and Specification. Versión 2.0. pp. 463. Disponible por Internet.

Orfali, Robert. et al. (1996). The essential Distributed Objects Survival Guide. New York: John Wiley & Sons, Inc., pp. 604.

Pree, Wolfgang. (1994). Design Patterns for Object Oriented Software Development. Wilmington: Addison-Wesley/ACM Press.

Pressman, Roger. (1988). Ingeniería del Software. Un enfoque práctico. 2a. Ed. Madrid: McGraw-Hill.

Sheldon, Tom. (1994). LAN TIMES - Enciclopedia de Redes. Madrid: Osborne McGraw-Hill.

Zetlin, Minda. (1997). 97...98...99...00. ¿Está preparada su empresa para el caos computacional?. En: Semana Plus No. 121, Julio 1997. Tomado de: Sloan Management Review.

En Internet:
http://www.omg.org
http://www.rational.com
http://www.javasoft.com