UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

UNIDAD ACADÉMICA DE LOS CICLOS PROFESIONAL Y DE POSGRADO DEL COLEGIO DE CIENCIAS Y HUMANIDADES

UNA GUÍA PRÁCTICA PARA LA EVALUACIÓN DE PLATAFORMAS DE DESARROLLO DE SOFTWARE ORIENTADO A OBJETOS

TESIS QUE PARA OBTENER EL GRADO DE MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA

Francisco Alejo Ríos Gascón

DIRECTOR: DRA. HANNA OKTABA

México D.F. 1995

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

RECONOCIMIENTOS

A la Dirección General de Asuntos del Personal Académico (D.G.A.P.A.) de la U.N.A.M. por el apoyo económico que me brindó durante mi estancia en la maestría y para la elaboración de este trabajo de tesis.

A la M. en C. María Garza Vigil por haber aceptado fungir como mi asesor nacional ante D.G.P.A. para la obtención de mi beca.

A la Dra. Hanna Oktaba por la amistad que me brindo, la paciencia que me tuvo a lo largo de estos años y la dirección de este trabajo de tesis.

DEDICATORIA

A mis maravillosos padres, Francisco y Ana María por ser el faro que con su luz siempre me ha guiado por el camino correcto y haber apoyado la decisión de convertirme de nueva cuenta en estudiante.

A mis hermanos, Iván y Omar por la compañía y camaradería que siempre han demostrado.

A mi novia y prometida Alicia por las palabras de aliento que siempre tenía en los momentos más difíciles.

A mis compañeros de la maestría por el inagotable intercambio de ideas y conocimiento que cada una de nuestras charlas siempre dejaba.

CONTENIDO

INTRODUCCIÓN i

CAPÍTULO 1 EL ENFOQUE ORIENTADO A OBJETOS VS. LA DESCOMPOSICIÓN FUNCIONAL DEL SOFTWARE 1

Introducción. 1

1.1 Características deseables en toda pieza de software. 2

1.2 El ciclo de vida del software basado en la descomposición funcional. 5

1.3 El ciclo de vida del software orientado a objetos. 12

1.4 Comparación de ambos enfoques. 15

CAPÍTULO 2 LA METODOLOGÍA CASE Y LAS CARACTERÍSTICAS INDISPENSABLES EN UN AMBIENTE DE DESARROLLO ORIENTADO A OBJETOS 18

Introducción. 18

2.1 La metodología CASE. 19

2.2 Las características deseables en un ambiente de desarrollo orientado a objetos. 25

2.3 Conclusiones. 32

CAPÍTULO 3 SNAP (STRATEGIC NETWORKED APPLICATIONS PLATFORM) 35

Introducción. 35

3.1 El Templete de SNAP. 36

3.2 El Ambiente de Desarrollo de SNAP. 38

3.3 El Modelo de Objetos. 41

3.4 El Componente dedicado a la Interfaz Gráfica de Usuario. 46

3.5 El Componente de Almacenamiento Permanente. 51

3.6 El Componente de Comunicaciones. 56

3.7 La Interfaz con Software de Aplicación Externo. 60

CAPÍTULO 4 NEXTSTEP 63

Introducción. 63

4.1 El sistema operativo Mach. 65

4.2 El modelo de objetos: Objective-C y las bibliotecas de clases de NEXTSTEP. 68

4.3 El servidor de Ventanas y el Display de Postcript. 70

4.4 Herramientas de Desarrollo. 71

4.5 El componente GUI de NEXTSTEP: Interface Builder. 72

4.6 El administrador de proyectos: Project Builder. 74

4.7 El componente de persistencia de NEXTSTEP: Enterprise Objects Framework. 75

CAPÍTULO 5 DEFINICIÓN DE LOS PARÁMETROS DE EVALUACIÓN DE UNA PLATAFORMA DE DESARROLLO DE SOFTWARE ORIENTADO A OBJETOS Y UN CASO DE ESTUDIO: SNAP VS. NEXTSTEP 77

Introducción. 77

5.1 Parámetros de evaluación que se deben considerar para cualquier plataforma de desarrollo de software orientado a objetos. 78

5.1.1 Evaluación del modelo de objetos. 79

5.1.2 Evaluación del componente GUI. 80

5.1.3 Evaluación del componente de almacenamiento permanente. 81

5.1.4 Evaluación del componente de comunicaciones. 82

5.1.5 Parámetros adicionales de evaluación. 83

5.2 Evaluación de SNAP. 84

5.3 Evaluación de NEXTSTEP. 87

5.4 Comparación. 90

CONCLUSIONES 96

BIBLIOGRAFÍA 97

INTRODUCCIÓN

Durante los últimos tres años dado que a nivel mundial se ha observado un creciente interés por la tecnología de software orientada a objetos. Poco a poco la industria de la informática en México ha ido adoptando esta tecnología por las ventajas que ofrece con respecto a otros paradigmas de análisis, diseño y programación de sistemas de información.

Sin embargo, la transición ha sido lenta debido a que los sistemas orientados a objetos requieren de una nueva forma de trabajo, los esquemas de descomposición funcional para el análisis y diseño de software se tienen que substituir por un enfoque orientado más hacia el modelado de los datos que a la serie de transformaciones que estos sufren durante su ciclo de vida útil.

Debido a la dinámica actual, las organizaciones dependen cada vez más de la calidad y la eficacia de sus sistemas de información. La capacidad para desarrollar aplicaciones en intervalos de tiempo muy cortos es una necesidad fundamental si se quiere ser competitivo.

El desarrollo rápido de aplicaciones usando el modelo orientado a objetos se puede lograr fácilmente cuando se tienen las herramientas adecuadas y personal calificado y con experiencia en el paradigma.

Del universo de herramientas que se emplean en el desarrollo de sistemas de software, surgen con una fuerza cada vez mayor los ambientes de desarrollo de software orientado a objetos. Su principal objetivo es facilitar el desarrollo rápido de aplicaciones orientadas a objetos ofreciendo una serie de herramientas útiles en la construcción de éstas.

En este trabajo se analizan las ventajas que el modelado orientado a objetos tiene con respecto a las técnicas tradicionales de análisis y diseño (capítulo 1), las características indispensables que un ambiente de desarrollo de software orientado a objetos debe tener (capítulo 2); se describen dos ambientes de desarrollo: SNAP (capítulo 3) y NEXTSTEP (capítulo 4), se definen los parámetros de evaluación a considerar cuando se está planeando la adquisición de un ambiente

de desarrollo orientado a objetos y se realiza una comparación entre SNAP y NEXTSTEP (capítulo 5).

Es importante hacer énfasis que las conclusiones de este trabajo son totalmente imparciales y que no hay predilección hacia ninguno de los ambientes comparados. Definitivamente creemos que los parámetros de evaluación que planteamos a lo largo de esta tesis cubren los aspectos básicos de una plataforma de desarrollo de software orientado a objetos, pero también somos conscientes de que pueden ser enriquecidos de acuerdo a las necesidades particulares de cada organización.

Ciudad Universitaria, Marzo 28 de 1995.

CAPÍTULO 1

EL ENFOQUE ORIENTADO A OBJETOS VS. LA DESCOMPOSICIÓN FUNCIONAL DEL SOFTWARE

Introducción.

El objetivo de este capítulo es discutir las cualidades deseables que toda pieza de software debe tener y los dos caminos que existen para el modelado de sistemas de información: el enfoque propuesto por la descomposición funcional del software y el enfoque basado en el modelado de los datos. La descomposición funcional es el mecanismo de abstracción que nos permite plasmar las transformaciones que se deben realizar sobre los datos durante su periodo de vida a lo largo de la serie de procesos a los que son sometidos en un negocio o actividad del mundo real.

Por otro lado, el modelado de las clases de datos que participan de manera activa en un sistema de información es indispensable para evitar la duplicidad de información, la heterogeneidad de formatos, y toda una serie de problemas derivados de la construcción inadecuada de las entidades y sus relaciones.

Ambos enfoques han probado ser efectivos, sin embargo la eficacia de un buen diseño se basa en el uso de cada uno de ellos de manera complementaria. El paradigma orientado a objetos por su naturaleza requiere de ambas perspectivas, es por ello que es necesario su análisis para identificar los requisitos indispensables con los que debe contar un buen modelo de objetos.

1.1 Características deseables en toda pieza de software.

Actualmente los sistemas de información juegan un papel fundamental en el esquema operativo de cualquier organización ya que proveen los medios para el procesamiento de los datos relevantes al negocio. Su finalidad va más allá del registro histórico de información al hacerla disponible no solo a ciertos departamentos o funciones especificas, sino a toda la institución en su conjunto. De esta manera la toma de decisiones estratégicas se facilita pues se tiene una panorámica global del estado del negocio en cualquier instante de tiempo.

Acordes a la dinámica de nuestro mundo moderno, los sistemas de software actuales tienden a ser extremadamente complejos [2] [8]. Para ilustrar el grado de complejidad al que se puede llegar con los sistemas de software modernos consideremos el siguiente ejemplo: supongamos que nuestro negocio es la comercialización de papel reciclado y que entre nuestros clientes principales tenemos a un grupo de imprentas y a varias casas editoriales. Con el paso del tiempo, observamos que la demanda de nuestro producto se incrementa considerablemente debido a que es mucho más barato usar papel reciclado y a la conciencia ecológica que se ha despertado en los consumidores. Para poder satisfacer la demanda, hemos tenido la necesidad de ampliar nuestra capacidad de producción abriendo una serie de plantas en el interior de la república. Por cuestiones administrativas requerimos de un sistema de información que nos indique los niveles de productividad de cada una de las plantas y el total disponible de papel listo para su venta, entre otras cosas. Una de las posibles soluciones que se le puede dar a este problema es diseñando una aplicación que aproveche las cualidades de una base de datos distribuida sobre plataformas heterogéneas de hardware (esto se debe a que no todas las plantas tienen que contar necesariamente con el mismo tipo de equipo de cómputo). Para poder implantar esta aplicación se necesita de un grupo de trabajo formado por expertos en comunicaciones y redes, diseño y administración de bases de datos y programadores de alta calidad para cada tipo de plataforma de hardware disponible en cada planta. Como se puede observar, la complejidad no solo radica en la

aplicación *per se*, sino que también se manifiesta en la administración y coordinación de los diferentes grupos de trabajo.

Sin embargo, no todas las necesidades de procesamiento de información dentro de una organización deben ser complejas. Booch [2] establece una diferencia precisa entre dos tipos diferentes de software fácilmente identificables:

- El ***software sencillo*** (término que proponemos a falta de una definición precisa del autor) es cualquier aplicación de baja complejidad que puede ser especificada, construida y mantenida por una sola persona, cuyo propósito por lo regular es de carácter muy específico y con una vida útil corta, y
- El ***software de poder-industrial (industrial-strength software)*** representado por aplicaciones que tienen un conjunto muy rico de comportamientos (por ejemplo, los sistemas financieros) o que modelan fenómenos muy intrincados (por ejemplo, los sistemas para simular los mecanismos de la inteligencia humana). La vida útil de este tipo de software tiende a ser muy amplia y, con el paso del tiempo, una gran cantidad de usuarios depende de su correcto funcionamiento. Una de las características principales que lo distingue es su alto grado de complejidad, la cual rebasa la capacidad individual del ingeniero de software para comprender *todas* las sutilezas que su diseño implica. Esta complejidad es inherente a los sistemas de software de grandes magnitudes.

El software de poder-industrial representa un gran reto para cualquier grupo de desarrollo. Como se trata de aplicaciones de muy alta dificultad, debemos ser capaces de dar un orden al caos que representa abordar el problema en su totalidad. Dar orden significa poder identificar todos los aspectos relevantes en la aplicación: las necesidades reales del usuario, los datos y los procesos que se aplican sobre estos (de acuerdo a las explicaciones que nos hayan dado los expertos en el dominio del problema) y, finalmente, la manera en que vamos a resolver cada uno de los procesos determinados: esto implica decidir sobre la tecnología de software a usar (por ejemplo, bases de datos, lenguajes de 3ª y/o 4ª generación o sistemas orientados a objetos).

Obvio es pensar que debe haber alguna manera para lidiar con la complejidad, y resulta ser que la respuesta se encuentra en la antigua fórmula de *divide et impera* (divide y vencerás) [2] [8] [12]. Un claro ejemplo de la aplicación de ésta regla es el método de *refinamiento por pasos sucesivos* [8] [9] propuesto por Wirth: el problema inicial se descompone en una serie de subproblemas, los que a su vez son descompuestos en otra serie de sub-subproblemas hasta que logramos identificar todos aquellos componentes que son atómicos o de fácil solución (de acuerdo a ciertos criterios de terminación). Una vez logrado este objetivo, realizamos un recorrido de abajo hacia arriba para ir resolviendo los problemas inmediatos superiores de tal forma que al llegar al nivel más alto de abstracción ya hemos resuelto el problema original.

Cuando aplicamos la fórmula *divide y vencerás* a la Ingeniería de Software surge la necesidad de definir una serie de parámetros de calidad para los productos finales o intermedios que generamos. Pezzè [10] define las cualidades que se espera tenga toda pieza de software desarrollada, propiedades que definimos a continuación:

- ***Correctitud:*** un programa **P** es correcto con respecto a una especificación **S** si **P** siempre se comporta de acuerdo a lo especificado en **S**.
- ***Confiabilidad:*** un programa **P** es *confiable* si el usuario puede depender de él (por ejemplo, que garantice un correcto comportamiento para un cierto intervalo de tiempo).
- ***Robustez:*** un programa **P** es robusto si se comporta razonablemente incluso en circunstancias que no fueron anticipadas en las especificaciones.
- ***Rendimiento (Performance):*** es evaluado por la manera en la que el programa usa los recursos del sistema.
- ***Facilidad de uso (User friendliness):*** que tan fácil es usar el programa.
- ***Verificabilidad:*** que tan fácil es verificar las cualidades del programa.
- ***Mantenibilidad:*** definida por la facilidad para dar mantenimiento al programa.
- ***Reusabilidad:*** métrica para saber que tan fácil es reusar el programa (o parte de él).
- ***Portabilidad:*** que tan fácil es portar el programa a una nueva plataforma.

- ***Productividad:*** que tan fácil es producir el programa (aquí entra la pregunta: ¿Tenemos ambientes de desarrollo?).
- ***Líneas de Tiempos (Timeliness):*** definen la facilidad para entregar el producto en calendario.

Además de estas propiedades puede haber una serie de cualidades adicionales que son determinadas especificamente para el tipo de aplicación que se esta desarrollando y que dependen directamente del paradigma empleado (por ejemplo, los conceptos de ***abstracción***, ***modularización*** y ***encapsulación*** son cualidades fundamentales para el modelo de objetos[1], la ***concurrencia***, la ***tipificación (typing)*** y la ***persistencia*** no son privativos al modelo de objetos pero si muy útiles como se verá más adelante).

1.2 El ciclo de vida del software basado en la descomposición funcional.

Los seres vivos, los objetos y cualquier tipo de actividad (intelectual, artística o manual) cumplen un ciclo vital. Los seres vivos nacen, crecen, se reproducen y mueren; un par de zapatos se usa hasta que las suelas se acaban y el proceso de razonamiento que realizamos para analizar y resolver un problema llega a su "fin" cuando estamos seguros de que se ha encontrado la solución (o por lo menos concluimos que no es posible hallarla en ese momento debido a que los elementos con que contamos no son suficientes para guiar correctamente nuestros mecanismos de inferencia).

El ciclo de vida para los dos primeros ejemplos es obvio para todo mundo ya que se pueden establecer claramente los eventos que marcan el principio y el fin del mismo. Sin embargo, en lo que se refiere a las actividades de tipo intelectual este ciclo es difícil de identificar ya que no sabemos a ciencia cierta cómo se inician y cuando concluyen los procesos mentales de razonamiento y aprendizaje. El desarrollo de sistemas de software, aun cuando es una labor

[1] Esto no quiere decir que la modularidad, la encapsulación y la abstracción sean conceptos exclusivos del modelo de objetos y que no se puedan aplicar en el enfoque de descomposición funcional, sin embargo resulta más natural su interpretación y uso en el modelado conceptual de una aplicación orientada a objetos.

eminentemente intelectual [8], tiene un ciclo de vida bien definido (aunque en la práctica profesional, este ciclo de vida no necesariamente se cumple).

De acuerdo con Pressman[9], el ciclo de vida clásico usado en la Ingeniería de software es el llamado *modelo de cascada*. Este paradigma demanda de un enfoque sistemático y *secuencial* durante el desarrollo del software que inicia al nivel del sistema y evoluciona a través de las fases de análisis, diseño, codificación, pruebas y mantenimiento (véase fig. 1).

Cada una de las fases contempladas por el modelo de cascada se encuentra bien definida y provee una cierta cantidad de información que es usada para alimentar a las etapas posteriores. Cualquier equivocación en la interpretación del problema u omisión en las especificaciones que el ingeniero de software debe plasmar en la documentación, origina el acarreo y propagación de errores y, consecuentemente, la implantación de un producto carente de un certificado de calidad.

Las actividades que se deben realizar para cada etapa se encuentran descritas en [9], [16], [18], [31] y [32]. [32] Es una excelente referencia para la discusión *grosso modo* de metodologías de análisis y diseño de software. Este artículo aun cuando no es un análisis detallado de los métodos, presenta una panorámica general del enfoque que se aborda para manejar la complejidad del software. Cada uno de los métodos analizados en el artículo (el diseño estructurado discutido por Larry L. Constantine, la metodología de Warnier/Orr, el enfoque propuesto por Gane/Sarson, el método de Yourdon y la propuesta de Peter Chen) tiene una manera muy particular para abordar el problema de análisis y diseño del software. A excepción del método de Chen, el denominador común resulta ser la *descomposición funcional* del problema.

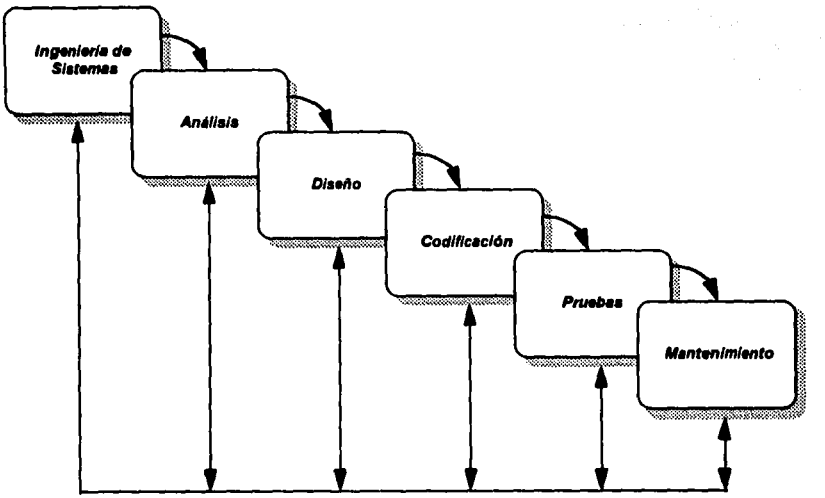

Figura 1. El ciclo de vida clásico de la Ingeniería de Software: El modelo de cascada.

De acuerdo con Constantine ([32]: "The Structured Design Approach") se puede resumir la arquitectura general del software desarrollado con la técnica de diseño estructurado como sigue:

1. Realizar una descripción no procedural (es decir, independiente de cualquier método) de los requerimientos del sistema, usualmente basada en un diagrama de flujo de datos (DFD).
2. Basados en la estructura del problema, se debe elegir un modelo de organización de software apropiado, o bien la combinación de modelos que más se ajuste al problema.
3. Guiados por el flujo de datos y el modelo organizacional elegido, descomponer las funciones globales en un grupo de subfunciones y combinar funciones primitivas en funciones de más alto nivel hasta que los requerimientos sean satisfechos completamente.
4. Usando reglas de diseño y de medición para los conceptos de acoplamiento y cohesión, refinar el diseño para incrementar la modularidad, extensibilidad y la reusabilidad de módulos.

El refuerzo práctico para la teoría desarrollada del método de diseño estructurado toma la forma de dos mediciones: el acoplamiento y la cohesión de módulos. La cohesión es una medida para el concepto de generalidad (***wholeness***) y el acoplamiento mide el nivel de interdependencia. Un

buen diseño es aquel que se distingue por estar construido por un conjunto de módulos, los cuales son altamente cohesivos y muy poco acoplados.

Para el caso de los modelos orientados a objetos, la cohesión de módulos que implantan clases de objetos y su acoplamiento con otros módulos es un concepto importante para la construcción de sistemas con componentes ***verdaderamente reusables***. Por consecuencia las métricas de acoplamiento y cohesión se han extendido y adaptado para evaluar la calidad de dichos módulos y de los tipos de datos abstractos[2].

Por otro lado, Yourdon ([32]: "The Yourdon Approach") hace énfasis en que su método consta fundamentalmente de dos partes: la primera formada por una serie de herramientas y la segunda cimentada por un conjunto de técnicas. Para él las herramientas son una variedad de diagramas gráficos que sirven para modelar los requerimientos y la arquitectura de un sistema de información (SI).

De estas herramientas la más familiar resulta ser el diagrama de flujo de datos (DFD), del que se muestra un ejemplo en la figura 2. Un DFD es un excelente artilugio para modelar las **funciones** que un sistema debe realizar, pero no expresa información suficiente sobre las relaciones de los datos y sobre el comportamiento de la aplicación con respecto al tiempo. Por consecuencia, su metodología ha sido extendida para usar diagramas entidad-relación (DE-R) y diagramas de transición de estados (DTE).

Ahora bien, debido a que los diagramas proveen únicamente de un efectivo medio para comunicar la información sobre diferentes aspectos del SI, Yourdon recomienda el uso de un diccionario de datos el cual debe detallar la composición de cada elemento de información (dato) y un conjunto de especificaciones de los procesos que describen el comportamiento requerido para cada burbuja en el DFD.

---

[2]Regularmente la gente poco informada sobre el paradigma orientado a objetos tiende a interpretarlo como una extensión de los tipos de datos abstractos al dotarlos de funcionalidad. Sin embargo se olvidan de las cualidades fundamentales que distinguen al paradigma: la herencia, el polimorfismo y el *dynamic binding*. Estas cualidades son las que hacen al modelo de objetos realmente potente y versátil.

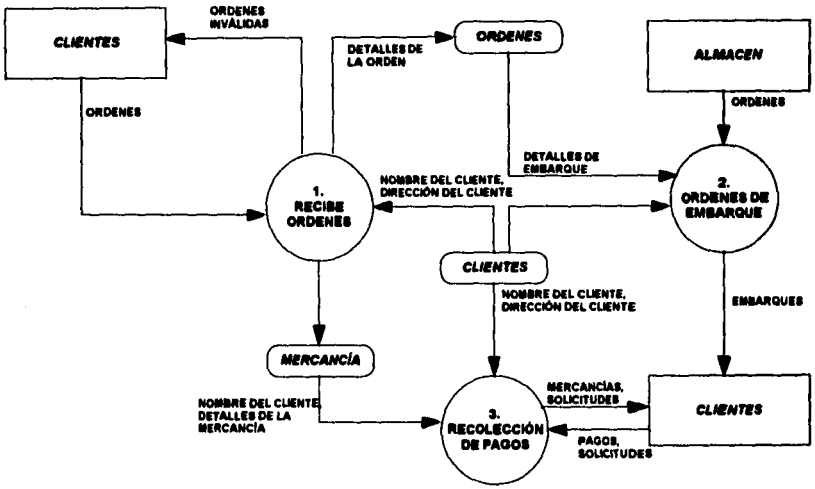

Figura 2. Ejemplo de un Diagrama de Flujo de Datos (DFD).

Las técnicas del método de Yourdon consisten básicamente en una suerte de "recetas de cocina" basadas en una procedimiento conocido como *partición por eventos*. Este enfoque comienza dibujando un diagrama de contexto de alto nivel, para identificar las fronteras del sistema y para definir las interfaces necesarias entre el SI y fuentes externas de información. Después de entrevistar al usuario, se puede escribir una lista de *eventos* que ocurren en el ambiente exterior y a los cuales el sistema debe responder (estos eventos son por lo regular transacciones que sirven para alimentar de información al SI).

El enfoque de partición por eventos provee una guía simple para elaborar el primer esbozo de un primitivo DFD: para cada evento identificado, se dibuja una burbuja cuya función es la de proporcionar la respuesta requerida para ese evento. Debido a que un sistema de información puede tener más de cien burbujas este enfoque nos ayuda a *desmenuzar hacia arriba*, esto es, el poder agrupar a varias burbujas de un DFD y representarlas como una sola en un DFD de más alto nivel. La estrategia para decidir cuales burbujas deben agruparse consiste en buscar aquellas burbujas que lidian con datos comunes. En este sentido, la partición por eventos es muy similar al enfoque propuesto por el diseño orientado a objetos.

Pero, ¿cual es la filosofía del método de Yourdon?. Abarca en concreto los siguientes puntos:

1. **Modelar es bueno.** El modelar una aplicación antes de construirla es una actividad fundamental. Sin embargo, para que esta actividad sea realmente productiva, el modelo debe ser de bajo costo y de fácil construcción, es decir, si modelar cuesta tanto como desarrollar el sistema entonces definitivamente se trata de una pérdida de tiempo. El modelo debe cumplir con dos características fundamentales:

   a) Debe ser exacto: no debe desviarnos del objetivo original o engañarnos sobre las necesidades del usuario y

   b) Debe ser fácil de entender: debe remarcar aquellos aspectos que sean realmente importantes y ocultar aquellos que no son relevantes o de interés para la aplicación.

Dado que la mayoría de los sistemas son muy complejos en tres dimensiones básicas: funcionalidad, datos y control de tiempos y eventos, Yourdon recomienda elaborar tres tipos diferentes de modelos: diagramas de flujo de datos para expresar la funcionalidad del sistema, diagramas entidad-relación para el modelado de los datos y diagramas de transición de estados (DTE) para modelar eventos y el *timing* de la aplicación.

2. **Iterar es bueno.** Por medio de una serie de revisiones periódicas, se pueden ir refinando y mejorando los modelos que hemos creado, pero de nueva cuenta, estos modelos deben ser fáciles de crear y revisar.
3. **Dividir es bueno.** Debido a que las aplicaciones del mundo real son muy complejas, el dividir a la aplicación en piezas de menor complejidad es fundamental para lograr un manejo adecuado del problema que se está analizando.

En este artículo, Yourdon comenta sobre los debates en torno a si la descomposición del software debe ser funcional o basada en los datos. De manera prudente, se abstiene de tomar partido por alguno de estos enfoques, pero hace énfasis en que el uso de cualquiera de los dos (si se hace con cierto rigor) es mucho más provechoso y productivo que no hacer ningún tipo de análisis. El desarrollo de una aplicación que no haga énfasis en este tipo de modelado da como resultado una serie de subsistemas delicados y con interconexiones patológicas.

También basada en la descomposición funcional del software, IBM desarrolló una metodología que hace énfasis en la necesidad de definición de un enfoque que permita administrar la evolución de un SI denominada *Bussines System Planning* (BSP) [16].

Un estudio BSP parte de los requerimientos iniciales de información de una organización y avanza gradualmente hasta concebir un esquema de logística que permite involucrar a los diferentes niveles ejecutivos de la empresa para lograr un objetivo común: conseguir un manejo apropiado de la información de tal suerte que su interpretación permita la toma de decisiones estratégicas y que además las aplicaciones generadas tengan un cierto grado de independencia de los lineamientos administrativos propios de cada división dentro de la organización. De esta manera, los SI que se emplean en cada una de las divisiones de la organización forman un grupo de aplicaciones que modelan a la empresa como un todo; evitandose que las aplicaciones se conviertan en "islas de información" solo útiles para el departamento o división que las usa.

Como dijimos anteriormente, el BSP es una metodología que basa su éxito en la combinación de un esquema de descomposición funcional cuyo propósito es el modelado del *ambiente y los objetivos del negocio* (que se traducen en una serie de *procesos del negocio*), en la identificación de los *datos del negocio* (las *entidades del negocio* y las *clases de datos*), así como en un método orientado a la administración del estudio.

La parte medular de un estudio BSP radica en la identificación de los procesos del negocio. Esto significa que el modelado preciso de las operaciones que se realizan sobre los datos en cada etapa es fundamental. Esto se logra usando un esquema *top-down* en el que se parte de la definición de los objetivos del negocio, la organización del mismo, los procesos del negocio, los datos hasta llegar a una arquitectura de información. A partir de este punto el análisis es *bottom-up* en el cuál se definen las bases de datos, las aplicaciones, los procesos de negocios y finalmente se llega a los objetivos del negocio.

Resulta interesante la concepción que el documento de IBM hace de una *clase de datos*. La definición que se da es la siguiente:

"Una clase de datos representa una categoría de información sobre una entidad. Por consecuencia, una elección obvia de clase de datos sería el asignar una categoría de información a cada entidad. Sin embargo, para ser capaces de manejar la integridad de los datos, no debe haber *más de una* fuente para su creación - una fuente que se mantiene por su facilidad para ser cuantificada. De esta forma, los datos de una entidad se dividen en múltiples clases de datos cuando varios procesos crean diferentes datos sobre la misma entidad. Típicamente un estudio de BSP define de 30 a 60 clases de datos".

Aun cuando estamos hablando de una metodología desarrollada en 1987, es importante recalcar que el modelado de los datos y la noción de clases de datos adquiere un papel relevante ya que el objetivo fundamental del método es evitar la redundancia e inconsistencia de información como se discute en [1], [2], [14], [15], [16] y [18], sin embargo, el génesis del SI sigue siendo la descomposición funcional de los requerimientos.

1.3 El ciclo de vida del software orientado a objetos.

A diferencia del ciclo de vida del software basado en la descomposición funcional, las aplicaciones que se desarrollan con el paradigma orientado a objetos tienen su origen en el modelado de los datos.

El modelo orientado a objetos ofrece un mecanismo de abstracción cuyo enfoque principal es la identificación de los datos relevantes de una aplicación en lugar de las operaciones que en ella se realizan. Bajo esta perspectiva, las entidades del mundo real son modeladas como **(clases de) objetos**. De esta manera, una clase encapsula las características o **atributos** de sus objetos y a los **métodos** que pueden ser usados para crear y destruir objetos y asignar y accesar valores a sus atributos.

La premisa que justifica tomar este camino en lugar de la descomposición funcional es que la *manera de procesar* la información es menos estable con el paso del tiempo que los *datos* sobre los cuales es aplicada. Al hacer a los objetos de la aplicación el punto de atención, el modelo básico de una aplicación puede mantenerse intacto cuando ésta evoluciona. Este enfoque requiere de:

- Identificar las clases, sus atributos y las relaciones entre clases.

- Identificar las operaciones que afectan a cada clase.
- Determinar qué es públicamente accesible y que debe ser ocultado en cada clase.
- Implantar cada clase.

Otras características importantes del enfoque orientado a objetos son:

- Las subclases pueden heredar y extender tanto a los atributos y los métodos de la(s) clase(s) padre(s).
- El ocultamiento de la información es soportado al definir métodos y atributos en una clase como públicos, protegidos o privados. Los aspectos privados son ocultados de las demás clases.
- El polimorfismo (es decir, la habilidad para que la misma operación se comporte de manera diferente en diversas clases) y el *dynamic binding* (la habilidad para asociar el método correcto en tiempo de ejecución dependiendo del objeto sobre el cual se esté operando) refuerzan de manera impresionante la flexibilidad y fomentan la reusabilidad de clases bien diseñadas.

El ciclo de vida del software más apegado al modelo de objetos es el denominado *modelo de fuente* (también llamado *modelo de espiral*) [3], [11], [30] (véase la fig. 3). Este ciclo de vida está basado en el desarrollo iterativo del software y tiene las siguientes propiedades de acuerdo con [18]:

- El desarrollo es realizado en incrementos de ciclos pequeños. Versiones de pre-liberación de la aplicación (**prototipos**) pueden ser producidos en periodos de tiempo cortos (de dos semanas a tres meses) y las versiones de *release* (**producción**) en ciclos que pueden ir de dos a seis meses.
- Debido a que el desarrollo iterativo es una parte integral del proceso, los requerimientos evolucionan de la misma manera en que lo hace la aplicación. No es necesario completar todo el diseño antes de proceder a la implantación de requerimientos ya conocidos.
- El usuario final participa en una serie de revisiones frecuentes durante el desarrollo. De esta forma se logra involucrarlo con la aplicación, virtud necesaria para este enfoque.

- Los costos y el calendario se mantienen invariantes durante el ciclo, y la funcionalidad es modificada tanto como sea necesario.

La planeación de un proyecto basado en el modelo de ciclo de vida de fuente, se debe complementar con las siguientes guías:

- **Particionar a las aplicaciones de grandes magnitudes en múltiples proyectos** de acuerdo a las fronteras del proceso, esto es, cada proceso independiente es una aplicación por si mismo.
- **Dividir la funcionalidad de la aplicación en tareas** que no requieran más allá de dos a cuatro semanas de esfuerzo.
- **Crear un orden parcial de las tareas** basándose en la necesidad de enfocar antes que nada aquellas áreas de riesgo identificadas anteriormente. Se deben implantar las tareas que sean más importantes para el uso operacional de la aplicación primero. Esto garantiza una versión totalmente funcional tan pronto como sea posible.
- **Calendarizar las tareas a implantar incrementalmente en ciclos** de una a tres semanas de tal forma que los resultados de cada ciclo se vean reflejados en una nueva versión funcional de la aplicación.

Figura 3. El modelo del ciclo de vida de fuente

1.4 Comparación de ambos enfoques.

Es indiscutible que el software desarrollado en base al esquema de descomposición funcional ha sobrevivido durante mucho tiempo. El enfoque por si mismo hace énfasis en el modelado de las transformaciones que se realizan sobre los datos, más que en los datos mismos. Esto no es malo, simplemente es una manera de lidiar con la complejidad inherente a los sistemas del mundo real. Sin embargo, este enfoque trae como consecuencia las siguientes limitaciones:

- Es difícil mantener y acomodar cambios evolutivos.
- El diseño no facilita la reusabilidad.
- Los datos son desenfatizados.

Por otro lado, el modelado basado en los datos hace énfasis en las entidades que participan de manera activa y significativa en la aplicación, sin embargo carece de una descripción funcional

detallada de los mecanismos de manipulación de la misma. En [32] el artículo de Peter Chen sobre modelado E-R ("The Entity-Relationship Approach") no toca de ninguna manera la parte funcional de una aplicación. Semánticamente, esto carece de sentido si se desea desarrollar un SI que realmente cumpla con las expectativas del usuario final.

Para el desarrollo de una aplicación orientada a objetos se debe combinar ambos enfoques de modelado: la descomposición basada en los datos para obtener a las clases, sus atributos y relaciones (usando el modelo E-R) y la descomposición funcional para modelar los procesos que se deben realizar en la aplicación, los cuales se reflejan directamente en los métodos de las clases.

Bajo esta perspectiva, las técnicas de descomposición funcional pasaran de ser los actores principales en el proceso de construcción de software a ser actividades complementarias de un proceso mucho más adecuado para el manejo de la complejidad: el enfoque de desarrollo orientado a objetos. Este enfoque produce clases más modulares, extensibles y modificables. Es usado para descomponer los requerimientos en una serie de tareas y subtareas hasta que las **clases** que ellas manipulan son identificadas. Entonces cada tarea es asociada con una clase específica. La descomposición funcional se usa para identificar a las **operaciones** asociadas con una clase.

Actualmente la Ingeniería de Software ha evolucionado del mero bosquejo en papel del análisis y diseño de una aplicación, al uso de herramientas CASE[3]. Una herramienta CASE es un excelente auxiliar en el modelado del negocio que se va a interpretar en un SI, en la generación de la documentación, en la generación de código y en la fase de mantenimiento de la aplicación. El uso de herramientas CASE durante el desarrollo de una aplicación es muy recomendable ya que tienen el potencial para automatizar todas las fases del ciclo de vida del software. Pero, es importante recalcar que una herramienta CASE **no substituye a ninguna metodología**, es tan solo un grupo de aditamentos para los métodos que ofrece una serie de mejoras para lograr la generación de productos de alta calidad.

[3]Del Inglés *Computer-Aided Software Engineering*, Ingeniería de Software Asistida por Computadora.

En el capítulo siguiente se describen con más detalle la perspectiva CASE, las características deseables en un ambiente de desarrollo orientado a objetos y la comunión que debe haber entre las dos filosofías de software.

CAPÍTULO 2

LA METODOLOGÍA CASE Y LAS CARACTERÍSTICAS INDISPENSABLES EN UN AMBIENTE DE DESARROLLO ORIENTADO A OBJETOS

Introducción.

La Ingeniería de Software Asistida por Computadora (CASE) engloba una colección de métodos y herramientas automatizadas para auxiliar a la ingeniería de software durante cada una de las fases del ciclo de vida que tiene el desarrollo de software.

Debido a su naturaleza, las herramientas CASE deben ser abordadas con un enfoque **disciplinado** ya que su objetivo no es el de substituir a los métodos existentes de análisis y diseño sino el de proporcionar una serie de artilugios automáticos que liberen al ingeniero de software de la parte tediosa de su trabajo (la generación de documentación, diagramas, etc.), permitiéndole concentrarse en el **modelado** de la aplicación.

Para explotar al máximo las cualidades y beneficios que ofrece el CASE es necesario contar con personal **bien entrenado** ya que la efectividad y el impacto que se puede lograr en una organización (desde el punto de vista informático) depende fundamentalmente del dominio que el personal tenga sobre las herramientas, metodologías y técnicas que usa y no de las herramientas *per se*.

Una vez que se ha modelado un SI usando CASE (aunque no necesariamente debe ser así, pero la conveniencia de modelar con un ambiente CASE se justificará más adelante en este capítulo), el siguiente paso es la fase de implantación.

En el capítulo anterior se discutieron una serie de parámetros de calidad que toda pieza de software debe cubrir. Uno de estos parámetros se refiere al concepto denominado *productividad*. La productividad está ligada a la fase de implantación de un SI ya que por definición, se debe interpretar como la **capacidad** para generar una aplicación o un prototipo *operacionalmente completo* en el menor tiempo posible.

Ahora bien, una aplicación (o prototipo) es *operacionalmente completa(o)* si cuenta con una interfaz gráfica de usuario (GUI)[4] que permita presentar la información en un ambiente de ventanas (MS-Windows, XWindows, OSF/Motif, etc.), con un modelo de persistencia eficiente y confiable para almacenar y recuperar la información relevante en la aplicación, y con un esquema de comunicaciones que explote las cualidades de una arquitectura cliente-servidor o de cualquier otro modelo.

Para mantener un nivel de productividad adecuado, se pueden tomar dos caminos: usar y explotar las capacidades de un ambiente CASE integrado o bien usar un ambiente de desarrollo[5] que nos permita generar prototipos que a la larga se convertirán en aplicaciones listas para ser integradas a un ambiente de producción, en un intervalo de tiempo razonable. Como se menciona en el capítulo anterior, el modelo de objetos plantea como estrategia el desarrollo de *prototipos extensibles*. Esta forma de trabajo permite atacar el problema que implica planear, costear y calendarizar el desarrollo de una aplicación de gran tamaño (que modela al negocio de manera global) de manera diferente: el desarrollo se van planteando como una serie de productos de menor tamaño pero que ofrecen el nivel de funcionalidad necesario para ponerlos a trabajar. Cada uno de estos productos evoluciona progresivamente hasta alcanzar un cierto nivel de madurez . Si la arquitectura de la aplicación es correcta, el producto final se consigue automatizando gradualmente al negocio.

---

[4] Del inglés *Graphic User Interface*.

[5] También denominado *plataforma de desarrollo*. En este trabajo usaremos de manera indistinta cualquiera de los dos términos para denotar al mismo concepto.

Una plataforma de desarrollo, por consecuencia, debe proporcionarnos las *facilidades*[6] necesarias para cubrir los tres aspectos básicos que una aplicación requiere para ser operacionalmente completa.

A continuación discutiremos con más detalle la metodología propuesta por la Ingeniería de Software Asistida por Computadora y las características que debe tener todo ambiente de desarrollo de software orientado a objetos.

2.1 La metodología CASE.

De acuerdo con Gibson [34], CASE implica el uso de la computadora como una herramienta de desarrollo en la construcción de modelos que describen un negocio, el ambiente del negocio, los planes corporativos, y como auxiliar en la documentación de sistemas de procesamiento de datos desde su planeación hasta su implantación.

Si somos congruentes con este punto de vista, un ambiente CASE integrado debe tener la capacidad para automatizar **todas** las fases del ciclo de vida del software. Al hablar de todas las fases del ciclo de vida no nos referimos a un modelo en particular (cascada, prototipos, espiral, etc.); más bien tratamos de ubicar la génesis de una aplicación a un nivel conceptual más alto, veamos porqué.

En el capítulo anterior una de las ideas de mayor importancia es la de concebir a una organización (desde el punto de vista informático) como un **todo**. De esta manera la información se encuentra siempre *actualizada* y *disponible* a todo mundo, evitándose con esto la aparición de "islas de información" que dificultan el acceso a la misma.

Para lograr esta meta, es necesario modelar a la empresa a tres diferentes niveles de abstracción:

El primer nivel que se debe considerar resulta ser el de mayor grado de complejidad: analizar y estudiar a la organización como *corporación*. El definir los objetivos a corto, mediano y largo plazo

[6] Traducción del término en inglés *facility*.

de la empresa no es una tarea sencilla, se requiere de una compenetración tan íntima con la organización que solo se puede lograr cuando se tiene una panorámica global del negocio. Por lo regular, quienes se encargan de la definición de los objetivos y las estrategias a seguir son los ejecutivos de la empresa.

Una vez que se tienen bien definidos los puntos neurálgicos del negocio, el siguiente paso es la planeación de los SI. Los sistemas de información siempre deben estar orientados a mejorar la calidad y productividad en toda empresa, es por ello que su planeación debe ser el resultado de un estudio serio sobre el impacto que la informática puede tener en la consecución de los objetivos de la organización. Aquí es donde se deben detectar las áreas de oportunidad, analizar los problemas de información y resolverlos diseñando soluciones óptimas que cumplan con los objetivos trazados por la empresa.

La tercera y última etapa resulta ser la de más bajo nivel de abstracción: programar e implantar los SI que surgieron como resultado del estudio realizado en la fase anterior. En esta etapa, el diseño pasa del nivel conceptual al físico. Aquí se evalúan las tecnologías de software a usar y se definen los modelos de datos, los procesos a realizar sobre los datos, las relaciones entre diferentes aplicaciones y las estrategias para la implantación de los productos generados (véase la fig. 4).

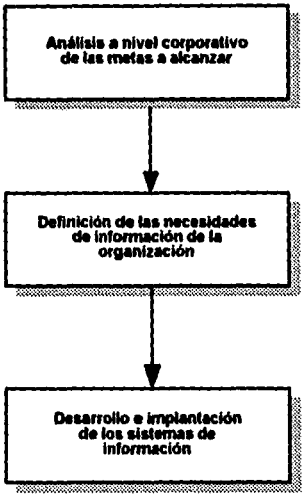

Figura 4. Los tres niveles de abstracción para modelar a una organización

Como puede observarse, los SI se deben planear y poner en marcha cuando se tiene una clara visión del negocio, de las necesidades reales de procesamiento de datos y con el objetivo de *integrarlos* a la infraestructura informática con la que cuente la organización. Esta filosofía de trabajo es difícil de abordar si no se cuenta con herramientas que permitan llevar un seguimiento adecuado de cada una de las etapas que hemos planteado. Una excelente alternativa para cubrir este ciclo de vida global es el uso de ambientes CASE.

Un ambiente CASE integrado se encuentra formado por tres componentes básicos que en conjunto cubren los niveles de abstracción requeridos para formar el modelo global de una organización: el CASE de nivel superior (*Upper CASE*) que permite modelar al negocio, el CASE intermedio (*Middle CASE*) de carácter un poco más técnico ya que está orientado a labores de análisis y diseño de SI, y el CASE de bajo nivel (*Lower CASE*) que representa a la fase de mayor tecnicismo: el desarrollo del sistema.

El *Upper CASE*, también denominado *planeación asistida por computadora*, se refiere al componente orientado a la descripción de la organización y de sus planes. Por lo regular los ejecutivos de una organización invierten gran parte de su tiempo tratando de comprender a la compañía y en la creación de planes para sus actividades. Los planes corporativos fijan las metas a alcanzar, las estrategias para cumplir esas metas, las políticas que se deben seguir para enfocar el esquema operativo de la empresa hacia los objetivos planteados, y una serie de normas de calidad y niveles de rendimiento que se esperan durante el desarrollo de cada una de estas fases.

El CASE de nivel superior usa herramientas de software para describir a la compañía y sus planes gráficamente. Los diagramas generados son útiles para descomponer y especificar a la empresa en todos sus aspectos relevantes (por ejemplo, algunos de estos aspectos pueden ser los diferentes departamentos y las funciones que realiza cada uno de ellos dentro de la organización, los objetivos, planes, recursos, responsabilidades, problemas, etc. de la empresa y de sus departamentos, y así sucesivamente). Poder desglosar las diferentes facetas del negocio suministra información gráfica y textual que ayuda a conseguir una visión más clara de la organización y de la situación actual por la que está pasando.

El uso de estas descripciones permiten la elaboración de planes estratégicos. Las especificaciones de los planes representan los recursos y las tareas a completar que son necesarios para cumplir con los planes corporativos.

La metodología de planeación del *Upper CASE* proporciona un "esqueleto" dentro del cual se insertan una serie de *atributos de planeación* que definen el esquema de un plan. Este armazón cuenta con funcionalidad para indicar las relaciones entre los componentes del plan y los suministros[7] que necesitan para seguir adelante. De esta manera para diferentes planes, la estructura de la empresa se mantiene invariable y solo los valores de los atributos de planeación cambian.

Una vez creado el modelo general del negocio y un buen número de modelos de planeación se facilita su reusabilidad ya que son almacenados como especificaciones dentro de un diccionario.

---

[7] Del inglés *accoutrements*

Entre más constante sea el uso del *Upper CASE* más reusables se hacen las especificaciones ya que se van refinando constantemente los modelos creados con la herramienta.

Todo plan que una empresa establece depende en forma directamente proporcional de la oportuna disponibilidad de información para asegurar su éxito. Los sistemas de *Upper CASE* contienen especificaciones de planeación para actividades de carácter funcional así como para el desarrollo de sistemas de información.

El CASE intermedio está orientado al análisis de los problemas que se tienen con la información y al diseño de soluciones para estos. La mayoría de los sistemas de *Middle CASE* cuentan con herramientas para la realización de diagramas y con un diccionario de datos muy similares a las facilidades proporcionadas por el CASE de nivel superior, sin embargo las metodologías contenidas por el CASE intermedio tienen una filosofía diferente: la combinación de una serie de diagramas y de un diccionario de datos **automatiza** el uso de las metodologías que un analista de sistemas emplea para desarrollar su trabajo.

Debido a que muy poca información sobre las funciones de la empresa está directamente relacionada con el software que soporta las operaciones de ésta, los sistemas de CASE intermedio ofrecen una estructura para almacenar este tipo de información y hacerla más accesible. Esta información involucra la descripción de las operaciones departamentales y su relevancia, el por qué las operaciones son realizadas de una cierta manera, qué tipo de información y cómo la usan, por qué ciertas condiciones influencian su desarrollo, etc. Gibson [34], acota que una de las ventajas del uso continuo del *Middle CASE* radica en que

> "Se pueden ganar años de experiencia sobre como opera la compañía y de cómo usa la información simplemente 'mousing around' (navegando con el ratón) dentro de las especificaciones de diseño."

Una limitante importante que tienen los sistemas de CASE intermedio es que solamente del 25 al 30 por ciento de las especificaciones generadas con ellos son transportables a sistemas CASE de bajo nivel. Los sistemas de *Lower CASE* crean las especificaciones para el desarrollo de los programas de un SI. Debido a que las especificaciones de CASE intermedio básicamente

documentan las actividades de una organización y la manera en que la información es usada, solo un pequeño porcentaje de estas especificaciones puede mapearse directamente a los sistemas CASE de bajo nivel. Sin embargo, una vez que se han modelado un buen número de sistemas usando el CASE intermedio, gran parte de las especificaciones de diseño pueden ser reusadas. Conforme el uso del *Middle CASE* se hace habitual, la cantidad de especificaciones de análisis y diseño de SI que pueden volver a emplearse se incrementa.

El CASE de bajo nivel utiliza componentes de desarrollo de software para crear un conjunto de especificaciones usadas en la generación de los programas y la documentación de usuario de un SI. Debido a que están orientados al diseño físico de las aplicaciones no es común que cuenten con una interfaz gráfica ya que por lo regular no la necesitan. Al igual que los dos niveles CASE anteriores, el *Lower CASE* cuenta con un *diccionario de sistema* y adicionalmente con un *diccionario de desarrollo.*

Tradicionalmente, un diccionario de sistema documenta las características de las entidades del mundo real que están siendo modeladas. Es por esto que las especificaciones almacenadas en este solo proporcionan información del fenómeno que se está analizando. Por otro lado, el diccionario de desarrollo del CASE, es un diccionario *activo*, en el cual se ingresan especificaciones que describen e influencian el desarrollo del objeto que se está modelando al suplir una serie de criterios para su desarrollo así como referencias a sus atributos.

Un diccionario activo, se forma por tres componentes principales:

- Una base de datos en la cual se tienen las características del ambiente de computación y las características explícitas de las aplicaciones,
- Una serie de marcos de trabajo (*frameworks*) que permitan expresar los mecanismos típicos de lógica, comandos específicos y módulos que son comunes a cualquier aplicación y
- Un *activador* capaz de combinar las características del ambiente y de la aplicación con los marcos de trabajo a nivel de comando y módulo para producir los programas de aplicación.

El flujo que la información debe seguir dentro de un ambiente CASE totalmente integrado se encuentra plasmado en la figura 5.

2.2 Las características deseables en un ambiente de desarrollo orientado a objetos.

Un ambiente de desarrollo (o plataforma de desarrollo) orientado a objetos se puede definir como todo sistema de software que permite producir aplicaciones orientadas a objetos en un lapso corto de tiempo (cuantificado en función del tipo de aplicación a producir) debido a que proporciona un grupo de facilidades que liberan al desarrollador[8] de la programación de tareas comunes a toda aplicación y que, por su naturaleza, consumen mucho tiempo.

El tipo de tareas a las que nos referimos en esta definición son, por ejemplo, la programación de la interfaz gráfica de usuario, la programación de mecanismos de persistencia con una base de datos relacional, etc., permitiendo de esta manera que el ingeniero de software pueda centrar su atención en los puntos neurálgicos de la aplicación sin perder tiempo en detalles de implantación. Por consecuencia, *una **buena** plataforma de desarrollo de software orientado a objetos es aquella que libera al desarrollador de la mayor cantidad de trabajo de bajo nivel.*

[8] Del inglés *Developer*. Decidimos utilizar el término *desarrollador* en español ya que no existe ninguna palabra en el idioma que denote el mismo significado que tiene la palabra en inglés. No usamos el término programador, porque deseamos hacer énfasis en el nivel de entrenamiento que se requiere para explotar convenientemente las capacidades de un ambiente de desarrollo orientado a objetos.

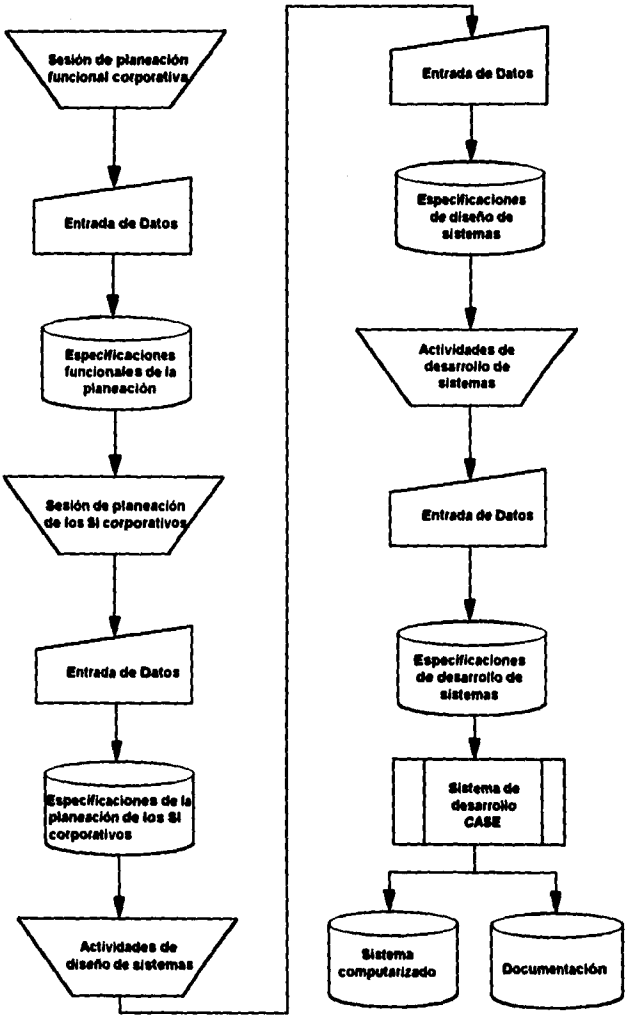

Figura 5. Un ambiente CASE totalmente integrado: De la planeación del negocio hasta la implantación de los SI.

La arquitectura básica de una plataforma de desarrollo orientada a objetos está formada por tres piezas fundamentales: un *componente GUI* dedicado a resolver el problema de diseño de la interfaz gráfica de usuario, un mecanismo dedicado a la persistencia de la información, al que denominaremos *componente de almacenamiento permanente* y un *componente de comunicaciones*. Cubriendo esta tres características básicas, podemos asegurar el desarrollo de aplicaciones operacionalmente completas.

A continuación discutiremos las características básicas de cada una de estas piezas de software.

El componente GUI.

De acuerdo con [37], la interfaz gráfica de usuario es más que la apariencia de una aplicación en la pantalla de la computadora, es la manera en la que se comunica el usuario con la aplicación y viceversa. Es por esto que el diseño del GUI en una aplicación dirigida al usuario final debe considerar las necesidades tanto de un usuario experto como las de un usuario novato. Para el usuario novato, la interfaz debe ser simple y fácil de aprender y recordar, para el usuario experto debe ser rápida y eficiente, es decir, no debe de haber obstáculos que lo distraigan para completar satisfactoriamente su trabajo.

El componente GUI en un ambiente de desarrollo debe cumplir con estos mismos objetivos: entre más simple sea la creación de menús, ventanas, tablas, gráficos, etc.; más rápida y eficiente será la generación de una aplicación.

Bajo este punto de vista, la cantidad de código que debe programar el desarrollador para manipular las ventanas y los menús de la aplicación debe ser mínimo. Aquellos que hayan trabajado con el SDK[9] de Microsoft o el OWL[10] de Borland han pasado por el tenebroso mundo de la programación Windows de bajo nivel. No solamente han tenido que aprender a crear y destruir ventanas, sino que además tienen que ser cuidadosos en mantener el control sobre los recursos de Windows (por ejemplo, la paleta de colores, las *plumas*, etc.) si no quieren que su aplicación

[9] *Software Development Kit.*

[10] *Object Windows Library.*

"tire" a todo el sistema. El componente dedicado a la interfaz gráfica de usuario de un ambiente de desarrollo debe encapsular al máximo la creación y uso de los recursos y hacerse cargo de todo el trabajo de administración de los mismos dentro del ambiente de ventanas.

El uso de un GUI en una aplicación obliga a cambiar el estilo de programación ya que se debe evolucionar de una arquitectura secuencial a una orientada a eventos. El componente GUI debe ofrecer un acoplamiento transparente con el código de la aplicación de tal forma que el desarrollador sólo deba preocuparse por la programación de aquellos *eventos relevantes* en la aplicación, es decir, de la manipulación de *callbacks* en los menús, tablas, gráficos y monitores que se presenten en la pantalla.

Una característica adicional que es deseable en el componente GUI es la portabilidad hacia otras plataformas. En este punto, se debe evaluar la estructura de la biblioteca GUI proporcionada por el componente: como un API[11] o como un *esqueleto de aplicación*[12].

Un API provee un conjunto de funciones de desarrollo del GUI comunes a toda una variedad de plataformas.

Un *esqueleto de aplicación* ofrece un API en capas al proporcionar un grupo de clases que trabajan en conjunto para encapsular el comportamiento de una aplicación de tipo genérico. Un verdadero *esqueleto de aplicación* se diferencia de una biblioteca de clases porque proporciona clases de datos y componentes de aplicación adicionales a los componentes estandar del GUI, permitiendo la construcción de una nueva aplicación tan solo derivando nuevas clases a partir de las clases genéricas contenidas en el *esqueleto*.

El uso de técnicas de programación orientada a objetos en el componente GUI de un ambiente de desarrollo ofrece una serie de ventajas en el diseño de herramientas y/o en la implantación de la interfaz gráfica de usuario. De Santis [39] indica las siguientes ventajas derivadas del uso de un GUI orientado a objetos:

---

[11] Del inglés, *Application Programming Interface* (interface de programación a nivel de aplicación).

[12] Traducción del término en inglés *Application framework*.

- La estructuración de una biblioteca como una colección de clases organizadas por funcionalidad. Tanto la biblioteca como las aplicaciones desarrolladas serán más fáciles de comprender y mantener.
- El proporcionar una interfaz de programación simple y consistente que encapsule los detalles complejos de implantación al desarrollador. Esto reduce considerablemente el tiempo invertido en la programación de un GUI.
- La creación de nuevos componentes fácilmente al derivarlos de las clases proporcionadas por la biblioteca. Estos nuevos elementos pueden operar con los componentes previamente construidos en la biblioteca.
- El reuso de componentes en otras aplicaciones. Esto puede ofrecer un ahorro significativo tanto en tiempo de desarrollo como en costos.

El componente de almacenamiento permanente.

La persistencia de la información es fundamental en toda aplicación independientemente del problema que ésta resuelva. El componente de almacenamiento permanente debe encargarse de proporcionar las facilidades necesarias para que los mecanismos de persistencia sean lo más simples y fáciles de usar. Actualmente, existen solo dos maneras confiables para implantar la persistencia de la información:

a) con el uso de archivos planos, o

b) mediante una base de datos relacional.

El uso de archivos planos.

Los archivos planos fueron el primer medio de almacenamiento que se usó en la computación. Tienen la desventaja de que su administración es difícil ya que se puede caer fácilmente en problemas de inconsistencia, duplicidad y multiplicidad de formatos [4], [14] y [15]. Sin embargo, no dejan de ser una buena solución para resolver el problema de persistencia si son administrados con cuidado.

El uso de archivos planos requiere de una íntima compenetración con el sistema operativo de la máquina que se esté usando. Por esta razón, el componente de almacenamiento permanente de un buen ambiente de desarrollo debe proporcionar una serie de facilidades que encapsulen los detalles de bajo nivel para la creación y manipulación de los archivos planos.

El uso de bases de datos relacionales.

Las bases de datos relacionales han probado ser la solución ideal para el problema de persistencia de cierto tipo de información. Basadas en el cálculo y álgebra relacional, ofrecen un esquema conceptual simple (basado en el uso de "tablas" para modelar a las entidades y relaciones que participan en la aplicación y el concepto de "renglones" que representan a las instancias de esas "tablas") que garantiza la calidad en la forma que está siendo almacenada la información, siempre y cuando su diseño haya sido realizado con el debido rigor que el modelo relacional exige.

Actualmente, los productos comerciales disponibles son bastante *maduros* y *confiables*. Ofrecen por lo regular algún dialecto del lenguaje SQL[13], una gran variedad de estructuras de datos para hacer eficiente el acceso a la información y una serie de mecanismos para el control de procesos concurrentes.

El componente de almacenamiento permanente debe proporcionar las facilidades para la interacción con el manejador de bases de datos relacionales y con el SQL para hacer más natural la programación de la persistencia. Un parámetro importante en la evaluación del componente es la versatilidad para usar diferentes manejadores de bases de datos.

En un ambiente de desarrollo orientado a objetos, se debe proporcionar una biblioteca de clases que permita manipular archivos planos o la interfaz con una base de datos relacional. Por ejemplo, SNAP proporciona una serie de facilidades para la manipulación de archivos planos a través del módulo de clases denominado FIO, así como una serie de facilidades que encapsulan los detalles de conexión e interacción con una base de datos relacional en el módulo de clases DBL.

El componente de comunicaciones.

[13]Siglas del término en inglés *Structured Query Language*, lenguaje estructurado de consulta.

Como apunta Booch en su libro [2], la complejidad del software actual es extremadamente alta. Esto hace pensar que una aplicación debe explotar las ventajas que da la distribución de tareas en diferentes procesadores. Cuando se distribuyen las tareas de procesamiento en varias computadoras, se logra un nivel adecuado de *rendimiento* ya que los tiempos de respuesta son abatidos al disminuir la carga de trabajo por procesador comparada con la que se tendría con una arquitectura monolítica de software. Por esta razón es recomendable que la plataforma de desarrollo soporte algún modelo de comunicaciones.

De la gran variedad de paradigmas de comunicaciones, un modelo que ha demostrado su eficiencia por la popularidad que ha adquirido es la arquitectura *cliente-servidor*.

En el modelo *cliente-servidor* las tareas son distribuidas entre dos tipos de procesos: los procesos *cliente* y los procesos *servidor*. Un *cliente* puede hacer una petición de información o enviar un mensaje para la ejecución de un procedimiento remoto en el *servidor*, cuando va a iniciar o ha completado un ciclo de procesamiento local bajo un esquema puramente *transaccional*. Dependiendo del tipo de conexión establecida (asíncrona o síncrona), el cliente continúa con sus tareas o espera la respuesta del servidor.

El componente de comunicaciones amen de que debe soportar algún modelo de comunicaciones, debe encapsular al máximo los mecanismos para el control de las transacciones. Es muy recomendable que soporte la arquitectura *cliente-servidor* sin embargo, puede estar basado en otras arquitecturas como la *peer to peer*, *maestro-esclavo*, etc.

El modelo de Objetos.

Además de los tres componentes básicos una plataforma de desarrollo debe contar con un *lenguaje de programación anfitrión*. Este lenguaje debe tener capacidades para hacer llamadas a lenguajes incrustados[14] (por ejemplo, contar con un preprocesador para la inserción de cláusulas SQL dentro del código), que soporte mecanismos de comunicación tales como las llamadas a

[14]Traducción del término en inglés *embeeded languages*.

procedimientos remotos (RPC) y que además su interacción con el componente GUI sea totalmente transparente.

Si se trata de una plataforma de desarrollo orientado a objetos, el lenguaje de programación anfitrión debe contar con un modelo de objetos robusto y fácil de aprender que explote al máximo las capacidades de la herencia, el polimorfismo y el *dynamic binding*. Entre más puro sea el lenguaje mejor. Por ejemplo, uno de los conceptos más difíciles de entender para el desarrollador novato en el paradigma de objetos es la creación y destrucción dinámica de instancias. En un lenguaje orientado a objetos puro, la creación y destrucción deben estar encapsuladas en un par de instrucciones propias del lenguaje y no se debe llegar a un nivel tan bajo de definición como se tiene que hacer en C++.

Además el lenguaje debe contar con una biblioteca de clases que ofrezcan una variedad de servicios de diferentes tipos: por ejemplo, ofrecer una clase que encapsule las llamadas SQL dentro del código, una clase que encapsule los mecanismos de comunicaciones, una clase que realice las operaciones necesarias para la manipulación de fechas, de *strings*, etc.

2.3 Conclusiones.

Como se puede observar, CASE plantea un enfoque integral en el modelado e implantación de un sistema de información. Los diferentes niveles de abstracción que tenemos que cubrir para la creación de una solución apropiada a las necesidades reales de una organización implican un esfuerzo notable, pero con el uso de herramientas CASE no es imposible conseguirlo.

Un ambiente de desarrollo de software orientado a objetos es una alternativa para la fase de programación de una aplicación, el otro camino es el uso de un componente CASE de bajo nivel para la conclusión del ciclo de vida integral de un SI. Debido a que la tecnología CASE no se encuentra lo suficientemente avanzada para cubrir todos los niveles que su mecánica de trabajo plantea, los ambientes de desarrollo son la mejor alternativa a seguir.

Para aliviar la carga de trabajo, una cualidad que sería muy recomendable (más no indispensable) en una plataforma de desarrollo es que contara con una interfaz CASE. Si el modelado de la aplicación se realiza con una herramienta CASE, entre menos información tengamos que generar "a pie" en el ambiente de desarrollo, más *productivos* podemos ser.

Por lo regular, los SI tienen que intercambiar datos entre sí para evitar el vicio de las "islas de información". Cuando nos enfrentamos al problema de desarrollar una nueva aplicación siempre buscamos resolver las necesidades de procesamiento con el menor esfuerzo posible. Si existen aplicaciones que han sido probadas durante un cierto intervalo de tiempo y que han demostrado un nivel de certidumbre bastante aceptable, el sentido común nos dice que es mejor buscar una manera de comunicarse con esa aplicación que tratar de rehacerla como un módulo del nuevo sistema. Por esto, es deseable que un ambiente de desarrollo cuente con un componente de interfaz con aplicaciones externas. Este componente debe encargarse de la transparencia en la comunicación con aplicaciones que pudieron desarrollarse con tecnología de software distinta a la de la plataforma.

Podemos resumir las características deseables en una plataforma de desarrollo orientado a objetos en los siguientes puntos (véase la figura 6):

1. Debe contar con un lenguaje de programación que explote el modelo de objetos de forma eficiente,
2. Contar con un componente dedicado a la interfaz gráfica de usuario que sea fácil de usar,
3. Tener un modelo de comunicaciones,
4. Contar con un esquema de persistencia robusto,
5. Ofrecer un módulo de interfaz con software de aplicación externo, y
6. Opcionalmente, contar con una interfaz para comunicarse con alguna herramienta CASE (preferiblemente un CASE orientado a objetos).

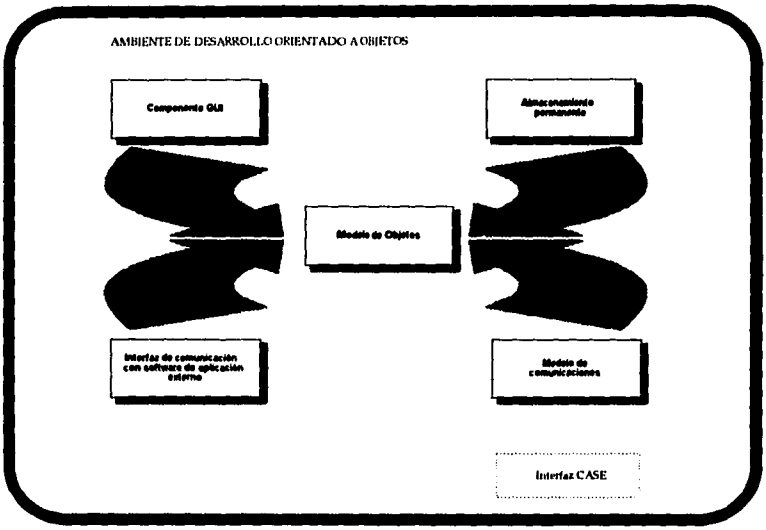

Figura 6. Arquitectura de un ambiente de desarrollo de software orientado a objetos.

CAPÍTULO 3

SNAP (STRATEGIC NETWORKED APPLICATIONS PLATFORM)

Introducción.

Una necesidad que los sistemas de información modernos deben resolver es la de proveer soluciones poderosas que se puedan ejecutar en múltiples ambientes de hardware conectados por redes locales y de área amplia[15]. Este tipo de computación distribuida por lo regular está orientada a soportar las actividades críticas de un negocio. Debido a su origen y naturaleza, estas aplicaciones ofrecen servicios que van más allá de los confines del centro de procesamiento de datos tradicional.

SNAP acrónimo de *Strategic Networked Applications Platform* (Plataforma de Aplicaciones Estratégicas en Red) es un ambiente de desarrollo orientado a objetos comercializado por la empresa Norteamericana *Template Software Inc*[16]. que ha tenido un fuerte impacto en la industria por su enfoque de trabajo basado en el concepto de *templete*[17].

La versión más reciente de SNAP (Release 6.0) puede correr en estaciones de trabajo SUN bajo Solaris 2.1 o SUN OS 4.1, en estaciones Hewllet Packard bajo HP UX 9.01, en otros sistemas que esten corriendo bajo UNIX System V Release 4; y en computadoras personales bajo Windows 3.1 o Windows NT. Una licencia de desarrollo de SNAP cuesta $20,000.00 dólares y las licencias para la ejecución de las aplicaciones estan en el rango de USD $500.00 a USD $1000.00.

---

[15] Del término *wide-area network*

[16] A lo largo de este capítulo para referirnos a esta empresa tan solo usaremos su primer nombre, es decir, Template.

[17] Traducción libre del término en inglés *template* (molde o patrón). Aun cuando en español la palabra templete no tiene este significado, la correspondencia entre los términos inglés y español es uno a uno, por lo cual, en este trabajo, se extiende la noción del vocablo a la de molde o patrón.

Una aplicación construida a partir de un templete permite el reuso a gran escala de diferentes piezas de software ahorrándonos esfuerzo ya que se tiene "capturada" (dentro del molde) la información básica recurrente para cualquier SI. Por ejemplo, un abogado usa templetes en la elaboración de documentos legales como testamentos o contratos debido a que gran parte del texto estándar empleado en ellos puede ser reusado. De ésta forma, el uso de un templete acelera la preparación, elimina errores de omisión e incrementa la calidad del producto final transformando lo complejo en rutina [17].

El templete de SNAP está enfocado a una amplia variedad de aplicaciones distribuidas, es por eso que soporta la arquitectura cliente-servidor y la arquitectura de redes pequeñas[11]. Además, proporciona facilidades para la implantación de gráficos orientados al usuario final, la comunicación entre procesos, el acceso a archivos planos y/o manejadores de bases de datos relacionales. Cuenta con un modelo de objetos que incorpora mecanismos de inferencia para la manipulación dinámica de eventos, fuentes de conocimiento, etc., y con una interfaz con software de aplicación externo.

A lo largo de este capítulo vamos a explorar cada uno de los componentes de SNAP a detalle con la finalidad de tener una panorámica global de la plataforma y así poder establecer puntos de comparación claramente definidos en capítulos posteriores.

3.1 El Templete de SNAP.

El diseño de SNAP soporta la forma más general de una arquitectura distribuida. Esto se debe a que una aplicación puede consistir de varios procesos corriendo sobre plataformas heterogéneas de hardware las cuales se encargan de administrar diferentes manejadores de bases de datos, resolver los problemas asociados al manejo de datos en tiempo real y de agrupar datos interactivos. Las capacidades necesarias para manejar la comunicación entre procesos y la

[11] En inglés *peer to peer*.

manipulación dinámica de eventos están construidas dentro de SNAP con lo que se permiten tantos niveles de interacción como sean necesarios.

El templete de SNAP es un *molde* empleado en la construcción rápida de aplicaciones de carácter crítico y de tipo distribuido que ofrece un conjunto de componentes de software de carácter genérico que por lo regular representan el 90% del código de una aplicación operacionalmente completa. La filosofía de trabajo del templete consiste en el "llenado" del molde proporcionado por SNAP ya que su arquitectura genérica cuenta con código predefinido para ser usado en la construcción de cada uno de los procesos que forman a una aplicación distribuida.

El templete de SNAP consiste de cinco componentes de software (véase la siguiente figura):

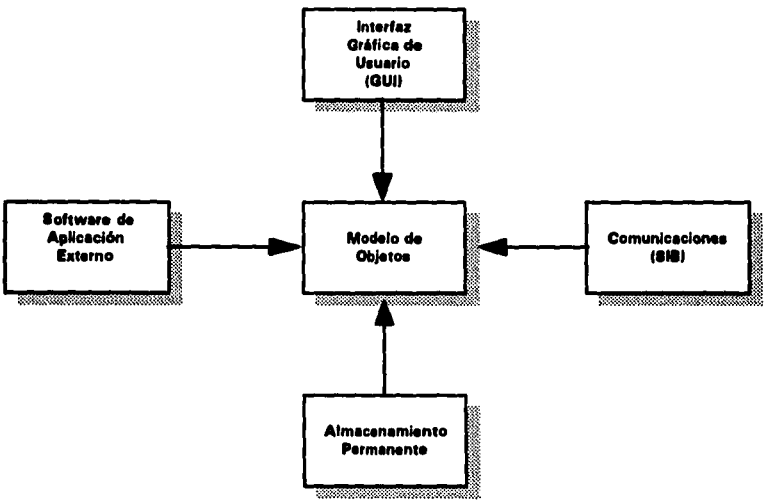

Figura 7. El templete de SNAP.

El componente principal es el modelo de objetos. En torno a él se encuentran:

- El componente dedicado a la interfaz gráfica de usuario (GUI)
- El componente de comunicaciones
- El componente de persistencia (denominado componente de almacenamiento permanente)

- El componente que sirve como interfaz de comunicación con software de aplicación externo

Cada uno de estos componentes juegan un papel importante en la generación de una aplicación operacionalemente completa. Debido a la arquitectura del templete, SNAP proporciona gran parte del código requerido para una aplicación (véase la figura 8), permitiendo que el desarrollador centre su atención en los detalles propios de la aplicación y no pierda su tiempo en la solución de problemas tediosos y complejos que nada tienen que ver con el desarrollo que está realizando (por ejemplo, detalles de programación de bajo nivel del ambiente gráfico de usuario).

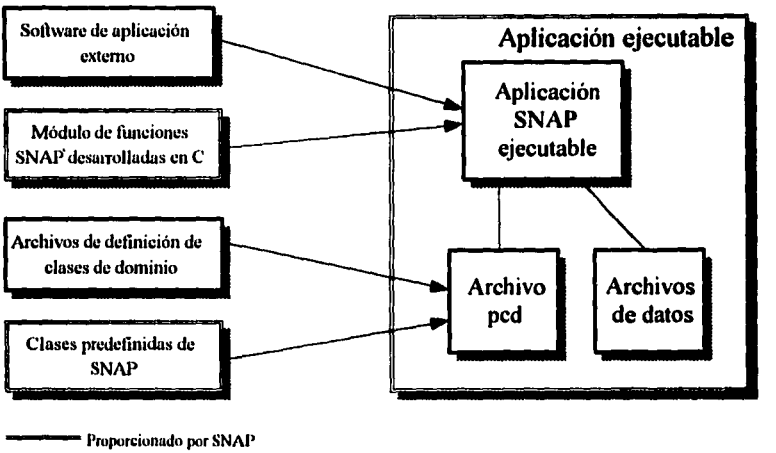

Figura 8. Arquitectura de una aplicación SNAP

3.2 El Ambiente de Desarrollo de SNAP.

El ambiente de desarrollo de SNAP (fig. 9) es una herramienta que hace fácil y rápida la creación y modificación de aplicaciones SNAP. Proporciona un conjunto de facilidades que permiten al desarrollador instanciar clases predefinidas en el templete de SNAP, así como definir las clases

que son específicas a la aplicación (*domain classes*[19]) que se está construyendo. Adicionalmente facilita una serie de herramientas para compilar y salvar el modelo de objetos, para rastrear errores (*debug*), para ver la definición de la aplicación (código), para ejecutar la aplicación, para realizar actividades de configuración, para establecer el mapeo del modelo de objetos a una base de datos y para generar la versión de usuario final de la aplicación.

Todas estas facilidades se accesan dentro del ambiente de desarrollo SNAP a través de los botones (tabla 1) o menús (tabla 2) que están en la ventana principal del ambiente.

Las facilidades proporcionadas en el ambiente de desarrollo incluyen:

- Comandos que permiten modificar la aplicación de una manera interactiva mientras se está ejecutando.
- Comandos que soportan actividades de administración de configuraciones durante el ciclo de desarrollo de la aplicación.
- Un editor que permite cambiar la forma en que la aplicación es construida.
- Un editor que permite crear y modificar las clases en la definición del modelo de objetos.
- Una serie de editores que permiten crear y modificar la interfaz gráfica de usuario.
- Un editor que permite crear y modificar la forma en que SNAP interactúa con las tablas de un RDBMS usado para resolver el problema de la persistencia de información.
- Un *browser* que permite observar las clases contenidas en la definición del modelo de objetos.
- Un *browser* que permite ver el código del software de aplicación externo.
- Un *browser* que permite crear, editar y ver a los miembros de clases predefinidas y de las clases de dominio.

[19]En español *clases de dominio*. Se debe establecer la diferencia entre las clases predefinidas en el templete de SNAP y las clases específicas de una aplicación. Cada vez que tengamos que hacer referencia a estas últimas el término *clases de dominio* será utilizado.

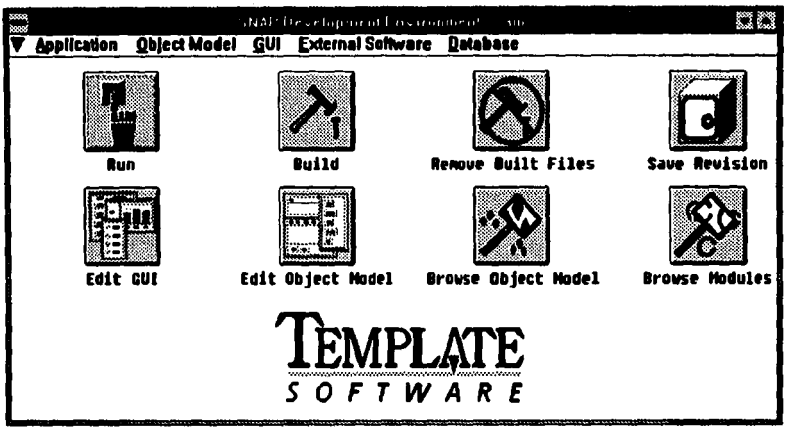

Figura 9. El ambiente de desarrollo de SNAP.

Nombre del Botón	Descripción
Run	Ejecuta la aplicación.
Build	Crea una versión de la aplicación que puede ser ejecutada.
Remove Built Files	Borra todos los archivos generados por el comando *Build* del directorio de la aplicación.
Save Revision	Guarda una versión específica de los componentes de definición de la aplicación que no fueron generados por el comando *Build*.
Edit GUI	Invoca el editor GUI.
Edit Object Model	Invoca el editor del lenguaje SNAP.
Browse Classes	Invoca el *Browser* (examinador) de las clases.
Browse Modules	Invoca el examinador de módulos.

Tabla 1. Botones dentro del ambiente de desarrollo SNAP

Nombre del Menú	Descripción
▼	Permite abandonar el ambiente de desarrollo SNAP.
Application	Proporciona los comandos para construir, ejecutar y administrar la aplicación SNAP.
Object Model	Provee los comandos para la creación, actualización e inspección de la definición del modelo de objetos. También permite la ejecución de la aplicación.
GUI	Proporciona los comandos para crear y adecuar la interfaz de usuario de la aplicación y de los monitores gráficos.
External Software	Permite inspeccionar los módulos de aplicación externos al modelo de objetos.
Database	Permite la definición del mapeo del modelo de objetos a las tablas de una base de datos relacional.

Tabla 2. Menús accesibles dentro del ambiente de desarrollo SNAP

3.3 El Modelo de Objetos.

En términos generales *un modelo de objetos es una representación abstracta del mundo real que permite el procesamiento de información.* En SNAP, la representación de los datos se realiza por medio del modelo de objetos, el cual sirve como punto de fusión entre la representación y el manejo de los datos específicos de la aplicación y las operaciones usadas para almacenar, manipular, desplegar e interpretar a los datos.

Un modelo de objetos manipula a los datos como objetos[20]. Un objeto tiene identidad, propiedades (atributos) y relaciones con otros objetos. Además tiene una serie de comportamientos (métodos[21]) que definen las capacidades de procesamiento del objeto. En el modelo de objetos de SNAP los objetos son agrupados en *clases* definidas por un conjunto de atributos y un grupo de métodos. La clase es la principal unidad de encapsulación en el modelo de objetos de SNAP y por ello las clases soportan los conceptos de modularización, encapsulación y ocultamiento de información de la siguiente manera:

Modularización:

[20] En SNAP a los objetos se les denomina *miembros* y para ser consitentes con la terminología de la plataforma de ahora en adelante así nos vamos a referir a este concepto.

[21] En SNAP a los métodos se les llama *funciones*

- Se tiene la capacidad para definir clases diferentes en *archivos de definición de clase* (class definition files[22]) diferentes.

Encapsulación:

- Nos da la capacidad para definir secciones dentro de una clase.
- Se pueden especificar las fuentes de conocimiento de una clase.
- Se tiene la capacidad para definir los métodos (funciones) de una clase.
- Se puede restringir el acceso a los elementos de una clase a través de la definición de su visibilidad: pública, protegida o privada.
- Hay la capacidad para definir elementos propios (*own*) que guardan datos sobre la clase como un todo y que actúan sobre la clase de la misma manera.

Ocultamiento de información:

- La definición pública, privada y protegida de los elementos de la clase permite que se especifique explícitamente la interfaz de la clase por separado de su implantación.

En SNAP se distinguen dos tipos diferentes de clases:

- Las clases de dominio del problema (*domain classes*) que son todas aquellas definidas por el desarrollador durante las fases de análisis y diseño de la aplicación. Están formadas por las siguientes secciones:
  - **Patrones (*patterns*):** Un patrón es un conjunto ordenado de construcciones que pueden contener caracteres, símbolos propios de la gramática para definir a un patrón, nombres de otros patrones y sus componentes. Cada construcción corresponde a una cadena de caracteres que va a ser empatada y puede o no tener un nombre definido. Aquellas construcciones que tienen nombre se les llama componentes y estos pueden ser extraídos fácilmente. Un patrón SNAP es un templete que puede ser usado para determinar si la cadena de caracteres que resulta de la evaluación de una expresión

[22] Un archivo de definición de clase (cdf por sus siglas en inglés) es un archivo que contiene el código en lenguaje SNAP para la definición de una o más clases de dominio. Se puede definir una o más cdf para una misma especificación.

empata una forma en particular. Si la verificación es satisfactoria, se pueden extraer fácilmente diferentes porciones del string.

- **Tipos (*types*):** Se usa la sección de tipos para definir tipos de datos abstractos específicos para cada aplicación. Cuando se declara un atributo se debe indicar el tipo al que pertenece, lo cual señala los posibles tipos de valores que el atributo puede tomar. La definición de un tipo está en función de catorce tipos predefinidos: *sgl, mlt, truth, int, list of int, real, list of real, fixed, list of fixed, str, list of str, member, member set,* y *pointer.* Se define un nuevo tipo en la sección *types* cuando hay varios atributos que comparten la misma lista de valores posibles. Por medio de este mecanismo nos liberamos de tener que repetir en el código los valores posibles de un atributo varias veces. El nombre del tipo definido por el desarrollador es usado cuando se declaran aquellos atributos que comparten la misma lista de valores posibles.
- **Atributos (*attributes*):** Los atributos son utilizados para representar los datos identificados en el dominio del problema. La sección de atributos (*attributes*) puede ser definida tanto en la clase global como en las clases de dominio. Los valores de los atributos son asignados en tiempo de ejecución y estos valores contienen datos específicos a una sesión específica del usuario final con el sistema. El formato para la declaración de un atributo consiste de un nombre para el atributo seguido de dos puntos, su visibilidad y el tipo al que pertenece (aquí es válido declarar tipos definidos por el usuario) y opcionalmente las cláusulas de restricción y de valor por omisión.
- **Funciones (*functions*):** Las funciones son usadas para agrupar lógicamente grupos relacionados de comandos que son frecuentemente usados. Las funciones pueden estar definidas ya sea en el lenguaje SNAP (internas) o en lenguaje C (externas a SNAP). Un tipo especial de función son los métodos para la ejecución de llamadas a procesos remotos (RPC) las cuales emplean las palabras reservadas *remote* para aquellas funciones que ejecutan la llamada y *exported* para aquellas que ejecutan el RPC.

- **Módulos externos a SNAP (*externals*):** Se usa la sección de **externals** cuando se desea comunicar a una aplicación SNAP con otra aplicación no necesariamente SNAP o cuando se desea ejecutar programas externos. La ejecución de un módulo externo se puede realizar a nivel de la sección de acciones, por medio de los procesos asíncronos o a nivel de usuario final (mediante un menú, por ejemplo).
- **Reglas (*rules*):** Se usan *reglas* para inferir los valores de atributos y los miembros de una clase. Una regla tiene la siguiente forma: ***si** antecedente **entonces** consecuente*. Un *antecedente* es un conjunto de condiciones (pruebas o comparaciones que involucran los valores de atributos o a las instancias de una clase). Un *consecuente* es un conjunto de acciones ejecutadas si el antecedente se cumple. Debido a que las reglas pueden ser usadas para determinar los valores de los atributos e instancias de una clase, éstas son fuentes de conocimiento.
- **Procesos asíncronos (*demons*):** Los procesos asíncronos se usan para realizar acciones basadas en eventos. SNAP usa a los *demonios* cuando ocurre un evento que ocasiona que el antecedente del proceso asíncrono se haga verdadero. A diferencia de las fuentes de conocimiento, los *demonios* no son usados cuando el valor de un atributo o una clase es requerido por la máquina de inferencia.

- La clase **global** que es un tipo especial de clase ya que tiene un nombre predefinido por el ambiente de desarrollo (ya que no es declarada explícitamente sino generada), no tiene miembros y no puede heredarse. El motivo de su existencia radica en que provee un área de intercambio y acceso común para todas las construcciones dentro de una aplicación. Esta clase puede tener las siguientes secciones adicionales a las enunciadas en el punto anterior:
  - **Constantes (*constants*):** Esta sección es útil para la declaración de aquellos valores estáticos de una aplicación.
  - **Texto (*text*):** Esta sección es empleada para la definición de información de ayuda para el usuario en formato de texto libre que puede ser accesada si se necesita.

- **Acciones (*actions*):** Se encarga de ejecutar los métodos de arranque de una aplicación SNAP.

El modelo de objetos de SNAP distingue tres tipos diferentes de métodos:

a) Los métodos invocados explícitamente

b) Los métodos manejados por eventos, y

c) Los métodos disparados por las fuentes de conocimiento

Los métodos invocados de manera explícita incluyen funciones internas (definidas en el lenguaje SNAP) y funciones externas (funciones que son invocadas en una aplicación SNAP pero escritas en lenguaje C). Tal y como se hace en los lenguajes tradicionales de programación, las funciones son usadas en aplicaciones SNAP para encargarse del tratamiento procedural de la información. La diferencia entre el uso tradicional de las funciones y el uso que se les da en SNAP radica en que en SNAP las funciones están encapsuladas dentro de las clases.

Los métodos manejados por eventos son empleados por la máquina de inferencia de SNAP o por el manejador de eventos cuando un acontecimiento sucede. Ejemplos claros de eventos pueden ser las acciones realizadas a nivel de la interfaz de usuario, los cambios en el valor de un atributo o miembro de una clase o aquellos eventos dependientes de ciertos intervalos de tiempo (*timer events*). Los dos tipos de métodos manejados por eventos que proporciona SNAP son: las funciones de respuesta a una llamada (*callback functions*) y los procesos asíncronos.

Este tipo de métodos son disparados siempre que uno o varios datos sufran algún tipo de transformación, es decir, el código asociado al método manejado por eventos será ejecutado cuando la guarda encargada de monitorear los cambios en los datos se da cuenta que estos han cambiado.

Los métodos basados en las fuentes de conocimiento son usados por el mecanismo de inferencia de SNAP cuando se intenta determinar el valor de una clase o de un atributo. Los métodos disparados por las fuentes de conocimiento proporcionados por SNAP son las llamadas a programas externos, las *reglas* y las cláusulas *default* útiles para asignar valores por omisión. Una regla expresa las condiciones bajo las cuales información conocida puede ser usada para inferir

información que no está determinada, esto es, son fuentes independientes de conocimiento que sirven para asignar miembros a una clase o valores a los atributos de una clase. Si la condición establecida en la regla es verdadera, entonces algún valor puede ser inferido. Una cláusula *default* es una palabra reservada que asigna una fuente de conocimiento por omisión a un atributo o una clase y que contiene una expresión que es evaluada durante el proceso de inferencia. La máquina de inferencia de SNAP puede explicar conclusiones (*findings*) y el razonamiento que las produjo, usando los datos específicos del caso.

3.4 El Componente dedicado a la Interfaz Gráfica de Usuario.

La arquitectura del componente GUI se ilustra en la figura 10. Como se puede observar, consiste de cuatro capas. En el nivel más bajo, el componente GUI de SNAP usa las interfaces estándar proporcionadas por el sistema de administración de ventanas que soporta. La Biblioteca WinTool normaliza las diferencias entre la mayoría de sistemas de ventanas soportados para lograr una interfaz común de programación.

La capa de unidades de despliegue genéricas provee un conjunto de implantaciones generales para un grupo común de unidades de despliegue de información de alto nivel que incluye tablas, gráficas de línea, gráficas de barras, diagramas de Gantt, topologías de red, así como ventanas de tipo multipanel (*multipane*) y de lienzo (*canvas*).

Finalmente, la capa del Generador de unidades de despliegue hace posible producirlas en tiempo de ejecución a partir de los datos proporcionados por los objetos de las clases del GUI.

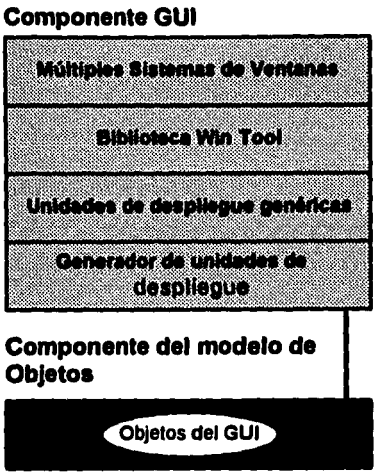

Figura 10. La arquitectura del componente GUI de SNAP.

El ambiente de desarrollo de SNAP ofrece una serie de editores para que se pueda crear fácil y rápidamente la interfaz gráfica de usuario (GUI). El GUI es el medio por el cual una aplicación presenta información e interactúa con el usuario final. SNAP provee de un conjunto de clases predefinidas que auxilian en la creación de los elementos que constituyen la interfaz con el usuario. Usando los editores gráficos de SNAP se puede rápidamente agregar miembros o adecuar las clases predefinidas en el GUI de SNAP de acuerdo a las necesidades propias de cada aplicación en particular.

El GUI de una aplicación SNAP esta basado en los conceptos de **ventana** y de unidades de despliegue (*displays*). La ventana es la unidad fundamental sobre la cuál todas las unidades de despliegue GUI son construidas, esto es, toda unidad de despliegue de una aplicación SNAP aparece en una ventana. Las ventanas ofrecen otras características adicionales, como son los menús, botones, iconos, etc., por medio de los cuales el usuario interactúa con la aplicación. Por lo regular los usuarios accesan las facilidades que tiene una aplicación por medio de menús disponibles en la ventana principal de la misma. Estos menús les permiten ejecutar comandos o accesar otras ventanas donde más opciones de procesamiento se encuentran disponibles.

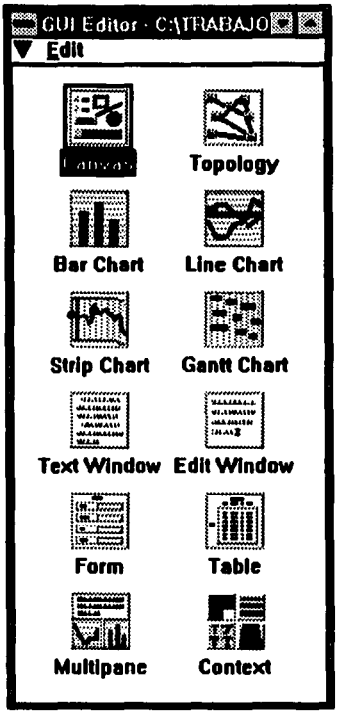

Figura 11. El editor GUI de SNAP.

Dentro de una aplicación, las unidades de despliegue (gráficas de barra, gráficas de líneas y diagramas GANTT) son útiles en la presentación de información del negocio, es decir, muestran un conjunto de datos pertenecientes al dominio del problema. Los tipos básicos de unidades de despliegue que se pueden construir y sus clases predefinidas en el GUI de SNAP se enumeran en la tabla 4. Estas unidades de despliegue y sus posibles combinaciones son usados en la creación de la interfaz gráfica de usuario de la aplicación.

Tipo de unidad de despliegue (*display*)	Nombre de la clase
Objetos Gráficos *ad-hoc*	CANVAS
Topologías	TOPOLOGY
Gráficas de barras	BAR_CHART
Gráficas de líneas	LINE_CHART
Gráficas de franjas	STRIP_CHART
Diagramas de GANTT	GANTT_CHART
Ventana de Texto	TEXT_WINDOW
Ventana de edición	EDIT_WINDOW
Formas	FORM
Tablas	TABLE
Unidad de despliegue que contiene a otras unidades de despliegue	MULTIPANE

Tabla 3. Los tipos de unidades de despliegue predefinidos en SNAP

Una unidad de despliegue y sus características se crea o modifica por medio de uno o más editores a los que se tiene acceso en la ventana del editor GUI de SNAP. Los tres pasos básicos para definir a una unidad de despliegue son los siguientes:

1. **Crear un miembro de una clase GUI usando cualquiera de los editores para las diferentes unidades de despliegue disponibles.**

   Cada unidad de despliegue es un miembro de una clase GUI. Cuando se agrega un miembro, el editor para ese tipo de unidad de despliegue en particular solicita la información requerida para definirla.

   La información que es solicitada depende del tipo de unidad de despliegue que se esté definiendo. El editor solicita la información mínima que se requiere para que se pueda ver en el menor tiempo posible. La mayoría de los atributos que definen a una unidad de despliegue tienen valores por omisión, los cuales son usados por el editor para crear la versión inicial de la unidad.

2. **Modificar la unidad de despliegue al cambiar los datos en el GUI y/o los datos de dominio.**

   Se usa el mismo editor que se empleó en la creación de la unidad de despliegue para modificar la apariencia, contenido y comportamiento de la unidad.

   A través del *browser* de SNAP (véase la fig. 12) se puede modificar la apariencia de la unidad de despliegue modificando los datos de dominio. Por lo regular cuando se está desarrollando una unidad de despliegue se usan datos de prueba para ver como será la apariencia de ésta. Es útil que se puedan alterar los datos de dominio sobre la marcha para observar como responde el unidad de despliegue con estos cambios y así determinar si el comportamiento del mismo es el adecuado.

3. **Integrar la unidad de despliegue a la aplicación.**

   Se usan los editores de las unidades de despliegue que sean necesarios para definir ya sea una opción de menú, botón, o cualquier otro tipo de unidad de despliegue que invoque a la unidad que se ha creado. En el lenguaje de SNAP se definen las funciones o mecanismos pertinentes para ejecutar la presentación de la unidad de despliegue.

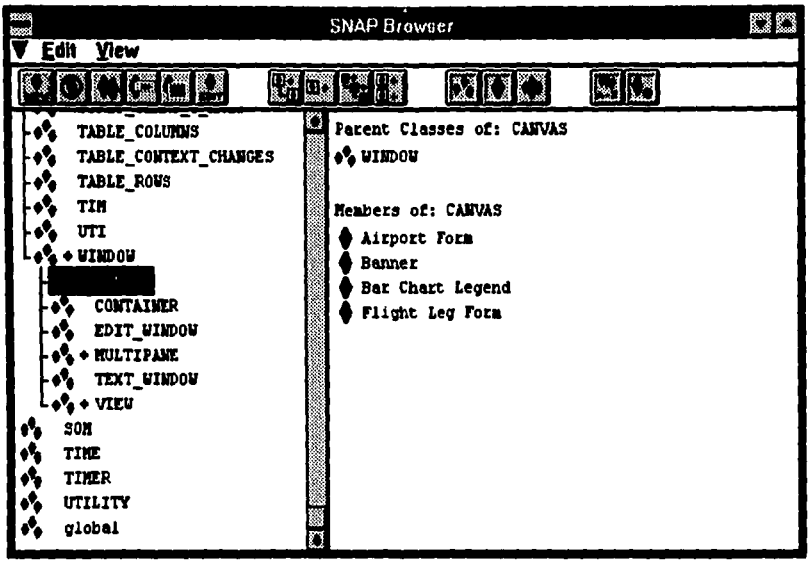

Figura 12. El browser de SNAP.

3.5 El Componente de Almacenamiento Permanente.

Toda aplicación usa algún medio de almacenamiento masivo (también denominado almacenamiento permanente) para salvar o recuperar aquellos datos que deben persistir más allá de su ejecución. Una aplicación desarrollada en SNAP no es la excepción y es por esto que Template provee en el ambiente un componente dedicado a resolver el problema de la persistencia de información.

La persistencia desde el punto de vista de SNAP puede tomar dos formas diferentes: mediante el uso de un RDBMS con interfaz SQL o por medio de archivos planos accesados secuencialmente. El importar o exportar datos vía alguna de estas facilidades permite al desarrollador un mecanismo por medio del cual la información puede preservarse entre sesiones

de trabajo con una aplicación SNAP o ser accesada por otras aplicaciones desarrolladas o no en SNAP.

La habilidad para poder accesar bases de datos relacionales y archivos planos también permite la fácil integración de aplicaciones SNAP dentro de ambientes de producción ya existentes. El acceso a RDBMS se hace mediante el uso de los métodos proporcionados por la clase DBL. La interacción con archivos planos es a través del módulo FIO.

El módulo DBL.

El módulo DBL contiene un conjunto de funciones en lenguaje C que se encargan de encapsular toda la interacción de bajo nivel con el RDBMS. Estas funciones también están declaradas en el archivo definición de clase *dbmap.cd* de tal forma que puedan ser accesadas desde el modelo de objetos usando el formato de referenciación del lenguaje SNAP (usando dobles dos puntos, ***DBL::función***) o desde un programa C usando el formato ***DBL_función*** siempre y cuando se haya incluido el header *dbl.h* y se haya ligado la biblioteca *libdtool.a*. Estas funciones proporcionan las siguientes capacidades:

- Iniciar y terminar la sesión con la base de datos
- Cargar la información de la base de datos al modelo de objetos de SNAP
- Actualizar un modelo de objetos de SNAP con información de la base de datos
- Actualizar a una base de datos con información de un modelo de objetos SNAP
- Insertar registros en la base de datos con información de un modelo de objetos SNAP
- Ejecutar comandos SQL.

El componente DBL provee un mecanismo flexible para **mapear** (es decir, el establecer la correlación entre clases del modelo de objetos y tablas en la base de datos) e intercambiar datos entre un RDBMS y un modelo de objetos de SNAP. A excepción de los procesos de inicialización y terminación de la sesión con la base de datos, la interfaz es independiente del RDBMS que se esté usando. Por ende, la portabilidad entre RDBMSs es fácilmente lograda. El mapeo entre las

clases de un modelo de objetos SNAP y las tablas en una base de datos relacional puede alterarse en tiempo de ejecución haciendo la implantación aún más flexible.

La definición de un mapa.

En una base de datos relacional, los datos son almacenados en tablas consistentes de columnas y renglones [4], [14], [15]. En SNAP, los datos son representados como clases, miembros y valores de los atributos de un miembro de una clase. Cada renglón (o registro) en una tabla de la base de datos corresponde a un miembro de una clase del modelo de objetos de SNAP. Las columnas en una base de datos corresponden a los atributos en una clase SNAP mientras que los valores de los campos en una columna de una tabla corresponden a los valores de los atributos en una clase.

Para transferir datos entre una tabla y una clase, a SNAP se le debe decir que columnas de la base de datos corresponden a que atributos de que clase, es decir, la tabla debe ser **mapeada** a una clase. Por consecuencia, *un mapa define la correspondencia entre una base de datos y un modelo de objetos.*

El mapeo entre una base de datos relacional y un modelo de objetos SNAP se realiza usando el archivo de definición de clase *dbmap.cd*. El mapeo consiste de cuatro partes:

- Identificación de la tabla relacional
- Identificación de la clase
- Correspondencia entre columnas y atributos
- Una descripción del "nombre del miembro"

Para poder definir el mapeo entre una base de datos y un modelo de objetos, se debe crear un miembro de la clase DBMAP para cada uno de los mapeos que se van a crear y entonces, para cada miembro los valores de los atributos que especifican el mapa son asignados. Para cada par clase/tabla se debe especificar un nuevo miembro de la clase DBMAP.

Como nuevos miembros de clase son agregados al modelo de objetos como resultado de importar los datos de la base de datos, los nombres de los miembros son creados usando los valores correspondientes a las columnas definidas por el atributo *memberkey*. El atributo

memberkey es una lista campos de la base de datos separada por comas y para asegurar que los nombres de los miembros sean únicos, por lo regular se especifica la *llave* de la tabla en la BD como el *memberkey* del modelo de objetos.

Las funciones que ofrece el módulo DBL para interactuar con la base de datos se enumeran en la tabla 4.

DBL::InitOracle	Inicia la conexión con la base de datos Oracle. Regresa *true* si la conexión fue exitosa.
DBL::InitInformix	Inicia la conexión con la base de datos Informix. Regresa *true* si la conexión fue exitosa.
DBL::InitInterbase	Inicia la conexión con la base de datos Interbase. Regresa *true* si la conexión fue exitosa.
DBL::InitSybase	Inicia la conexión con la base de datos Sybase. Regresa *true* si la conexión fue exitosa.
DBL::InsertPartialRecord	Inserta un registro en la base de datos. Solo aquellos atributos que fueron especificados se escriben al nuevo registro.
DBL::InsertRecord	Inserta un registro en la base de datos. Todos aquellos atributos que fueron especificados en el mapa son escritos en el nuevo registro.
DBL::LoadClass	Llena una clase a partir de la base de datos. Cada atributo de la clase es actualizado a partir del campo de la base de datos especificado en el mapa.
DBL::LoadPartialClass	Igual que en el método anterior con la variante de que solo aquellos atributos que se especificaron son los que se actualizan.
DBL::UpdateClass	Actualiza una clase a partir de la BD. No se crean miembros nuevos.
DBL::UpdatePartialClass	Actualiza una clase a partir de la BD. Solo los atributos especificados son actualizados.
DBL::WritePartialRecord	Escribe un registro en la base de datos. Solamente aquellos atributos que hayan sido especificados son escritos a los atributos definidos en el mapa. La cláusula *where* restringe al registro que se escribe en la BD.
DBL::WriteRecord	Escribe un registro en la base de datos. Todos los atributos mapeados son escritos en los campos de la BD como fueron definidos por el par clase/tabla. La cláusula *where* restringe al registro que se escribe en la BD.

Tabla 4. Funciones ofrecidas por la clase DBL.

DBL::Change Class	Actualiza los miembros de una clase que empaten las cláusulas **where** y **order** en la base de datos. Requiere que se haya definido un *memberkey* en el mapa.
DBL::ChangePartialClass	Actualiza los miembros de una clase que empaten las cláusulas **where** y **order** en la base de datos. Requiere que se haya definido un *memberkey* en el mapa. Sólo son actualizados los atributos que hayan sido especificados.
DBL::Command	Ejecuta un comando SQL que haya sido especificado. El único comando no válido es el *select*. Regresa el estatus de la operación.
DBL::ExitInformix	Desconecta la sesión de trabajo con el RDBMS Informix.
DBL::ExitInterbase	Desconecta la sesión de trabajo con el RDBMS Interbase.
DBL::ExitOracle	Desconecta la sesión de trabajo con el RDBMS Oracle.
DBL::ExitSybase	Desconecta la sesión de trabajo con el RDBMS Sybase.

Tabla 4. Funciones ofrecidas por la clase DBL (continuación).

El módulo FIO.

El módulo FIO es usado para leer y escribir datos a partir de archivos planos ASCII. Dos tipos de archivos ASCII son necesarios para poder leer o escribir en archivos planos. El primero es un archivo de formato el cual define la clase, los atributos y el tamaño del campo de los atributos. El segundo es en sí el archivo de datos, el cual contiene los datos de los atributos formateados de acuerdo a lo descrito por el archivo de formato.

La lectura o escritura a partir de una clase requiere de pares de archivos de formato y de información separados para cada una de estas operaciones y cada clase en el modelo de objetos requiere de la definición de estos archivos.

Un archivo de formato está estructurado de la siguiente manera:

Línea #:	Contenido:
1	<Nombre de la Clase>
2-n	<Nombre de Atributo>:<Tamaño del campo>

Las líneas a partir de la 2 en adelante en un archivo de formato para lectura de información pueden contener tan solo el <Tamaño del campo>. Esto ocasiona que el campo (de hecho, el número de caracteres especificado) en el registro del archivo de datos sea saltado. En un archivo

de formato para escritura es necesario indicar el nombre del atributo y su tamaño. Los tipos de los atributos están limitados a *str*, *int* y *real*.

Debido a la simplicidad de los mecanismos del módulo FIO son extremadamente sencillos, este módulo solo cuenta con dos funciones: FIO::ReadClass() que se encarga de leer la información de acuerdo a la especificación contenida en el archivo de formato para lectura y la función FIO::WriteClass() que se encarga de escribir la información de acuerdo a la especificación contenida por el archivo de formato para escritura.

3.6 El Componente de Comunicaciones.

Uno de los componentes fundamentales de la plataforma es el componente de comunicaciones. Por medio de este se define la arquitectura de una aplicación distribuida. Como se menciona en la introducción de este capítulo, SNAP permite dos modelos de comunicación interprocesos (*interprocess communication*) en una aplicación. Antes de definir que es la comunicación entre procesos, es conveniente definir el concepto de proceso.

> Un **proceso** es un programa que es ejecutado por el sistema operativo de una computadora

Cuando dos procesos necesitan comunicarse entre sí, surge la necesidad de contar con las facilidades necesarias para implantar la comunicación entre procesos. De esta manera podemos establecer la siguiente definición para el concepto de comunicación interprocesos:

> Por **comunicación interprocesos** se debe entender a cualquier mecanismo por medio del cual diferentes procesos pueden comunicarse entre sí, sin importar que estos procesos se encuentren en la misma computadora o en diferentes computadoras.

Por lo regular, dos procesos se comunican entre si a través de una **conexión**. La conexión permite que la información fluya de un proceso a otro. Una conexión se distingue de otras conexiones porque está identificada por el nombre del *host* y la dirección del proceso al cuál otro proceso se está comunicando. La conexión entre dos procesos se establece cuando un proceso

solicita conectarse a otro. Al proceso que inicia la conexión se le conoce como **proceso local** y al proceso que acepta la conexión se le denomina **proceso remoto**.

SNAP maneja dos modelos de comunicación interprocesos: el modelo *peer to peer* y el modelo *cliente-servidor*. En un modelo *peer to peer* la relación entre procesos es simétrica, esto es, cada proceso es un recurso compartido por los demás procesos procesos participantes. Una característica importante de este modelo de comunicaciones es que no se cuenta con una base de datos centralizada, es decir, la información que requieren los procesos entre sí es proporcionada por cada uno de ellos.

En el modelo *cliente-servidor*, un proceso (el proceso **servidor**) espera a ser contactado por otro proceso (el proceso **cliente**) para poder realizar alguna tarea de servicio para el cliente. Este modelo es usado cuando una aplicación requiere de la existencia de un proceso que controle a una base de datos centralizada y sea capaz de atender las solicitudes de información de otros procesos. Por ser éste el modelo de comunicaciones más empleado y porque con él se pueden realizar llamadas a procedimientos remotos (RPC[23]), solamente vamos a discutir como se implantaría una aplicación SNAP de esta naturaleza.

En el modelo de cliente-servidor, la base de información compartida (SIB[24]) es la facilidad que se emplea para establecer las relaciones de clases de datos compartidas, la manera en que se propagan las actualizaciones de los datos y el tipo de comunicación que se establece.

Una clase compartida en SNAP incluye la palabra reservada *shared* en su definición dentro del modelo de objetos. Las clases compartidas son útiles cuando se requiere que la información del cliente y del servidor sea exactamente igual. En la definición del modelo de objetos del cliente y del servidor, las clases compartidas deben aparecer en el mismo orden jerárquico y los atributos compartidos de la clase (declarados en el archivo plano del SIB) deben ser del mismo tipo y con igual nivel de visibilidad.

---

[23] por sus siglas en inglés *Remote Procedure Call*.

[24] por sus siglas en inglés *Shared Information Base*.

El SIB se apoya en un archivo plano que al ser leído asigna valores a las clases *SIB CONNECTION, TRANSFER* y *SIB CONNECTION ACCEPTER.*

A continuación presentamos el formato de un archivo plano empleado por el SIB para un proceso servidor:

```
\Descripción del Server
SIB CONNECTION ACCEPTER:Server>Address Name = "WEC5".
SIB CONNECTION ACCEPTER:Server>Local Name = "SERVER".
SIB CONNECTION ACCEPTER:Se,ver>Proto Auto Gets = unknown.
SIB CONNECTION ACCEPTER:Server>Proto Auto Sets = unknown.
SIB CONNECTION ACCEPTER:Server>Proto Initial Gets = unknown.
%
```

Debido a que un proceso servidor acepta conexiones de los procesos cliente, la clase que debe ser instanciada en *SIB CONNECTION ACCEPTER.* Los atributos de esta clase definen el socket de comunicaciones, el nombre local del proceso servidor y tres atributos adicionales que definen las actualizaciones automáticas a los datos del servidor cuando el modelo de objetos cambie en el cliente (*Proto Auto Gets*), las actualizaciones al modelo de objetos del cliente que se tienen que enviar automáticamente cuando el modelo de objetos del servidor cambie (*Proto Auto Sets*) y los datos que tienen que viajar automáticamente cuando se establece una conexión por primera vez (*Proto Initial Gets*).

El archivo plano SIB de un proceso cliente se muestra a continuación:

```
\ SIB-related attribute values
\ Operator's connection to the server
SIB CONNECTION:Oper>Host Name = "HAASC1".
SIB CONNECTION:Oper>Local Name = "md".
SIB CONNECTION:Oper>Address Name = "WEC5".
SIB CONNECTION:Oper>Auto Gets =
  SIB CLASS TRANSFER:StatusOp

SIB CONNECTION:Oper>Initial Gets =
  SIB CLASS TRANSFER:StatusOp

SIB CONNECTION:Oper>Auto Sets =
  SIB CLASS TRANSFER:Detalle Liquidacion

\ Transfers

SIB CLASS TRANSFER:StatusOp>Asynchronous = false.
SIB CLASS TRANSFER:StatusOp>Class Name = "STATUS OPS".
SIB CLASS TRANSFER:StatusOp>Member Atrs = unknown.
%
```

Este fragmento del archivo no contiene a todas las clases declaradas realmente en la aplicación. Decidimos reducirlo para explicar brevemente su contenido. Los atributos de la clase *SIB CONNECTION* a los que se les asigna un valor son: *Host Name*, *Local Name*, *Addres Name*.

Host Name define el nombre que tiene asignado en la red el proceso servidor, *Local Name* define el nombre del cliente, *Address Name* define el socket de comunicaciones a emplear. Los atributos *Auto Gets*, *Initial Gets* y *Auto Sets* se apoyan en las instancias definidas para la clase *SIB CLASS TRANSFER* cuyos valores son los nombres de las clases que son compartidas por el cliente y el servidor. En este ejemplo solamente se declara a la clase *StatusOp* pero puede haber tantas declaraciones como clases compartidas haya en la aplicación.

La clase *SIB CLASS TRANSFER* tiene una serie de atributos a los que se les debe asignar valores. Estos atributos definen si el tipo de comunicación es síncrona o asíncrona, el nombre de la clase que va a viajar del o hacia el servidor y los atributos de los objetos (*Member Atrs*) que van a estar viajando entre el servidor y el cliente.

De la administración del esquema de comunicaciones se encarga SNAP, liberando con ello al desarrollador de la tediosa tarea de la administración de estas funciones.

Adicionalmente el modelo *cliente-servidor* permite la declaración de funciones RPC. Para poder explotar ésta facilidad se debe incluir este tipo de funciones en clases que son compartidas. El formato de una función RPC se caracteriza por incluir las palabras reservadas *remote* o *exported*. El proceso local es el que va a usar la palabra reservada *remote* con lo que se indica que hace una llamada a un procedimiento remoto. El proceso remoto declara a la función RPC con la palabra reservada *exported* indicando con esto que la función es llamada desde otro proceso. El cuerpo de la función en el proceso remoto define el tipo de procesamiento que se va a realizar sobre los datos y una vez que ha finalizado envía al proceso local el resultado de estas computaciones.

3.7 La Interfaz con Software de Aplicación Externo.

El modelo de objetos de una aplicación SNAP es especificado usando el lenguaje propietario de Template. Esta especificación puede ser creada usando el ambiente de desarrollo de SNAP o bien desde un editor de textos. El software externo escrito en un lenguaje de tercera generación (3GL) puede accesar al modelo de objetos usando las funciones predefinidas en la clase *SOM* (*SNAP Object Model Module*) que forma parte de las bibliotecas proporcionadas por Template.

La razón principal que justifica el uso de estas funciones es la de proporcionar una interfaz eficiente entre un programa 3GL (por lo regular código C) y el modelo de objetos de una aplicación. Adicionalmente estas funciones extienden al lenguaje de SNAP ya que proporcionan funcionalidad adicional que no es disponible usando solamente los comandos propios del lenguaje.

La interfaz con software de aplicación externo ofrece dos niveles de acceso al modelo de objetos: el nivel básico y el nivel avanzado. Estos dos niveles difieren en el número de funciones ofrecidas, el poder de manipulación proporcionado por estas y en la complejidad de los tipos de datos y de las funciones. Cada nivel contiene una biblioteca y un grupo de funciones de *callback*. En cada uno de estos niveles, diferentes tipos de datos son usados.

El nivel básico.

El nivel básico proporciona las funciones más rudimentarias y, por consecuencia, es el más fácil de usar ya que nos da un grado de acceso similar al que tiene el usuario final; las funciones disponibles a este nivel proporcionan la capacidad para ejecutar comandos en tiempo de ejecución (*run-time*). Este nivel incluye funciones básicas útiles para cargar la especificación parseada (*parsed specification*), ejecutar la sección de *acciones*, y ejecutar comandos del lenguaje de SNAP. Toda entrada y salida de un programa externo que use este nivel es en forma de cadenas de caracteres y por lo regular el uso del nivel básico es apropiado cuando se va a encomendar la tarea de escribir un programa externo a un desarrollador novato en C.

Debido a su orientación, el nivel básico contiene solo ocho funciones: cinco funciones de biblioteca y tres de callback. Estas ocho funciones son todo lo que se requiere para realizar tareas

básicas tales como iniciar y detener una aplicación, ejecutar comandos SNAP, desplegar mensajes o ingresar información a la aplicación.

Cada nivel usa diferentes tipos de datos. Los tipos de datos que corresponden al nivel básico son los siguientes:

boolTP
SOM_cmdKindTP
SOM_errTP
SOM_MsgKindTP
stringTP

El nivel Avanzado.

El nivel avanzado provee acceso a datos mucho más específicos con respecto al nivel básico, es decir permite el acceso a todas las partes de la aplicación. El uso de este nivel requiere que la tarea sea encomendada a un desarrollador experto en C; siempre y cuando se desee manipular información de la aplicación que es más *específica* que *disponible* a través del lenguaje SNAP. Por otro lado, si lo que se desea es poder reusar un programa C con otras aplicaciones, el nivel avanzado proporciona las funciones necesarias para obtener información del modelo de objetos (por ejemplo, poder conocer el número de atributos de una clase).

La habilidad para obtener información sobre el modelo de objetos de una aplicación permite escribir programas de acceso y manipulación independientes de cualquier aplicación.

Las funciones en el nivel avanzado usan manejadores (*handles*). Un manejador "apunta" o hace referencia a un elemento o secuencia de elementos en la definición de una aplicación, o a un *valor parseado*. Un elemento es un componente que tiene nombre dentro de la definición de una aplicación SNAP, por ejemplo, un atributo, una clase, un proceso asíncrono, una regla, un valor posible, etc. Un *valor parseado* es una representación interna de un valor SNAP. Una vez que se ha creado un manejador, se pueden realizar múltiples operaciones sobre los elementos o valores parseados que son apuntados por este.

Un manejador es siempre único, esto es, se acota a un solo elemento y siempre hace referencia al mismo hasta que:

- El elemento sea borrado. El único elemento que puede ser eliminado es un miembro de una clase. Si el miembro es eliminado, entonces el manejador puede o bien apuntar a *null* o puede ser reasignado a otro elemento. Ambas posibilidades pueden generar errores y para evitarlos no se debe reusar un *handle* después de que se eliminó el miembro al que apuntaba.
- Se realice la siguiente llamada al método *SOM_AtrValParse()*, con la cual el manejador hace referencia al nuevo valor parseado que ha sido creado.
- Se realice la siguiente llamada al método *SOM_ClassParseMembers()*, en cuyo caso el *handle* hace referencia al nuevo valor parseado que ha sido creado.

De esta manera, con el uso de las funciones avanzadas se pueden accesar los elementos de una aplicación SNAP de una manera más elegante y eficiente, ya que el uso de los manejadores evita el tener que convertir un elemento SNAP en una cadena de caracteres lográndose con esto un acceso directo sobre el mismo.

Las funciones proporcionadas por el nivel avanzado son agrupadas en seis categorías:

- Funciones de acceso a elementos
- Funciones de parsing
- Funciones de prompt
- Funciones de comando
- Funciones de despliegue de comandos
- Funciones descriptoras de conversión entre *handles* y atributos

CAPÍTULO 4

NEXTSTEP

Introducción.

NeXT Computer, Inc. desarrolla y vende NEXTSTEP el sistema operativo orientado a objetos que se ha consolidado en la industria de la informática como uno de los productos más innovadores ya que revolucionó la mecánica de desarrollo de aplicaciones de carácter estratégico para los negocios. Su valor radica en la simplificación y aceleración del desarrollo de aplicaciones complejas *cliente-servidor* ya que ofrece el primer esqueleto de aplicaciones (*framework*) orientado a objetos de la industria para computación distribuida.

NeXT fue fundada en 1985 por Steven P. Jobs y otros cinco gerentes de Apple Computer, Inc. En Septiembre de 1989, NeXT introdujo la computadora NeXT y el ambiente de desarrollo NEXTSTEP V.1.0.

Una computadora NeXT consistía de tres componentes de hardware propietario (véase la fig.13): la computadora Next (conocida como el "cubo" por la ausencia de indicadores luminosos en su armazón), el monitor de alta resolución denominado *MegaPixelDisplay* y una impresora laser. La configuración típica del cubo incluía un disco duro de 660 Mb, un lector/escritor para disco óptico de 256 Mb y 16 Mb de memoria principal.

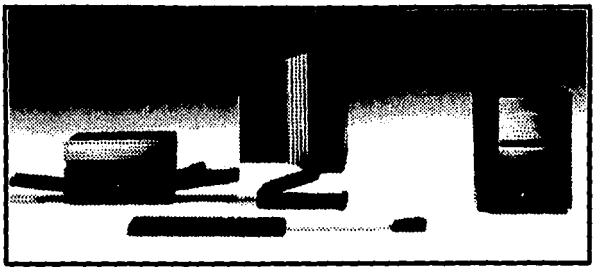

Figura 13. La computadora NeXT.

La concepción original de la computadora NeXT seguía la tendencia que en la década de los 70 pretendía eliminar todo indicador luminoso del hardware, es por esto que no había switches o perillas en el sistema (salvo el requerido para encender el CPU). El encendido de otros periféricos, el control de la brillantéz, volúmen y reset del sistema eran controlados desde el teclado de la computadora.

NeXT dota al cubo (tanto en el hardware como en el software) de los mecanismos necesarios para poder conectarse a cualquier red de computadoras bajo Ethernet. Como parte del hardware, la computadora NeXT contaba con una interfaz SCSI (siglas de Small Computer Standard Interface[25]). Con esta interfaz se tenía la capacidad de conectar hasta siete dispositivos de este tipo en una configuración denominada "encadenamiento margarita" (*daisy chain*). Como cualquier otra computadora, el cubo contaba con dos puertos seriales basados en el estándar RS-422 con los cuales se lograban velocidades de transmisión de 50 Kilobaudios (5000 bytes por segundo). Adicionalmente, toda computadora NeXT incluía en chip DSP56001 de Motorola. Este procesador digital de señales era el encargado de la generación de los sonidos y música sintetizada que el cubo podía generar. Por su versatilidad, este circuito contaba con su propio conector serie y por medio de este puerto, el DSP se podía comunicar con dispositivos seriales tales como un fax o un modem de alta velocidad; o con equipo que fuera útil en el muestreo y digitalización de sonidos.

La computadora NeXT representaba por sus características una buena alternativa y debido al gran éxito comercial que logra NeXT, ese mismo año Canon Inc. decide invertir en la compañía.

---

[25] Interfaz estándar para computadoras pequeñas

Versiones posteriores del software fueron desarrolladas y vendidas con los cubos NeXT y en 1990 la familia de productos NeXTstation fue introducida al mercado. En Febrero de 1993, cerca de la fecha de liberación de la primera versión de NEXTSTEP para procesadores Intel, NeXT anuncia que va a dejar de producir su hardware propietario y que se va a convertir en una compañía de software cuyo objetivo central era el lograr que NEXTSTEP se convirtiera en el estándar de la industria para computadoras basadas en procesadores Intel.

El sistema operativo NEXTSTEP (versión 3.2) corre sobre computadoras basadas en procesadores Motorola e Intel de 32-bits y se le ha considerado como un ambiente poderoso y fácil de usar. Sus puntos más fuertes son la interfaz de usuario y el ambiente de desarrollo.

El desarrollo y liberación de aplicaciones usando el ambiente NEXTSTEP es significativamente más simple y rápido que con otros ambientes de computación. Desafortunadamente para NeXT esta sofisticación técnica se ha logrado a costa de un grave distanciamiento con los estándares de la industria (por ejemplo Xwindows, C++, etc.).

Debido a que NeXT implantó sus sistemas antes de que los estándares *de facto* se fueran estableciendo, el sistema está escrito en Objective-C, lenguaje que no es tan popular como C++ o Smalltalk (aún cuando Objective-C es un híbrido entre C y Smalltalk), el sistema operativo base es Mach 2.5 en vez del más reciente Mach 3.0 microkernel y la interfaz gráfica (o ambiente de ventanas) de NEXTSTEP no sigue a ninguno de los GUI ya establecidos por la industria.

NEXTSTEP viene en dos versiones diferentes: la versión para usuario final y la versión para el desarrollador. NEXTSTEP Devoloper es el producto que se emplea para el desarrollo de aplicaciones cliente-servidor el cual incluye las herramientas de desarrollo *ApplicationKit*, *Project Builder*, *Interface Builder* y *Enterprise Objects Framework*.

Los requerimientos de hardware para ejecutar NEXTSTEP en una computadora personal son los siguientes: Un procesador 486 a 33 Mhz o superior, 16 MB o más de memoria principal (32 Mb recomendado), un disco duro de 500 Mb o superior (1 Gb recomendado) y CD-ROM SCSI, un monitor SVGA con 2 Mb de memoria de video (local bus recomendado) y el hardware necesario para trabajo en red con Ethernet. Los requerimientos de hardware para las estaciones de trabajo

son similares. Una licencia de desarrollo (NEXTDEVELOPER) cuesta al rededor de USD $2000.00 y cada una de las licencias de usuario USD $800.00.

El ambiente de desarrollo puede dividirse en tres componentes básicos: el sistema operativo (que provee de un ambiente operativo completamente multitarea), las bibliotecas de clases escritas en Objective-C y el ambiente de ventanas (que usa una maquinaria de Postcript Extendido).

A continuación discutiremos a cierto detalle cada uno de estos componentes para establecer un marco de referencia general de la plataforma.

4.1 El sistema operativo Mach.

El sistema operativo Mach fue diseñado originalmente para ser un microkernel. Sin embargo, la versión 2.5 de Mach que usa NEXTSTEP está íntimamente ligada con un server BSD UNIX de arquitectura monolítica. Desde un punto de vista de programación, Mach 2.5 es un BSD UNIX con *threads* y un mecanismo adicional para lograr la comunicación entre programas denominado *Mach ports*. En contraste, la versión 3.0 de Mach es un microkernel real que puede soportar múltiples servidores o *personalidades*.

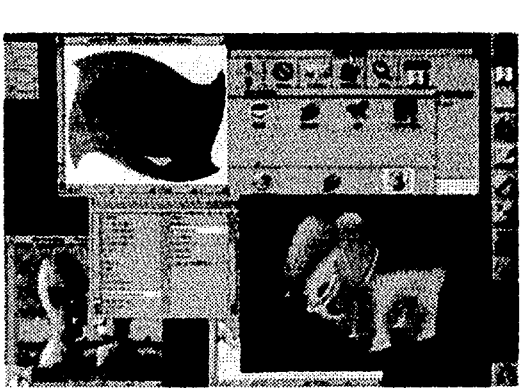

Figura 14. El sistema operativo NEXTSTEP: Mach.

De hecho, para la mayoría de los desarrolladores NEXTSTEP, Mach no es tan importante. Las clases definidas sobre Mach y las llamadas a esas clases son relevantes solo cuando una aplicación explota las características propias de Mach, por ejemplo, los *Mach ports*.

Resulta interesante discutir brevemente porqué se eligió Mach. De acuerdo con [24], cuando los diseñadores de NeXT tuvieron que decidir cuál sería el sistema operativo apropiado para el cubo, se dieron cuenta que tenían dos opciones:

a) Diseñar su propio sistema operativo (la cuál no era viable debido a la complejidad de la labor, el tiempo que se tenía que invertir y los recursos que se iban a distraer) o

b) Usar un sistema operativo ya desarrollado que cubriera el nivel de complejidad que se necesitaba.

Cuando se deciden por la segunda opción, el universo de sistemas operativos disponibles se fue reduciendo debido a que la mayoría eran específicos a un tipo de computadora, procesador o arquitectura de hardware, razón que hacía a los SOP evaluados inadecuados al hardware propietario de NeXT. Este camino los condujo directamente a una familia de sistemas operativos: UNIX.

UNIX fue originalmente diseñado para que se ejecutara en una computadora pequeña y simple. Una de sus primeras versiones usaba solamente 16 Kb de memoria para el kernel mientras se ejecutaba. Sin embargo, UNIX había crecido de manera impresionante durante 20 años con la consecuente adición de mayores características y adaptación de ambientes más sofisticados y complejos (hoy día se incluyen características como memoria virtual y capacidades para trabajar con múltiples procesadores). Bajo esta perspectiva, el kernel de UNIX pasó de ser pequeño y sencillo a grande y elaborado. Agregar nuevas características al SOP o modificar las ya existentes se convirtió en un serio desafío.

Por otro lado, un grupo de investigadores de la Universidad de Carnegie-Mellon decidieron regresar al punto de partida de UNIX y desarrollar un kernel pequeño que realizara un cierto grupo de tareas muy bien. Todas las operaciones adicionales serían manejadas por un grupo de piezas de software que residirían fuera del kernel, siguiendo la misma filosofía de las utilerías de UNIX. Este proyecto fue desarrollándose a lo largo de diferentes fases y el resultado final fue el sistema operativo Mach.

Debido a su características, NeXT seleccionó a Mach como su plataforma base y contrató a uno de los principales desarrolladores, el Dr. Avadis Tevanian, para que puliera a Mach y lograra una versión comercial del SOP. Debido a la creciente aceptación de UNIX en el mercado universitario, NeXT decide enfocar sus esfuerzos para lograr una versión de Mach compatible con UNIX con la finalidad de hacerlo competitivo con otras implantaciones de UNIX.

El Kernel de Mach fue diseñado para manejar tres tareas básicas:

- **Administración de memoria virtual,** orientada a la asignación de la memoria del sistema a cada programa.
- **Calendarización del procesador,** que consiste en la asignación de los recursos de computación a cada programa.
- **Comunicación entre tareas,** que son los mecanismos que permiten programas separados compartir información o enviarse mensajes entre si.

El kernel realiza estas tareas desde un punto de vista abstracto orientado a objetos con la finalidad de que Mach sea fácilmente portable a otros sistemas. Adicionalmente, al limitar las funciones del kernel a sus características básicas, el diseño de Mach permite que las modificaciones a otras funciones del sistema operativo sean más sencillas.

El enfoque orientado a objetos de Mach se basa en un conjunto de objetos abstractos que describen a los programas que están en ejecución. Estos objetos son: **tareas** (*tasks*), ***threads***, **mensajes** y **puertos** (*ports*).

Una **tarea** es la unidad básica de ejecución. Se consideran como tareas a un programa o a una aplicación. Una tarea no es solamente el código a ejecutar por si mismo, sin todos los recursos que requiere el programa: memoria, procesadores, dispositivos de almacenamiento secundario, puertos.

Un ***thread*** es un conjunto independiente de instrucciones de CPU que siempre está incrustado dentro de una tarea. El *thread* es en sí el código del programa que está contenido dentro de una tarea. La mayoría de las tareas contienen un solo *thread*, pero es posible que para una tarea dada existan más de un *thread*. Un *thread* siempre pertenece y está contenido en una sola tarea.

Un **mensaje** es una colección de información que va a ser enviada entre tareas o *threads*. El mensaje puede contener datos, instrucciones, o lo que sea necesario.

Un **puerto** es el lugar donde una tarea recibe mensajes. Para que una tarea pueda enviar un mensaje a otra, debe tener permisos para accesar el puerto de la tarea a la que se desea enviar la información. El uso de los puertos permite a las tareas establecer comunicación entre sí sin tener que saber mucho sobre la tarea receptora, lográndose un alto nivel de flexibilidad que permite comunicar tareas que están corriendo en diferentes computadoras.

El uso de Mach 2.5 puede presentar algunos problemas: si se desea portar aplicaciones NEXTSTEP a OpenStep[26] se enfrenta la dificultad de que las aplicaciones son altamente dependientes de las características de Mach 2.5. Otro de los problemas que tiene Mach es el

---

[26] OpenStep es un proyecto desarrollado por Next Computer, Inc en conjunto con Sun Microsystems y cuyo objetivo es el de proporcionar el ambiente de desarrollo NEXTSTEP de manera independiente al SOP Mach, haciéndolo por ende más accesibles a diferentes firmas de hardware.

soporte de dispositivos (*device driver support*). Los manejadores de dispositivos en NEXTSTEP como en cualquier otro sistema operativo basado en UNIX deben interactuar con el kernel. La interfaz para manejadores de dispositivos no es compatible con ninguno de los formatos para *device drivers* de las principales versiones comerciales de UNIX.

4.2 El modelo de objetos: Objective-C y las bibliotecas de clases de NEXTSTEP.

Objective-C es un lenguaje híbrido desarrollado por Brad Cox el cual consiste de ANSI-C con extensiones de objetos tipo Smalltalk. Objective-C provee *dynamic binding* y herencia sencilla de acuerdo a la tradición de Smalltalk.

La sintaxis de Objective-C para los aspectos procedurales es C y de tipo Smalltalk para las llamadas a los métodos de los objetos. Al igual que en Smalltalk, a la invocación de métodos se le denomina envío de mensajes. Por ejemplo el siguiente fragmento de código recorre a un arreglo de objetos y los despliega en diagonal a lo largo de una ventana:

```
for (i=0; i<max;i++) {
    rect.x = i+10;
    rect.y = i*10;
    myObj = array[i];
    [myWindow pos: &rect unidad de despliegue: myObj]
}
```

Las bibliotecas de clases para Objective-C bajo NEXTSTEP son muy diferentes con respecto a las de Smalltalk. Muchas de las clases de Smalltalk, por ejemplo las relacionadas con la magnitud, no son necesarias en Objective-C ya que C provee una semántica bien definida para ese tipo de objetos. Adicionalmente NEXTSTEP ofrece una amplia variedad de bibliotecas especializadas que facilitan varias funciones (por ejemplo, procesamiento de sonido, acceso a bases de datos) que complementan a la biblioteca principal (el *AppKit*) que es usada en la construcción de la interfaz de usuario y para conectar los objetos de dominio de la aplicación. Varias de las bibliotecas especializadas vienen con NEXTSTEP, otras pueden adquirirse a terceros.

El *AppKit* (*Application Kit*) de NEXTSTEP tiene como base a la clase raíz tradicional de Smalltalk: *Object*. De esta clase descienden todas las demás clases. La jerarquía de clases está orientada a un ambiente manejado por ventanas. Clases típicas como *Window*, *View*, *Panel*, *Text*, etc. se encuentran dentro de la biblioteca de clases.

Gran parte de la funcionalidad del ambiente NEXTSTEP es el resultado del diseño de las bibliotecas de clases y de Objective-C. Los conceptos de *enchufe* y *conexión* proveen gran parte de la base sobre la cual herramientas importantes como el constructor de interfaces (*Interface Builder*) operan.

4.3 El servidor de Ventanas y el Display de Postcript.

El sistema de ventanas de NEXTSTEP está basado en Adobe Display Postcript y, por consecuencia, tiene un modelo de imágenes muy diferente al de cualquier sistema de ventanas orientado a pixels.

El servidor de ventanas (*windows server*) es un proceso de bajo nivel que corre en *background* y que administra las labores de despliegue. El servidor de ventanas contiene una maquinaria de despliegue postcript la cual maneja un superconjunto del lenguaje Adobe Postcript Level 2. El servidor utiliza a los *Mach ports* ya que permiten conexiones en red, lográndose con esto un ambiente distribuido de ventanas entre máquinas NEXTSTEP.

La clase *Application* de NEXTSTEP es utilizada por todos los programas NEXTSTEP que interactúan con la unidad de despliegue. Esta clase maneja toda la interacción con el servidor de ventanas y por lo tanto aisla al desarrollador de la complejidad que significa interactuar con este.

La utilización en NEXTSTEP de un servidor de despliegue de tipo postcript ofrece una serie ventajas, por ejemplo, contar con un modelo unificado de imágenes así como independencia con respecto a los diferentes tipos de dispositivos. Un problema potencial derivado del uso de postcript es que es interpretado y por lo tanto lento. Esto en realidad no era un problema cuando NEXTSTEP corría en el cubo (el cual tenía hardware específico para procesamiento de video), sin

embargo se ha convertido en una dificultad para algunos de los sistemas basados en Intel que son mucho menos poderosos que el cubo.

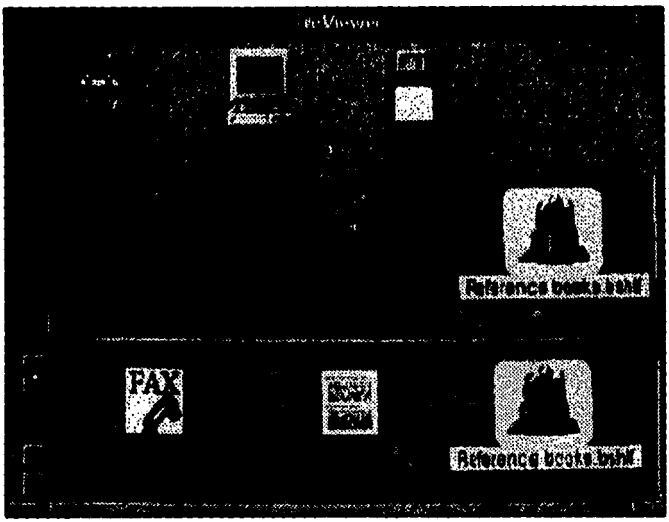

Figura 15. El ambiente de ventanas de NEXTSTEP.

4.4 Herramientas de Desarrollo.

NEXTSTEP provee todas las herramientas UNIX tradicionales ya que es UNIX. El compilador de NEXTSTEP es **gcc** de *Free Software Foundation* el cual puede compilar y ligar una mezcla de C, C++ y Objective-C. El debugger es **gdb** el cual ha sido extendido para que comprenda Objective-C y las características de Mach.

Las principales herramientas de desarrollo son *Interface Builder*, *Project Builder* y el recientemente liberado *Enterprise Objects Framework (EOF)*. Existen adicionalmente una serie de

kits cada uno de los cuales define a un grupo de clases reusables que ofrecen diferentes tipos de funcionalidad. De hecho, la biblioteca de clases descrita en el apartado 4.2 es conocida como *AppKit* debido a que es usada para definir el marco de referencia de una aplicación básica NEXTSTEP. Ejemplos de otros kits son el *PhoneKit* (para aplicaciones ISDN), *3DKit* (para aplicaciones con gráficos) y el *DBKit* (el cual es una extensión de EOF). Las otras herramientas de desarrollo como *Digital Librarian* (para indexar y buscar documentación), *Header Viewer* (para ver las definiciones de las clases) y *Edit* (para edición inteligente de código fuente) completan un ambiente de desarrollo completamente integrado.

4.5 El componente GUI de NEXTSTEP: *Interface Builder.*

Interface Builder es la herramienta con la cual los desarrolladores de aplicaciones NEXTSTEP pasan la mayor parte de su tiempo ya que la interfaz de usuario en la mayoría de las aplicaciones se lleva hasta el 70% del esfuerzo de desarrollo.

Interface Builder se puede comparar con el componente GUI de SNAP y con otros constructores GUI (*GUI builders*).

Por lo regular, los constructores GUI proveen mecanismos para la planeación gráfica de la interfaz de usuario y generan el código requerido para desplegarla. Adicionalmente permiten al desarrollador animar a la interfaz, aunque el soporte para lograr la animación por lo regular es muy limitado y, por consecuencia, el desarrollador tiene que realizar el oneroso trabajo de escribir el código de animación e integrarlo al código generado por el constructor GUI; lo cual es una seria desventaja.

Otra de las desventajas que se tiene al trabajar con constructores GUI consiste en que la creación rápida de prototipos está limitada a los aspectos de despliegue del sistema. Una vez que se ha desarrollado el código de animación a la medida y que ha sido integrado en la aplicación, el desarrollador debe asegurarse manualmente de que tanto el código generado por el constructor GUI como el código desarrollado *ad-hoc* todavía puedan operar entre sí.

El *Interface Builder* de NEXTSTEP retoma los conceptos derivados de un constructor GUI y los lleva hasta su conclusión lógica. Está fuertemente integrado a las bibliotecas de clases de NEXTSTEP y puede rápidamente construir o destruir el *parsing* de clases Objective-C.

También permite al desarrollador extender las paletas predefinidas (las cuales proveen objetos estándar de la interfaz tales como botones, *sliders*, etc.) con paletas definidas por el desarrollador (que pueden constar de objetos *ad-hoc* a una aplicación, por ejemplo, objetos para conectarse a la base de datos u hojas de cálculo, o bien objetos de la interfaz con un comportamiento diferente, etc.) Estas extensiones tienen la misma funcionalidad que los objetos predefinidos en el *Interface Builder*.

Tanto NEXTSTEP como *Interface Builder* utilizan el paradigma *Model-View-Controller* de Smalltalk. La biblioteca de clases de NEXTSTEP tiene una clase llamada *View* la cual proporciona el comportamiento básico de todas las "vistas" (*views*). La clase *View* y sus subclases proveen los principales mecanismos a través de los cuales una aplicación NEXTSTEP interactúa con el usuario. En NEXTSTEP cualquier variable de tipo **id** en Objective-C es considerada como un "enchufe" (*outlet*). *Interface Builder* permite que un "enchufe" se conecte con una "vista" o con cualquier otro objeto de interfaz usando el *mouse* en vez de tener que escribir código para hacer esto.

Una de las principales actividades de desarrollo en NEXTSTEP es la creación de las clases controladoras y sus objetos. Una clase controladora proporciona cualquier comportamiento definido "a la medida" para la aplicación así como las computaciones que sean necesarias. Debido a que *Interface Builder* puede hacer el *parsing* de las definiciones de clases Objective-C, éste puede importar las definiciones de objetos controladores. Una vez que las definiciones de controlador están disponibles vía *Interface Builder*, el desarrollador puede fácilmente definir la animación apropiada para la interfaz de usuario. Debido a que los controladores son definidos como clases es posible extender el comportamiento de clases controladoras ya existentes, por consecuencia, la reusabilidad en el ambiente NEXTSTEP es una actividad natural.

Los desarrolladores de aplicaciones NEXTSTEP definen la interfaz de usuario por medio de *Interface Builder*. Como se requiere integrar funcionalidad a la interfaz de usuario, el desarrollador crea una clase usando el editor de NEXTSTEP *Edit* e importa la definición a *Interface Builder*. Por lo tanto, el desarrollo rápido de aplicaciones se facilita al usar *Interface Builder*.

4.6 El administrador de proyectos: *Project Builder*.

Project Builder es una interfaz gráfica de usuario que es usada para agrupar a los archivos de código fuente, los generados por *Interface Builder* y los archivos de construcción (*Makefiles*) asociados con una aplicación dentro de un conjunto de directorios. *Project Builder* soporta la anidación de proyectos de tal forma que aplicaciones muy complejas integradas por más de un componente principal puedan ser manipuladas.

Otras características importantes de *Project Builder* incluyen algunas herramientas muy útiles para *debuggear* y generar código relocalizable para diferentes plataformas de hardware. De hecho, cuando ocurre un error en tiempo de compilación, el mensaje estándar de diagnóstico es presentado, sin embargo, el desarrollador puede hacer doble click sobre el mensaje y automáticamente el archivo fuente es abierto y la línea que ocasionó el error es resaltada. Para determinar errores en tiempo de ejecución, el *debugger* gráfico **gdb** puede invocarse dentro de *Project Builder*. La generación de código para diferentes plataformas de hardware se ejecuta automáticamente una vez que se ha elegido la opción deseada de una lista presentada por una ventana de tipo *pop-up*. Si se requiere generar código para más de una plataforma, el sistema crea un archivo binario multi-arquitectura (*MAB* por sus siglas en inglés) que duplica la información específica para cada plataforma (por ejemplo, llamadas de bajo nivel) pero no información independiente a la plataforma (por ejemplo, las especificaciones de la interfaz de usuario). De esta manera, esta aplicación puede ser ejecutada desde cualquier plataforma que se haya incluido en el MAB.

4.7 El componente de persistencia de NEXTSTEP: *Enterprise Objects Framework.*

Enterprise Object Framework (EOF) habilita al desarrollador de aplicaciones NEXTSTEP para la interacción con sistemas comerciales de bases de datos relacionales como Oracle y Sybase. EOF se usa como el "pegamento" para unir a un programa NEXTSTEP (el cual manipula y despliega a los objetos de la aplicación dentro del ambiente) y a un RDBMS el cual ofrece las facilidades para resolver el problema de persistencia de los objetos.

EOF es en realidad un esqueleto de aplicación (*application framework*) y un conjunto de facilidades que definen la manera en que NEXTSTEP y la base de datos interactúan. Para construir una aplicación que use EOF se tienen que crear o usar tres componentes: el *modelo*, la *capa de acceso* y la *capa de interfaz*.

Primero, debe existir un *modelo* que especifique la correspondencia entre los objetos de la aplicación y los elementos del esquema de la base de datos. Usando una herramienta denominada *EOModeler*, se pueden construir estructuras entidad-relación y conectarlas con las tablas de la base de datos con la misma mecánica de *Interface Builder* (se establecen las conexiones visualmente por medio del mouse). Estas conexiones son usadas en tiempo de ejecución para asociar a los objetos con sus correspondientes entidades y los valores apropiados de la base de datos. Por ejemplo, para un conjunto de variables de instancia de un objeto en particular, los valores que se van a tomar pertenecen al renglón de una tabla en la base de datos.

Segundo, la *capa de acceso* de EOF usa al modelo desarrollado para buscar y salvar los valores de los objetos desde y hacia la base de datos. Esta capa consiste de dos niveles: el nivel de adaptador (*adaptor level*) y el nivel de base de datos (*database level*).

El nivel de adaptador provee la interfaz entre un manejador de base de datos específico (por ejemplo Oracle) y el resto de EOF y es el responsable de la traducción de las llamadas realizados a funciones genéricas de base de datos a los comandos específicos de un RDBMS en particular. También es responsable de la comunicación entre EOF y el RDBMS.

El nivel de base de datos está construido sobre el nivel de adaptador y consiste de un conjunto especial de clases para manejar a los objetos que requieran de la base de datos. Por ejemplo, la capa de base de datos puede ser usada para alterar la manera en que los datos son buscados y, de esta manera, mejorar el tiempo de respuesta de la interfaz de usuario, o para especificar diferentes estrategias de actualización de la base de datos.

Tercero, la *capa de interfaz de usuario* proporciona la liga entre objetos de la interfaz de usuario (por ejemplo, botones, Vistas de tablas, etc.) y el modelo. Por ejemplo, mapea las propiedades de los objetos en los lugares apropiados de la interfaz de usuario con lo que se asegura que los valores de los datos y los presentados en la pantalla son consistentes. Quienes están familiarizados con Smalltalk podrán observar que el papel de la capa de interfaz de usuario se parece mucho al que juega el *Controller* en el paradigma de Smalltalk *Model-View-Controller*. De hecho, la clase principal de la capa se llama *EOController*.

Los beneficios que se tienen de trabajar con una estructura dividida en capas es que se pueden agregar fácilmente nuevos RDBMS comerciales simplemente escribiendo un nuevo adaptador y la interfaz de una aplicación con el nuevo RDBMS puede ser alterada sin tener que cambiar ninguno de los procedimientos de acceso a la base de datos ya que estos son un conjunto de funciones genéricas. Adaptadores para Oracle y Sybase vienen con EOF, si se desea usar otro manejador de base de datos se puede comprar el adaptador a terceros.

CAPÍTULO 5

DEFINICIÓN DE LOS PARÁMETROS DE EVALUACIÓN DE UNA PLATAFORMA DE DESARROLLO DE SOFTWARE ORIENTADO A OBJETOS Y UN CASO DE ESTUDIO: SNAP VS. NEXTSTEP

Introducción.

Siempre que se va a establecer una comparación entre una familia de productos de software, se debe contar con una serie de parámetros de evaluación que permitan definir claramente las ventajas que tiene un producto sobre los otros. Evidentemente, algunos criterios de evaluación pueden ser puramente subjetivos, pero estos deben evitarse al máximo si se desea contar con una evaluación objetiva de los productos.

El paradigma orientado a objetos es muy conveniente para el desarrollo de sistemas de carácter crítico en los negocios, por ejemplo, un sistema de información para el control de las operaciones de mercado de dinero de un intermediario financiero puede ser modelado de manera muy natural empleando las características de la orientación a objetos.

Los objetos están logrando cada vez más aceptación dentro de la industria informática en el país, sin embargo, todavía es un área obscura para la mayoría de las instituciones que comienzan a incursionar en el paradigma.

El desarrollo rápido de aplicaciones orientadas a objetos operacionalmente completas es prácticamente imposible cuando se está inmerso en el proceso de aprendizaje del modelo de objetos. Aún cuando ya se tiene experiencia en el paradigma, si se cuenta únicamente con un

lenguaje orientado a objetos y, si a caso, con un *application framework*; el desarrollo rápido de aplicaciones sigue siendo un gran problema.

Las plataformas de desarrollo de software orientado a objetos surgen como la mejor alternativa para este tipo de desarrollos ya que ofrecen al menos las tres características básicas necesarias para la implantación de una aplicación operacionalmente completa (amen del modelo de objetos, claro está): un componente de comunicaciones, un componente GUI y un componente de persistencia de la información.

Si se está considerando adquirir una plataforma de desarrollo de software es necesario comparar entre los diferentes ambientes que se ofrecen en el mercado para hacer la mejor elección. Definitivamente, la selección final obedece a las necesidades específicas de una organización en particular, sin embargo, se debe contar con una serie de criterios de evaluación independientes que puedan aplicarse a toda plataforma de desarrollo.

En este capítulo se van a establecer los parámetros de evaluación generales a todo ambiente de desarrollo orientado a objetos y vamos a realizar una comparación entre dos plataformas: SNAP y NEXTSTEP.

5.1 Parámetros de evaluación que se deben considerar para cualquier plataforma de desarrollo de software orientado a objetos.

Los componentes principales de todo ambiente de desarrollo orientado a objetos que se deben evaluar son:

- El modelo de objetos.
- Las facilidades para la generación de la interfaz gráfica de usuario (componente GUI).
- Las facilidades para la implantación de los esquemas de persistencia de los objetos (componente de almacenamiento permanente).
- Las facilidades para la implantación de modelos de comunicaciones (componente de comunicaciones).

- Los costos y configuración del hardware y los costos de las licencias de desarrollo y de usuario.

Adicionalmente, se pueden considerar en la evaluación del ambiente los siguientes componentes en caso de que los haya:

- Las facilidades para la integración y comunicación con software de aplicación externo.
- La interfaz CASE.

5.1.1 Evaluación del modelo de objetos.

Definitivamente el componente principal de toda plataforma de desarrollo orientada a objetos es el modelo de objetos. En torno a él se encuentran los demás componentes, ya que la construcción de cualquier aplicación depende del lenguaje anfitrión del ambiente.

El primer punto que se debe considerar en la evaluación del modelo de objetos es la pureza del lenguaje, ya que a partir de este concepto se puede analizar el soporte que el lenguaje ofrece a los conceptos fundamentales de la programación orientada a objetos: *clasificación, modularidad, encapsulación, ocultamiento de información, herencia* y *polimorfismo*.

Las preguntas que deben responderse al evaluar el modelo de objetos de un ambiente de desarrollo orientado a objetos son las siguientes:

- ¿El lenguaje es un híbrido (como C++, Pascal Objetivo, etc.) o es un lenguaje puro (como Eiffel, Smalltalk) o es un lenguaje propietario (como SNAP)? ¿Es fácil de aprender?
- En la declaración de una clase ¿Se pueden definir elementos adicionales a los atributos y métodos (por ejemplo tipos, reglas, *demons*, etc.)?
- Para efectos de modularidad ¿Se pueden definir las clases en diferentes archivos?
- Para efectos de encapsulación ¿Se pueden definir diferentes secciones dentro de una clase? ¿Se puede restringir el acceso a los elementos de una clase a través de la definición de su visibilidad?
- Para efectos de ocultamiento de la información ¿Se pueden definir los tres niveles de visibilidad (pública, privada y protegida) a los elementos de una clase?

- ¿El lenguaje soporta mecanismos de herencia sencilla y múltiple?
- ¿El lenguaje permite el polimorfismo? ¿El polimorfismo es paramétrico, por sobrecarga o múltiple?
- ¿El lenguaje permite *static bining* y *dynamic binding*?

5.1.2 Evaluación del componente GUI.

En el capítulo 2 se distinguen dos tipos de estructuras que puede tomar la biblioteca GUI del ambiente de desarrollo: como un API o como un *Application framework*. Como se trata de un ambiente de desarrollo orientado a objetos, entonces el componente GUI debe explotar las cualidades de la programación orientada a objetos. De Santis [39] establece una serie de puntos a considerar en la evaluación de una biblioteca GUI. Tomando como base esos criterios se pueden establecer los siguientes parámetros de evaluación del componente GUI:

- ¿Existe una jerarquía de clases?. Si no es así el componente GUI no es orientado a objetos.
- ¿La jerarquía de clases fue diseñada con la finalidad de cubrir necesidades de uso o para simplificar la implantación del GUI?
- ¿Se necesita conocer con detalle la estructura interna del ambiente GUI que se encuentra debajo para usar el producto?
- ¿Que tan fácil es derivar nuevas clases a partir de las clases proporcionadas por la biblioteca del componente GUI?
- ¿Como son manejados los eventos? ¿Se pueden procesar por medio de la redefinición de métodos polimórficos y con la asignación de funciones de *callback* o se tienen que procesar mensajes numéricos usando estructuras de control tipo *case*?
- ¿El componente GUI contiene elementos y clases de alto nivel tales como hojas de cálculo, editores de texto, *tool bars*, *status bars*, etc.? Esto puede indicar que tan fácil es construir elementos de despliegue complejos y derivar clases nuevas y más portables.

- ¿El diseño del componente GUI soporta conceptos puros de la orientación a objetos como el paradigma *Model-View-Controller*?
- ¿Que plataformas soporta el componente GUI - Windows, Windows NT, OS/2 PM, Macintosh, OSF/Motif? ¿Qué compiladores son soportados? Si existe una versión que soporte OSF/Motif, ¿Qué plataformas están soportadas - SCO, Unixware, Solaris, HP UX, RS/6000 AIX, VMS? ¿Soporta el modo carácter? y de ser así, ¿Para qué plataformas: DOS, Extended DOS, UNIX, VMS?
- ¿Se puede tener acceso a la capa del API sobre la que está construido el componente GUI?
- ¿Se pueden crear *dialogs*, *menus*, *bitmaps* e *iconos*? ¿Se pueden probar estos elementos conforme se van construyendo? ¿Se pueden usar objetos propios que fueron "hechos a la medida"?

5.1.3 Evaluación del componente de almacenamiento permanente.

Una aplicación que no pueda guardar información para usarla posteriormente es inútil. Por esto, un ambiente de desarrollo de software orientado a objetos debe contar con un componente dedicado a resolver el problema de persistencia.

Como se indica en el capítulo 2, las opciones que se tienen para implantar los mecanismos de persistencia del modelo de objetos son dos:

a) usando archivos planos o

b) con RDBMS

Para efectos de evaluación del componente de almacenamiento permanente se deben responder las siguientes preguntas:

- ¿Qué interfaces de persistencia ofrece el componente: archivos planos, RDBMS o ambos?
- Si se soporta el uso de archivos planos para implantar la persistencia ¿Cómo se define el mapeo del modelo de objetos a la estructura del archivo plano? ¿Qué tipos de datos se

pueden hacer persistentes con archivos planos? ¿Hay limitaciones en el crecimiento de los archivos?

- En caso de que se soporte RDBMS ¿Cómo se define el mapeo de las clases del modelo de objetos con respecto a las entidades en la base de datos? ¿Qué sistemas de manejo de bases de datos son soportados?
- ¿Existe una jerarquía de clases que encapsule el comportamiento del componente? ¿La jerarquía fue diseñada para cubrir necesidades de uso?
- ¿Se pueden derivar nuevas clases a partir de la jerarquía y extender su funcionalidad?
- En el caso de que no se soporte un RDBMS en particular ¿Qué tan difícil es la implantación de los mecanismos necesarios para interactuar con ese manejador de bases de datos?
- ¿Cómo está organizado el componente: por capas o monolítico? De esta manera es posible medir el nivel de flexibilidad del componente.

5.1.4 Evaluación del componente de comunicaciones.

El componente de comunicaciones es el encargado de la implantación de modelos de comunicación como el *cliente-servidor*. Las características que se deben evaluar del componente de comunicaciones son:

- ¿Qué modelos de comunicaciones soporta el componente (*cliente-servidor*, *maestro-esclavo*, *peer to peer*, etc.)?
- ¿Es fácil implantar los mecanismos de comunicación?
- ¿El componente ofrece facilidades para la programación de llamadas a procedimientos remotos (RPC)?
- ¿Que tan independiente del sistema operativo es el componente de comunicaciones?
- ¿Qué estándares de comunicaciones emplea el componente (TCP/IP, UDP/IP, etc.)?
- ¿Existe una jerarquía de clases que encapsule el funcionamiento de bajo nivel del componente?

- Si existe una jerarquía de clases ¿Su concepción fue orientada a resolver necesidades de uso o se pueden derivar nuevas clases y extender su funcionalidad?
- Supóngase que se ha desarrollado una aplicación basada en el modelo cliente-servidor corriendo en plataformas heterogéneas de hardware ¿Es difícil hacer funcionar los mecanismos de comunicación entre procesos?
- Si la plataforma de desarrollo puede correr bajo UNIX ¿Se usan los *sockets* o los *streams* como interfaces para operaciones I/O dentro del kernel de UNIX y entre el kernel y el resto del sistema UNIX?

5.1.5 Costos

Indiscutiblemente el dinero que se tiene que invertir es uno de los parámetros más importantes a considerar cuando se está planeando adquirir un ambiente de desarrollo orientado a objetos. Las preguntas que se deben contestar son las siguientes:

- ¿Cuál es la configuración mínima de hardware (y su costo) que se necesita para correr el ambiente de desarrollo?
- ¿Cuál es el costo por licencia de desarrollo que tiene la plataforma y cuál es el costo de la licencia de usuario?
- ¿Hay que invertir en software adicional y cual sería el precio de éste?

5.1.6 Parámetros adicionales de evaluación.

Si el ambiente de desarrollo cuenta con características adicionales como la facilidad de comunicación con software de aplicación externo se deben considerar los siguientes parámetros de evaluación:

- ¿Qué tipos de comunicación con software de aplicación externo se puede establecer? ¿Las aplicaciones desarrolladas con el ambiente pueden compartir datos con aplicaciones foráneas?

- Si se trata de aplicaciones desarrolladas en un 3GL ¿Qué lenguajes y compiladores están soportados?
- ¿Es necesario definir una interfaz en el modelo de objetos para interactuar con el software de aplicación externo (por ejemplo, crear una clase que pueda ser instanciada para llamar a la aplicación externa) o se puede interactuar con este directamente?

Si el ambiente cuenta con una interfaz CASE se deben considerar los siguientes parámetros de evaluación:

- ¿Con que soporte de CASE se cuenta: se pueden importar o exportar las definiciones desde o hacia un CASE de bajo nivel?
- ¿Se cuenta con interfaces para el uso de herramientas CASE tradicionales y/o orientadas a objetos?
- ¿Que tan transparente es la interacción entre la herramienta CASE con el ambiente de desarrollo? ¿Se tiene que invertir mucho tiempo para echar a andar una definición CASE dentro del ambiente?

5.2 Evaluación de SNAP.

El modelo de objetos.

El modelo de objetos de SNAP es un lenguaje propietario de *Template Software Inc.* Es un lenguaje fácil de aprender ya que la sintaxis es muy natural para cualquier desarrollador. Se puede clasificar como un lenguaje *puro* orientado a objetos.

En la declaración de una clase se tienen una serie de secciones que permiten la definición de diferentes elementos tales como reglas, procesos asíncronos y tipos, entre otros (véase capítulo 3) con diferentes niveles de visibilidad ya que se pueden definir como públicos, privados y protegidos.

El lenguaje soporta el concepto de modularidad ya que permite la definición de las clases en múltiples archivos de definición de clases (class definition files). También permite la definición de mecanismos de herencia sencilla y múltiple y es capaz de resolver aquellas definiciones que solo se pueden conocer en tiempo de ejecución (*dynamic binding*).

El uso de los procesos asíncronos o *demons* si no se hace con cuidado puede llevar a resultados inesperados. Para ejercer un control exacto sobre el disparo de los *demons* es necesario implantar una suerte de semáforos para evitar colisiones entre estos.

Una severa desventaja del lenguaje es la imprecisión por problemas de redondeo que se tienen con tipos de datos definidos como *fixed* o *real* sobre todo cuando se tienen que programar métodos que desarrollan una gran cantidad de cálculos numéricos. Aún cuando es posible declarar métodos externos que podrían explotar la exactitud de C y con esto resolver el problema de redondeo, los tipos de retorno que se emplean para enviar los datos calculados en C a SNAP no son muy útiles ya que se pierde exactitud cuando los parámetros llegan a SNAP.

El componente GUI.

El componente GUI de SNAP está en un punto intermedio entre un esqueleto de aplicación y un constructor de interfaces. Cuenta con una jerarquía de clases que encapsula la interacción del componente GUI con el servidor de ventanas del sistema operativo en el que está corriendo la plataforma de desarrollo. Esto hace que los detalles propios del sistema de ventanas no los necesite conocer el desarrollador y por consecuencia las aplicaciones son mucho más independientes y portables. El esfuerzo que se tiene que invertir en llevarse una aplicación corriendo en OSF/Motif a MS-Windows es mínimo y si acaso se tiene que invertir tiempo en el mapeo de los colores en Windows.

La biblioteca de clases facilitada por el componente GUI permite la definición de funciones de *callback* para responder a los eventos que se dan en el ambiente de ventanas. La construcción de las unidades de despliegue es a la manera de un constructor de interfaces, ahorrándonos por consecuencia mucho trabajo de programación del ambiente de ventanas de bajo nivel.

El GUI de SNAP soporta diferentes plataformas de hardware y sistemas operativos, lo que permite la interacción con más ambientes de ventanas.

El componente de almacenamiento permanente.

SNAP cuenta con un componente de almacenamiento permanente que soporta el uso de archivos planos y manejadores de bases de datos relacionales. Si se desea hacer persistente al modelo de objetos por medio de archivos planos tenemos que definir un par de archivos (el de definición y el de datos) para cada clase lo cual es una seria desventaja ya que si se tienen que hacer persistentes doscientas clases debemos contar con cuatrocientos archivos planos. Una desventaja adicional radica en que los tipos de datos soportados para el uso de archivos planos, están limitados a tres: *str, int, real.*

Si se desea interactuar con una base de datos relacional, SNAP ofrece dentro del ambiente una facilidad para la definición del mapa entre el modelo de objetos y el modelo relacional. Proporciona además una jerarquía de clases que encapsula los mecanismos de bajo nivel para establecer la interacción con el RDBMS. Las relaciones de uso que se pueden establecer con esta jerarquía son únicamente a nivel de implantación. Los sistemas comerciales de manejo de bases de datos relacionales que soporta el componente son varios, las operaciones de bajo nivel con cualquiera de estos son encapsuladas por el componente.

Para poder emplear un RDBMS que no esté soportado en el componente se tiene que establecer contacto con *Template* ya que no se cuenta con los fuentes para la implantación de estos mecanismos.

El componente de comunicaciones.

Uno de los componentes más importantes de SNAP es el componente de comunicaciones. Los modelos de comunicación que soporta son dos: el modelo *peer to peer* y el modelo *cliente-*

servidor. Cuando se trabaja con un modelo *cliente-servidor* el componente facilita la implantación de llamadas a procedimientos remotos.

SNAP cuenta con una facilidad denominada *Shared Information Base* la cual es el corazón de la implantación del modelo de comunicaciones. La base de información compartida permite que por medio de archivos planos se pueda establecer que clases son compartidas en el modelo de objetos, cuando y como se deben refrescar los datos en los procesos que se están comunicando, así como el tipo de comunicación a emplear: síncrona o asíncrona. El uso de archivos planos hace que el componente de comunicaciones sea independiente de la aplicación, ya que si cambia la definición de una clase compartida o se necesita agregar otra clase solo se deben alterar dos archivos planos.

Parámetros adicionales de evaluación.

Template hace referencia en su documentación al componente de software de aplicación externo. En realidad el objetivo de este componente es el ofrecer facilidades para la programación de rutinas en lenguaje C y no propiamente el "pegar" aplicaciones externas como tales, sin embargo se permite la ejecución de piezas de software externo.

La interfaz CASE con la que cuenta SNAP sirve para leer las definiciones de un CASE de bajo nivel no orientado a objetos. Por ejemplo, si el componente de almacenamiento permanente está interactuando con ORACLE, la interfaz CASE se comunica con ORACLE CASE.

5.3 Evaluación de NEXTSTEP.

El modelo de objetos.

El lenguaje que eligió NeXT para su modelo de objetos es Objective-C. Objective-C es un lenguaje híbrido que combina la potencia de C y la sintaxis de Smalltalk. El uso de un lenguaje híbrido tiene la desventaja de que la curva de aprendizaje es más lenta debido a que el diseño

original del lenguaje empleado como base no fue pensado para cubrir los conceptos del modelo orientado a objetos y, por consecuencia, las extensiones realizadas al mismo son una especie de "parches" para ofrecer las bondades de la programación orientada a objetos.

Objective-C permite la encapsulación ya que la definición de una clase se divide en su *declaración* (header file con extensión .h) y en su *definición* (archivos con extensión .m) que contienen la implantación de la clase.

Objective-C encapsula los mecanismos de instanciación de las clases ya que para cada clase crea un objeto especial denominado *class object* el cual tiene toda la información necesaria para construir instancias de esa clase.

Objective-C soporta únicamente mecanismos de herencia sencilla y también polimorfismo por resolución dinámica (*dynamic binding*).

El componente GUI.

Interface Builder (IB) es el componente GUI de NEXTSTEP. Es un generador de interfaces que permite la construcción visual de la interfaz gráfica de usuario. Esta facilidad permite la animación del GUI y la prueba del funcionamiento de la interfaz aún antes de que se escriba una sola línea de código, por lo que facilita la toma de decisiones durante el periodo de diseño del GUI.

Los objetos gráficos que se usan con IB son de carácter general, y el código que se tiene que desarrollar propio para una aplicación se encapsula dentro de las clases programadas en Objective-C. La interfaz diseñada con IB se almacena en archivos con extensión .nib los cuáles son utilizados por la aplicación cuando está corriendo.

Interface Builder al igual que NEXTSTEP soporta el modelo *Model-View-Controller* por lo que hablamos de un componente GUI totalmente orientado a objetos.

Una de las desventajas del componente GUI de NEXTSTEP es que no sigue a ninguno de los estándares *de facto* o formales para interfaz gráfica de usuario ya que está basado en Adobe

Display Postcript. Debido a que las aplicaciones NEXTSTEP solo corren en ambientes NeXT, el componente GUI no ofrece ningún mecanismo para el soporte de las aplicaciones en otros ambientes de ventanas.

El componente de almacenamiento permanente.

Enterprise Objects Framework es el componente de almacenamiento permanente del ambiente de desarrollo NEXTSTEP. Este componente es uno de los más interesantes si consideramos que su diseño fue pensado en capas para encapsular los niveles de complejidad de uso de un RDBMS. El componente no ofrece mecanismos para el uso de archivos planos ya que Objective-C permite la definición y administración de los mismos.

EOF encapsula la interacción con la base de datos a tres diferentes niveles: el *modelo*, la *capa de acceso* y la *capa de interfaz con el usuario*. Las ventajas que ofrece este esquema de trabajo son claras: por un lado la correspondencia entre el modelo de objetos y las entidades definidas en el esquema relacional se define con el *modelo*. La *capa de acceso* se apoya en el modelo para buscar y salvar los valores desde y hacia la base de datos relacional. Esta capa también está definida en dos niveles: el nivel de adaptador y el nivel de base de datos. El nivel de adaptador se dedica a la traducción de instrucciones de tipo genérico para el acceso a la información en la base de datos a las instrucciones específicas de un manejador en particular.

El nivel de base de datos está construido sobre el nivel de adaptador y consiste de un conjunto especial de clases para manejar a los objetos que requieren de una base de datos. Finalmente la *capa de interfaz con el usuario* se encarga de establecer la interacción entre los objetos gráficos y el modelo.

Este esquema basado en responsabilidades ofrece un nivel de flexibilidad bastante bueno ya que si se tienen que hacer cambios a la aplicación que no modifiquen el esquema de persistencia (o viceversa) estos pueden ser fácilmente implantados. EOF nace como una herramienta poderosa que supera por mucho a su predecesor, el DBKit. Sin embargo, sólo están soportados

dos manejadores comerciales de bases de datos relacionales: ORACLE y Sybase. Aún cuando la filosofía de trabajo de EOF permite la creación de adaptadores nuevos para trabajar con RDBMS no soportados por el componente, la carencia de una variedad aceptable de adaptadores para diferentes RDBMS es una seria desventaja pues no se puede probar al componente (véase [40] por ejemplo).

El componente de comunicaciones.

En NEXTSTEP no existe un componente de comunicaciones como tal. La implantación del modelo *cliente-servidor* y la definición de información compartida se logra apoyándose en el sistema operativo Mach por medio de los *Mach ports*. Este esquema de trabajo hace muy dependiente a NEXTSTEP del sistema operativo, pero hay que recordar que la concepción original del ambiente no fue orientada hacia la independencia del componente sino a la mejor forma de explotar los recursos de un sistema de hardware bastante poderoso.

Desde un punto de vista de portabilidad esta dependencia limita seriamente a las aplicaciones que se puedan implantar con NEXTSTEP ya que a diferencia de SNAP obliga a cambiar el SOP de las máquinas en que se pretendan ejecutar dichas aplicaciones. NEXTSTEP no sigue a ninguno de los estándares.

Parámetros adicionales de evaluación.

A diferencia de SNAP, que es una aplicación que se comporta como un ambiente de desarrollo, NEXTSTEP nace como un verdadero ambiente ya que integra un todo al sistema operativo y a las facilidades necesarias para la creación de aplicaciones. Del conjunto de facilidades ofrecidas por NEXTSTEP, uno de los componentes que marca una clara ventaja sobre otros ambientes es *Project Builder*.

Project Builder se encarga de la administración de los recursos de un proyecto así como de las diferentes versiones que del mismo puedan existir. Ofrece la ventaja de la generación de código

MAB ya que las diferentes versiones de NEXTSTEP pueden ejecutar en computadoras basadas en Intel, Motorola, DEC Stations, etc.

Desafortunadamente *Project Builder* no resuelve los problemas inherentes al esfuerzo que significa desarrollar proyectos grandes y complejos, los cuales requieren del uso de aplicaciones para el control de configuraciones de software. Sin embargo, tanto en NEXTSTEP como en los sistemas UNIX típicos, la integración de este tipo de componentes es simple. Existen aplicaciones desarrolladas por terceros que proveen de este tipo de funcionalidad.

Adicionalmente, se ha anunciado que la versión 3.3 de NEXTSTEP va a incluir más soporte para el control de configuraciones.

5.4 Comparación.

Todo mundo sabe que las comparaciones son odiosas pero necesarias. Sin embargo, en el caso de que se esté considerando la adquisición de una plataforma de desarrollo de software orientado a objetos es necesario contar con los elementos indispensables para hacer la mejor elección.

SNAP es un ambiente de desarrollo independiente del sistema operativo, ya que puede correr en computadoras basadas en Intel bajo Windows como sistema operativo o en estaciones de trabajo UNIX sin importar el ambiente de ventanas con el que se cuente.

El templete de SNAP provee de una arquitectura que fomenta de manera importante la reusabilidad. El modelo de objetos de SNAP es un lenguaje propietario que se aprende fácilmente. Una de las serias desventajas de SNAP radica en los graves problemas que se tienen con la aritmética del lenguaje ya que la mezcla de los diferentes tipos en una expresión provoca errores de redondeo (por ejemplo, la mezcla de variables *fixed* de diferentes tamaños ocasiona la pérdida de precisión en los resultados que son computados). Sin embargo, estas eventualidades pueden ser corregidas por *Template* fácilmente.

El componente GUI se apega a los estándares *de facto* de la industria y es, por consecuencia, más flexible y portable. El componente de almacenamiento permanente ofrece interfaz con cuatro de los RDBMS más solicitados de la industria (ORACLE, INFORMIX, Interbase y Sybase). El componente de comunicaciones brinda un buen nivel de independencia con respecto al SOP anfitrión.

En contraste NEXTSTEP ofrece la versatilidad de C con una extensión de objetos. La biblioteca de clases proporcionada por NEXTSTEP fomenta la reusabilidad de componentes comunes a toda aplicación. la generación de la interfaz gráfica de usuario es mucho más sencilla que en SNAP pero con la desventaja de que no es portable a otros ambientes de ventanas. El componente de almacenamiento permanente está limitado a dos manejadores de bases de datos (Sybase y ORACLE) pero por su arquitectura y concepción permite una interacción de más alto nivel que SNAP. El componente de comunicaciones no existe ya que se apoya fundamentalmente en el SOP Mach, lo que hace al ambiente excesivamente dependiente del mismo.

Para ampliar la versatilidad de NEXTSTEP hacia otras plataformas, NeXT ha anunciado la liberación en un futuro cercano de OpenStep. Este producto es una especificación que va a permitir la integración de la tecnología NEXTSTEP en diferentes plataformas de hardware sin tener que contar con Mach. En este momento cualquier compañía que desee ofrecer NEXTSTEP en su plataforma de hardware debe implantar un *puerto nativo* de NEXTSTEP, es decir, debe implantar NEXTSTEP desde sus bases (lo que genera la necesidad de programar una versión del SOP Mach *ad hoc* al hardware en el que se desea correr). En contraste, OpenStep solo va a contar con la parte de desarrollo de aplicaciones (como son los objetos de interfaz de usuario del AppKit, los servicios del sistema de ventanas y los mecanismos para permitir la comunicación entre objetos distribuidos) lográndose de esta manera que los fabricantes entreguen NEXTSTEP sobre una capa adicional que se encuentra arriba de su sistema operativo, en lugar de tener que substituirlo. Para cualquiera de los dos ambientes discutidos en este trabajo, es necesario someter al personal de desarrollo a un programa de entrenamiento serio que lo califique en el uso de las facilidades proporcionadas por el mismo, ya que solo así se podrá obtener el mayor rendimiento

posible en el uso de la plataforma. Los resultados de la comparación de los ambientes se resume en las siguientes tablas:

Parámetro de Evaluación	SNAP	NEXTSTEP
¿El lenguaje es un híbrido o es un lenguaje puro o propietario?	Lenguaje propietario.	Híbrido (Objective-C).
¿El lenguaje es fácil de aprender?	Si.	Relativamente, por su naturaleza híbrida.
En la declaración de una clase ¿Se pueden definir elementos adicionales a los atributos y métodos?	Si, se pueden definir *demons*, reglas y tipos.	No.
Para efectos de modularidad ¿Se pueden definir las clases en diferentes archivos?	Si.	Sí.
Para efectos de encapsulación ¿Se pueden definir diferentes secciones en una clase?	Si.	Si.
¿Se puede restringir el acceso a los elementos de una clase a través de la definición de su visibilidad?	Si. Debido a que los atributos y métodos puede ser públicos, privados o protegidos, el nivel de visibilidad en su declaración es el que restringe el acceso.	Por definición del lenguaje los atributos son privados y los métodos públicos (aunque la documentación no es muy explícita al respecto).
Para efectos de ocultamiento de la información ¿Se pueden definir los tres niveles de visibilidad (pública, privada y protegida) a los elementos de una clase?	Si, se pueden definir los tres niveles.	No. Los atributos son privados y los métodos públicos.
¿ El lenguaje soporta mecanismos de herencia sencilla y múltiple?	Sencilla y múltiple.	Sencilla.
¿El lenguaje permite el polimorfismo? ¿Es paramétrico, por sobrecarga o múltiple?	Si. Polimorfismo por sobrecarga.	Si. Polimorfismo por sobrecarga.

Tabla 5. Comparación del modelo de objetos de SNAP y de NEXTSTEP.

Parámetro de Evaluación	SNAP	NEXTSTEP
¿Existe una jerarquía de clases?	Si.	Si.
¿La jerarquía de clases fue diseñada con la finalidad de cubrir necesidades de uso o para simplificar la implantación?	Para cubrir necesidades de uso y simplificar la implantación.	Para simplificar la implantación.
¿Se necesita conocer con detalle la estructura interna del ambiente GUI que se encuentra debajo para usar el producto?	No.	No.
¿Qué tan fácil es derivar nuevas clases a partir de las clases proporcionadas por la biblioteca GUI?	Si.	Si.
¿Cómo son manejados los eventos? ¿Se pueden procesar por medio de la redefinición de los métodos polimórficos y con la asignación de funciones de callback o se tienen que procesar mensajes numéricos usando estructuras de control tipo case?	Los eventos son manejados por medio de funciones de callback.	Los eventos son manejados por medio de los "enchufes" (variables de tipo *id*) que en concepto son similares a las funciones de callback.
¿El componente GUI contiene elementos y clases de alto nivel tales como hojas de cálculo, editores de texto, tool bars, status bars?	Si.	Si.
¿El diseño del componente GUI soporta conceptos puros de la orientación a objetos como el paradigma *Model-View-Controller?*	No.	Si.
¿Que plataformas soporta el componente GUI? ¿Qué compiladores son soportados? Si existe una versión que soporte OSF/Motif ¿Qué plataformas están soportadas?	MS-Windows, OSF-Motif, Windows NT. Compiladores soportados C y C++. Plataformas soportadas: HP UX, VMS, Solaris.	No. El GUI está basado en Adobe Display Postcript y no sigue a ningún estandar.

Tabla 6. Comparación del componente GUI de SNAP y de NEXTSTEP.

Parámetro de Evaluación	SNAP	NEXTSTEP
¿Se puede tener acceso a la capa del API sobre la que está construido el componente GUI?	Si. Como el GUI de SNAP es de tipo intermedio entre un esqueleto de aplicación (application framework) y un constructor de interfaces.	No es necesario. El GUI de NEXTSTEP es de tipo constructor de interfaces.
¿Se pueden crear dialogs, menus, bitmaps e íconos? ¿Se pueden probar los elementos conforme se van construyendo? ¿Se pueden usar elementos "hechos a la medida"?	Si.	Sí.

Tabla 6. Comparación del componente GUI de SNAP y de NEXTSTEP (Continuación).

Parámetro de Evaluación	SNAP	NEXTSTEP
¿Qué interfaces de persistencia ofrece el componente: archivos planos, RDBMS o ambos?	Ambos. Archivos planos y manejadores relacionales de bases de datos.	RDBMS.
Si soporta el uso de archivos planos ¿Cómo se define el mapeo del modelo de objetos a la estructura de un archivo plano? ¿Qué tipos de datos se pueden hacer persistentes con los archivos planos? ¿Hay limitaciones en el crecimiento de los archivos?	Si. El mapeo del modelo de objetos se realiza apoyándose en un archivo de formato el cual define a la clase, sus atributos y los tamaños de los campos. Los tipos de datos que se pueden almacenar son: *str, int y real.* Las limitaciones en el crecimiento de los archivos no se especifican.	No aplica.
En caso de que se sporten a los RDBMS ¿Cómo se define el mapeo de las clases del modelo de objetos con respecto a las entidades en la base de datos? ¿Que sistemas de manejo de bases de datos son soportados?	Por medio de archivos planos que definen las relaciones entre clases y entidades.Los RDBMS sopoprtados son: Informix, Oracle, Sybase e Interbase.	Por medio de *EOModeller* se establece el mapeo del modelo de objetos a las entidades de la BD por medio de un *modelo*. Los RDBMS soportados son: Oracle y Sybase.

Tabla 7. Comparación del componente de almacenamiento permanente de SNAP y de NEXTSTEP.

Parámetro de Evaluación	SNAP	NEXTSTEP
¿Existe una jerarquía de clases que encapsule el comportamiento del componente? ¿La jerarquía fué diseñada para cubrir necesidades de uso?	No existe una jerarquía de clases, solo se proporciona la clase de servicios DBL (por lo que se cubren relaciones de uso).	Si, diseño en capas que encapsula gran parte de la interacción con el RDBMS por medio de *Enterprise Object Framework*.
¿Se pueden derivar nuevas clases a partir de la jerarquía y extender su funcionalidad?	Si.	No es necesario, se cuenta con una arquitectura altamente eficiente.
En el caso de que no se soporte un RDBMS en particular ¿Qué tan fácil es la implantación de los mecanismos necesarios para interactuar con ese manejador de base de datos?	Difícil, se requiere del soporte de Template Software.	Teóricamente si, pero no es práctico ya que se tiene que invertir mucho tiempo.
¿Cómo está organizado el componente: por capas o monolítico?	Monolítico.	Por capas

Tabla 7. Comparación del componente de almacenamiento permanente de SNAP y de NEXTSTEP (Continuación).

Parámetro de Evaluación	SNAP	NEXTSTEP
¿Qué modelos de comunicaciones soporta el componente?	Cliente-Servidor y Peer to Peer.	Cliente-Servidor.
¿Es fácil implantar los mecanismos de comunicación?	Si, se puede recurrir en el modelo cliente-servidor a la facilidad denominada SIB o en el modelo peer to peer por medio de la facilidad IPC. En ambos casos todo se hace apoyándose en archivos planos, lo que hace al proceso más flexible	Relativo, se apoyan en la programación de los Mach ports para definir los mecanismos de comunicación
¿El componente ofrece facilidades para la programación de llamadas a procedimientos remotos (RPC)?	Si, cuando se emplea el modelo de comunicaciones cliente-servidor.	No.

Tabla 8. Comparación del componente de comunicaciones de SNAP y de NEXTSTEP.

Parámetro de Evaluación	SNAP	NEXTSTEP
¿Que tan independiente del SOP es el componente de comunicaciones?	Altamente independiente. Se apoya en los servicios proporcionados por el SOP anfitrión.	Altamente dependiente de Mach (el sistema operativo de NEXTSTEP).
¿Qué estándares de comunicaciones emplea el componente?	TCP/IP y UDP/IP.	Ninguno.
¿Existe una jerarquía de clases que encapsule el funcionamiento de bajo nivel del componente?	Si.	No.
¿Se puede extender fácilmente a la jerarquía y extender su funcionalidad o solo resuelve necesidades de uso?	Si.	No aplica.
Para aplicaciones cliente-servidor corriendo en plataformas heterogéneas de hardware. ¿Es difícil hacer funcionar los mecanismos de comunicación entre procesos?	No siempre y cuando se conozcan las direcciones de los sockets en UNIX y de windows.	No porque todas las aplicaciones siempre corren en el sistema operativo Mach.
¿Qué interfaces de comunicación se pueden emplear para interactuar con el SO Unix?	Sockets.	Mach ports, pero solo están disponibles en el SOP Mach.
Facilidad para adecuar cambios en el esquema de comunicaciones	Buena, el uso de archivos planos facilita las cosas.	Regular, depende en gran medida de las definiciones que se hayan realizado.

Tabla 8. Comparación del componente de comunicaciones de SNAP y de NEXTSTEP (Continuación).

Parámetro de Evaluación	SNAP	NEXTSTEP
¿Cuál es la configuración mínima de hardware y su costo que se necesita para correr el ambiente de desarrollo?	Configuración más barata: Una computadora 386 o superior con disco duro de 250 Mb y 16 Mb de memoria principal, con tarjeta y monitor SVGA (opcional). Costo aproximado: USD $1,500.00	Configuración más barata: Una computadora 486 a 33 Mhz. o superior con un disco duro de 500 Mb (1 Gb es mejor), tarjeta SVGA de 2 Mb local bus y monitor del mismo tipo y CD-ROM SCSI. Costo aproximado: USD $4,500.00
¿Cuál es el costo por licencia de desarrollo y por licencia de usuario?	Licencia de desarrollo: USD $20,000.00. La licencia de usuario: de USD $500.00 a USD $1,000.00	Licencia de desarrollo: USD $2,000.00. Licencia de usuario: USD $800.00
¿Hay que invertir en software adicional y cuál sería su costo?	Windows 3.1 o superior con un costo de USD $40.00 Borland C++ V. 4.0 con un costo de USD $150.00.	No aplica.

Tabla 9. Comparación de los costos de SNAP y de NEXTSTEP.

Parámetro de Evaluación	SNAP	NEXTSTEP
¿Qué tipos de comunicación con software de aplicación externo se puede establecer? ¿Las aplicaciones desarrolladas en el ambiente pueden compartir datos con aplicaciones foráneas?	Se pueden ejecutar aplicaciones externas empleando el comando *run*. La única manera de compartir información con aplicaciones foráneas es por medio de un RDBMS.	No, solo aplicaciones NEXTSTEP
¿Qué lenguajes (y compiladores) de 3a. generación son soportados?	C, C++.	Sólo Objective-C.
¿Interfaces con herramientas CASE tradicionales y orientadas a objetos?	Si, pero limitada	Si, como NEXTSTEP es un sistema operativo además de un ambiente de desarrollo, hay herramientas CASE disponibles que son desarrolladas por diferentes firmas de software

Tabla 10. Comparación de los parámetros adicionales de evaluación que aplican a SNAP y a NEXTSTEP.

CONCLUSIONES

El paradigma orientado a objetos es sin lugar a dudas una excelente alternativa para el desarrollo de sistemas complejos de información.

Los ambientes de desarrollo de software orientado a objetos tienen la finalidad de simplificar el proceso de desarrollo de aplicaciones basadas en el modelo de objetos al ofrecer una serie de facilidades que liberan al desarrollador de gran parte del trabajo de bajo nivel (por ejemplo, la programación de la interfaz gráfica de usuario, los mecanismos de comunicaciones y de persistencia de la información).

El objetivo de esta tesis no es decidir si SNAP es "mejor" que NEXTSTEP o viceversa. El objetivo que perseguimos (y que creemos haber alcanzado) es solamente el de orientar a toda persona involucrada en un proceso de toma de decisiones, planteando tan solo un conjunto de parámetros básicos que sirvan como una guía de carácter general durante el proceso de evaluación. Estos parámetros no son los únicos que se deben considerar y esperamos se vean enriquecidos de acuerdo a las experiencias de todos aquellos que pongan en práctica lo que aquí describimos. Se podría llegar a pensar que en este trabajo se estan comparando "peras" con "manzanas", debido a que SNAP es una aplicación que se comporta como un ambiente de desarrollo y a que NEXTSTEP es un sistema operativo con un ambiente de desarrollo. Sin embargo, los parámetros de evaluación que aquí establecemos solo consideran las características que un ambiente de desarrollo orientado a objetos debe tener. En el caso de NEXTSTEP se debe realizar una evaluación adicional como sistema operativo (cuyos parámetros de evaluación están fuera del alcance de este trabajo) con la finalidad de establecer un punto de comparación más justo, ya que si no se considera esta segunda evaluación se corre el riesgo de descalificar al ambiente NEXTSTEP de una manera poco objetiva.

Un aspecto muy importante que se debe considerar son los costos que implica el integrar un ambiente de desarrollo de software orientado a objetos, no solo desde el punto de vista del precio que tienen las licencias de desarrollo y de usuario; sino que además se tiene que considerar el

costo del hardware necesario para poder ejecutarlo y la inversión que se tiene que realizar en el entrenamiento del personal que va a desarrollar en el ambiente. Como se indica en el capítulo 2 de éste trabajo, la efectividad en el empleo de las herramientas depende del nivel de maestría con el que los desarrolladores puedan usarlas. Esta perspectiva implica la necesidad de someter constantemente al personal de desarrollo a una serie de programas de entrenamiento y actualización sobre el ambiente de desarrollo orientado a objetos que se esté empleando y también sobre el paradigma orientado a objetos.

A final de cuentas, las elección de un ambiente de desarrollo de software orientado a objetos siempre va a depender de las cualidades que el ambiente pueda demostrar despues de ser evaluado y de la estimación de la relación costo-beneficio que se espera obtener cuando se va a realizar una inversión tan fuerte.

BIBLIOGRAFÍA

Referencias bibliográficas sobre Sistemas Orientados a Objetos:

1. Wirfs-Brock, Rebecca, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall, 1990.
2. Booch, Grady. *Object Oriented Design With Applications*. The Benjamin/Cummings Publishing Company, Inc. 1991.
3. Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
4. Michel Adiba, *Object Oriented Database Management Systems*. Memorias del curso impartido en la Segunda Escuela Internacional de Invierno en Temas Selectos de la Computación, Xalapa, Ver. (1991).
5. Antoni Kreczmar. *Object-Oriented Programming: Implementation Problems*. Memorias del curso impartido en la Tercera Escuela Internacional de Invierno en Temas Selectos de la Computación, Cholula, Puebla. (1992).
6. Adolfo Guzmán Arenas. *Diseño y construcción de aplicaciones con objetos*. Memorias del curso impartido en la Cuarta Escuela Internacional en Temas Selectos de la Computación, Mérida, Yucatán (1993).
7. Andrzej Salwicki. *Construction of Software: from the Specification to Implementation in an Object Oriented Programming Language*. Memorias del curso impartido en la Cuarta Escuela Internacional en Temas Selectos de la Computación, Mérida, Yucatán (1993).

Referencias bibliográficas sobre Ingeniería de Software:

8. Fairley, Richard E. *Software Engineering concepts*. Mc.Graw-Hill (1985).
9. Pressman, Roger S. *Software Engineering: A practitoner's approach*. McGraw-Hill (1987).

10. Mauro Pezzè. *Software Engineering: Testing and Verificación.* Memorias del curso impartido en la Segunda Escuela Internacional de Invierno en Temas Selectos de la Computación, Xalapa, Ver. (1991).

11. Carlo Ghezzi. *The Software Process.* Memorias del curso impartido en la Cuarta Escuela Internacional en Temas Selectos de la Computación, Mérida, Yucatán (1993).

Referencias bibliográficas sobre CASE:

12. Barker, Richard. *CASE Method, Tasks and Deliverables.* Addison-Wesley (1990).

13. Sodhi, Jag. *Software Engineering Methods, Management and CASE Tools.* McGraw-Hill (1991).

Referencias bibliográficas sobre sistemas de manejo de bases de datos:

14 Dionysios C. Tsichritzis and Frederick H. Lochovsky. *Data Base Management Systems,* Academic Press (1977).

15 H.F. Korth and A. Silberschatz, *Database Systems Concepts,* Mc.Graw-Hill (1986).

La siguiente referencia es sobre una metodología propietaria de I.B.M.

16. IBM Corp. *Business Systems Planning.* 1987.

Referencias sobre SNAP:

17. Template Software, Inc. *SNAP Technical Brief.*

18. Template Software, Inc. *SNAP User's Guide to the SNAP Development Methodology.*

19. Template Software, Inc. *SNAP Language Reference Guide.*

20. TemplateSoftware, Inc. *User's Guide to the SNAP Graphic User Interface Component.*

21. Template Software, Inc. *User's Guide to the SNAP Permanent Storage Component.*

22. Template Software, Inc. *User's Guide to the SNAP Communication Component.*

23. Template Software, Inc. *User's Guide to the SNAP External Application Software Component.*

Referencias sobre NEXTSTEP:

24. Webster, Bruce F. *The NeXT Book.* Adisson-Wesley (1989)

25. NeXT Computer, Inc. *The NEXTSTEP Advantage: A technical guidebook to object-oriented programming.*

26. NeXT Computer, Inc. *Programming NEXTSTEP.*

27. NeXT Computer, Inc. NEXTSTEP DATABASE KIT CONCEPTS.

Las siguientes referencias son una serie de artículos consultados sobre sistemas orientados a objetos:

28. Henderson-Sellers, Brian, and Edwards, Julian M. "The Object-Oriented Systems Life Cycle". *Communications of the ACM*, September, Vol. 33, No. 9 (1990). pp. 142-159.

29. McGregor, John D. and Tim Korson. "Understanding Object Oriented: A Unifying Paradigm". *Communications of the ACM*, Vol. 33, No. 9 (1990), pp. 40-60.

30. Wong, William G. "Object-Oriented Program Construction". *Dr. Dobb's Journal.* No. 193 October (1992). pp. 36-42.

31. Jicha, Henry. "Object technology explodes with visual programming", *Object Magazine*, Vol. 4, No. 4, July-August (1994), pp. 33-36.

Las siguientes referencias son una serie de artículos consultados sobre Ingeniería de Software:

32. Orr, Ken, Chris Gane, Edward Yourdon, Peter P. Chen, and Larry L. Constantine. "Methodology: The Experts Speak". *Byte Magazine.* April, Vol. 14, No. 4 (1989). pp. 221-233.

33. Swaine, Michael. "Programming Paradigms". *Dr. Dobb's Journal.* No. 193 October (1992). pp. 133-137.

Las siguientes referencias son una serie de artículos consultados sobre CASE:

34. Gibson, Michael Lucas. "The CASE Philosophy". *Byte Magazine.* April, Vol. 14, No. 4 (1989). pp. 209-218.

35. McClure, Carma. "The CASE Experience". *Byte Magazine.* April, Vol. 14, No. 4 (1989). pp. 235-244.

La siguiente referencia es un artículo consultado sobre RDBMS:

36. E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, Vol. 13, No. 6, (June 1970), pp. 377-387.

Artículos consultados sobre las tendencias para la programación de la interfaz gráfica de usuario y sobre la evaluación de ambientes de desarrollo:

37. Walrath, Kathy and Ronald Hayden. "The philosophy of designing a GUI", *Object Magazine*, Vol. 4 No. 4, July-August (1994), pp. 28-32, 44.

38. Lord, D. Hambleton. "Visual programming for visual applications: *A new look for computing*", *Object Magazine*, Vol. 4, No. 4, July-August (1994), pp. 37-40.

39. DeSantis, Joseph. "Evaluating multiplatform development tools", *Object Magazine*, Vol. 4, No. 4, July-August (1994), pp. 41-44.

40. Ibarra Enrique, Gabriel Vergara. "Evaluación del sistema operativo NEXTSTEP en el Grupo Financiero InverMéxico", *Soluciones Avanzadas*, Año 3, No. 18, Febrero 1995.