Universidad Nacional Autonoma de México
Escuela Nacional de Estudios Profesionales
Aragon

" Diseño Orientado a Objetos
Fundamentos y Aplicaciones "

TESIS

Para Obtener el Titulo de:

Ingeniero en Computación

P R E S E N T A

OLGA HERNANDEZ SANCHEZ

ENEP
ARAGON

SAN JUAN DE ARAGON, EDO. DE MEXICO 1995

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Agradecimientos

A Dios :

Por la existencia,
por ser el amigo más fiel,
y por contar contigo siempre.

A mi Alma Mater :

Por la oportunidad que me proporcionó,
por permitirme ampliar mis horizontes,
y por formar una parte esencial en mi vida...

A mi Directora de Tesis, Ing. Silvia Vega Muytoy :

Por la infinita paciencia al revisar bosquejos de una tesis que al inicio, ni yo misma comprendía en su totalidad y, por el apoyo brindado durante tanto tiempo.

A mis revisores :

Ing. Amilcar Monterrosa Escobar,
por su gran disposición hacia el alumno, por ayudar a que nuestro querido Campus "Aragón" prospere, por sus sugerencias y, por los estupendos comentarios que estimularon mi crecimiento...

Ing. Ernesto Peñaloza Romero,
por invitarme a corregir, en repetidas ocasiones, infinidad de detalles, y por mantenerse pendiente del avance alcanzado.

Ing. Roberto Blanco Bautista,
por sus acertadas observaciones y por motivarme a finalizarla.

Ing. José Francisco Montiel Villa,
porque la visión que tuvo del producto terminado, aportó significativas ideas.

A mi papá, Sr. José Hernández Márquez :

Por inculcarnos desde pequeños el mayor legado que la vida nos puede brindar : la educación.

Porque te quiero y ...

" Porque lo que el árbol tiene de florido,
vive de lo que tiene sepultado ... "

A mi mamá Sra. Guadalupe Sánchez Gómez :

Porque te agradezco infinitamente la enorme entrega que la maternidad trae consigo...

Porque no obstante tu renuencia a la profesión elegida, estudiar Ingeniería fue un gran estímulo dentro de una carrera que, ya por sí misma, es todo un reto.

Porque no sólo me has creído capaz de alcanzar mis objetivos, sino también porque me has alentado a tenerlos.

A mi hermana Lupita :

Por el cariño, por la amistad y el apoyo que siempre me has otorgado y, muy especialmente, por respetar mi forma de ser, alentando mi vocación por sobre todas las cosas.

A mi hermano Jaime :

Por la total y absoluta confianza que generosamente has depositado en mi para el logro de los objetivos propuestos...

Por ser el amigo más sincero que haya existido jamás...

Porque estoy plenamente convencida de que aún no existe meta alguna que no consigas alcanzar.

A mi hermanita, Myrna :

Por ser una gran amiga,
por apoyarme con tenacidad,
por soñar con alcanzar las estrellas
y, por la oportunidad de crecer juntas.

A mis pequeñas sobrinas :

Miriam , Citlalli y Angélica ...

A Manuel :

Por brindarme su amistad, su comprensión, su apoyo incondicional y, por invitarme a compartir buenos y malos momentos.

A Angel :

Porque finalmente tenías razón, " El fracaso no te sobrecogerá nunca, si tu determinación para alcanzar el éxito es lo suficientemente poderosa ".

A mi amiga Carmen :

Por "dejarme ser" en todo momento, por estar presente siempre que lo necesité, por tus palabras de aliento y, por valorar la sinceridad.

A mis amigos, Lulú y César, y al pequeño Alex :

Por demostrar siempre y en todo momento que la lealtad y el amor no están disgustados entre sí.

A los CD's, Odis, Efras y su beba, Miriam :

Por mostrarme panoramas diferentes y, por ser tan agradables.

A Juan Manuel :

Por ser el exagerado más grande del mundo y, sencillamente, por hacerme sentir la dicha de contar con un gran amigo.

A tanta gente bonita :

A Rita Carolina Rodriguez Martinez, por ser tan especial.

A mis amigas, Lulú, Lupita, Ariadna, Araceli y Erika.

A los nuevos y ya grandes amigos, Fidel, Eusebio y, Mauricio.

A los amigos que no se olvidan, Selene, Alice, Jenis, Lalo y, Martín.

Con profundo aprecio y admiración, Ing. Margarito Pineda Díaz.

Al prof. de leyenda : Ing. Juan Méndez Moreno.

A un gran amigo, Ing. Martín Contreras.

Y a todas las personas que han hecho posible la culminación de mis estudios profesionales...

A todos ustedes,

¡ Mil y un Gracias !

A Ernesto :

Si la vida se sostiene por instantes
y un instante es el momento de existir,
si tu vida es otro instante, no comprendo,
tantos siglos, tantos mundos,
tanto espacio y, coincidir. ..

Porque en tantos momentos difíciles
me hiciste recordar que :

" Seguro que hay Sol mañana,
dime cuánto apuestas que mañana ,
sale el Sol... "

Por creer en las personas,
por eternizar la niñez en tu corazón,
por ponerle a todo entusiasmo y amor,
por luchar día tras día y,
por confiar en que todo lo logra una sonrisa...

Pero especialmente,
por estimular mi superación de mil y un maneras,
por amar mi libertad,
tolerar mis desvaríos,
y por infundirme confianza suficiente
para lanzar las alas al viento...

Con un nudo en la garganta
sólamente alcanzo a pronunciar : / Gracias /

DISEÑO ORIENTADO A OBJETOS : FUNDAMENTOS, Y APLICACIONES

CAPITULO II) LENGUAJES ORIENTADOS A OBJETOS

CAPITULO III) ANALISIS ORIENTADO A OBJETOS

El Diseño Orientado a Objetos :

La técnica que revolucionará el estilo de programación .
ohs-1995

INTRODUCCION

¿ Cuál es la motivación para realizar el proyecto de tesis profesional bajo ésta metodología ?

En realidad la respuesta no es en modo alguno complicada, como miembro del Area de Ingeniería, afirmo nuestra permanente tendencia a crear, descubrir y, desarrollar. Así es como surge ésta inquietud, manifestándose finalmente en un texto que permita introducir al lector de manera amena, al ámbito de la Metodología Orientada a Objetos.

Tómese en cuenta que hasta hace poco tiempo, en 1992, el Plan de Estudios de la Carrera de Ingeniería en Computación tuvo una actualización (dichos planes de estudio contaban ya con 15 años de vida y, especialmente en la Ciencia de la Computación, es recomendable efectuar una revisión por lo menos cada 2 años), de tal manera que la materia sobre computación que se imparte en el primer semestre de la carrera : Computadoras y Programación, venía impartiéndose bajo el esquema de la Programación Estructurada, con Basic y Fortran como Lenguajes de Programación, actualmente se pretende que el alumno sea capaz de analizar de manera lógica un problema del mundo real, bajo los Paradigmas Estructurado y Orientado a Objetos, cuya herramienta de trabajo es el Lenguaje de Programación C y C++, respectivamente. Entonces éste texto resulta apropiado como referencia bibliográfica para el alumno y como material de apoyo para el profesor, siempre y cuando se desee consultar acerca del Paradigma Orientado a Objetos.

Por otro lado, la principal inquietud consistió en dar vida a un proyecto capaz de representar una aplicación real, procurando utilizar tanto las herramientas que la Carrera aportó, así como la Investigación Científica, Práctica y Metodológica que fuese necesaria. Sin embargo, aún cuando el reto es grande, no lo es así el enorme deseo de plasmar en la tesis un trabajo interesante y, verdaderamente útil para las nuevas generaciones, con inquietudes aún mayores a las nuestras pues su momento así lo exige, ya que :

*" La Computación es una ciencia con evolución creciente e indefinida, provocando esto que su vigencia resulte verdaderamente efímera , lo cual no significa que no sea útil, sino por el contrario, se mantiene en permanente crecimiento, implicando esto : **actualización continua** ... "*

No se descarta la posibilidad y la necesidad de que existan trabajos de tesis que empleen la "Metodología Orientada a Objetos" con enfoques diferentes, dado que el área citada es realmente extensa y, como referencia pueden citarse : **el Análisis Orientado a Objetos, el Diseño Orientado a Objetos , la Programación Orientada a Objetos y, las Bases de Datos Orientadas a Objetos, entre otros;** lo cual nos proporciona una idea de la amplitud de este apasionante tema. El enfoque perseguido : **DISEÑO ORIENTADO A OBJETOS** es, finalmente, una luminosa estrella dentro de la constelación de tópicos que conforman el "Universo la Computación".

Se considera relevante comentar que con la finalidad de estudiar a detalle éste texto y poder aprovecharlo al máximo, es deseable que el lector sepa programar, es decir, que posea la capacidad de resolver de manera lógica un problema del "mundo real", que sea capaz de modelar un problema dado auxiliado por una Computadora; también es deseable que el lector cuente con nociones del Lenguaje de Programación "C". Sin embargo, influirá en gran medida, el esfuerzo y empeño que el lector brinde al presente trabajo para lograr su total aprovechamiento.

Como parte de una breve reseña, durante la lectura del presente material se observa que :

EL CAPITULO I proporciona al lector un esquema de lo que representa el **"Paradigma Orientado a Objetos"**, introduciéndolo en los Fundamentos de la Tecnología Orientada a Objetos.

EL CAPTIULO II da a conocer diferentes **"Lenguajes Orientados a Objetos"**, así como los factores que todo programador debería considerar en la elección de alguno, se

hace un análisis comparativo entre los lenguajes "C++", **Eiffel** y **Smalltalk**, finalmente, se define el por qué se emplea el Lenguaje "C++" como herramienta de desarrollo en el presente trabajo.

EL CAPITULO III, habla sobre el **"Análisis Orientado a Objetos"**,en qué consiste, sus requerimientos básicos, cómo se desarrolla, y, se muestra un ejemplo de su aplicación.

EL CAPITULO IV, nos introduce finalmente en la materia en cuestión : **"El Diseño Orientado a Objetos"**, presentando la utilidad, ventajas y desventajas que su empleo nos proporciona, el concepto de reusabilidad que nos permite ahorrar tiempo, esfuerzo y, costos; Además muestra en qué punto se considera que inicia la fase de Diseño y lo que ello implica.

EL CAPITULO V, muestra la aplicación práctica del ejemplo que se maneja durante los capítulos III y IV, un Videojuego : ***" Citlalli " (estrella)***, se podrá apreciar al inicio del mismo, un esquema que define las relaciones entre las clases y las instancias empleadas, así mismo, se cita nuevamente la Especificación de Requerimientos manejada en el Análisis del juego, de tal manera que, si el lector se encuentra interesado en consultar de manera independiente del resto del texto el presente capítulo, podrá hacerlo. Se citan todos los listados que conforman el juego de video.

Finalmente, en el **CAPITULO VI**, se presentan las conclusiones originadas a raíz del conocimiento de la Tecnología Orientada a Objetos, contemplándola entre otras cosas, como una herramienta eficaz de trabajo siempre que se desee desarrollar y optimizar los recursos disponibles para el Diseño de Sistemas.

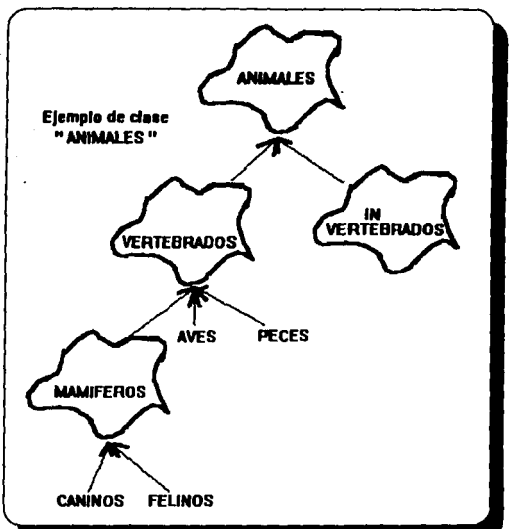
Con la finalidad de no restringir al lector en el uso de una Metodología de Análisis y de Diseño Orientado a Objetos en particular, se incluyen los **Apéndices "A" y "B"** que exponen diversas técnicas empleadas por autores expertos en la materia.

Así pues, deseando aportar al lector panoramas e inquietudes distintos en el mismo contexto del ámbito de la Computación y, deseando profundamente que los temas presentados le resulten tan apasionantes como a mí me lo han parecido, quedo de usted:

ATENTAMENTE,

Olga Hernández Sánchez

CAPITULO I
CONCEPTOS FUNDAMENTALES
DE LA METODOLOGIA ORIENTADA A OBJETOS



Ninguno de los mejores métodos orientados a objetos son perfectos. La educación podría incluir *conceptos fundamentales* tan buenos como métodos específicos y, algunas adecuaciones podrían ser necesarias.

Martín Fowler.

CAPITULO I
CONCEPTOS FUNDAMENTALES
DE LA METODOLOGIA ORIENTADA A OBJETOS

I.1) ¿ Qué es un Objeto ?

Para comenzar con el estudio de los objetos, lo más conveniente para nosotros sería tener claro el concepto objeto, con tal fin iniciamos con su definición :

" Un **objeto** *es una unidad de software que contiene datos y procedimientos interrelacionados ". Los datos se denominan variables, porque definen el estado del objeto en cualquier momento y, los procedimientos suelen llamarse Métodos, quienes se encargan de la definición del comportamiento de un objeto.*

Como ejemplo de **objeto**, considérese un reloj, pero de un tipo en particular, un cronómetro, con él será posible considerar : * tiempo inicial, * tiempo final, * tiempo transcurrido, para que exista un objeto, éste debe ser enmarcado en una clase (conjunto de objetos), la cual se denominará en lo sucesivo clase **CRONOMETRO**, ésta permitirá definir los objetos "cronómetros". La clase como módulo ofrece la funcionalidad siguiente :

* una operación **Arranca** para echar a andar un cronómetro;
* una operación **Para** para detener el cronómetro y,
* una operación **Muestra** para desplegar el tiempo que marca el cronómetro.

Obsérvese que, para cuando se aplique "Muestra" y el cronómetro se encuentre detenido, se deberá desplegar el tiempo que marcaba el cronómetro en el momento de pararse. Estas operaciones sólo pueden solicitarse a través de un objeto de la clase (porque es el objeto el que las lleva a cabo). Bajo el nombre **"MiCrono"**, se nombrará a un objeto de la clase CRONOMETRO, al cual podrán enviársele mensajes, éstos mensajes serán las operaciones que podrán realizarse sobre el objeto (cronómetro), hasta éste momento se han definido dos tipos de operaciones, las de transformación (Arranca y Para) y, las de acceso (Muestra).

Expresado en lenguaje C++, la instrucción tendría el formato siguiente :

CRONOMETRO MiCrono; // Se indica que el objeto MiCrono es del tipo de la clase Cronómetro

MiCrono.Arranca(); // Se llama a la operación Arranca sobre el objeto MiCrono

MiCrono.Para(); // Se llama a la operación Para sobre el objeto MiCrono

MiCrono.Muestra(); // Se llama a la operación Muestra sobre el objeto MiCrono

En ocasiones, el diseño de un sistema resulta más sencillo cuando procuramos auxiliarnos de alguna herramienta gráfica de software. Por tal razón, podría surgirnos la inquietud : ¿ Cómo representar a un objeto ?. Este cuestionamiento se resuelve fácilmente : ¿ De qué manera ? en el próximo apartado se indica la forma.

I.1.1) Notación en Objetos

En casi todas las ramas de la Ciencia (ó al menos en la gran mayoría), se emplean diagramas, dibujos, esquemas, etc., que apoyan un trabajo didáctico, por ejemplo obsérvese el caso de la Enseñanza, la cual se apoya fundamentalmente con el empleo de pizarrones; o bien constituyen la parte medular de una profesión, por ejemplo : los planos Arquitectónicos, los diseños que efectúan los Ingenieros Eléctricos, Mecánicos, Civiles; las gráficas para la realización de Estadísticas muy útiles en el terreno de la Economía y, particularmente, en la Ingeniería de Software, la representación de las ideas es mucho más sencilla y proporciona mayor entendimiento si empleamos esquemas o figuras que nos representen quizá un tipo de diseño, la estructura de un programa o como en éste caso, la definición de un concepto.

Pero, antes de continuar, analicemos las ventajas que generosamente nos brinda la *"notación"*, iniciemos con ***"la estandarización"***, mundialmente hablando existen símbolos conocidos que son del dominio público, tales como : la muerte, una señal de alto, cruce de ferrocarriles, etc., dichos símbolos no requieren asociación de texto en un idioma determinado,

se explican por sí mismos, además no es necesaria la traducción o la investigación que los fundamente, si empleamos entonces ***"notación estandarizada"*** tendremos por resultado accesibilidad al concepto, independientemente de la metodología empleada, lenguaje de programación u enfoque citado, lo cual nos lleva a notar que no se desvirtuará la atención del tema en cuestión y, por consiguiente, habrá consistencia en los temas abordados.

La **Fig. 1.1** representa a un objeto en notación icónica, el nombre del objeto es básico pues le proporciona identidad y, es también indispensable dado que de ésta forma puede ser llamado (aún cuando el sistema sea de grandes proporciones). Por otro lado, los nombres de los objetos necesitan no ser únicos, pues de ésta forma pueden denotar alguna instancia indefinida que, sin duda alguna, es representativa de la abstracción. A consideración del diseñador, es posible colocar las propiedades del objeto en la esquina inferior izquierda de la figura, los iconos para clases son los únicos que pueden incluir adornos tales como cardinalidad (Norte, Sur, Este, Oeste). Las relaciones entre objetos son interpretadas como el envío de mensajes entre objetos, dado que éstos mensajes son bidireccionales *(parten del objeto_1 y llegan al objeto_2 y, a su vez, el objeto_2 envía una respuesta al objeto_1, es decir, en dos direcciones)*, éstas relaciones se pueden señalar mediante líneas que se encargan de indicar el acceso o salida del sistema.

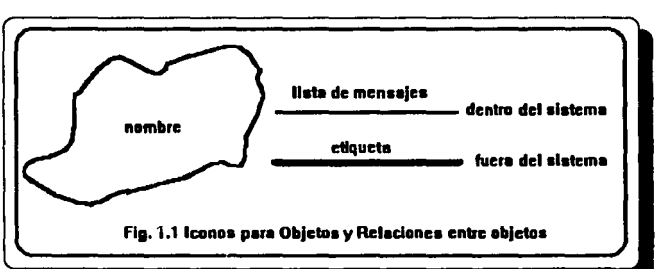


Fig. 1.1 Iconos para Objetos y Relaciones entre objetos

La programación orientada a objetos comienza con una idea más abstracta : el ¿Qué...?. Primeramente se averigua sobre lo que intenta el programa, se buscan los objetos y sus conexiones, para lograrlo se investiga qué operaciones se necesitan desarrollar y qué información resulta de esas operaciones. Cada objeto sabe lo que puede hacer (operaciones) y

sobre qué lo puede hacer (datos). *Entonces se llega a descomponer un sistema en entidades que se ajustan a las reglas dentro de dicho sistema, dado que conocen todo sobre sí mismas.*

La información normalmente se maneja en dos tipos diferentes : ***funciones y datos. En la programación procedimental inmediatamente se trabaja sobre las funciones y las particularidades de los datos que van a intervenir, es decir, se maneja el*** **¿Cómo...?**

Booch nos dice que ***"El diagrama de un objeto se emplea para mostrar la existencia de objetos y su relación en el Diseño Lógico de un Sistema",*** razón que ciertamente nos parece del todo congruente, ¿por qué?, porque si concebimos que la creación de un objeto es de tipo transitorio (puede ser destruido en cualquier momento según se requiera), lo cual no ocurre así en una clase, entonces **tenemos como factor decisivo para el diseño de diagramas de objetos la relación existente entre los objetos y la(s) clase(s) a utilizar,** si no estamos conscientes -desde el momento del diseño- de la relación que emplearán los objetos entre sí, nuestro diagrama con el más mínimo cambio, resultaría obsoleto.

Para ejemplificar el concepto de objeto contemplemos la representación automatizada de un robot (un brazo mecánico), el cual a su vez es capaz de efectuar diversas acciones : movimientos laterales (izquierda, derecha), longitudinales (arriba, abajo), rotaciones de su "antebrazo" y su "mano" en un ámbito de 360 grados, movimientos predeterminados en la dirección que el usuario designe : recoger artículos entre sus "dedos", soltarlos, etc. También es deseable que se conserve la información que resulta inherente en su funcionamiento, tal es el caso de : capacidad de carga, capacidad de movimiento, velocidades : máxima, mínima, etc.; similarmente concierne a nuestro interés su estado actual : contenido, posición, orientación y velocidad.

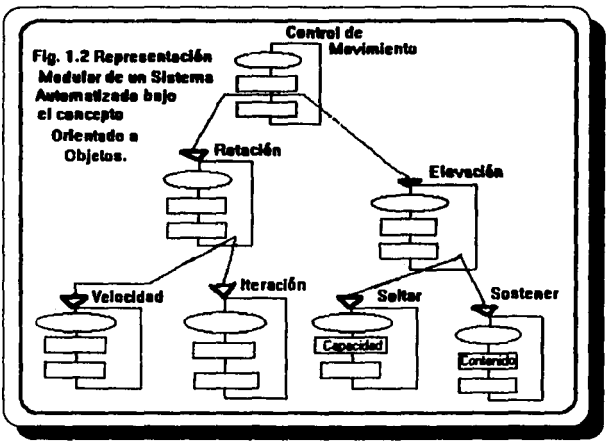


Fig. 1.2 Representación Modular de un Sistema Automatizado bajo el concepto Orientado a Objetos.

En la **Fig. 1.2**, las *variables* mostradas son *"Capacidad"* y *"Contenido"*, los *métodos* son los encabezados de cada módulo : *"Control de Movimiento"*, *"Rotación"*, *"Elevación"*, etc.

Es interesante saber que : ***" Todo lo que un objeto conoce está expresado en sus variables y, todo lo que un objeto puede hacer está expresado en sus métodos ".***

Dado que el concepto de Objeto ya ha sido identificado, es posible y viable, considerar ahora el término de ***"Abstracción de datos"***, quien resulta de particular importancia para nuestro fin, si ponderamos que **la base del Diseño Orientado a Objetos (DOO) es : *"La construcción de sistemas como colecciones estructuradas de tipos de datos abstractos".***

I.2) Abstracción de Datos

El término ***"abstracción"*** se acompaña de diversas ideas y es un poco más difícil de entender. El Oxford English Dictionary (OED) señala a la abstracción como *"El acto de la*

separación en en el pensamiento", esto no parece tener relación alguna con el tema en cuestión, sin embargo, una definición más apropiada para los fines deseados podría ser ***"Representación de las características esenciales de algo sin incluir conocimiento previo o detalles no esenciales".***

La gente comprende el mundo al modelar porciones de él en versiones simplificadas. Un modelo debe ser un bosquejo y no debe intentar manejar hasta el último detalle, ya que de lo contrario resultaría igualmente complicado de lo que se intenta entender y modelar. En realidad, lo que se pretende es ***simplificar***, comúnmente las personas modelan el mundo real en pequeñas porciones, es decir, en versiones simplificadas. Al modelar no se pretende detallar todo lo más posible, al contrario, entre más sencillo sea el modelo creado, más útil será.

Por ejemplo, evaluemos el uso de un aparato electrodoméstico, dícese una lavadora, una secadora, una licuadora, una batidora, etc.; con ellos el ama de casa NO requiere comprender muchos de los detalles que implican su funcionamiento (si tuviera que comprenderlo, ¿, qué tan comerciales serían estos aparatos ?), es decir, se abstrae el conjunto de cientos de piezas distintas a una sola idea: el aparato electrodoméstico. Debido a ello es que ***"la abstracción constituye la clave para diseñar un buen software".***

Si al concepto de ***"abstracción"*** aunamos el de ***"encapsulación"*** de todas las propiedades esenciales de una cosa, resulta entonces que el concepto de abstracción se encontrará más completo. El término en cuestión lo es ahora la ***encapsulación(1)***, en donde las estructuras de los datos y detalles de la implantación son ocultados por otros objetos en el sistema. La única forma disponible para accesar el estado de un objeto la obtenemos mediante el envío de uno o varios mensajes, quienes provocarán que uno de los métodos se ejecute. Estrictamente hablando, los atributos son escritos en código especial para los métodos, por ejemplo : obtener y colocar valores.

Hablando de Metodología Orientada a Objetos, dos ideas principales son :

(1) "Object Oriented Methods", Ian Graham, 1992.

¿ Qué puede hacer éste objeto? y ¿ Qué conoce éste objeto?.

La primer pregunta se refiere a las operaciones o funciones que puede realizar y, la segunda se refiere a los datos que se implican en esas operaciones.

Finalmente, ***encapsulación*** significa : ***reunión en un sólo objeto de los datos y las operaciones que afectan esos datos.*** *Con la encapsulación se logra ocultar cierta complejidad hacia otros objetos y hacia el diseñador; esta complejidad es el* ***cómo*** *hace las cosas, lo cual no es de interés para otras partes del sistema, lo que sí les interesa es el* ***qué*** *puede hacer. El* ***qué*** *es la parte* ***pública****, lo único que los otros objetos* ***deben*** *conocer.*

Se entiende entonces que, un tipo de dato abstracto es una abstracción. Los tipos de datos abstractos pueden ser definidos por un usuario en la construcción de una aplicación, de preferencia por el Diseñador. Un tipo de dato abstracto define datos conjuntamente con las operaciones que los manipulan. Obsérvese la **Fig.1.3** :

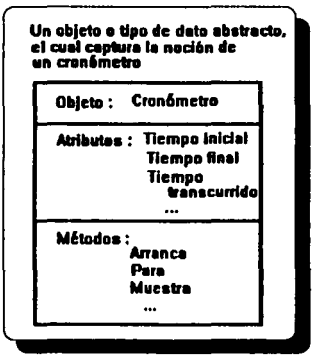


Fig. 1.3 Ejemplo cotidiano del concepto de abstracción.

Ahora, en la **Fig. 1.4** se muestra el mismo ejemplo de abstracción en su expresión icónica representando a la Clase Figura Gráfica.

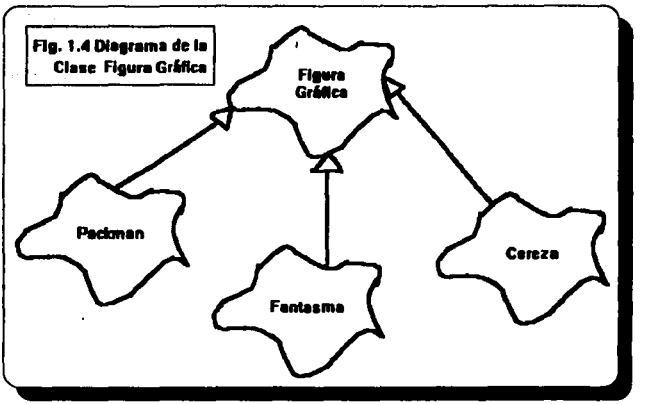


Fig. 1.4 Diagrama de la Clase Figura Gráfica

Concretando, consideremos que la ***abstracción es el proceso de identificación relevante de objetos en la aplicación e, ignora el conocimiento previo irrelevante.*** *La abstracción libera la reutilización a través de la encapsulación u ocultamiento de la información.* ***"La encapsulación consiste en ocultar la implantación de objetos y la declaración pública de la especificación de su conducta por medio de atributos y operaciones".*** Las estructuras de los datos y los métodos que los implantan son privados al objeto.

Los beneficios obtenidos con la abstracción son múltiples :

- *disminuye la complejidad al permitir al usuario manipular los valores sin conocer los detalles ligados a una implantación y,*
- *protege los datos evitando que el usuario acceda a su estructura interna y la cambie sin usar las funciones especialmente diseñadas para ello, afectando así la confiabilidad del sistema.*

Los lenguajes de programación que apoyan la abstracción de datos permiten agrupar en una misma declaración : los datos y las operaciones que los manipulan, ocultando los detalles

de implantación. Tradicionalmente, en los lenguajes de programación no existía la posibilidad de ocultar la representación concreta elegida para los valores, ahora, con el empleo de los LOO's *(Lenguajes Orientados a Objetos)*, la abstracción es realmente aplicable y, constituye el objetivo por alcanzar en el desarrollo de los sistemas implantados mediante éste enfoque.

Es deseable comentar brevemente sobre un tema que se encuentra relacionado de alguna forma con los conceptos "abstracción" y "encapsulación", el de ***ocultamiento de la información***, al manejar la complejidad en una caja negra, debe haber alguna manera que indique lo que algún objeto en particular puede hacer, es decir, se está empleando una ***representación privada y una interfaz pública :*** el denominado *ocultamiento de la información*, cuya representación gráfica se muestra en la **Fig. 1.4.1 :**

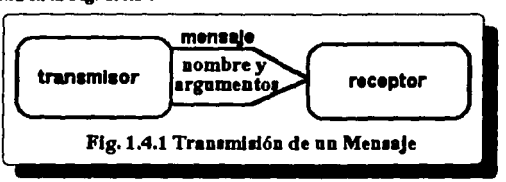


Fig. 1.4.1 Transmisión de un Mensaje

Al ocultar la información, un objeto no dice cómo hace su trabajo, pero públicamente se sabe lo que el objeto es capaz de realizar. Adicionalmente, como un objeto tiene una parte privada, ésta puede ser modificada sin alterar el equilibrio del sistema, ya que si un error es introducido, éste queda enclaustrado en la parte privada del objeto y es más fácil detectarlo y corregirlo.

Continuando con el ejemplo del objeto Cronómetro, consideremos que internamente en su memoria, existe una componente **"pri_tm"**, de tipo **time (2)** que tendrá la hora que marcaba el sistema cuando se echó a andar el cronómetro (si está parado tendrá el tiempo que marcó el cronómetro mientras estuvo andando). Para definir el funcionamiento de la función Muestra, necesitamos saber si el cronómetro está andando (para en este caso mostrar el tiempo que lleva andando) o si está parado (para que nos dé el tiempo que marcaba cuando se paró). Se necesita entonces, que el cronómetro posea también en su memoria una indicación de si está andando o

parado. Lo cual podrá apreciarse con una componente "andando" de tipo short int (0 parado, 1 andando).

(2) C++, como híbrido con C, incluye todos los tipos primitivos predefinidos de C con todas sus operaciones y funciones conocidas. Particularmente, Borland C++ incluye en la librería dos.h el tipo time y la función gettime que nos dá la hora marcada por el reloj del sistema (Este ejemplo depende de la comunicación de BORLAND C++ con el sistema operativo MS-DOS, cualquier otro sistema operativo y compilador de C++ deben incluir recursos similares). La definición del tipo time es :

```
struct time
{  unsigned char ti_min;     // minutos
   unsigned char ti_hour;    // horas
   unsigned char ti_hund;    // centésimas de segundo
   unsigned char ti_sec;     // segundos
}
```

el esquema de gettime es : `void gettime (struct time *t);`

Estas dos componentes *pri_tm* y *andando* forman parte de cada objeto CRONOMETRO y, podrán ser usadas por las funciones que éste lleva a cabo. Esta funcionalidad de un cronómetro se puede encerrar (encapsular) dentro de la definición de una clase CRONOMETRO. La forma en la que se encapsula consiste en que, ésta clase caracteriza el tipo de todos los objetos que pueden llevar a cabo esta funcionalidad.

NOTA : Con la finalidad de tener una idea concreta de lo que se pretende realizar con el objeto C (cronómetro), se colocó en primera instancia el programa principal correspondiente, sin embargo, recuérdese que para efectos de compilación, al menos en Turbo C++, se deben incluir, en primer término, las bibliotecas a usar, en seguida la definición de las clases o, en su caso, incluir los archivos con las definiciones tanto de clases como de métodos requeridos por el programa principal. Se emplea la convención, archivos con **extensión ".h"**, para aquellos **archivos auxiliares** y, se emplea la **extensión ".cpp "**, para el **programa principal**

En el **Listado I.1** se muestra el programa principal para el manejo de un cronómetro, se pretenden realizar las siguientes acciones :

```
int main(void)
{
CRONOMETRO MiReloj;          // Definición del objeto MiReloj, tipo CRONOMETRO
 int calculo;
```

```
 MiReloj.Arranca();                    // Se inicializa contador, operación : Arranca
 printf("Evalue la expresión ((2*(1+1)-1)+3) y diga su valor ");    // Intervalo de tiempo
 scanf("%d",&calculo);                 // Lee valor
 MiReloj.Para();                       // Detiene contador , operación : Para
 printf("Usted tardo ");
 MiReloj.Muestra();                    // Muestra tiempo transcurrido, operación : Muestra
 return 0;
}
```

Listado I.1 Programa principal para la manipulación de un objeto cronómetro

En el **Listado I.2** se hace referencia a las operaciones empleadas para el objeto cronómetro.

```
#include "dos.h"          // Funciones requeridas en ésta biblioteca : struct time, gettime ...
#include "iostream.h"     // Funciones requeridas en ésta biblioteca : printf, scanf ...

class CRONOMETRO
{
 private:
   struct time pri_tm;
   short int andando;
   void dif_tm(time t2, time t1, time *dif)
   {
    // Diferencia entre dos horas
    // Usada internamente por Muestra y Para

    if (t2.ti_hund < t1.ti_hund)
      { t2.ti_hund+=100; t1.ti_sec++; };

    dif->ti_hund = t2.ti_hund - t1.ti_hund;

    if (t2.ti_sec < t1.ti_sec)
      { t2.ti_sec+=60; t1.ti_min++; };
    dif->ti_sec = t2.ti_sec - t1.ti_sec;

    if (t2.ti_min < t1.ti_min)
      { t2.ti_min+=60; t1.ti_hour++; };
    dif->ti_min = t2.ti_min - t1.ti_min;

    if (t2.ti_hour < t1.ti_hour)
      t2.ti_hour+=24;

    //Se supone que la diferencia de tiempo nunca es mayor de 24 horas

    dif->ti_hour = t2.ti_hour - t1.ti_hour;
   }
```

```
 public:

// Inicializa el cronómetro
   void Arranca( )
   {
     andando=1;
     gettime(&pri_tm);
   }

// Despliega el resultado en pantalla
   void Muestra( )
    {
      if (andando)
       {
         // Está  andando, dar el tiempo que marca y seguir andando
         time actual_tm, dif;
         gettime(&actual_tm);

         dif_tm(actual_tm, pri_tm, &dif);
         printf("%d:%d:%d:%d\n", dif.ti_hour, dif.ti_min,dif.ti_sec, dif.ti_hund);
       }
      else
      {
        // Está  parado, la componente pri_tm indica el tiempo que marcó cuando fue detenido
        printf("%d:%d:%d:%d\n", pri_tm.ti_hour,   pri_tm.ti_min, pri_tm.ti_sec,
pri_tm.ti_hund);
      }
    }

// Detiene el cronómetro
   void Para( )
   {
     if (andando)
      {
        // Está andando, pararlo y dejar en la componente pri_tm, el tiempo marcado actual_tm;

     time actual_tm;
     gettime(&actual_tm);
     andando=0;
     dif_tm(actual_tm, pri_tm, &pri_tm);
     };
     //Si ya está  parado no hacer nada
    }
};
```

Listado I.2 Definición de la Clase Cronómetro

I.3) Tipificación

Imaginemos que somos un programador y tratemos de pensar como tal, entonces encontraríamos diferencia entre clases y tipos, consistiendo ésta en que la clase describe especificaciones que pueden compartirse entre colecciones de objetos y otras clases, no sólo entre objetos con identidad única o instancias. Sin embargo, si fuésemos Analistas de Sistemas, las clases y los tipos de datos abstractos serían lo mismo para nosotros.

La clasificación de objetos en tipos se genera para categorizar objetos de acuerdo a su uso y comportamiento entonces, naturalmente la clasificación existirá en cualquier dominio.

"Un tipo determina las características estructurales y de comportamiento de los objetos que le pertenecen. El objetivo de un sistema de tipos es estructurar el Universo de objetos de tal forma que se regule su uso en diversos contextos".

Para ejemplificar lo anterior emplearemos el Sistema Numérico como medio de referencia, como es sabido, existe una amplia gama de "tipos de números" existentes, entre ellos existen :

Número Abstracto
Número Concreto
Número Primo
Número Romano

de ésta forma notamos que el **número** es el **tipo** que nos permite modelar las características de los elementos que lo componen y con él podemos aplicar, en función de su tipificación, el tipo de número que necesitemos.

Existen tres variantes en la validación de errores de tipos :

I.3.1) Tipificación : "Sin validación"

En ella se ignora el problema : ***"El ambiente no realiza validación alguna y el error puede ser detectado inmediatamente en caso de que la computadora no pueda efectuar la operación adscrita, pero es posible que el error no sea detectado y un cálculo erróneo prosiga como si hubiera sido correcto".*** Este tipo de validación es adecuado por ejemplo, para un Lenguaje de Bajo Nivel.

Un punto importante a señalar en éste tipo de validación radica en que si el sistema desarrollado podría contener problemas de lógica (en la construcción de los resultados deseados), la tipificación sin validación posiblemente no los detecte; aún cuando el programa no presente problemas en su ejecución, si ésta situación se presenta y el diseñador no lo detectáse, las consecuencias podrían ser fatales, dado que los cálculos que precise éste algoritmo no serían correctos *-por cierto, todo proceso es importante no obstante el impacto que produzca su magnitud-*.

I.3.2) Tipificación : " Validación Dinámica "

Cada objeto tiene asignado un tipo (o familia de tipos). ***"La validación de tipos se realiza en tiempo de ejecución"***, es decir, la validación dinár ıica se efectúa al momento de compilar el programa.

Este tipo de validación es útil en ambientes usados para experimentar ideas o generación de prototipos. Proporciona un margen de seguridad confiable puesto que los errores se detectan dentro de una gran flexibilidad, lo cual es básico en todo buen sistema de cómputo.

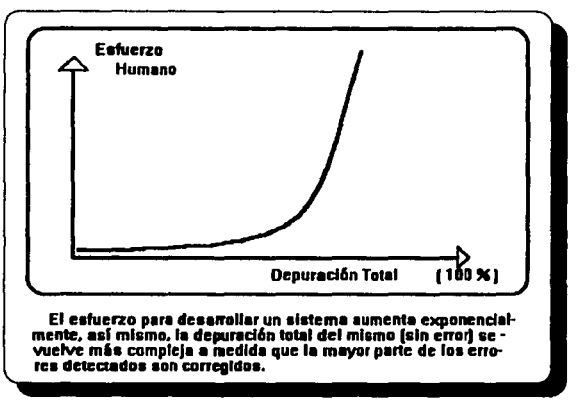
I.3.3) Tipificación : " Validación Estática "

"La validación estática es aquélla en la cual la tipificación ocurre en tiempo de compilación. Las validaciones ocurren antes de ejecutar el proceso y son fuertes y

anticipadas. Si desea hacerse una modificación es necesario recompilar. Este tipo de validación otorga una confianza muy grande en el programa, pero es muy rígida".

La validación estática dá el grado de seguridad necesaria para sistemas empleados en producción, donde se requiere que los programas se encuentren tan libres de errores como sea posible.

Al respecto, existe una gráfica que expresa ampliamente como la depuración total de un sistema es un objetivo difícil de alcanzar, pues ésta posibilidad se decrementa a medida que la mayor cantidad de errores detectados son corregidos.



El esfuerzo para desarrollar un sistema aumenta exponencialmente, así mismo, la depuración total del mismo (sin error) se vuelve más compleja a medida que la mayor parte de los errores detectados son corregidos.

1.4) ¿ Qué es una Clase ?

*Cuando se clasifica a un **objeto** como miembro de una **clase**, se establece que el objeto posee una serie de características propias de su clase.* Formalmente hablando :

" Una clase es un prototipo que define los métodos y variables que serán incluidas en un tipo de objeto particular "

Expresándolo de otra manera :

" Una clase representa una agrupación de objetos que comparten una estructura común y un comportamiento común "

Los objetos que pertenecen a una clase se denominan **"instancias de la clase"** *y, contienen solamente los valores particulares para las variables, compartiendo el código de los métodos ".*

Al codificar un sistema, definiremos las descripciones tanto de los métodos como de las variables, esto ocurrirá exclusivamente una sóla vez dentro de la definición misma de la clase.

Empleando ahora el ejemplo de "Figuras Geométricas", definamos ahora una clase que sea capaz de manipularlas, en la **Fig. 1.5** se aprecian algunas instancias de dicha clase.

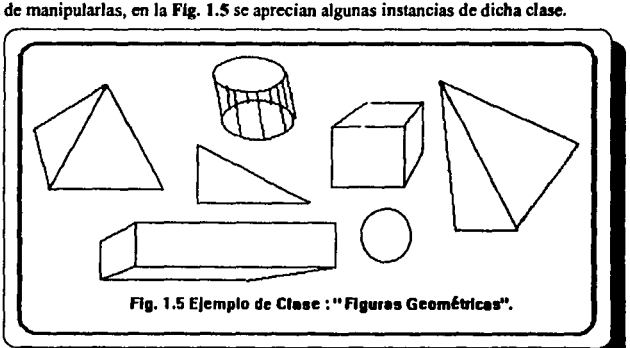


Fig. 1.5 Ejemplo de Clase : " Figuras Geométricas".

En la POO (Programación Orientada a Objetos), las clases soportan la abstracción de datos ya que comprenden la declaración de los datos que contendrá cada objeto, así como las

operaciones que podrán invocarse sobre ellos. La interfaz de una clase cumple con una doble tarea, por un lado especifica qué operaciones están disponibles sobre los objetos de la clase y, por otro lado, regula el acceso a las operaciones, ocultando los servicios correspondientes de acuerdo al cliente.

*Casos especiales de una clase se conocen como **subclases de esa clase**; la clase más general de las **subclases** se conoce como la **superclase o clase base** de sus clases especializadas. Y, contamos inclusive, con clases concretas, para profundizar un tanto en éstas definiciones, véanse los próximos apartados.*

1.4.1) Clases en C++

Según *Naba Barkakati* en su libro Programming Oriented Object : *"una clase también es conocida como un tipo de dato abstracto"* (los cuales nos ayudan a ligar datos y funciones en una misma definición), por cierto, la clase efectivamente define un nuevo tipo de datos con sus operaciones propias.

Para definir el formato de una clase en C++ se emplea la palabra **class**, la cual encabeza la definición tanto de los datos como de los métodos por emplear. Es importante observar que la definición de una clase es similar a la de una estructura en el Lenguaje "C", sin embargo una clase no solamente puede definir los datos a emplear, también se encarga de los métodos que podrán manipular ésos datos, es decir, ***los miembros de una clase son : datos y funciones.***

La declaración de una clase en C++ es a su vez la interfaz de la clase. Una clase específica en forma sintáctica todos los atributos y servicios, llamados datos y funciones miembro de los objetos que se crearán a partir de ella, éste tipo de interfaz es meramente sintáctica.

Para diferenciar cuáles recursos son internos del objeto y cuáles públicos, C++ introduce dos tipos de secciones : la sección **private** y la sección **public**. El uso del término **private** es opcional ya que se asume por omisión que todos los recursos son privados. En cada sección pueden describirse datos y funciones. Los recursos que aparecen en la sección **private** son

privados de la clase y sólo pueden ser usados en las implantaciones de las propias funciones de la clase. Los recursos que aparecen en la sección **public** pueden ser usados por funciones que no pertenezcan a la clase pero siempre a través de un objeto de la clase.

Las reglas de acceso *(visibilidad)* a los miembros de una clase se dividen en tres niveles :

* ***público***.....accesibles a cualquier clase o función del sistema.

* ***privado***.....accesibles solamente a las funciones de la clase misma.

* ***protegido***.....accesibles únicamente a las funciones de la clase misma y de las clases derivadas o subclases.

Dados éstos filtros de acceso o bien de visibilidad, tenemos la posibilidad de distinguir entre los clientes externos de una clase y los clientes que son sus herederos de ella, de ésta forma, nosotros mismos somos capaces de otorgar ciertos privilegios a los clientes herederos.

```
//Formato general para la declaración de una clase
class nombre_clase
{
public:
      tipo variable;
      tipo variable;

      tipo nombre_función(tipo variable,...);
      tipo nombre_función(tipo variable,...)
      {
        bloque de sentencias
      }
protected:
      tipo variable;
      tipo variable;

      tipo nombre_función(tipo variable,...);
      tipo nombre_función(tipo variable,...)
      {
        bloque de sentencias
      }
```

```
private:
      tipo variable;
      tipo variable';

      tipo nombre_función(tipo variable,...);
      tipo nombre_función(tipo variable,...)
      {
        bloque de sentencias
      }
};
```

Listado 1.3 Formato general para la declaración de una Clase.

Suele ser práctica común entre los programadores el utilizar archivos denominados de encabezado para las declaraciones de clases y otros para las implantaciones de funciones. Los archivos de encabezado se distribuyen junto con la biblioteca de funciones compiladas. De tal forma que, la relación entre la declaración de una clase *(la interfaz)* y, la implantación de sus funciones miembro puede significar carácter de acoplado y/o desacoplado, según convenga al implantador.

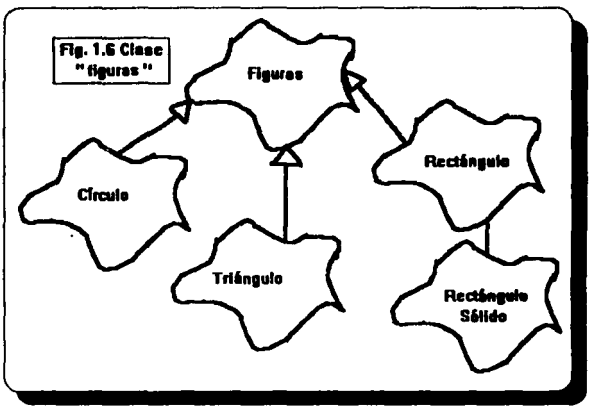
I.4.1.2) Las clases figuras

El primer paso que se recomienda seguir en la implantación de figuras geométricas en C++ es definir las clases. Para comenzar tenemos una clase base abstracta, la clase llamada *"figura"*. La consideramos abstracta porque no pensamos crear instancias de ésta clase, se utilizará para encapsular datos y funciones comunes a todas las clases derivadas, se empleará herencia y polimorfismo *(Véase 1.11 y 1.15 respectivamente).*

La Fig. 1.5 contiene a todas las *"figuras"* que son derivadas de ésta clase base, es decir, las clases derivadas. En la **Fig. 1.6**, se representa en notación icónica la clase "Figuras Geométricas".

Una clase se encuentra compuesta de objetos, un objeto realmente demostrativo es : una figura geométrica. Para comenzar crearemos un círculo (se apreciará solamente su perímetro), pero para hacerlo aún más ameno, también se desea poder manipular al objeto círculo por medio

de un dispositivo periférico, el ratón. Se procederá entonces con la codificación de la clase figuras en el lenguaje C++, obsérvense los nombres de archivo empleados.



Entonces es recomendable no profundizar en éste momento en nuestro objeto, sino más bien en el dispositivo que servirá como auxiliar en su manipulación : *el ratón*. El diseño de una buena clase, radica en el diseño de los objetos que poseerá. Particularmente, la clase ratón ha sido creada modularmente, los archivos "Raton.h" y "Raton.cpp" son incluidos en el programa principal "Prinrat.cpp"; el primero, ***Raton.h***, contempla la definición formal de la clase "raton" y de las bibliotecas necesarias para su correcto funcionamiento (notable es la biblioteca "graphics.h" para el manejo de gráficos en la computadora), así como de las macros empleados a lo largo de la clase. El segundo archivo, ***Raton.cpp***, contempla los constructores definidos en la clase, propiamente, los métodos que serán accesados en el archivo. El tercer archivo, ***Prinrat.cpp***, éste es el programa principal, en él se pueden apreciar mensajes enviados al usuario vía monitor, que facilitan el entendimiento de la aplicación, en la corrida del programa. Será posible navegar con el ratón a lo largo y ancho de nuestra pantalla, indicándose siempre el punto

exacto de ubicación en la misma (pixel) y, también, si es el caso, indicará qué botón fue digitado (por supuesto, el dispositivo considerado tiene solamente dos botones : derecho e izquierdo).

```
// Archivo : PRINRAT.CPP
// Programa principal de la Clase Ratón

#include "raton.h"
#include "raton.cpp"
#include <conio.h>

void main(void)
{
      raton miguelito;
      int boton=0;
      int x,y;

      clrscr();
      miguelito.aparece();
      while(BO_AMBOS!= boton)
      {
       boton=miguelito.boton();
       miguelito.posicion(x,y);
       gotoxy(5,5); printf("Coordenadas:%4i,%4i",x,y);
       gotoxy(5,6);printf("Boton: ");
       switch(boton)
       {
       case BO_NINGUNO :
              printf("ninguno     ");
                    break;
       case  BO_DERECHO :
                    printf("derecho     ");
                    break;
              case BO_IZQUIERDO :
                    printf("izquierdo   ");
                    break;
       }
      }
      miguelito.esconde_raton();
}
```

Listado I.4 Programa principal de la Clase Ratón

```
// Archivo : RATON.H
// Interfaz de la clase del raton
#ifndef RATON_H
      // Definición de Macros
      #define RATON_H 1
      #define RATON 0x33                    // Número de la interrupción
      #define BO_NINGUNO    0               // Indica que ningún botón ha  sido oprimido
      #define BO_DERECHO    1               // Indica que el botón derecho  fue oprimido
      #define BO_IZQUIERDO  2               // Indica que el botón izquierdo fue oprimido
      #define BO_AMBOS      3               // Indica que ambos botones fueron oprimidos

      #include <iostream.h>
      #include <conio.h>
      #include <dos.h>
      #include <graphics.h>
      class raton
      {
      public:
        raton()
        {
              inicia_raton();
        };
        void inicia_raton(void);
        void apaga_raton(void);
        void coloca_raton(int x, int y);
        void posicion(int & x, int & y);
        void esconde_raton(void);
        void aparece(void);
        int boton(void);
      };
      #include "raton.cpp"
#endif
```

Listado I.5 Interfaz de la clase del raton

```
// Archivo : RATON.CPP
// Clase raton
// Utiliza la interrupcion 33 del microprocesador
// Supone la existencia de un manejador para Ratón Microsoft
#ifndef RATON_CPP 1

      #define RATON_CPP

      // Activa el raton
      void raton::inicia_raton( void )
      {
```

```
  union REGS regs;

  regs.x.ax = 1;          // Activa el raton
  regs.x.cx = 100;
  regs.x.dx = 100;
  regs.x.bx = BO_NINGUNO;
  int86(RATON, &regs, &regs);
}

// Coloca el raton en una posición determinada
void raton::coloca_raton( int x, int y)
{
  union REGS regs;

  regs.x.ax = 4;      //Establece el raton
  regs.x.cx = y;
  regs.x.dx = x;
  regs.x.bx = BO_NINGUNO;
  int86(RATON, &regs, &regs);
}

// Desactiva el raton
void raton::apaga_raton( void )
{
  union REGS regs;

  regs.x.ax = 0; /* Desactiva el raton  */
  int86(RATON, &regs, &regs);
}

//Obtiene la posición del raton
void raton::posicion( int & x, int & y )
{
  union REGS regs;

  regs.x.ax = 3;          /*  Modo de pregunta  */
  regs.x.bx = BO_NINGUNO;
  int86(RATON, &regs, &regs);
  x = regs.x.dx;          /* posición del ratón: renglon */
  y = regs.x.cx;          /* posición del ratón: columna */
}

// Aparece el raton
void raton::aparece( void )
{
  union REGS regs;
```

```
    regs.x.ax = 3;          /*  Modo de pregunta  */
    regs.x.bx = BO_NINGUNO;
    int86(RATON, &regs, &regs);
  }

  // Obtiene el boton que ha sido oprimido
  int raton::boton( void )
  {
    union REGS regs;

    regs.x.ax = 3;          /*  Modo de pregunta  */
    regs.x.bx = BO_NINGUNO;
    int86(RATON, &regs, &regs);
    return( (int) regs.x.bx);
  }

  // Desaparece el indicador grafico de la pantalla, sin desactivar el raton
  void raton::esconde_raton( void )
  {
    union REGS regs;

    regs.x.ax = 2;  /* Esconde el raton */
    regs.x.bx = BO_NINGUNO;
    int86(RATON, &regs, &regs);
  }
#endif
```

Listado I.6 Definición de la clase raton

Una vez definida la interfaz (clase raton) par. nuestro objeto *"circulo"*, ya es posible codificarlo, recuérdese que el objeto va a ser manipulado por medio del ratón, por ello se decidió definir su clase en primera instancia. El Listado I.7 muestra el programa principal **"princirc.cpp"**, el cual incluye los **Listados I.5 "raton.h" y I.8 "circulo.h"**,

```
// Archivo : PRINCIRC.CPP
#include <graphics.h>
#include <stdlib.h>
#include "raton.h"
#include "circulo.h"

int graficos_ok(void);
void main(void)
{
```

```
int ManejaGraf,ModoGraf;
int ErrorCod;
raton Jerry;                        // Creación de la instancia "Jerry" de la clase ratón
circulo esferita(100,100,50);       // Creación de la instancia "esferita" de la clase circulo

ManejaGraf = DETECT;
initgraph( &ManejaGraf, &ModoGraf, "" );
ErrorCod = graphresult();
if( ErrorCod != grOk )
{
     // En caso de error, emite mensaje
     printf(" Error en el Sistema Gráfico: %s\n", grapherrormsg( ErrorCod ) );
     exit( 1 );
}
cout<< "Pulse un boton para continuar";
Jerry.inicia_raton();               // Inicializa la instancia Jerry
Jerry.aparece();                    // Dibuja en pantalla a Jerry
esferita.dibuja();                  // Dibuja la instancia esferita
while(Jerry.boton() ==BO_NINGUNO);  // Espera mientras ningún botón sea oprimido
circulo bolita(Jerry);              // Creación de bolita ( circulo ) con el ratón
while(Jerry.boton() ==BO_NINGUNO);
bolita.dibuja();                    // Se dibuja la bolita
while(Jerry.boton() ==BO_NINGUNO);
esferita.borra();                   // Se elimina la esferita
while(Jerry.boton() ==BO_NINGUNO);
closegraph();                       // Termina uso de gráficos
}

// Manipulación de gráficos
int graficos_ok(void)
{
     int errorcode = graphresult();

     if (errorcode != grOk)
       return 0;
     else
       return 1;
}
```

Listado I.7 Programa principal para la manipulación del objeto Jerry (ratón).

En el **Listado I.8**, se define la clase circulo, así como los métodos (operaciones) que se efectuarán sobre las instancias (elementos de la clase) que sean creadas.

```
// Archivo : CIRCULO.H
#ifndef CIRCULO_H
      #define CIRCULO_H 1
      #include "raton.h"

      class circulo
      { // Definición de la clase circulo
      public:
        circulo(raton &);
        circulo(int,int,int);

        double area(void) ;
        void dibuja(void) ;
        void datos(void)  ;
        void borra(void)  ;

      protected:
        int xc,yc; // Coordenadas del centro
        int radio;
      };
      #include "circulo.cpp"

#endif
```

Listado I.8 Definición de la Clase circulo

A continuación, en el **Listado I.9**, se definen las operaciones relativas a la clase circulo.

```
// Archivo : CIRCULO.CPP
#include <math.h>          // para la definicion de la variable M_PI
#include <conio.h>
#include <iostream.h>
#include <graphics.h>

#define PREVIO LIGHTBLUE

int graficos_ok(void);
// Metodos para la clase circulo

// Constructor
circulo::circulo(raton &mimi)
{
      int color;
      int radiox, radioy;
      // Centro del circulo
```

```
        // Hasta que se oprima el boton derecho
        while (mimi.boton() != BO_DERECHO);
        // Obtiene posicion del cursor del raton
        mimi.posicion (xc,yc);

        color=getcolor();
        setcolor(PREVIO);

        // Radio
        while (mimi.boton() !=BO_DERECHO);
        while (mimi.boton() == BO_DERECHO)
        {
          mimi.posicion (radiox,radioy);
          radio = sqrt( pow((radiox-xc),2)+pow((radioy-yc),2) );
          mimi.esconde_raton();
          dibuja();
          borra();
          mimi.inicia_raton();
        }
        mimi.posicion (radiox,radioy);
        radio = sqrt( pow((radiox-xc),2)+pow((radioy-yc),2) );
        setcolor(color);
        mimi.esconde_raton();
        dibuja();
        mimi.inicia_raton();
}

// Parámetros requeridos para la construcción del circulo
circulo::circulo(int iCentrox, int iCentroy, int iRadius)
{       xc=iCentrox;
        yc=iCentroy;
        radio=iRadius;
}

// Cálculo del área del circulo
double circulo::area(void)
{
        return( M_PI * radio* radio);
}

// Captura de parámetros
void circulo::datos(void)
{
        cout << endl << "Circulo: \n\tCentro: ("<< xc <<","<<yc<<")";
        cout << "\n\tRadio:"<<radio;
}
```

```
// Pinta el circulo en la pantalla
void circulo::dibuja(void)
{
  /* dibuja un circulo */
  if (graficos_ok())
  circle(yc, xc, radio);
}

// Elimina el circulo de la pantalla
void circulo::borra(void)
{
      int color;
      if (graficos_ok())
      {
        color = getcolor();
        setcolor(getbkcolor());
        circle(yc,xc,radio);
        setcolor(color);
      }
}
```

Listado I.9 Definición de los métodos empleados en la clase circulo

Si deseamos incorporar las implantaciones de las funciones miembro de una clase con la interfaz misma, podemos hacerlo con aquellas interfaces conocidas como en línea *(inline)*, también es posible presentarlas por separado, como se haría con cualquier definición de una función en C, esto lo logramos anteponiendo el nombre de la clase al nombre de la función. Queda a juicio del programador, la libertad de agrupar declaraciones de clases y las implantaciones de funciones miembro en archivos separados o juntos.

I.5) Constructores y Destructores

Contamos con dos tipos de funciones miembro : **constructores y destructores**, dichas funciones nos son útiles para la creación de objetos de una clase, esto ocurre en el caso de las funciones constructoras, quienes comparten el nombre con la clase misma.

Ahora bien, la destrucción de los objetos en C++ es realizada por el sistema en forma implícita, de ésta manera se libera el espacio de un objeto cuando éste sale de su alcance.

También se puede encargar ésta tarea al programador para que implante una liberación más completa al tratarse de objetos de los cuales cuelgan otros objetos que para ése momento resulten inútiles. La función miembro destructora de objeto deberá tener el mismo nombre de la clase, precedido por un símbolo de "~".

I.5.1) Constructores

En su labor cotidiana, el programador emplea variables, al ser éstas declaradas, el compilador creará el espacio necesario para alojar el dato en cuestión. Cuando un objeto es creado, la acción realizada es semejante, es decir, la creación de un objeto origina la reservación de un espacio para el almacenamiento del (los) dato(s) al (los) que haga referencia.

" ***Un constructor es uno de los tipos de función miembro que podemos emplear en una clase, la función constructora es verdaderamente útil para la creación de un objeto de una clase dada, dicha creación puede surgir en dos modalidades: estática o dinámica. Hablamos de una creación estática al momento de declarar una variable, el operador "new" nos permite hacerlo*** "

Es importante resaltar la posibilidad de que existan varias funciones constructoras para los objetos de la misma clase, la utilidad que podemos encontrar en ello es : la inicialización de estados diversos para los objetos creados.

Comúnmente, un constructor es definido con el mismo nombre de su clase. **Téngase en cuenta que un constructor no devuelve ningún tipo.** De hecho, un constructor puede ser definido para distintos tipos de argumentos. El argumento correcto será establecido en función del tipo de argumentos con los cuales sea llamado, es posible definir tantos constructores como sean necesarios, la única condición es que todas las listas de argumentos sean efectivamente distintas.

Para ejemplificar, recordemos el objeto "MiCrono" de la clase CRONOMETRO estudiado en el Apartado I.1, el nombre de la clase (CRONOMETRO), representa un nuevo tipo y puede ser usado en cualquier contexto de declaración donde se use un tipo. Es decir :

CRONOMETRO Crono1, Crono2;

declara a las variables Crono1 y Crono2 de tipo CRONOMETRO. Con ésta declaración a partir de la clase se crean (instancian) dos objetos de la clase CRONOMETRO que serán los valores de las variables Crono1 y Crono2. En este ejemplo nos conviene que la componente *andando* tenga valor inicial 0 para indicar que el cronómetro está parado y que las componentes de pri_tm fuesen todas 0 para que Muestra escriba la hora 0:0:0:0.

En una declaración adicional de C, habría que ***inicializar*** estas componentes. Que esta ***inicialización*** sea responsabilidad del usuario (cliente) de la definición de CRONOMETRO ha sido tradicionalmente gran fuente de errores. Por otra parte, para que el cliente pueda hacer esta inicialización tendría que tener acceso a las componentes andando y pri_tm las cuales se han querido proteger poniéndolas en una sección *private*.

Para hacer esta inicialización, C++ introduce el concepto de ***constructor***. En C++ un constructor es una función que tiene el mismo nombre de la clase. Por ejemplo :

```
CRONOMETRO()
{
  andando=0;
  pri_tm.hour=0;
  pri_tm.min=0;
  pri_tm.sec=0;
  pri_tm.hund=0;
}
```

Listado I.10 Constructor de un Cronómetro

Esta función constructora será aplicada implícitamente por cada variable declarada del tipo de la clase, añadiendo con ello las acciones necesarias para la creación de un objeto de la clase. En caso de no existir esta función constructora, C++ sólo reservará memoria para las componentes del objeto (pero no les dará ningún valor inicial).

La definición final de la clase CRONOMETRO tendría la forma :

```
class CRONOMETRO
{
  private:
    //... Componentes andando, pri_tm y función dif_tm
  public:
       CRONOMETRO( ){ andando=0; pri_tm.hour=0; pri_tm.min=0; pri_tm.sec=0;
pri_tm.hund=0;}
    //... Funciones públicas Arranca, Muestra y Para
};
```

Listado I.11 Clase CRONOMETRO

En el ejemplo anterior, se empleó un recurso de clase para modelar un objeto del mundo real. El concepto de clase puede utilizarse también para describir objetos de software propiamente. Por ejemplo para clasificar las estructuras clásicas de datos conocidas por tipos de datos abstractos. De estas estructuras de datos tal vez la más conocida y, ampliamente utilizada como ejemplo en la literatura, es la pila (STACK). Supóngase que se desea definir una clase de objetos que representen pilas para guardar números enteros con el principio de funcionamiento de una pila : el último en empilar es el primero en desempilar (LIFO). Las operaciones de acceso serían : saber si una pila está vacía (Empty), saber cuál es el elemento que está en el tope de la pila (Top), saber cuántos elementos hay en la pila (Total) y saber si la pila está llena (Full). Las operaciones de transformación de la pila son : poner un elemento en tope de la pila (Push) y botar el elemento que esté en el tope de la Pila (Pop).

A continuación veremos una definición para expresar esta clase de Pila.

NOTA : Por uniformidad con la literatura existente el nombre de la clase y los nombres de las funciones se han dejado en inglés.

```
class STACK
{
 public:
   int *ptr_tope;
   int *ptr_fondo;
```

```
void Push(int x)
{
   if ((ptr_tope-ptr_fondo)<longitud)
   *ptr_tope++ = x;
}

void Pop(void)
{
  if (ptr_tope > ptr_fondo) ptr_tope--;
}

int Top(void)
{
   if (ptr_tope > ptr_fondo) return *(ptr_tope-1);
}

int Total(void)
{
  return (ptr_tope-ptr_fondo);
}

int Empty(void)
{
   return (ptr_fondo==ptr_tope);
}

int Full(void)
{
   return ((ptr_tope-ptr_fondo)==longitud);
}

};
```

Listado I.12 Definición de la Clase Pila (Stack)

A diferencia del uso de una pila al estilo tradicional de C, no es posible llamar directamente a ninguna de las funciones descrita en la clase. Es decir, no puede hacerse : ***Push(3); ni Pop();*** De acuerdo al modelo de objetos, estas funciones deben ser llamadas a través de un objeto de la clase, es decir a través de un objeto de tipo STACK. Si p es un objeto de la clase STACK entonces puede hacerse : ***p.Pus(3); y p.Pop();***

Protección de las partes privadas de un stack.- En la definición de la clase STACK, los punteros ptr_tope y ptr_fondo forman parte de la memoria (de los datos del objeto). Pus, Pop, Top, Total, Empty y Full son las operaciones (funciones miembro en C++) que dan la funcionalidad del objeto.

Si tenemos que p es un objeto de la clase STACK, entonces desde fuera de la clase puede hacerse : ***p.Pus(3); p.Top(); p.Empty(); p.Full(); p.Total(); p.Pop();*** para hacer referencia a la memoria del objeto. Sin embargo, a los efectos de la imagen exterior (interfaz) del objeto no es de interés conocer estos punteros ptr_tope y ptr_fondo. Conocerlos no sólo no aporta ninguna información de interés sino que puede facilitar una modificación de los mismos provocándose con ello un comportamiento erróneo de la pila.

La definición anterior debe queda ahora :

```
class STACK
{
 private:
   int *ptr_tope;
   int *ptr_fondo;

 public:

   void Push(int x)
   {
      if ( (ptr_tope-ptr_fondo)<longitud )
      *ptr_tope++ = x;
   }

   void Pop(void)
   {
      if (ptr_tope > ptr_fondo) ptr_tope--;
   }

   int Top(void)
   {
     if (ptr_tope > ptr_fondo) return *(ptr_tope-1);
   }

   int Total(void)
   {
      return (ptr_tope-ptr_fondo);
   }
```

```
  int Empty(void)
  {
    return (ptr_fondo==ptr_tope);
  }

  int Full(void)
  {
     return ((ptr_tope-ptr_fondo)==longitud);
  }
};
```

Listado I.13 Definición de la Clase Stack, incluyendo datos de tipo private.

El nombre de la clase (STACK en este caso), representa un nuevo tipo y puede ser usado en cualquier contexto de declaración donde se use un tipo. Así por ejemplo : ***STACK p, q;*** declara a las variables p y q de tipo STACK. Ellas denotarán a objetos STACK. La función constructora sería entonces :

```
STACK (void)
{
  ptr_tope=ptr_fondo = new int[100];
}
```

Listado I.14 Constructor de una pila

La creación de objetos puede ser estática o dinámica, la creación estática se logra al declarar una variable, la diferencia la podemos lograr con el operador **"new"**. Lo interesante es que puede haber varias funciones constructoras para los objetos de la misma clase, las cuales se distinguen por el conjunto de parámetros. Varias funciones constructoras sirven para poder crear objetos con estados iniciales diversos.

Al igual que C, C++ permite manipular los objetos por valor o por punteros. Esto queda a la decisión explícita del programador. Si tenemos por ejemplo la clase :

```
class A
{ //...
  public:
    ...A.../ Constructor de A
    ...F.../ F es un recurso público de A
}
```

podemos tener declaraciones de la forma : *"A x"*, que significa que la variable "x" podrá tener como valor un objeto de tipo A. El constructor de A es aplicado para construir estos objetos en el momento en que el programa en ejecución alcanza el contexto en que ocurre la declaración anterior. Siguiendo el mismo modelo de ejecución de C, la memoria para el objeto, de acuerdo a las componentes internas del mismo, es separada por C++ en la pila de ejecución (esto es independiente de si el constructor de la clase separa memoria a su vez, como es el caso de la clase STACK).

Extendiendo ortogonalmente el concepto de puntero de C, C++ permite definir punteros a objetos, es decir, podemos tener declaraciones de la forma : *"A *ptr"*, lo que significa que la variable ptr puede tener como valor un puntero a un objeto de tipo A. Un objeto podrá ser creado entonces dinámicamente con una instrucción de la forma : *"ptr=new A"*. En este caso la memoria para las componentes del objeto será asignada dinámicamente por C++ en ese momento y luego se aplicará el constructor de A (que podrá a su vez hacer nuevas asignaciones de memoria para asociar a alguna de las componentes del objeto). Por ejemplo, si se tiene : ***STACK *Stack_ptr=new STACK.***

Al hacer new STACK se asigna espacio a las componentes ptr_tope, ptr_fondo y longitud y se pone a Stack_ptr a apuntar a este espacio. Luego se aplica el constructor quien internamente hace new para separar espacio para el arreglo.

I.5.2) Destructores

Simétricamente al concepto de constructor, C++ introduce el concepto de destructor.

" Un destructor se encarga de la liberación de espacio de un objeto dado, al momento de salir de su alcance "

Para algunos lenguajes la destrucción de objetos se realiza en forma implícita, particularmente, en C++ no es indispensable declararla, sin embargo, el programador por razones de espacio puede desear destruirlos en un momento determinado; ***un destructor también recibe el nombre de la clase a la cual pertenece pero le antecede el símbolo (~). Es vital notar cuando una clase hereda de otra, debido al orden de llamada en el que los constructores y destructores sean solicitados.***

Una función destructora será una función con el mismo nombre de la clase y precedida del símbolo ~. En el ejemplo de STACK, será la función :

```
~ STACK (void)
{
  delete ptr_fondo
};
```

Listado I.15 Destructor de una pila

que libera el espacio separado para la pila.

Esta función destructora es invocada implícitamente para cada variable u objeto de la clase STACK cuando se abandona el bloque donde fue declarada. De esta manera se garantiza que el objeto haga una serie de acciones antes de que se "pierda" el acceso al mismo. Por ejemplo en el caso de STACK liberar la memoria reservada para cada pila.

El destructor también se aplica cuando se libera un objeto apuntado a través de un puntero. Por ejemplo si tenemos :

```
STACK *PtrS;
PtrS = new STACK(200);
```

y hacemos :

```
delete PtrS
```

La clase STACK completa quedaría ahora :

```
class STACK
{
 private:
   int *ptr_tope;
   int *ptr_fondo;
   int longitud;

 public:

   STACK(int talla=100)
   {
      ptr_tope = ptr_fondo = new int[ longitud = talla ];
   };

   int Max_Long(void)
   {
      return longitud;
   }

   void Push(int x)
    {
      if ((ptr_tope-ptr_fondo)<longitud)
       *ptr_tope++ = x;
    }

   void Pop(void)
    {
      if (ptr_tope > ptr_fondo) ptr_tope--;
    }

   int Top(void)
    {
      if (ptr_tope > ptr_fondo) return *(ptr_tope-1);
    }

   int Total(void)
    {
      return (ptr_tope-ptr_fondo);
     }

   int Empty(void)
    {
       return (ptr_fondo==ptr_tope);
     }
```

```
  int Full(void)
   {
     return ((ptr_tope-ptr_fondo)==longitud);
   }

  ~STACK(void)
   {
    delete ptr_fondo;
   }
};
```

Listado I.16 Listado completo de la clase Stack

Esta función destructora no puede ser invocada explícitamente por lo que no puede ser parametrizada. Con esto aparentemente se garantiza que no podamos destruir la integridad del objeto asociado a una variable si aún no hemos abandonado el bloque donde fue declarada dicha variable. Sin embargo, como C++ incluye, al igual que C, el concepto de variables globales, puede haberse hecho una asignación s=q donde s es global. Por lo que al destruir automáticamente el objeto asociado a q, al abandonar el bloque al cual q pertenece, aún pueden quedar variables como s que aparentemente están asociadas a un objeto que ha sido destruido.

El programa principal para la manipulación de la clase STACK podría ser :

```
int main(void)
{  STACK s(3);
   int i;

   s.Push(1);
   s.Push(2);
   s.Push(3);
   while (!s.Empty())
   {
    i = s.Top();
    s.Pop();
    printf("saque el elemento %d\n",i);
   }
   i = s.Total();
   printf("quedaron %d elementos\n",i);
   return 0;
}
```

Listado I.17 Programa principal de la clase STACK

Este tipo de problemas está asociado al hecho de poder usar explícitamente punteros. En C++ la mayoría de los casos en los que se usan destructores están asociados a la liberación de memoria. El uso de destructores no nos libera de provocar este tipo de errores. Otros lenguajes orientados a objetos más puros, como Eiffel y SMALLTALK, no manejan el concepto de punteros. Es el sistema el que maneja internamente la reservación y liberación de memoria. Esto implica que deben incluir subsistemas de recolección de basura *(garbage collector)*.

I.6) Mensajes

Los objetos del mundo real pueden verse afectados en diversas modalidades entre sí, por ejemplo, un objeto podría verse afectado con las acciones : crear, agregar, mover, enviar, doblar, etc. Esta infinita variedad genera un problema interesante: ¿ Cómo es posible representar todas estas interacciones en software ? Una solución elegante a este problema es el ***mensaje***. Los objetos se ven involucrados mutuamente mediante la comunicación que se origina en la *"transmisión-recepción"* de mensajes, en donde la ejecución de un método específico es solicitada.

" Un mensaje consiste del nombre del objeto a quien se envía seguido del nombre del método que el objeto receptor sabe cómo ejecutar. Si el método requiere información adicional, el mensaje incluye esa información como parámetros ".

El objeto que inicia el mensaje se conoce como "transmisor" y el que lo recibe como "receptor".

Un SOO *(Sistema Orientado a Objetos)* consiste de varios objetos interactuando unos con otros enviándose mensajes entre sí. Todo lo que un objeto puede hacer se expresa en sus métodos, este simple mecanismo soporta todas las posibles interacciones entre objetos.

Hasta éste momento, se ha hecho referencia al manejo de métodos, sin embargo, se considera oportuno definir ¿Qué es un "Método"? ***El término método (bajo el punto de vista orientado a objetos), se refiere a muchas cosas, se puede decir que constituye una filosofía más en el Desarrollo de Sistemas, cuyas etapas son : Especificación de Requerimientos o***

Análisis, Diseño, Codificación, Programación, Implantación, Mantenimiento,etc., es decir, es el procedimiento por medio del cual se definen las instrucciones que el objeto deberá realizar en cuanto éste sea llamado.

Concretamente, ***"un método es la función ejecutada en respuesta a un mensaje dentro del objeto, el nombre del método es el nombre del mensaje".***

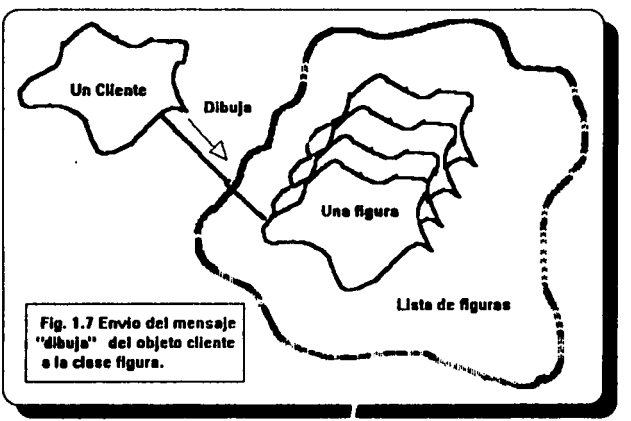


Fig. 1.7 Envío del mensaje "dibuja" del objeto cliente a la clase figura.

Un objeto se puede comunicar con otro mandando un mensaje. A dicha acción se le conoce como ***transmisión de un mensaje*** y, es la única manera en que un objeto se puede comunicar con otro. Recuérdese que, en el epígrafe I.1 se hizo referencia al objeto Cronómetro, perteneciente a la Clase CRONOMETRO, al cual podrían enviársele mensajes del tipo : ***MiCrono.Arranca();***, ésta instrucción indica la llamada de la operación Arranca sobre el objeto MiCrono (instancia de CRONOMETRO).

I.7) Subclase o Clase Base

Una subclase es una clase que hereda las características de otra llamada ***subclase*** o ***clase base.*** Si consideramos el ejemplo de Figuras Geométricas, ***la subclase es : figura,*** quien define

los métodos para su manipulación, ocultamiento, dibujo de la figura, movimiento, etc., (su comportamiento) y, creamos una figura, por ejemplo, un triángulo, también requerirá dibujarse, ocultarse, desplazarse, lo que variará aquí es la forma en la que será construido, pues la fórmula para calcular el área de un círculo, no es la misma para calcular un triángulo.

Las clases que no cuentan con instancias suelen denominarse ***"clases abstractas"***. A la clase más generalizada en la estructura de una clase se le llama ***"clase base"***.

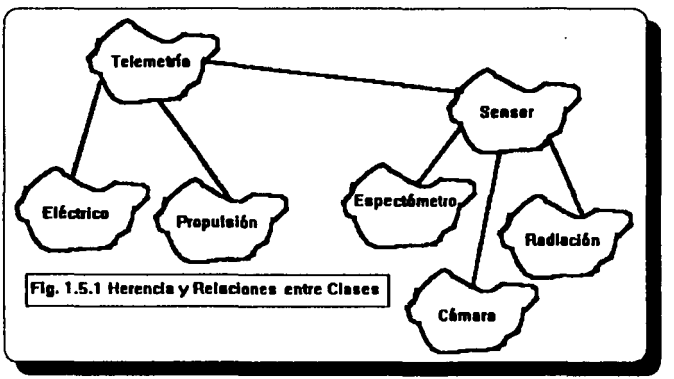


Fig. 1.5.1 Herencia y Relaciones entre Clases

Necesariamente de la **Fig. 1.5.1** se espera que algunas clases cuenten con instancias y otras de ellas no las tengan, a decir verdad, es de las clases especializadas de donde esperamos contar con instancias tales como: *"Eléctrico"* y *"Espectómetro"* y, no así en el caso de clases generalizadas, ejemplos de ellas en la misma figura lo son : *"Sensor"* y *"Telemetría"*.

I.8) Superclase

Una ***superclase*** es la clase de la cual se hereda su comportamiento. Una clase puede tener una o varias superclases, de ésta manera al poner limitaciones y hacer modificaciones en las

características heredadas se tiene una nueva clase con vida propia. Una subclase puede ser la superclase para otras subclases.

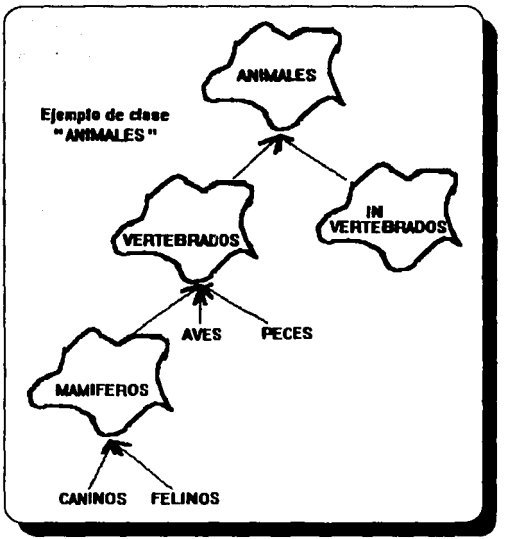
I.9) Clase Abstracta

Una ***clase abstracta*** es una clase que sirve para la creación de nuevas clases derivadas (mediante herencia) y no puede crear instancias (objetos). Es una clase que provee de varias funcionalidades en forma de prototipo, es decir, la subclase que la hereda deberá implantar algunos de los métodos para sus propósitos específicos, lo cual origina que el código sea reutilizable.

La clase abstracta capturará la *comunalidad* de varias clases derivadas que sí podrán crear objetos. Este recurso ayuda a expresar la abstracción que hacemos en la modelación de la realidad cuando agrupamos por clasificación objetos que tienen un comportamiento o características comunes. Este proceso de clasificación lo desarrollamos en una jerarquía de clases que van desde las clases más abstractas, que agrupan las características más generales (comunes a una mayor familia de objetos), a las clases más específicas, que expresan las características más particulares (propias de un conjunto más reducido de objetos).

Esta jerarquía de clasificación es característica de muchas disciplinas científicas como recurso para tener un mejor dominio y conocimiento del universo que queremos modelar. Tal es el caso por ejemplo de la Biología, cuando clasificamos los animales en vertebrados e invertebrados, los vertebrados a su vez en mamíferos, peces, aves, reptiles, etc. y así sucesivamente con cada una de las nuevas clases, hasta llegar a las clases que expresan las especies más particulares.

En C++ una clase abstracta es una clase que tiene al menos alguna función virtual pura. Una función virtual pura es una función de la cual sólo se especifica su encabezamiento pero que no se le define su cuerpo (esto se especifica en la sintaxis de C++ con la notación = 0).



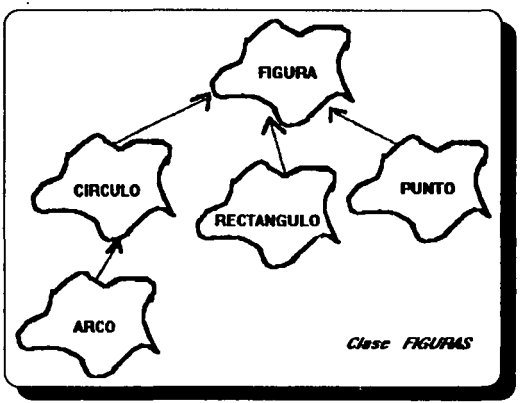
I.10) Clase Concreta

Una clase abstracta puede, sin embargo, tener componentes y funciones con una instrumentación concreta que expresen lo común, ya conocido, a todas las clases que la sucederán. En este sentido el papel de la clase abstracta es el de servir de patrón o modelo para la creación de nuevas clases aportando en estas clases derivadas la ***funcionalidad concreta*** que quedó pendiente en las funciones virtuales puras de la clase base. Por lo que, ***las clases que permiten definir instancias de ellas se les conoce como clases concretas.***

Una clase derivada de una clase abstracta : o define las funciones virtuales puras no definidas en las clase base o vuelve a especificar como puras aquellas que no defina. De esta

manera se garantiza que el diseñador de la nueva clase : o da una instrumentación concreta o conscientemente vuelve a especificar que ésta también es abstracta.

A continuación desarrollaremos una jerarquía de clases para expresar figuras geométricas. La clase abstracta más general FIGURA expresará lo común a cualquier figura : un par de coordenadas X y Y que la ubicarán en la pantalla, información sobre si la figura está visible o no, dos funciones Muestra y Oculta que mostrarán y ocultarán las figuras y, una función Traslada para trasladarla de coordenadas. Las funciones Muestra y Oculta serán virtuales puras porque no es posible definirlas hasta que no nos encontremos en la clase una figura en particular. Sin embargo, la función Traslada se puede definir utilizando las funciones Muestra y Oculta que están por definir.




```
class FIGURA
 {
   protected:
     int pri_x;
     int pri_y;
     int pri_visible;
   public:
     FIGURA(int InicX=0, int InicY=0) { pri_x=InicX; pri_y=InicY;pri_visible=0; }
     int X(void) const { return pri_x; }
```

```
    int Y(void) const { return pri_y; }
    int Visible(void) const { return pri_visible; }
    virtual void Muestra(void) = 0;
    virtual void Oculta(void) = 0;
    void Traslada( int NuevoX, int NuevoY );
  };
```

Listado I.3 Clase abstracta : FIGURA (figuras geométricas)

```
  // Desplaza la figura
  void FIGURA::Traslada(int NuevoX, int NuevoY)
  {
    if (pri_visible)
    {
      Oculta();   // hacerlo invisible
      pri_x = NuevoX;  // cambiar las coordenadas pri_x y para la nueva posición
      pri_y = NuevoY;
      Muestra();   // muestra el punto en las nuevas coordenadas
    }
    else
    {
      pri_x = NuevoX; pri_y = NuevoY;
    }
  };
```

Listado I.18 Constructor de la operación Traslada

Obsérvese que, ésta clase abstracta FIGURA tiene un constructor. Aún cuando la clase abstracta no permite la creación de objetos, tiene sentido que ésta pueda tener un constructor. Esto garantiza que cualquier clase derivada tenga que aplicar este constructor.

De ésta clase abstracta podemos ahora derivar tres clases concretas PUNTO, CIRCULO y CUADRADO, que sí definirán las funciones para Muestra y Oculta.

```
#include "graphics.h" // Funciones requeridas : circle, rectangle ...
#include "math.h"     // Funciones requeridas : sqrt

 class PUNTO : public FIGURA
 { public:
    PUNTO(int InicX, int InicY): FIGURA(InicX, InicY) {;}
    virtual void Muestra(void);
    virtual void Oculta(void);
 };
```

Listado I.19 Definición de la Clase concreta PUNTO

```
void PUNTO::Muestra(void)
{
    pri_visible=1;
    putpixel(X(), Y(), getcolor());
}

void PUNTO::Oculta(void)
{
  if (pri_visible)  // pintar en el color de fondo provoca el efecto de borrar
  {
    pri_visible=0;
    putpixel(X(), Y(), getbkcolor());
  };
}
```

Listado I.20 Instrumenta las funciones puras Muestra y Oculta para PUNTO

```
class CIRCULO: public FIGURA
{
  private:
    int pri_radio;                    // añade una nueva componente radio
  public:
    CIRCULO(int InicX, int InicY, int InicRadio): FIGURA(InicX,InicY)
    {
      pri_radio=InicRadio;
    }
    int Radio(void) const
    {
      return pri_radio;
    }
    virtual void Muestra(void);
    virtual void Oculta(void);
    void Inflar(int EnCuanto);
    void Desinflar(int EnCuanto);
};
```

Listado I.21 Definición de la Clase concreta CIRCULO

```
void CIRCULO::Muestra(void)
{
  pri_visible=1;
  circle(pri_x, pri_y, pri_radio);
}

void CIRCULO::Oculta(void)
{
    if (pri_visible)
```

```
    {
     pri_visible=0;
     unsigned int ColorTemp;
     ColorTemp = getcolor();
     setcolor(getbkcolor());
     CIRCULO(X0,Y0, pri_radio);
     setcolor(ColorTemp);
    };
};
```

Listado I.22 Instrumenta las funciones puras Muestra y Oculta para CIRCULO

```
void CIRCULO::Inflar(int EnCuanto)
{
    if (pri_visible)
    {
     Oculta();
     pri_radio+=EnCuanto;
     if (pri_radio<0) pri_radio=0; // Radio negativo es tomado como 0
        Muestra();
    }
    else
    {
     pri_radio+=EnCuanto;
     if (pri_radio<0) pri_radio=0; // Radio negativo es tomado como 0
    };
}

void CIRCULO::Desinflar(int EnCuanto)
{
    Inflar(-EnCuanto);
}
```

Listado I.23 Añade dos nuevas funciones Inflar y Desinflar para CIRCULO

```
class RECTANGULO: public FIGURA
{
    private:
      int pri_ancho, pri_alto;
      // Las coordenadas pri_x y pri_y son las del vertice superior izquierdo
   public:
      RECTANGULO(int InicX, int InicY, int InicAncho, int InicAlto): FIGURA(InicX,
InicY)
       {
         pri_ancho=InicAncho; pri_alto=InicAlto;
       }
       int Ancho(void) const
```

```
    {
      return pri_ancho;
    }
    int Alto(void) const
    {
      return pri_ancho;
    }
    virtual void Muestra(void);
    virtual void Oculta(void);
    double Diagonal (void) const;
};
```

Listado I.24 Definición de la clase RECTANGULO

```
void RECTANGULO::Muestra(void)
{
  pri_visible=1;
  rectangle(pri_x, pri_y, pri_x+pri_ancho, pri_y-pri_alto);
}

void RECTANGULO::Oculta(void)
{
  if (pri_visible)
  {
    pri_visible=0;
    unsigned int ColorTemp = getcolor();
    setcolor(getbkcolor());
    rectangle(pri_x, pri_y, pri_x+pri_ancho, pri_y-pri_alto);
    setcolor(ColorTemp);
  };
}

double RECTANGULO::Diagonal(void) const
{
   return sqrt(pri_ancho*pri_ancho + pri_alto*pri_alto);
}
```

Listado I.25 Instrumenta las funciones puras Muestra, Oculta y Diagonal para RECTANGULO

Para un tipo definido por una clase abstracta no pueden declararse variables, ni parámetros pasados por valor, ni pueden ser retomados por una función, pero sí pueden definirse variables apuntadores a la clase o pasarse parámetros por referencia. Esto permite aprovechar el polimorfismo cuando trabajamos con variables que pueden tener punteros o referencias a objetos

creados para alguna clase derivada que sí tenga una instrumentación concreta de las funciones puras.

De ésta manera la clase abstracta FIGURA aunque no permite directamente crear objetos (como corresponde a una clase abstracta), sirve para definir referencias o punteros que puedan recibir referencias o punteros a objetos de alguna de sus clases derivadas PUNTO, CIRCULO o RECTANGULO, aprovechando de ésta manera el polimorfismo. Se puede entonces, tener una función en biblioteca :

```
#include "FIGURA.h"
void FuncionAplicacion(FIGURA & S)
  {
   S.Muestra();
  }
```

Listado I.26 Biblioteca de la clase FIGURA

que ahora se puede usar en una aplicación en la forma :

```
#include <iostream.h>        // Función requerida : printf
#include <conio.h>   //                             getch
#include <stdlib.h>  //                             exit
int main(void)
{
    // Solicitud de auto detección
    int gdriver = DETECT, gmode, errorcode;

    // Inicializa modo gráfico
    initgraph(&gdriver, &gmode, "G:\\APPS\\BC\\BGI");

    // Lee el resultado de la inicialización
    errorcode = graphresult();

    if (errorcode != grOk)
    {
    printf("Error: %s\n", grapherrormsg(errorcode));
    printf("Oprima una tecla");
    getch();
    exit(1);
     }

    PUNTO APoint(100,100);
```

```
    CIRCULO ACircle(200,200,50);
    RECTANGULO ARectangle(250,300,150,100);
    //...
    FuncionAplicacion(APoint);
    FuncionAplicacion(ARectangle);
    FuncionAplicacion(ACircle);
    return 0;
}
```

Listado I.27 Programa principal de la clase FIGURA, instancias : PUNTO, CIRCULO y RECTANGULO.

Se ha procurado con éste ejemplo de jerarquía a partir de la clase FIGURA, presentar una clase no muy compleja, con el objetivo de ilustrar el concepto de clase abstracta y de función pura. Este recurso de clase abstracta puede ser utilizado como una herramienta importante de ingeniería de software ya que podemos, en una etapa inicial de desarrollo de un proyecto, definir una clase con aquellas funciones conocidas y de uso común para diferentes usuarios. Cada usuario podría, en una etapa posterior del desarrollo de trabajo, completar las definiciones inconclusas de aquellas funciones virtuales especificadas puras. Lamentablemente, C++ no incluye ningún recurso para especificar la semántica de la clase abstracta de modo que luego tenga que ser garantizado por la instrumentación concreta en una clase derivada.

En ocasiones no siempre es posible, en la fase inicial de desarrollo de un proyecto, concebir esta jerarquía partiendo de una clase abstracta. Ocurre entonces que, sólo luego de haber desarrollado varias clases por separado, nos damos cuenta de la comunalidad existente entre las mismas y de la conveniencia de haber concebido estas como derivadas de otras clases más generales. C++ no tiene ningún recurso para facilitar esta "reestructuración" de nuestras clases y a la vez tratar de provocar el mínimo de alteraciones en nuestras aplicaciones. Algunas herramientas extralingüísticas como los Sistemas de Bases de Datos Orientados a Objetos, que incluyen el concepto de versión de clases, han comenzado a tratar este problema.

I.11) Herencia

Las características de cada clase se transfieren a las clases inferiores a través de un proceso conocido como **"herencia"**.

" ***La herencia es una relación entre clases que permite declarar una clase (subclase) como extensión o especialización de otra clase (superclase)*** "

Los LOO's presentan dos variantes básicas: herencia sencilla *(una clase hereda de otra clase)* ***y, herencia múltiple*** *(una clase puede heredar de más de una clase).*

Al declarar una clase podemos indicar además, cuando existe herencia desde una clase hacia otra clase. Al declarar la clase, en la primer línea de la codificación se coloca el signo ":" (dos puntos) seguidos por una lista de las clases base desde las cuales ésta clase hereda. Por ejemplo, suponiendo que deseamos declarar la ***clase circulo***, emplearíamos el formato :

```
class circulo: public figura
{
    //Declaración de las variables miembro y de las funciones miembro
}
```

Listado I.28 Formato para la definición de Herencia con C++.

Aquí, la ***clase "figura"*** es la ***clase base*** y ***"circulo"*** es la ***clase derivada.***

La palabra **"public"** que precede a *"figura"* significa que algunas variables y funciones miembro son públicas de *"figura"*, la cual será accesible a la clase *"circulo"*.

A continuación se presenta el Listado I.8 que muestra la clase Circulo (en ella se aprecia la formación del perímetro de un círculo) que hereda al objeto Circulo Lleno (por medio de herencia, se forma el perímetro de un círculo dado y, al oprimir un botón del ratón, se aprecia la superficie del círculo). Obsérvense los listados empleados para representar herencia :

```
// Archivo: CIRLLENO.H
// Definición de la clase : "circulo lleno"
#ifndef CIRC_LLENO
    #define CIRC_LLENO
    #include "circulo.h"

    class circulo_lleno: public circulo
    {
      // Definición de la clase circulo_lleno
```

```
        public:

                circulo_lleno(int,int,int);
                circulo_lleno(raton &);
                void dibuja(void);
                void borra(void);
        };
        #include "cirlleno.cpp"
#endif
```

Listado I.29 Definición de la clase : "circulo lleno"

```
// Archivo : CIRLLENO.CPP
// Métodos del objeto "circulo lleno"
// Parámetros para el cálculo del circulo
void circulo_lleno::circulo_lleno(int iCentrox, int iCentroy, int iRadius):circulo(iCentrox,
iCentroy, IRadius)
{
      xc=iCentrox;
      yc=iCentroy;
  radio=iRadius;
}

// Dibuja el circulo en la pantalla
void circulo::dibuja(void)
{
  /* dibuja un circulo */
      if (graficos_ok())
      {
        setfillstyle(1,WHITE);
        fillellipse(yc,xc,radio,radio);
      }
}

// Elimina el circulo de la pantalla
void circulo::borra(void) {
      int color;
      if (graficos_ok())
      {
        color = getcolor();
        setfillstyle(1,getbkcolor());
        fillellipse(yc,xc,radio,radio);
        setcolor(color);
      }
}
```

Listado I.30 Métodos del objeto "circulo lleno"

```
// Archivo : HERCIRC.C
// Programa principal para Circulo Lleno
#include <graphics.h>
#include <stdlib.h>
#include "raton.h"
#include "circulo.h"

int graficos_ok(void);
void main.(void)
{

 int ManejaGraf,ModoGraf;                  // Definición de variables
 int ErrorCod;
 raton Jerry;                              // Definición de la instancia Jerry de la clase
ratón
 circulo esferita(100,100,50);             // Creación de la instancia esferita de la clase
circulo
 circulo_lleno llenito(200,200,50);        // Creación de la instancia llenito de circulo_lleno

 // Manipulación de gráficos
 ManejaGraf = DETECT;
 initgraph( &ManejaGraf, &ModoGraf, "" );
 ErrorCod = graphresult();
 if( ErrorCod != grOk )
 {
      printf(" Error en el Sistema Gráfico: %s\n", grapherrormsg( ErrorCod ) );
      exit( 1 );
 }
 cout<< "Pulse un boton para continuar";

 Jerry.inicia_raton();                     // Inicializa a Jerry
 Jerry.aparece();                          // Dibuja a Jerry

 llenito.dibuja();                         // Dibuja a llenito (area)
 esferita.dibuja();                        // Dibuja a esferita (perimetro)
 while(Jerry.boton() ==BO_NINGUNO);        // Espera mientras ningún botón se oprima
 circulo holita(Jerry);                    // Crea a holita con ayuda de Jerry
 while(Jerry.boton() ==BO_NINGUNO);        // Dibuja a holita
 holita.dibuja();
 while(Jerry.boton() ==BO_NINGUNO);
 esferita.borra();                         // Borra a esferita
 while(Jerry.boton() ==BO_NINGUNO);
 closegraph();                             // Fin de uso de gráficos
}
```

```
// Detecta gráficos en el equipo
int graficos_ok(void)
{
      int errorcode = graphresult();

      if (errorcode != grOk)
        return 0;
      else
       return 1;
}
```

Listado I.31 Programa principal para Circulo Lleno

```
// Archivo : HERCIRC.CPP
#include <graphics.h>
#include <stdlib.h>
#include "raton.h"
#include "circulo.h"
#include "cirlleno.h"

int graficos_ok(void);

void main(void)
{

 int ManejaGraf,ModoGraf;          // Definición de variables
 int ErrorCod;
 raton Jerry;                      // Definición de la instancia Jerry de la clase ratón
 circulo esferita(100,100,50);     // Creación de esferita, parámetros dados por
simplicidad
 circulo_lleno llenito(200,200,50); // Creación de llenito, parámetros dados por
simplicidad

 // Manipulación de gráficos
 ManejaGraf = DETECT;
 initgraph( &ManejaGraf, &ModoGraf, "" );
 ErrorCod = graphresult();
 if( ErrorCod != grOk )
 {
      printf(" Error en el Sistema Gráfico: %s\n", grapherrormsg( ErrorCod ) );
      exit( 1 );
 }
 cout<< "Pulse un boton para continuar";

 Jerry.inicia_raton();
 Jerry.aparece();
```

```
 llenito.dibuja();
 esferita.dibuja();
 while(Jerry.boton() ==BO_NINGUNO);
 circulo bolita(Jerry);
 while(Jerry.boton() !=BO_NINGUNO);
 while(Jerry.boton() ==BO_NINGUNO);
 circulo_lleno otro(Jerry);
 while(Jerry.boton() !=BO_NINGUNO);
 while(Jerry.boton() ==BO_NINGUNO);
 esferita.borra();
 llenito.borra();
 while(Jerry.boton() !=BO_NINGUNO);
 while(Jerry.boton() ==BO_NINGUNO);
 closegraph();
}

int graficos_ok(void)
{
      int errorcode = graphresult();

      if (errorcode != grOk)
        return 0;
      else
        return 1;
}
```

Listado I.32 Programa principal para Circulo Lleno, se aprecia herencia de las clases circulo y raton.

Como se aprecia, se pueden crear nuevas clases que complementen con nuevas funciones dicha superclase sin necesidad de reescribir todo nuevamente, únicamente bastará con escribir la nueva funcionalidad. Esta estrategia fue una primera forma de herencia y, hoy en día, es uno de los pilares de la POO.

Por lo que ***en la Programación Orientada a Objetos, la herencia asocia a cada clase derivada, las conductas y los datos asociados con la clase de la que hereda.*** Una subclase puede tener todas las características de una superclase heredadora. Cuando una clase hereda conductas de otra clase, el código asociado no tiene que reescribirse, lo cual ahorra a los programadores muchas horas en la codificación de sistemas que reutilizan código a través de la herencia, cuando contamos con clases que han heredado las conductas de otra, podemos apreciar claramente el concepto de **"reutilización"**, así como el de **"extensibilidad"**, ninguno de ellos

posee restricción alguna. Para poder explotar al máximo dichas ventajas se requiere que la documentación generada por el Análisis y el Diseño sean almacenadas en bibliotecas y, a su vez, éstas sean reutilizadas o extendidas una y otra vez. Si el proyecto a desarrollar es grande, es posible formar equipos de trabajo en función de las clases emparentadas por la "herencia".

La sintaxis utilizada para la clase derivada presenta el siguiente formato :

```
class nombre_clase_derivada : tipo_de_acceso nombre_clase_base
{
    //Cuerpo de la clase
}
```

Listado 1.33 Ejemplo de herencia y/o paso de mensajes.

donde :

nombre_clase_derivada	*Se refiere al nombre de la nueva clase.*
tipo_de_acceso	*Trata de los tipos : público o privado, declaradas con las palabras : public y private respectivamente.*
nombre_clase_base	*Esta es la clase que heredará sus caracteristicas.*

Una de las grandes ventajas de la herencia consiste en acelerar la rapidez con la cual se realizan los prototipos de las funciones y asegurar que las interfaces sean consistentes para todas las clases que la utilicen, como consecuencia de ello, se evitan problemas con la compatibilidad entre objetos similares. Si un error es encontrado en una clase base, éste se corrige en un sólo punto y se refleja en todas las clases herederas. De la misma manera, si en una clase base se reimplanta un método para que éste sea más eficiente, el cambio se verá reflejado en cada clase derivada.

Sin embargo, es necesario tomar en cuenta que la herencia es una herramienta muy valiosa cuando se trata de modelar problemas grandes, aún cuando adiciona una gran complejidad a los

programas que la utilizan. Quizá una consecuencia no muy recomendable de la herencia la representa *"el tiempo de ejecución"*, el cual puede llegar a incrementarse ligeramente, ya que los métodos a través de los cuales se implanta ocupan mayor tiempo de procesamiento que una simple llamada. Sin embargo, sería cuestión de evaluar entre tiempos de ejecución y rapidez en la producción de un sistema, así como la compatibilidad entre los objetos similares ahí empleados puesto que : son mayormente redituables.

I.12) Sobrecarga

> ***" Se denomina sobrecarga, a la posibilidad de definir diversas funciones con el mismo nombre en los Lenguajes Orientados a Objetos"***

En los lenguajes de programación tradicionales, también es posible tal definición, aunque de manera limitada, virtualmente todos los lenguajes emplean el mismo símbolo para operaciones del tipo aritmético, independientemente del tipo de dato.

Un ejemplo típico de sobrecarga podemos apreciarlo en los operadores aritméticos como en el caso del símbolo + que está definido para más de un tipo de datos :

> Si "a" y "b" son dos números enteros, entonces "a + b" ,
> simboliza la suma de dos números enteros.
>
> Si "a" y "b" son dos números reales, entonces "a + b" ,
> simboliza la suma de dos números reales, etc.

Sin embargo, no es del todo frecuente encontrar que el Diseñador de un Lenguaje presente la misma flexibilidad de su producto de la que él mismo se permitió en su definición. La sobrecarga de operadores es una característica que facilita la programación proporcionada por los *LOO's (Lenguajes Orientados a Objetos)*.

Si un Lenguaje como C++ incluye la notación y la instrumentación tradicional para denotar las operaciones con los tipos simples predefinidos entonces sería deseable que permita utilizar la misma notación, para los nuevos tipos que se definan. Para lograr esto, C++ permite redefinir los operadores de C. Con este recurso podemos utilizar los mismos símbolos y la misma sintaxis de los operadores predefinidos de C pero con la semántica conforme a las nuevas clases que se definan.

A continuación, se muestra una clase que define números racionales. La intención es definir para los números racionales operaciones similares a las que define C para los tipos aritméticos predefinidos como : **asignación (=), suma y asigna (+=), comparación (==) y suma (+)** (por simplificidad, de las operaciones aritméticas sobre racionales, sólo se ha incluido la suma).

```
#include <iostream.h>
class RACIONAL
{
    private:
      int num, den;
      void simplifica(void);
      int abs(int n)
      {
      if (n<0) return -n;
      else return n;
      };
    public:
     RACIONAL(int n=0, int d=1)                    // Constructora
     {
      num=n; den=d;
     }
     RACIONAL operator= (RACIONAL &r);    // Es preferible no copiar el objeto
     RACIONAL operator+=(RACIONAL &r);    // & Indica acceso a la parte privada de r
     RACIONAL operator+ (RACIONAL &r);
     int          operator==(RACIONAL &r);
};
```

Listado I.34 Clase Números Racionales

Esta clase va a permitir aplicar a operandos de tipo RACIONAL los operadores "=", "+=", "+" y "==" con la misma sintaxis que C aplica a los aritméticos. Si tenemos por ejemplo la declaración ***"RACIONAL p, q;"*** entonces podemos escribir :

p=q ***p+=q*** ***p+q*** ***p==q***

lo que será interpretado como enviar los mensajes :

p.operator=(q); ***p.operator+=(q);*** ***p.operator+(q);*** ***p.operator==(q);***

Obsérvese que la descripción anterior está utilizando dos funciones privadas para transformar al número racional y representarlo en forma simplificada cuando sea necesario. Su definición es :

```
void RACIONAL::simplifica(void)
{
  if (den<0)
  {
    num=-num; den=-den;
  };

 for (int i = abs(num)< den ? abs(num) : den; (i>=2) &&(den>1); i--)
     if ((num%i)==0 && (den%i)==0)
     {
       num/=i; den/=i;
     };
 };
```

Listado I.35 Constructor simplifica para la clase RACIONAL

Para mantener la misma semántica de la asignación con los tipos predefinidos, en una asignación **p=q**, debe retornarse el valor de p. Si en la notación de objetos esta asignación se lee como **p.operator=(q)** esto significa que la función **operator=** debe retornar el propio objeto actual después de haber cambiado los valores de sus componentes. Esto se hará utilizando el identificador **this**. Utilizado dentro de una función miembro de una clase este identificador representa a un puntero al objeto actual. La definición del **operador=** queda entonces en la forma :

```
RACIONAL RACIONAL::operator=(RACIONAL &r)
{
  num=r.num;
  den=r.den;
  return (*this);
};
```

Listado I.36 Uso del identificador this en el Constructor de Asignación (paréntesis opcionales).

La definición de las operaciones + y += hacen también uso de este recurso :

```
RACIONAL  RACIONAL::operator+(RACIONAL &r)
{
      return RACIONAL(num *r.den+r.num+den,den*r.den);
};

RACIONAL RACIONAL::operator+=(RACIONAL &r)
{
      num=num*r.den+r.num*den;
    den*=r.den;
      return(*this);
};
```

Sería deseable poder usar un valor entero (de tipo int) como un caso particular de racional. Esto es posible en este ejemplo, es decir, si k es una variable de tipo int se puede escribir :

```
p=2        p=k        p+=k        p=2        p==4
```

ya que son interpretadas como :

```
p.operator=  (RACIONAL(2));
p.operator=  (RACIONAL(k));
p.operator+= (RACIONAL(k));
p.operator+  (RACIONAL(k));
p.operator== (RACIONAL(4));
```

ya que el parámetro formal de operator= es de tipo RACIONAL y los parámetros k y 4 pueden ser convertidos a RACIONAL en virtud de la aplicación implícita del constructor RACIONAL con valor por omisión 1 para el parámetro d.

El lector preocupado por la eficiencia puede querer evitar estas operaciones de conversión porque hacen perder tiempo. Esto puede lograrse dando otras definiciones para las funciones operator=, operator+=, operator+ y operator==, pero ahora con un argumento de tipo int.

C++ permite que se use un mismo nombre de función en distintas clases. Esto no provoca ambigüedad porque las funciones sólo pueden llamarse a través de objetos de la clase. C++ también permite tener más de una definición para una función dentro de la misma clase. Las diferentes definiciones de una misma función tienen que diferenciase en la cantidad y/o en el tipo de los parámetros de manera que esto no genere ninguna posible situación ambigua.

La definición de la clase RACIONAL quedaría :

```
class RACIONAL
{
 private:
      int num, den;
      void simplifica(void);
      int abs(int n);
 public:
      RACIONAL(int n=0, int d=1)
      {
        num=n;
        den=d;
      }
      RACIONAL operator=(RACIONAL &r);
      RACIONAL operator=(int n);
      RACIONAL operator+=(RACIONAL &r);
      RACIONAL operator+=(int n);
      RACIONAL operator=+(int n);
      RACIONAL operator+(RACIONAL &r);
      RACIONAL operator=(int n);
      int operator==(RACIONAL &r);
      int operator==(int n);
};
```

Listado I.37 Clase RACIONAL, incluye parte privada.

En este caso para una construcción como : *"p=2"*, de las dos posibles interpretaciones :

p.operator=(RACIONAL(2)) *p. operator=(2)*

El lector preocupado por la eficiencia puede querer evitar estas operaciones de conversión porque hacen perder tiempo. Esto puede lograrse dando otras definiciones para las funciones operator=, operator+=, operator+ y operator==, pero ahora con un argumento de tipo int.

C++ permite que se use un mismo nombre de función en distintas clases. Esto no provoca ambigüedad porque las funciones sólo pueden llamarse a través de objetos de la clase. C++ también permite tener más de una definición para una función dentro de la misma clase. Las diferentes definiciones de una misma función tienen que diferenciase en la cantidad y/o en el tipo de los parámetros de manera que esto no genere ninguna posible situación ambigua.

La definición de la clase RACIONAL quedaría :

```
class RACIONAL
{
 private:
      int num, den;
      void simplifica(void);
      int abs(int n);
 public:
      RACIONAL(int n=0, int d=1)
      {
        num=n;
        den=d;
      }
      RACIONAL operator=(RACIONAL &r);
      RACIONAL operator=(int n);
      RACIONAL operator+=(RACIONAL &r);
      RACIONAL operator+=(int n);
      RACIONAL operator=+(int n);
      RACIONAL operator+(RACIONAL &r);
      RACIONAL operator=(int n);
      int operator==(RACIONAL &r);
      int operator==(int n);
};
```

Listado I.37 Clase RACIONAL, incluye parte privada.

En este caso para una construcción como : *"p=2"*, de las dos posibles interpretaciones :

p.operator=(RACIONAL(2)) *p. operator=(2)*

La implantación de las funciones quedaría de la forma :

```
RACIONAL RACIONAL :: operator=(int n)
{
  num=n;
  den=1;
  return(*this);
};

RACIONAL RACIONAL ::operator+=(int n)
{
      num+=n*den;
      return(*this);
};

RACIONAL RACIONAL :: operator+(int n)
{
  return RACIONAL(num+n*den, den);
}

int RACIONAL :: operator==(int n)
{
  simplifica(); // Llevar el objeto en curso a forma simplificada antes de comparar
  return(num==n && den==1);
};
```

Listado I.38 Constructores de las operaciones aritméticas definidas en la clase RACIONAL

Con la finalidad de completar nuestro listado, se introducirán en éste momento funciones amigas para el caso de los operadores : "+" e "==", una ***función friend (amiga)*** es una función asociada ("amiga") de una clase. Esta función puede ser miembro de otra clase o una función global (no parte de ninguna clase). Que la función haya sido definida como amiga le da derecho a usar las componentes privadas de un objeto de la clase de la cual es amiga. Finalmente, el programa a continuación ejemplifica el uso de la clase RACIONAL :

```
#include <iostream.h>
class RACIONAL
{
  private:
    int num, den;
    void simplifica(void);
    int abs(int n);
  public:
```

```
    RACIONAL(int n=0, int d=1) { num=n; den=d; }      //constructora
    RACIONAL operator= (const RACIONAL &r);           //es preferible no copiar el objeto
    RACIONAL operator= (int n);
    RACIONAL operator+=(const RACIONAL &r); // Con & se accesa a la parte privada de r
    RACIONAL operator+=(int n);
    RACIONAL operator+ (const RACIONAL &r);
    RACIONAL operator+ (int n);
    friend RACIONAL operator+ (int n, const RACIONAL &r);
    int     operator==(RACIONAL &r);
    int     operator==(int n);
    friend int     operator==(int n, RACIONAL &r);
    void    imprime(void);
};
```

Listado I.39 Definición de la Clase Racional

```
int RACIONAL::abs(int n)
{
      if (n<0) return -n; else return n;
};

void RACIONAL::simplifica(void)
{
    if (den<0)
      {
        num=-num; den=-den;
      };
  for (int i = abs(num)< den ? abs(num) : den; (i>=2) &&(den>1); i--)
     if ((num%i)==0 && (den%i)==0) { num/=i; den/=i;};
};

RACIONAL RACIONAL::operator=(const RACIONAL &r)
{
      num=r.num; den=r.den; return (*this);
};

RACIONAL  RACIONAL::operator=(int n)
{
      num=n;  den=1; return (*this);
};

RACIONAL RACIONAL::operator+=(const RACIONAL &r)
{
    num=num*r.den+r.num*den;
    den*=r.den; return(*this);
};
```

```
RACIONAL  RACIONAL::operator+=(int n)
{
      num+=n*den; return (*this);
};

RACIONAL  RACIONAL::operator+(const RACIONAL &r)
{
      return RACIONAL(num*r.den+r.num*den,den*r.den);
};

RACIONAL  RACIONAL::operator+(int n)
{
      return RACIONAL(num+n*den,den);
};

RACIONAL operator+(int n, const RACIONAL &r)    //Esta función amiga no ha sido
{
      return RACIONAL(n*r.den+r.num, r.den);
};//Calificada con el nombre de la clase inclusive si regresa un resultado del tipo RACIONAL

int RACIONAL::operator==(RACIONAL &r)
{
  simplifica();  r.simplifica();
  return (num==r.num && den==r.den);
};

int RACIONAL::operator==(int n)
{
      simplifica(); // Llevar el objeto en curso a forma simplificada antes de comparar
    return (num==n && den==1);
};

int operator==(int n, RACIONAL &r)  //Esta función amiga tampoco es calificada
{
    r.simplifica();                       //Da con el nombre de la clase; e incluso
    return (n==r.num && r.den==1);        //ni su resultado es del tipo de la clase
};                                        //RACIONAL, pero accede a su parte privada

void RACIONAL::imprime(void)
{
      simplifica();
    if (den==1) printf("%d  ",num);
    else printf("%d/%d  ",num,den);
};
```

Listado I.40 Constructores de la clase RACIONAL

```
void main()
{
     RACIONAL p(-1,2), q(12,-15), s(-21,-7);

   printf("\nRESULTADOS DEL PROGRAMA DE NUMEROS RACIONALES\n");
   p.imprime();
   q.imprime();
   s.imprime();
   (p+=RACIONAL(3,4)).imprime();
   RACIONAL(2,7).imprime();
   (p=q+1).imprime();
   (1+p).imprime();
   int k=1; p=RACIONAL(-1,12);
   (k+p).imprime();
   RACIONAL(1).imprime();
   if (RACIONAL(6,7)==RACIONAL(-2,-5))
       printf("\ncierto");
       else
       printf("\nfalso");

       if (1==k+p+RACIONAL(2,24))
       printf("\ncierto");
       else
       printf("\nfalso");
}
```

Listado I.41 Programa principal de la clase RACIONAL

La corrida debe dar :

"-1/2", "-4/5", "1/4", "2/7", "1/5", "6/5", "11/12", "1", "falso", "verdadero".

Muchos programadores, pensando al estilo tradicional C, abusan del uso de las funciones amigas, utilizando con el recurso de friend lo que debían haber definido como una función miembro. Situaciones como las anteriores, para redefinir operadores predefinidos, son de los pocos casos que justificar el uso de las mismas.

" Si se combinan la sobrecarga y la herencia, se puede crear un mecanismo para la simplificación de la programación potencialmente superior ".

I.13) Métodos Estáticos y Virtuales : Enlaces Estáticos y Enlaces Dinámicos

" El enlace dinámico es de vital importancia para poder implantar el polimorfismo ".

Para ejemplificar, considérese el Control Administrativo del Personal Escolar de la Facultad de Ingeniería, podemos imaginar el comportamiento del mensaje que deseamos transmitir en función de una ecuación, del tipo :

E <— dar_de_baja

donde :

E* = *Empleado*,** éste es su ***tipo estático

esto significa que la variable puede asumir exclusivamente valores de ése tipo, incluyendo el valor de sus subtipos, lo que a fin de cuentas redunda en que el valor de "E" puede asumir cualquiera de los subtipos, siempre y cuando tengamos en cuenta que un subtipo de **empleado** será también un **empleado**.

Por otro lado, el ***enlace dinámico*** se refiere al objetivo que tendrá el lenguaje al determinar éste ***tipo dinámico*** y, basado en él, invocará el método indicado.

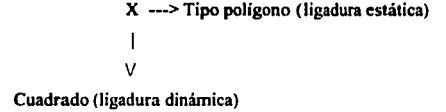
Analizando lo anterior, podemos observar que en realidad por medio de éstos dos tipos de comportamiento que puede presentar una variable (dinámico y estático), se está haciendo referencia al ya citado *"polimorfismo"* : la rutina en cuestión es capaz de adoptar formas distintas, tomando en cuenta que puede llevar a cabo acciones variadas en función de los tipos de argumentos que reciba.

I.13.1) Enlace Dinámico

Si bién la función definida en una clase concede un grado de flexibilidad al permitir que sea utilizada en sus subclases, en ocasiones se desea especializar la funcionalidad de una

operación. Consideremos, por ejemplo, la función *"área"* definida en la clase *"polígono"*, ésta función puede ser redefinida a una versión más eficiente dentro de la subclase *"cuadrado"*. Sería deseable que, cada vez que la función área sea llamada -sobre una variable que dinámicamente apunte a un objeto de tipo cuadrado-, la misma función sea ejecutada, independientemente del contexto de activación, de ésta forma se logra la activación de la versión especializada para cuadrados.

La diferencia esencial entre la ligadura dinámica y la ligadura estática reside en que, ***la ligadura estática se genera al momento de la compilación,*** en contraposición, ***la ligadura dinámica permite especializar tipos, al asociar el significado de una función al momento de su activación.*** Por ejemplo, si la variable "X" fuera definida de tipo polígono con liga estática, independientemente de que dinámicamente "X" referencia a un cuadrado, al llamar la función área de "X" se activaría siempre a la función área de polígono.



Función área (X), es lo mismo que decir que se pretende obtener la función área de un Polígono, a raíz de la ligadura estática que presenta la variable "X".

I.14) Funciones virtuales (Funciones diferidas)

Las ***funciones virtuales*** se especifican en una clase y su implantación se pospone para ser provista por los descendientes de la clase. Este tipo de funciones sólo tiene sentido si se trabajan en conjunto con el enlace dinámico. Cuando una clase tiene una o más funciones virtuales se le denomina ***"abstracta"***. Recordando el epígrafe 1.9 (clases abstractas), una clase abstracta no puede ser instanciada, es decir, no se pueden generar objetos de esa clase.

Las funciones virtuales sirven para especificar servicios que deben proporcionar todos los descendientes de una clase. Son una herramienta valiosa para modelar conceptos abstractos que son implantados de diversas maneras.

Una función virtual es una función precedida en su definición por la especificación ***virtual***, si tuviésemos una clase CIRCULO definida de la forma :

```
class CIRCULO
{
  private:
       int pri_x, pri_y;
       int pri_visible;
  protected:
       int pri_radio;
  public:
       CIRCULO(int InicX, int InicY, int InicRadio)
       {
         pri_x=InicX;
         pri_y=InicY;
         pri_radio=InicRacio;
     }
     int X(void) const
       {
        return pri_x;
       }
       int Y(void) const
       {
                return pri_y;
       }
       int Radio(void) const
       {
        return pri_radio;
       }
       int Visible const
       {
        return pri_visible;
       }
       virtual void Muestra(void);
       void Oculta(void);
       void Traslada(int NuevoX, int NuevoY);
};
```

Listado I.42 Clase CIRCULO

Ahora creemos una clase heredera llamada CIRCULO_COLOREADO, la cual heredará las características propias de un circulo definidas en el anterior listado, pero agregará color al circulo, obteniéndose un código del tipo :

```
class CIRCULO_COLOREADO : public CIRCULO
{
  private:
       unsigned int Color;
  public:
       CIRCULO_COLOREADO(int InicX, int InicY, int InicRadio, unsigned int InitColor) :
CIRCULO
       (InicX, InicY, InicRadio) { Color=InitColor}
       virtual void Muestra(void);
       void CambiaColor(unsigned int NuevoColor);
       unsigned int MiColor(void) const
       {
       return Color;
       }
};
```

Listado I.43 Clase CIRCULO_COLOREADO, uso de función virtual.

Se logra un enlace con el cual la función Muestra es invocada y, se realiza dinámicamente, por enlace dinámico, es decir, en tiempo de ejecución. este es el objetivo de la especificación virtual.

C++ exige que una vez que en cada clase se especifica una función como virtual entonces la especificación virtual se le debe poner a la función en todas clases herederas o derivadas en donde la función se redefina con los mismos parámetros.

I.15) Polimorfismo

En los Lenguajes de Programación, ***el polimorfismo se genera al admitir que una expresión o valor tome más de un tipo,*** es decir, cuando existe manipulación de datos de múltiples *(poli)* formas *(morfos)*. La misma rutina adopta formas distintas en el sentido de que puede llevar a cabo diferentes actividades dependiendo de los tipos de argumentos que reciba. Esto permite definir funciones que son aplicables a objetos de diferentes tipos, es decir, en más de un caso los objetos pueden responder de muy diferente manera a un mismo mensaje.

" Polimorfismo es la capacidad que tiene una clase de responder de manera diferente a un mismo mensaje "

Retrocediendo un poco, recordemos el concepto ***"tipificación"***, notamos que se encuentra estrechamente relacionado con el de Polimorfismo : En la tipificación se citó que, al tipificar estamos procurando identificar un objeto en función del uso y comportamiento que se le asignen y, el Polimorfismo, se genera a raíz de que ese objeto (expresión o valor) tome más de un tipo, ambos trabajan en función del tipo según sea el caso.

El polimorfismo nos permite implantar clases que respondan a un sólo mensaje. Las características y los métodos con los cuales se implante el mensaje es algo que en general, no le interesa al objeto que realiza la llamada. En los LOO's, el polimorfismo está muy relacionado con las jerarquías de herencia. Una operación definida sobre objetos de una clase puede ser aplicada sobre objetos de sus subclases.

Para asentar un poco más el concepto de polimorfismo, podemos imaginarnos una clase de ***"ANIMALES"***, en cuyo caso existirán subclases del tipo : ***"Animales Domésticos"***, ***"Animales Salvajes"***, así como métodos relacionados que indiquen : "Mi_Clase es", "Como_Soy" y "Quien_Soy".

```
class ANIMAL
{
  //...
};
```

y sus descendientes :

```
class ANIMAL_DOMESTICO : public ANIMAL
{ //...
  public :
  //...
  void ComeDeMiMano(...);
};
```

Se considera en éste caso, como Animal Doméstico aquél que es alimentado por el hombre.

```
class ANIMAL_SALVAJE : public ANIMAL
{ //...
  public :
  //...
  void ComeAnimalVivo(...);
}
```

El Animal Salvaje, por el contrario, es considerado como aquél (carnívoro) que se alimenta de otros animales.

Dado que los mensajes *"Mi_Clase", "Como_Soy", "Quien_Soy"* dentro de la clase ANIMAL tienen distintas formas (interpretaciones), se observa que se está empleando polimorfismo. Esta interpretación dinámica del llamado a una función virtual es aplicada cuando la función es llamada a través de una referencia o puntero al objeto.

El listado completo que ilustra el uso de polimorfismo en la Clase citada (ANIMAL), se puede apreciar a continuación :

```
int main(void)
{
   ANIMAL a;
   a.Quien_Soy();
   ANIMAL_DOMESTICO UnPerro("Pluto"), UnGato("Tom");
   ANIMAL_DOMESTICO *jaula;
   jaula=&UnGato;
   jaula->Quien_Soy();
   ANIMAL_SALVAJE UnMono("el plátano");
   UnMono.Quien_Soy();
   ANIMAL *jaula_cerrada;
   jaula_cerrada = &UnPerro;
   jaula_cerrada->Quien_Soy();
   jaula_cerrada=new ANIMAL_SALVAJE("la carne");
   jaula_cerrada->Quien_Soy();

   return 0;
}
```

Listado I.44 Programa principal que emplea la clase ANIMAL

La corrida esperada del programa anterior emite los mensajes :

```
soy animal desconocido
no se quien soy
soy animal doméstico
mi nombre es Tom
soy animal salvaje
me gusta el plátano
soy animal doméstico
mi nombre es Pluto
soy animal salvaje
me gusta la carne
```

La implantación de la clase se observa en el listado I.45 :

```
#include <iostream.h>
#include <string.h>

class ANIMAL
{
  public:
    virtual void Mi_Clase(void) const;
    virtual void Como_Soy(void) const;
    void Quien_Soy(void) const;
};
```

Listado I.45 Definición de la clase ANIMAL

```
void ANIMAL::Mi_Clase(void) const
{
  printf( "\n soy animal desconocido");
}

void ANIMAL::Como_Soy(void) const
{
  printf( "\n no se quien soy");
}

void ANIMAL::Quien_Soy(void) const
{ Mi_Clase();
  Como_Soy();
}
```

Listado I.46 Métodos de la clase ANIMAL

```
class ANIMAL_DOMESTICO :public ANIMAL
{
  private:
```

```
    const char *nombre;
  public:
    ANIMAL_DOMESTICO(const char *mi_nombre)
    {
      nombre = mi_nombre;
    }
    virtual void Mi_Clase(void) const;
    virtual void Como_Soy(void) const;
};

void ANIMAL_DOMESTICO::Mi_Clase(void) const
{
  printf( "\n soy animal domestico");
}

void ANIMAL_DOMESTICO::Como_Soy(void) const
{
  printf("\n mi nombre es %s", nombre);
}

class ANIMAL_SALVAJE :public ANIMAL
{
  private:
    const char *alimento;
  public:
    ANIMAL_SALVAJE(const char *me_gusta)
    {
      alimento=me_gusta;
    };
    virtual void Mi_Clase(void) const;
    virtual void Como_Soy(void) const;
};

void ANIMAL_SALVAJE::Mi_Clase(void) const
{
    printf( "\n soy animal salvaje");
}

void ANIMAL_SALVAJE::Como_Soy(void) const
{
   printf("\n me gusta %s", alimento);
}
```

Listado I.47 Clases herederas ANIMAL_SALVAJE y ANIMAL_DOMESTICO, así como los constructores respectivos.

De ésta forma el concepto de herencia introduce una nueva relación de subtipo entre las clases : la relación is-a (es un). Todo objeto de una clase derivada puede verse (es un) como un objeto de la clase base que puede llevar a cabo todas las mismas acciones (posiblemente mejoradas) que el objeto de la clase base puede realizar.

A un puntero o referencia a un objeto de una clase se le puede asignar un puntero o referencia a un objeto de una clase heredera o derivada.

Puede apreciarse entonces que, una Clase es a la vez un módulo y un tipo. Como módulo encierra un número determinado de recursos. Como tipo caracteriza a un conjunto de entidades (objetos).

La herencia aporta una interpretación significativa a ambas perspectivas. Si una clase hereda de otras, extiende y/o mejora los recursos que ésta ofrece como módulo. Desde el punto de vista de tipo, la herencia introduce una relación de subtipo is-a entre la clase heredera o derivada y la clase base, es decir un objeto de la clase heredera puede verse (es un) como un objeto de la clase base.

I.16) La estrecha relación entre el polimorfismo y el enlace dinámico

C++ provee una forma de imponer una función definida en una clase base con una función definida en una clase derivada. Además, otra característica importante de C++ se refiere a que podemos emplear apuntadores a una clase base para referirse a objetos de una clase derivada. La combinación de esas dos características nos habilita para implantar el comportamiento del polimorfismo en clases codificadas en éste lenguaje de programación.

Para aclarar el concepto, recordemos el ejemplo de la clase "figura", la cual es la clase base, se encarga de encapsular datos y funciones comunes a otras clases de figura, tales como *"circulo"* y *"rectángulo"*, quienes son a su vez derivadas de la clase figura. Una de las funciones que se emplean es la llamada función *"dibuja"*, la cual se encarga de dibujar a la clase base figura.

Porque cada figura se dibuja de manera diferente es que, la clase base define a la función dibuja con la palabra ***"virtual"***, como se muestra en el Listado 1.48 :

```
class figura
{
 public:
     virtual void dibuja (void) const { };
  //Otras funciones miembro...
};
```

Listado 1.48 Ejemplo de polimorfismo

La palabra "virtual" le indica al compilador de C++ que la función "dibuja" definida en la clase base será utilizada solamente si las clases derivadas no la definen. En ese caso, la clase base define que "dibuja" sea una función que en realidad no hace nada.

En una clase derivada, es posible que impongamos ésta definición simplemente suministrando una función con el mismo nombre, como lo muestra el Listado 1.49 :

```
class circulo_figura : public figura
{
  //Datos privados...
public:
  //Otras funciones miembro...
    virtual void dibuja (void) const;
};
```

Listado 1.49 Listado muestra de una clase derivada

A diferencia del listado 1.48, en el listado 1.49 notamos que las otras funciones miembro son definidas antes de la clase base.

Una vez definida la función *"dibuja"* para la clase derivada *"circulo"*, podemos realizar lo mismo para la clase derivada : *"rectángulo"*, posteriormente aplicaremos la función miembro a instancias de clases diferentes y la correcta función "dibuja" es la que será llamada, por ejemplo:

```
//Crea instancias de circulo y rectángulo.
circulo_figura c1(100.,100.,50.);
rectangulo_figura r1(10.,20.,30.,40.);
```

```
c1.dibuja();    //dibuja de la clase circulo es llamada.
r1.dibuja();    //dibuja de la clase rectangulo es llamada.
```
Listado 1.50 Creando instancias de la Clase "Figuras"

Aún cuando el listado 1.50 representa una conducta polimórfica, el compilador de C++ puede determinar (analizando el código), exactamente cuál función deberá ser llamada. De hecho, podemos comentar que este caso se refiere al ***enlace estático*** de funciones virtuales, porque el compilador puede determinar la función que será llamada al momento de compilar.

Quizá un caso de estudio más interesante lo presente el **enlace dinámico**, que se efectúa cuando una función virtual es invocada a través de un apuntador a un objeto y el tipo de objeto es desconocido durante la compilación. Lo cual se logra porque C++ es capaz de utilizar un apuntador a una clase base para referirse a una instancia de una clase derivada. Analicemos éste asunto con calma, supongamos que deseamos crear un número de *"figura"* almacenándolo en un arreglo y pretendemos también dibujarlo, si codificáramos lo anterior, posiblemente obtendríamos un listado del tipo :

```
//Crea instancias de circulo_figura y rectangulo_figura

int i;

figura *figuras[2]; // Arreglo de apuntadores a la clase base

//Crea una "figura" y salva los apuntadores

figuras[ 0 ]=new circulo_figura(100.,100.,50.);
figuras[ 1 ]=new rectangulo_figura(80.,40.,120.,60.);

//Dibuja los arreglos
for(i=0;i<2;i++)figuras[i]->dibuja();
```
Listado 1.51 Muestra el uso de polimorfismo empleando enlace dinámica.

Es muy sencillo apreciar que podemos crear un ciclo por medio de los apuntadores al arreglo *"figuras"* y llamar a la función miembro *"dibuja"*.

Esto es, si figuras[i] apunta a una instancia de *"circulo"*, entonces la función *"circulo::dibuja()"* es llamada, de igual forma, si figuras[i] apunta a una instancia de *"rectangulo"*, entonces la función *"rectangulo::dibuja()"* será llamada. En el epígrafe correspondiente a Clases Concretas, se mostró el listado completo de la Clase "Figuras Geométricas".

Según Naba Barkakati :

***"En tiempo de ejecución, los apuntadores del arreglo figuras pueden apuntar a alguna instancia de las clases derivadas desde la clase base figura, de ésta forma la función actual que está siendo llamada en repetidas ocasiones, se encuentra acorde al tipo de apuntador de figura determinado en tiempo de ejecución, éste estilo en el que la función virtual es llamada se denomina :* enlace dinámico"**

I.17) Genericidad

En la programación de clases es frecuente encontrarnos con clases que son muy semejantes y que sólo se diferencian en el tipo de algunas de las entidades de la clase (componentes, parámetros de algunas funciones de la clase, valor retornado por una función). Estas clases ofrecen una misma funcionalidad pero con parámetros o valores de retorno de tipo diferente.

La genericidad permite implantar clases parametrizadas.

Por ejemplo, recordando la Clase Stack, definida con amplitud en el epígrafe ***I.5 Constructores y Destructores***, en ése caso, las componentes de la pila son de tipo "int" (entero), si se deseara definir una clase STACK donde las componentes sean de tipo "float" (flotante), se tendría un listado del tipo :

```
class STACK
{
  private :
     float *ptr_tope;
```

```
    float *ptr_fondo;
    int longitud;
public:
    STACK(int large=100)
    {
      ptr_tope=ptr_fondo = new float[longitud=large];
    }

    int Max_Long(void) const
    {
      return longitud;
    }

    void Push (float x)
    {
      if ( (ptr_tope-ptr_fondo) < longitud)
            *ptr_tope++ = x;
    }

    void Pop(void)
    {
       if (ptr_tope > ptr_fondo)
            ptr_tope--;
    }

    float Top(void) const
    {
       if ( ptr_tope > ptr_fondo )
            return *(ptr_tope-1);
    }

    int Total(void) const
    {
      return (ptr_tope-ptr_fondo);
    }

    int Empty(void) const
    {
       return (ptr_fondo==ptr_tope);
    }

    int Full(void) const
    {
       return ((ptr_tope-ptr_fondo==longitud);
    }
```

```
    ~STACK(void)
    {
      delete ptr_fondo;
    }
};
```

Listado I.52 Definición de la Clase STACK con parámetros de tipo "float:

Como puede observarse esta definición es idéntica a la anterior salvo que las componentes son ahora de tipo "float" (por supuesto si queremos que ambas coexistan en una misma aplicación, entonces no pueden tener el mismo nombre STACK). Análoga situación se presentaría, si se deseara definir una Clase Stack donde las componentes sean de tipo (por ejemplo) CIRCULO.

Esta situación es característica de clases como STACK conocidas como ***clases contenedores***, es decir, clases que contienen objetos de algún otro tipo. Pilas, colas, listas, conjuntos, diccionarios, arreglos, son ejemplos de clases contenedoras. El tipo del objeto contenido es, por lo general, intranscendente para el que define la clase pero importante para el que la utiliza. Es deseable poder definir clases donde el tipo de los objetos contenidos pueda ser un argumento de la clase contenedora. Es entonces el usuario de la clase quien deberá especificar el tipo del argumento de la clase sin tener que ver con la definición interna de la misma.

Esta forma de reusabilidad parametrizando clases, es conocida como genericidad. Definiendo clases con parámetros que representen tipos cualesquiera podemos evitar tener que reescribir clases muy parecidas que se diferencien sólo en el tipo de alguna de las entidades con las que se trabaja dentro de la clase.

La genericidad no es un recurso original de la Programación Orientada a Objetos, pero es un arma que combinada con el modelo de objetos potencia la flexibilidad y la reusabilidad que éste promueve.

I.18) Concurrencia

Para ciertos tipos de problemas, un sistema automatizado puede mantener diferentes eventos simultáneamente. Otros problemas que puede involucrar mucho la computación es que exceden la capacidad de algunos procesadores. En cada uno de esos casos, es natural considerar un conjunto distribuidos de computadoras para la implantación destino o usar procesadores capaces de multiprocesamiento. Un sólo proceso -conocido como un ***hilo de control***- es la raíz de la cual la acción dinámica independiente ocurre dentro de un sistema. Todo programa tiene al menos un hilo de control, pero un sistema involucrando concurrencia puede tener muchos hilos : algunos que son transitorios y, otros que ocurren en la última ejecución del tiempo de vida. Los sistemas ejecutándose a través de CPU's múltiples permiten varios hilos de control para contemplar verdaderamente la concurrencia. Sin embargo, para aquéllos sistemas que corren en un sólo CPU, solamente puede archivar la ilusión de hilos de control concurrentes, usualmente para algunos algoritmos en tiempo-distribuido.

Lim y Johnson señalan que "las características del diseño para la concurrencia en los lenguajes de Programación Orientada a Objetos no son muy diferentes de otros tipos de lenguajes - la concurrencia es ortogonal a la Programación Orientada a Objetos en los niveles más bajos de abstracción. En la Programación Orientada a Objetos o no, todos los problemas tradicionales que suelen acontecer en la programación concurrente, aún permanecen".

En efecto, la construcción de una pieza larga de software es difícilmente suficiente; diseñar una que abarque múltiples hilos de control es aún más difícil. "En el más alto nivel de abstracción, la POO puede aliviar el problema de concurrencia para la mayoría de los programadores, debido al ocultamiento de la concurrencia dentro de abstracciones reusables". Además sugieren que "un modelo objeto es apropiado para un sistema distribuido porque define implícitamente las unidades de distribución y movimiento y las entidades que comunican".

La POO se enfoca a la abstracción de datos, encapsulación y herencia, la concurrencia se enfoca al proceso de abstracción y sincronización.

El objeto es un concepto que unifica esos dos diferentes puntos de vista : cada objeto puede representar un hilo de control separado. Tales objetos son llamados ***activos***. En un sistema basado en un diseño orientado a objetos, algunos de los cuales son activos y así sirven como centros de actividad independiente. Dada ésta concepción definimos la concurrencia como sigue :

"Concurrencia es la propiedad que distingue un objeto activo de uno que no es activo".

I.19) Persistencia

Un objeto en software, ocupa una cantidad de espacio y existe para una cantidad particular de espacio. Atkinson sugiere que ésta es una existencia de objeto contínuo, partiendo de objetos transitorios que surgen dentro de la evaluación de una expresión, a objetos en una base de datos que sobreviven la ejecución de un sólo programa. Este espectro de persistencia de objeto abarca los siguientes pasos :

1) ***Resultados transitorios en la evaluación de la expresión.***
2) ***Variables locales en las activaciones del procedimiento.***
3) ***Variables propias, variables globales y apilación de los detalles cuya extensión sea considerada diferente de su ámbito.***
4) ***Los datos que existen entre ejecuciones de un programa.***
5) ***Los datos que existen entre varias versiones de un programa.***
6) ***Los datos que sobreviven al programa.***

Los lenguajes de programación tradicionales usualmente direccionan solamente los primeros tres tipos de persistencia de objetos; la persistencia de los tres últimos tipos es típicamente el campo de la tecnología de bases de datos. Esto nos conduce al choque de culturas que algunas veces resulta en diseños muy extraños : programadores que finalizan con la habilidad de esquemas "ad hoc" para el almacenamiento de objetos cuyos estados deben ser preservados entre las ejecuciones del programa y, los diseñadores de bases de datos aplican en forma incorrecta su tecnología para arreglárselas sin objetos transitivos.

Unificando los conceptos de concurrencia y objetos dados se incrementan los lenguajes de programación orientados a objetos concurrentes. En una moda similar, introduciendo el concepto de persistencia al modelo objeto, se provoca el incremento de bases de datos orientadas a objetos. En la práctica se cuenta con tales bases de datos construidas sobre tecnología ya probada, tales como modelos de bases de datos : secuencial, indexada, jerárquica, de red o relacionales, pero entonces ofrecen a los programadores la abstracción de una interfaz orientada a objetos, aún cuando los filtros de bases de datos y otras operaciones son completadas en términos de objetos cuyo tiempo de vida trasciende el tiempo de vida de un programa individual. Esta unificación simplifica bastamente el desarrollo de ciertos tipos de aplicaciones.

La persistencia va más allá que el tiempo de vida de un dato. En las bases de datos orientadas a objetos, no solamente hace que el estado de un objeto persista, pero esta clase debe también trascender algún programa individual a todos esos programas intérpretes, esto salva el estado en la misma forma. Se aprecia claramente lo difícil que es mantener la integridad de una base de datos conforme ésta va creciendo, particularmente si nosotros debemos cambiar la clase de un objeto.

En la mayoría de los sistemas, un objeto, una vez creado, consume la misma memoria física hasta que cesa de existir. Sin embargo, para los sistemas que ejecutan un conjunto distribuido de procesos, debemos algunas veces estar interesados en que la persistencia esté contemplada en el espacio disponible. En tales sistemas, es útil pensar en que los objetos deban moverse de máquina a máquina y, que podrían tener diferentes representaciones en diferentes máquinas.

Sintetizando, Grady Booch define la persistencia como sigue :

" La persistenca es la propiedad de un objeto por medio de la cual ésta existencia trasciende el tiempo (el objeto continúa existiendo después de que su creador termina de existir) y/o el espacio (las ubicaciones de los objetos se mueven de las direcciones de espacio en las cuales fueron creadas) "

I.20) Alojamiento dinámico de memoria

Debido a la forma dinámica con que se crean y eliminan objetos durante la ejecución de un *SOO (Sistema Orientado a Objetos)*, es necesario contar con mecanismos que optimicen el uso de los recursos de memoria disponibles. Los mecanismos de recolección de basura liberan automáticamente los segmentos de memoria utilizados por objetos que ya no son referenciados. Algunos lenguajes permiten hacerlo explícitamente y otros implícitamente.

I.21) Enforzamiento de principios

Dependiendo de las características particulares de su diseño, cada lenguaje ***"enforza"*** en mayor o menor medida el uso de los principios de la *TOO (Tecnología Orientada a Objetos)*, cuando se utiliza para el desarrollo de sistemas. Entre las características particulares que influyen en éste sentido de un lenguaje, se encuentran aspectos como : la medida en que se apoya el uso de abstracciones y la seguridad de las mismas, el poder de sus mecanismos de herencia, el uso de enlace dinámico, etc.

I.22) Estándares

Se utilizan para regular las características de un lenguaje en particular y garantizar la compatibilidad de distintos compiladores del mismo, habilitando la portabilidad entre distintas plataformas.

I.23) Conclusiones

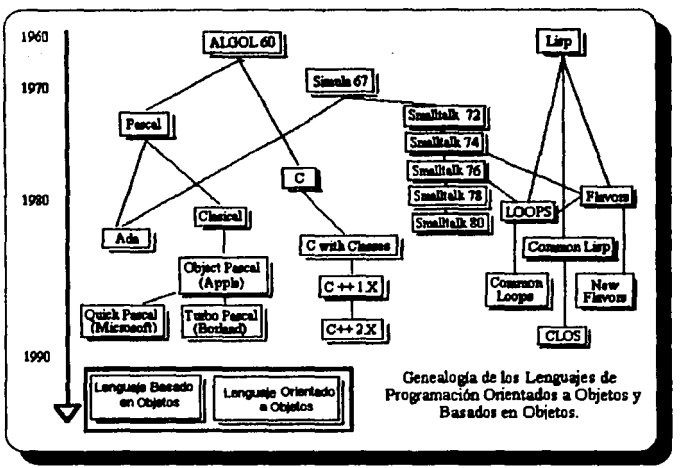
Aún cuando el tema fundamental del presente texto se refiere al "Diseño Orientado a Objetos", es indispensable conocer la Metodología Orientada a Objetos en función de su terminología y, representación gráfica (notación) . No es nada recomendable intentar abordar la

fase de Diseño del Ciclo de Vida de un Sistema Orientado a Objetos, si no se conoce a ciencia cierta, en qué consiste dicho Paradigma.

Obsérvese que, si súbitamente se comenzara a hablar de objetos, clases, relaciones, herencia y polimorfismo por citar algunos de los conceptos que conforman la Metodología Orientada a Objetos, el lector tendría cada vez mayores interrogantes, menor claridad y sin duda alguna, una confusión múltiple.

Por otro lado, hasta éste momento se conocen los fundamentos del Paradigma, como es sabido, es indispensable llevar estos conocimientos a la práctica mediante alguna aplicación dada, pero antes de entrar en materia, ¿ Qué Lenguajes de Programación la soportan ?. Esta interrogante y algunas otras que posiblemente el lector tenga en mente. serán resueltas en el siguiente *Capítulo : Lenguajes Orientados a Objetos*.

CAPITULO II
LENGUAJES ORIENTADOS A OBJETOS



Genealogía de los Lenguajes de Programación Orientados a Objetos y Basados en Objetos.

La computadora es muy quisquillosa. Es peor que el profesor más "hueso" que haya habido jamás;mientras que el profesor puede que ponga mala cara cuando te saltas alguna regla lingüística, o dejarte castigado después de las horas de clase, el ordenador hace algo mucho peor : ***te ignora por completo...***

"La Máquina Superinteligente", Adrián Berry

CAPITULO II
LENGUAJES ORIENTADOS A OBJETOS

II.1) ¿ Qué es un Lenguaje Orientado a Objetos ?

Si pretendiésemos en éstos momentos delimitar la definición precisa de lo que un *LOO (Lenguaje Orientado a Objetos)* representa, no sería apropiada, ya que existen diversas teorías respecto al enfoque orientado a objetos y ello, sin duda alguna originaría polémica, de cualquier forma, lo que si se puede hacer con la certeza de no dañar ningún criterio, es comentar que : en esencia, un **LOO** soporta cuatro características principales :

* **Abstracción** (***Modelación de porciones del mundo real en versiones simplificadas***)
* **Encapsulación** (***Reunión de datos y operaciones que afectan esos datos en un sólo objeto***)
* **Clase Base** (***La clase más generalizada en una estructura de clases***)
* **Herencia** (***Derivación de una nueva clase a partir de otra, permite reutilizar código***)

El hecho de que existan diversas ideologías con respecto al término Lenguaje Orientado a Objetos, se debe entre otras cosas a que no existe una teoría formal de la programación lógica o funcional. Adentrándonos un poco más en éste campo, es conocido por todo programador que, en función de la práctica que desarrolle respecto a alguna teoría de programación y/o el lenguaje de programación que suela emplear, es capaz de crear mayores herramientas en su uso cotidiano, esto da por resultado la creación de "bibliotecas".

II.2) ¿ Qué es una "Biblioteca de Objetos" ?

Una biblioteca de objetos es una colección completa, examinada, documentada y reutilizable de objetos disponibles como un empaque comercial o como una biblioteca de software (en casa). Sin ellas algunos beneficios de la reutilización no podrían ser obtenidos. La mayoría de los LOO cuentan con bibliotecas básicas o genéricas de objetos, el detalle es que se venden adecuadas a una aplicación productiva en desarrollo, no obstante, no hay que descartar

(como comentábamos en el apartado II.1), la posibilidad de que nosotros mismos generemos nuestra propia biblioteca de objetos en función de nuestras necesidades.

II.3) El comportamiento del Modelo Objeto por medio de su "Evolución"

Para poder apreciar la evolución que el modelo objeto ha ido adquiriendo a través del tiempo, es conveniente repasar brevemente las etapas que presentan los lenguajes de programación en todas sus generaciones subsecuentes. Enseguida podemos apreciar un tipo de clasificación de lenguajes de programación, quizá ésta clasificación sea la más conocida, la emitió Wegner, en función de aquello que caracterizó al lenguaje en cuestión :

- Primera Generación de Lenguajes (1954-1958)

Fortran I	expresiones matemáticas
Algol 58	expresiones matemáticas
Flowmatic	expresiones matemáticas
IPL V	expresiones matemáticas

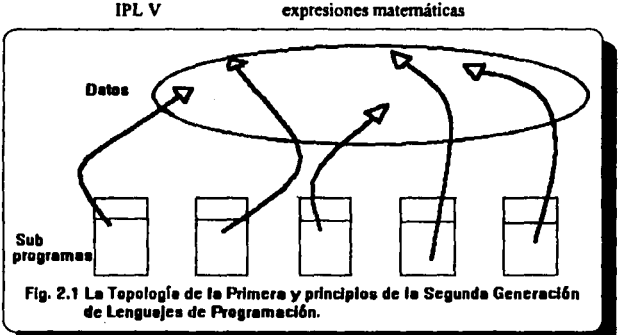


Fig. 2.1 La Topología de la Primera y principios de la Segunda Generación de Lenguajes de Programación.

Es decir, los subprogramas accesan a los datos por vía de una zona común.

- Segunda Generación de Lenguajes (1959-1961)

FORTRAN II	Subrutinas, compilación separada
ALGOL 60	Estructura de bloques, tipos de datos
COBOL	Descripción de datos, manejo de archivos
Lisp	Procesamiento en lista, apuntadores

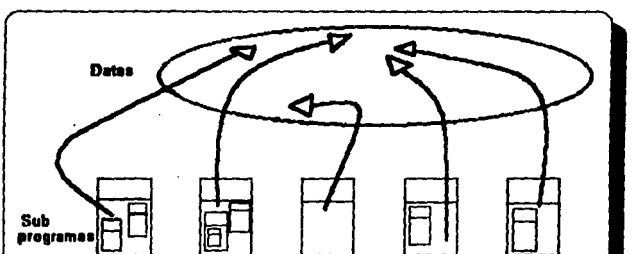


Fig. 2.2 La Topología de la última parte de la Segunda y principios de la Tercera Generación de Lenguajes de Programación.

Los programas son segmentados, pero no así los datos.

- Tercera Generación de Lenguajes (1962-1970)

PL/1	FORTRAN+ALGOL+COBOL
ALGOL 68	Sucesor riguroso de ALGOL 60
Pascal	Sucesor simple de ALGOL 60
Simula	Clases, abstracción de datos

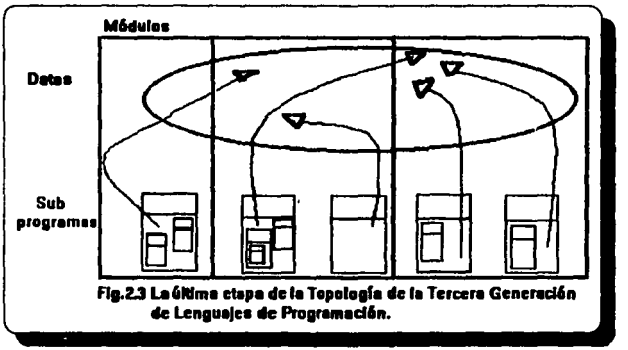


Fig.2.3 La última etapa de la Topología de la Tercera Generación de Lenguajes de Programación.

En base a los estudios sobre estructuras de datos, éstos son organizados para vincularse y estructurarse. Los programas cambian el modo de accesarlos.

- La Generación vacía (1970-1980)

En ésta etapa muchos lenguajes diferentes fueron inventados pero pocos permanecieron. Se acentúa la búsqueda de nuevos enfoques. La programación queda estancada en la práctica.

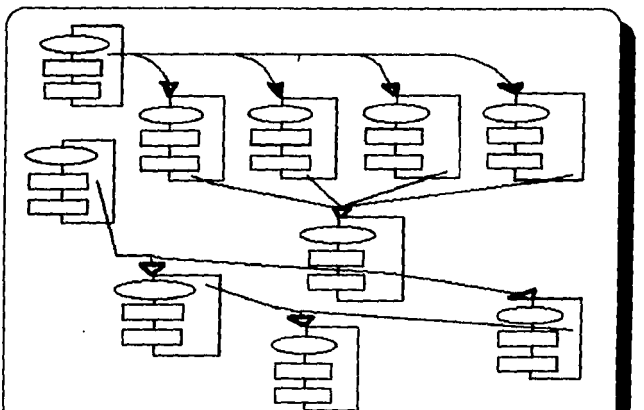


Fig. 2.4 La Topología de unas aplicaciones de tamaño moderado empleando Lenguajes de Programación Orientados a Objetos y Basados en Objetos.

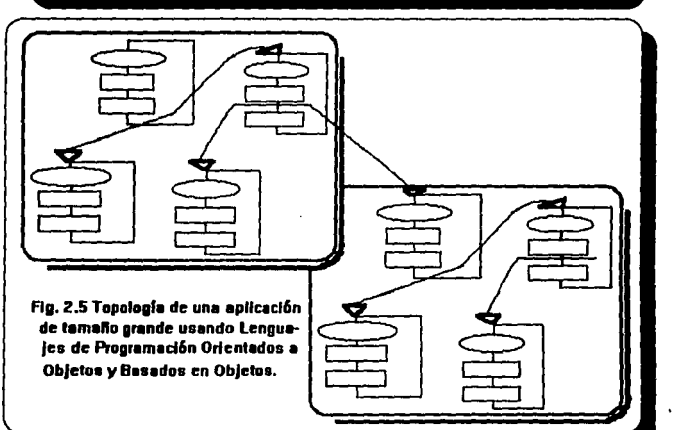


Fig. 2.5 Topología de una aplicación de tamaño grande usando Lenguajes de Programación Orientados a Objetos y Basados en Objetos.

II.4) ¿ Cuántos Lenguajes Orientados a Objetos existen actualmente?

El ***DOO (Diseño Orientado a Objetos)*** no se restringe al empleo de un lenguaje de programación en particular, de cualquier forma, tomemos en cuenta los tres factores que Wulf consideró como fundamentales en la creación de ellos, donde :

Un lenguaje de programación :

- ***Es una herramienta de diseño.***
- ***Es una herramienta para el consumo humano.***
- ***Es el medio por el cual se instruye a una computadora.***

El conocimiento de algunos ***LOO's (Lenguajes Orientados a Objetos)***, especialmente de aquellos con los que contamos con mayor material de apoyo, nos permitirán poseer tecnicidad y rigor en su manejo, dado que una de las propuestas a cumplir en éste trabajo es el desarrollo de pequeños e ilustrativos programas, es conveniente entonces estudiar en primera instancia, qué LOO's existen y posteriormente definir, cuál se adapta a nuestras necesidades.

La presentación histórica es más breve de lo que podríamos imaginarnos, así es que :

Para el año de 1991 existían cerca de 200 lenguajes de programación, los cuales surgieron de requerimientos particulares, la complejidad de los problemas por resolver se incrementó, los desarrolladores, al existir mayor cantidad de lenguajes de programación, contaban cada vez con mayores herramientas para la realización de su labor, muchos aspectos han influído en la evolución de los lenguajes de programación, por ejemplo : el progreso de la teoría de la computación y, el aprendizaje de nuevas lecciones (tal como lo muestra la clasificación anterior).

En términos de Software, aún cuando éste evoluciona en menor medida con respecto al Hardware, se ha ido incrementando gradualmente su crecimiento, los avances más recientes en los lenguajes de programación se deben a que : **actualmente existen cerca de 100 LOO (Lenguajes Orientados a Objetos) y LBO (Lenguajes Basados en Objetos), así como LEO (Lenguajes Extendidos a Objetos),** contando entonces con una gama tan amplia de lenguajes

de programación, la labor de elegir uno de ellos no resulta sencilla. Pero antes de continuar, analicemos ¿ Qué diferencia existe entre los LOO's, LBO's y los LEO's ?

La diferencia es básica, se considera que :

* **Un LBO (Lenguaje Basado en Objetos)** ***es capaz de soportar directamente abstracción de datos y clases.***

* **Un LOO (Lenguaje Orientado a Objetos)** ***es aquél que no es un LBO, pero proveé soporte adicional para la herencia como un significado de expresiones jerárquicas de clases.***

* **Un LEO (Lenguaje Extendido a Objetos)** ***es un lenguaje de programación que originalmente no contenía la programación orientada a objetos y al cual, posteriormente se le hizo un añadido para soportar los objetos, el lenguaje primitivo se convierte entonces en un subconjunto del Lenguaje Extedido a Objetos.***

A continuación citamos una lista con algunos de los LOO's, LBO's y LEO's, en sus correspondientes categorías :

Lenguajes Orientados a Objetos exclusivamente :

- CLOS
- Eiffel
- Simula
- Smalltalk
- Prolog++ y DPL

Lenguajes convencionales extendidos :

- C++
- Objective C
- Object Pascal y Turbo Pascal
- Modula 3 y Classic Ada
- Object COBOL

Lenguajes extendidos a ambientes LISP y AI :

- KEE, Joshua y Art
- KBMS y ADS
- Nexpert Object y Nivel % de Object
- ProKappa, Kappa, ObjectiQ y XShell

Lenguajes basados en objetos :

- Ada
- Modula 2
- Ellie

Lenguajes basados en clases :

-CLU

El empleo de lenguajes de programación convencionales no suele ser muy recomendable, (por supuesto en términos orientados a objetos), ya que algunos beneficios que nos aporta el DOO se perderían con su uso. El ciclo de vida en cascada no era apropiado para la POO, el prototipo esencial lo constituía el trabajo en grupo en el cual era notable la intención del uso de las bibliotecas de clases, esto aconteció hasta la llegada de C++ , por cierto, C++ es considerado como un lenguaje de programación práctico y, de propósitos generales. No obstante, en el campo de la computación el interés por el conocimiento de los lenguajes de programación : Smalltalk y Eiffel se ha ido incrementado. En el mundo de sistemas comerciales y mainframes el empleo de Object COBOL (Orientado a Objetos), ha sido trascendental. En la **Fig. 2.6** mostramos el trabajo desarrollado por **Schmucker,** así como la genealogía de los cinco LBO's y LOO's que **Grady Booch** considera han influído ampliamente en el empleo de ambas metodologías : Smalltalk, Object Pascal, C++, CLOS (Common Lisp Object System) y Ada.

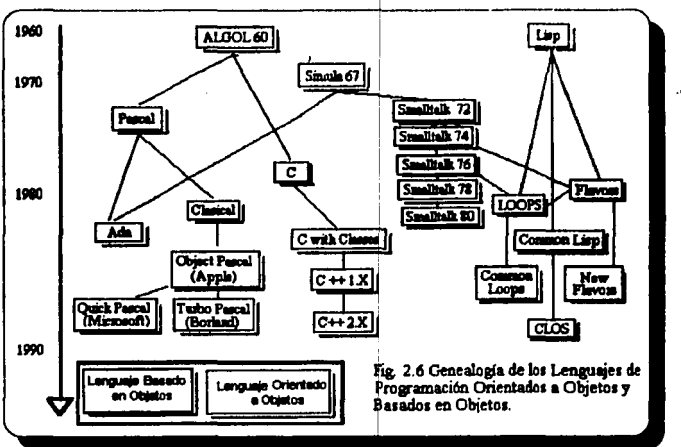


Fig. 2.6 Genealogía de los Lenguajes de Programación Orientados a Objetos y Basados en Objetos.

II.5) Haciendo Historia

El Lenguaje considerado por diversos autores doctos en la materia como origen del paradigma orientado a objetos es Simula, una de sus ramificaciones es Simula 67, el cual introdujo la noción de clases y herencia.

II.5.1) Simula

Los orígenes de Simula se remontan a 1967, entre sus creadores encontramos a Dahl y Nygaard, 1966; Dahl, Myrhaug y Nygaard, 1968; quienes se basaron en los primeros trabajos de un lenguaje especializado : **Simula 1**, éste y sus descendientes fueron estrechamente influenciados por ALGOL con la idea de bloques de programa.

En 1949, Simula fue empleado en eventos de simulación e investigación operacional, fecha en la que Nygaard trabajó en el diseño de reactores nucleares.

Los lenguajes convencionales presentaban una abstracción difícilmente representable en procesos de cómputo, Simula, en contraste, era capaz de describir procesos reales. Además, Simula tenía una biblioteca especial de clases que contenía la mayoría de los requerimientos primitivos para simulación de eventos discretos, poseía también *"co-rutinas"* que podían ser invocadas mientras que, el proceso principal continuaba ininterrumpidamente hasta necesitar el resultado de la co-rutina. Simula contempla las ideas de ALGOL, además de los conceptos de encapsulación y herencia. Algunas características de Simula se presentan a continuación en :

Chequeo de tipo	**Dinámico o estático**
Polimorfismo	**Si**
Ocultamiento de la información	**Si**
Concurrencia	**Si**
Herencia	**Si**
Herencia Múltiple	**No**
Persistencia	**No**
Genericidad	**No**
Bibliotecas objeto	**Simulación**

Tabla 2.1 Características de Simula

Con Simula la idea de la abstracción se encuentra implícita, pero el paso de mensajes arribó junto con Smalltalk (actualmente los desarrolladores de Simula trabajan en un nuevo lenguaje BETA que incluye programación funcional y también son responsables de la especificación de un lenguaje formal orientado a objetos llamado ABEL).

II.5.2) Smalltalk

Smalltalk es considerado como la representación pura del ideal de los LOO's, las versiones que sucedieron a Smalltalk son identificadas por el año de su creación y son cinco : Smalltalk 72-74-76-78-80, además de Smalltalk V.

Smalltalk fue creado en los años 70's por Alan Kay, Adele Goldberg, Daniel Ingalls y por el grupo de investigación de Palo Alto Xerox. Simula fue su primer influencia, también influyeron en Simula : el lenguaje FLEX y el trabajo de Seymore Papert Y Wallace Feurzeig. Smalltalk posee un ambiente completo de programación, tanto en lenguaje como en ambiente de desarrollo, con editores, mostrando clases jerárquicas y aspectos de 4GL (Lenguajes de Cuarta Generación). Para éste lenguaje todas las cosas son visualizadas como objetos, es quizá el más importante LOO y, ha influído positivamente en la mayoría de los LOO subsecuentes.

La evolución de Smalltalk tomó cerca de una década de trabajo, estableció la comunicación entre objetos, no es un lenguaje rápido en términos de ejecución, también introdujo el concepto de interfaces icónicas. Un concepto realmente importante aportado por Smalltalk, lo es la ***"metaclase", la cual es una clase de clases, con ella no existe instanciación (indica cuando un objeto pertenece a determinada clase, una instancia posee estado, conducta e identidad, considérense intercambiables los conceptos de objeto e instancia), sin embargo algunos conceptos concretos pueden ser heredados.***

En las **Tablas 2.2 y 2.3** se muestran algunas características de Smalltalk :

Chequeo de tipo	**Dinámico**
Polimorfismo	**Si**
Ocultamiento de la información	**Si**
Concurrencia	**Pobre**
Herencia	**Si**
Herencia Múltiple	**No**
Persistencia	**No**
Genericidad	**No**
Bibliotecas objeto	**Gráficas en su mayoría**

Tabla 2.2 Características de Smalltalk

Smalltalk 80 se comercializó en primer término en computadoras personales y estaciones de trabajo.

Abstracción	**Instancia de variables** **Instancia de métodos** **Variables de Clase** **Métodos de Clase**	**Sí** **Sí** **Sí** **Sí**
Encapsulación	**De variables** **De métodos**	**Privadas** **Públicas**
Modularidad	**Tipos de módulos**	**Ninguno**
Jerarquía	**Herencia** **Unidades genéricas** **Metaclases**	**Solamente** **No** **Sí**
Tipificación	**Fuertemente tipificable**	**No**
Concurrencia	**Multiprocesamiento (definido por clases)**	**Sí**
Persistencia	**Objetos persistentes**	**No**

Tabla 2.3 Siete características del modelo objeto de Smalltalk

II.5.3) Object Pascal

Fue creado por los desarrolladores de la Compañía Apple conjuntamente con Niklaus Wirth, creador de Pascal. Su antecesor es Clascal, el cual es una versión orientada a objetos de Pascal por Lisa. Estuvo disponible en 1986 y es el primer LOO soportado por la Macintosh Workshop y programadores. Fue la inspiración de dos versiones más de LOO : Quick Pascal de Microsoft y Turbo Pascal 5.X de Borland. Object Pascal no contempla métodos de clase, variables de clase, herencia múltiple o metaclases, la **Tabla 2.4** muestra sus características:

Abstracción	**Instancia de variables** **Instancia de métodos** **Variables de Clase** **Métodos de Clase**	**Sí** **Sí** **No** **No**
Encapsulación	**De variables** **De métodos**	**Públicas** **Públicas**
Modularidad	**Tipos de módulos**	**Unica**
Jerarquía	**Herencia** **Unidades genéricas** **Metaclases**	**Solamente** **No** **No**

Tipificación	**Fuertemente tipificable** **Polimorfismo**	**Sí** **Sí**
Concurrencia	**Multiprocesamiento (definido por clases)**	**No**
Persistencia	**Objetos persistentes**	**No**

Tabla 2.4 Siete características del modelo objeto de Object Pascal

II.5.4) Eiffel

Por otro lado, en 1988 surgió Eiffel, creado por Meyer y, constituye un lenguaje que direcciona las ideas de corrección, robustez, portabilidad y eficiencia. Las clases de Eiffel no son objetos, es capaz de emplear la tipificación estática para ayudar a eliminar errores al momento de correr el programa, además otorga eficiencia.

Su terminología difiere de otros lenguajes, para Eiffel los métodos se denominan "rutinas". Una característica importante de Eiffel es su capacidad de especificar propiedades formales, en donde unas operaciones de un objeto deben ser respetadas escribiendo sus *"aserciones o afirmaciones"*. La aserción permite a Eiffel incorporar características de manejo de excepciones.

La **Tabla 2.5** muestra algunas características de Eiffel :

Chequeo de tipo	**Estático**
Polimorfismo	**Sí**
Ocultamiento de la información	**Sí**
Concurrencia	**Prometida**
Herencia	**Sí**
Herencia Múltiple	**Sí**
Persistencia	**Algún soporte**
Genericidad	**Sí**
Bibliotecas objeto	**Un poco**

Tabla 2.5 Características de Eiffel

II.5.5) C++

El lenguaje "C++" es una extensión orientada a objetos del lenguaje de programación imperativo C. C++ fue creado a principios de los años 80's, por Bjarne Stroustrup de los laboratorios Bell de AT&T, surgió cuando Stroustrup se encontraba adicionando características a C para lograr mayor eficiencia en el manejo de programas con eventos de simulación, su inspiración provino del Lenguaje Simula67, el cual soportaba el concepto de una "clase", AT&T realizó innumerables mejoras a C++ antes de su comercialización alrededor de 1985, desde entonces, C++ ha ido evolucionando bajo el control de las versiones de AT&T.

Una aproximación en la eficiencia de la POO (Programación Orientada a Objetos) ha sido la extensión de un lenguaje existente a uno que incluya características orientadas a objetos. El lenguaje "C" es un ejemplo palpable de ello, en sus sucesivas modalidades : Objective-C, C++ (Cox, 1986; Cox y Novobilski,1991), C++ 1.0 y C++ 2.0, C++ 4.0 por Borland y C++ 7.0 por Microsoft.

Las modificaciones y/o cambios al compilador de C y extensión de su sintaxis mediante un nuevo tipo de datos primitivo para las clases fue introducido por los laboratorios Bell AT&T's : C++ (Stroustrup 1986).

C++ es una modificación genuina de C, es quizá una extensión misma de C, también fue influído por Simula. El prinicipal cambio lo representa la introducción de un nuevo y primitivo tipo de datos : *"clase"*, el cual no contempla tipos de datos de alto nivel, de tal forma que los nuevos tipos se definen en el lenguaje mismo. C++ se diseñó pensando en portabilidad y eficiencia, soporta abstracción, herencia, referencia a sí mismo, adicionalmente, C++ cuenta con las características de C.

Las declaraciones de clase se dividen en una parte pública y una privada. ***Los métodos en C++ se denominan "funciones miembro".*** El paso de mensajes corresponde a una llamada(s) a función. C++ corrige muchas de las deficiencias de C y añade al lenguaje clases,

tipificación, sobrecarga, tipos constantes, referencias, funciones en linea, clases derivadas y, funciones virtuales.

C++ 1.0 no contemplaba la adición de las todas características de C, la versión 2.0 (1989) es el producto mejorado entre versiones previas en una variedad de formas, como lo es la introducción de herencia múltiple. Algunas características de C++ se muestran en :

Chequeo de tipo	**Dinámico o Estático**
Polimorfismo	**Si**
Ocultamiento de la información	**Si**
Concurrencia	**Pobre**
Herencia	**Si**
Herencia Múltiple	**Si**
Persistencia	**No**
Genericidad	**Si (templates)**
Bibliotecas objeto	**Un poco**

Tabla 2.6 Características de C++

Abstracción	**Instancia de variables** **Instancia de métodos** **Variables de Clase** **Métodos de Clase**	**Sí** **Sí** **No** **No**
Encapsulación	**De variables** **De métodos**	**Públicas** **Públicas**
Modularidad	**Tipos de módulos**	**Unica**
Jerarquía	**Herencia** **Unidades genéricas** **Metaclases**	**Solamente** **No** **No**
Tipificación	**Fuertemente tipificable** **Polimorfismo**	**Sí** **Sí**
Concurrencia	**Multiprocesamiento (definido por clases)**	**No**
Persistencia	**Objetos persistentes**	**No**

Tabla 2.7 Siete características del modelo objeto de C++

II.6) Análisis Comparativo de tres reconocidos lenguajes en Programación Orientada a Objetos : C++, Eiffel y Smalltalk

En el siguiente cuadro presentamos un resumen de los conceptos citados en Capítulo I, respecto a tres de los Lenguajes Orientados a Objetos más populares. El número de cruces en cada casilla nos representará el grado de evolución y/o soporte que presenta cada lenguaje con respecto al concepto en evaluación, por otro lado, un guión significará que el lenguaje en cuestión no proporciona el mecanismo.

CUADRO SINOPTICO COMPARATIVO

Capítulo I	C++	Eiffel	Smalltalk
Abstracción de Datos	++	+++	++
Herencia	++	++	+
Validación de Tipos	+(estática)	+(estática)	+(dinámica)
Enlace Dinámico	+	++	++
Sobrecarga	++	+	+
Funciones Virtuales	+	+	-
Genericidad	+	++	-
Alojamiento de Memoria	+	++	++
Principios	+	++	++
Estándar	+	+	+

Tabla 2.8 Soporte a los Elementos de la POO de C++, Eiffel y Smalltalk

Otras características a considerar en estos lenguajes son las siguientes :

LENGUAJE *C++*

- Compatibilidad con C a Nivel Lenguaje.
- Amplia gama de bibliotecas disponibles.
- Estándar de hecho en el mercado actual.

LENGUAJE *Eiffel*

- Ambiente de programación visual.
- Especificaciones.
- Sistema de documentación automática.
- Soporte automático de mantenimiento de Bibliotecas.
- Depurador.

LENGUAJE *Smalltalk*

- Ambiente de programación visual.
- Depurador.
- Poder para el Desarrollo de Sistemas de Graficación.

Existen en gran cantidad lenguajes que no son dedicados por completo a la metodología orientada a objetos, usualmente tratan sobre abstracción o herencia, tales como : Ada, Modula-2 y aún Object Pascal, la amplia gama de lenguajes de programación enfocados u orientados a objetos es enorme (Véase epígrafe II.4).

II.7) ! Más Lenguajes de Programación Orientados a Objetos !

Sanders en su artículo *"Trabajo de la Progra nación Orientada a Objetos"*, proveé una encuesta acerca de ocho diferentes LOO, sugiriendo que éstos deben ser agrupados dentro de 7 categorías :

- **Actor.....Lenguajes soportando delegación**
- **Concurrente.....Enfatizando la concurrencia de los LOO**
- **Distribuídos.....Enfatizando los objetos distribuídos en los LOO**
- **Estructura.....Lenguajes que soportan la teoría de la estructura**
- **Híbridos.....Extensiones o-o de los lenguajes tradicionales**
- **Basado en Smalltalk.....Smalltalk y su dialéctica**
- **Ideológico.....Aplicación de características o-o a otros dominios**
- **Misceláneos.....LOO que no se colocan en ninguna otra categoría**

La **Fig. 2.7** tomada del libro *"Oriented Object Design" de Grady Booch*, muestra una lista de los diversos lenguajes de programación basados en objetos y orientados a objetos.

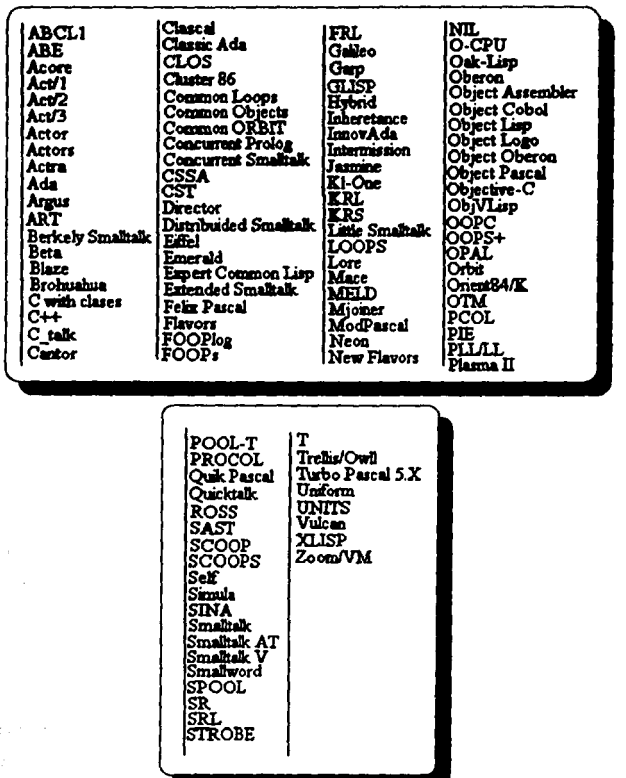


Fig. 2.7 Lenguajes de Programación Orientados a Objetos y Basados en Objetos.

II.8) ¿ Qué pasos seguir para seleccionar un Lenguaje Orientado a Objetos ?

La pregunta básica es : ¿ Cómo elegir una herramienta que otorgue los beneficios de los Métodos Orientados a Objetos ? La respuesta, dada la gran variedad de LOO's, no es simple. Quizá podríamos comenzar por sugerir la búsqueda de las características y requerimientos para cada aplicación y tratar de elegir la mejor herramienta para ese proyecto. Posteriormente, la siguiente pregunta a realizar sería : ¿ Es correcto el impacto organizacional de nuestra elección? Es posible que hallemos la solución en los beneficios que pueden obtenerse mejor a través del AOO (Análisis Orientado a Objetos) y el DOO (Diseño Orientado a Objetos), independientemente del lenguaje de programación elegido.

Como apoyo didáctico es posible que nos auxilie en nuestra elección una relación que incluya las características de los diferentes LOO, sería imposible redactar una guía que nos indique cuál debemos elegir como una receta de cocina, todo depende de la finalidad (aplicación real) que tengamos en mente. Como se observa, el AOO y el DOO aún cuando se encuentren enlazados estrechamente, merecen ser conceptualizados de manera independiente, como se apreciará en los capítulos III y IV respectivamente.

II.9) En lo sucesivo : ¿ Por qué codificar en C++ ?

Por supuesto, aún cuando la elección de un lenguaje deberá realizarse a juicio de nuestras necesidades, se ha optado por citar uno en particular : el Lenguaje de Programación "C++", el cual representa el material de estudio requerido en nuestro texto, considerando que :

- Es un Lenguaje muy dinámico y posee características reales de la Metodología Orientada a Objetos, dado que éste lenguaje de programación representa una versión extendida del Lenguaje C, además de que se originó a raíz de una mejor aplicación bajo éste enfoque.

- En México existe muchísimo material bibliográfico que trata con tanta profundidad como se deseé, programación bajo éste lenguaje (libros, software,etc.).

- Porque nos encontramos con la posibilidad de que nuestros lectores (usuarios finales de éste texto) aún cuando no lo dominen, seguramente han escuchado hablar del Lenguaje "C" y habrán tenido cursos básicos del mismo, en caso de no ser cierto esto, familiarizarse con el lenguaje "C" ó con "C++" no deberá observarse como un punto insuperable para el lector.

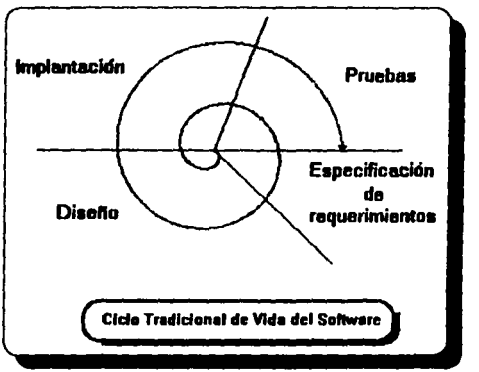
- Por otro lado, "C++" es un lenguaje cuya codificación breve y muy concisa emplea poco espacio, lo cual significa la gran oportunidad de manejar diversos ejemplos, útiles a los fines perseguidos a largo del presente trabajo.

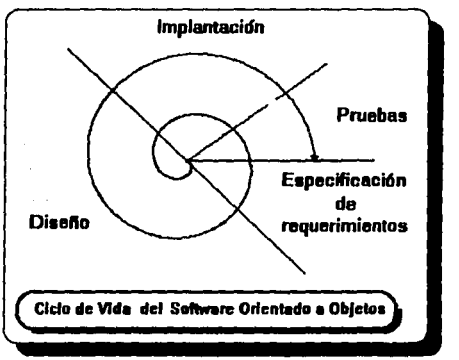
Podría pensarse que, la elección del lenguaje de trabajo es un tópico muy personal y, de alguna forma a decir de las razones presentadas, así es, sin embargo, el lector podrá emplear como material de referencia todo aquello que se presente a partir del siguiente subtema, de tal forma que, si decide utilizar cualquier otro LOO, procure solamente considerar la idea mostrada en éste texto, en cualquier caso :

! Felicidades por su elección ! y, adelante ...

En los Capítulos III y IV, se presenta el proceso de Análisis y Diseño Orientado a Objetos con una aplicación real, útil y verdaderamente amena : "Un Juego de Computadora", la idea no es programar un "Videojuego" espectacular con efectos multimedia, resaltando esencialmente las características y capacidades del equipo de cómputo disponible, en lo absoluto, la intención es mostrar que tal como el Paradigma Orientado a Objetos pretende modelar un problema del mundo real por medio de una computadora, de la misma forma se muestre una aplicación latente y real, un juego para computadora que no requiera mayores elementos electrónicos que una computadora, un compilador C++ y el entusiasmo del lector para llevarlo a la práctica.

CAPITULO III
" ANALISIS ORIENTADO A OBJETOS "





La capacidad de verificar y validar sistemas a gran escala es una medida fundamental de la madurez tecnológica. La investigación bidireccional brinda esto a la Tecnología Orientada a Objetos.

Kenneth S. Rubin & Adele Goldberg.

CAPITULO III
" ANALISIS ORIENTADO A OBJETOS "

El "boom" de nuevas propuestas de lenguajes de programación que incorporaban los conceptos de objeto, clases, herencia, etc., ocurrido en la primera mitad de los 80's, ha causado una gran desorientación entre los diseñadores y programadores de sistemas. El lenguaje C que empezó a ganar terreno como el estándar para la programación de sistemas comerciales dejó de ser el "rey invencible". La amenaza llegó con olas de nuevas propuestas, sobre todo de C++ y, las personas empezaron a preguntarse si ya era hora de cambiar.

Por otro lado, la variedad de nuevos lenguajes que estaban saliendo bajo la bandera "Orientada a Objetos" llevó a los investigadores a buscar una síntesis de las propuestas, un denominador común que permitiera hablar del modelo de objetos sin necesidad de referirse a un lenguaje de programación en particular. Además, se observó que éste modelo, por ser lo suficientemente abstracto, puede aprovecharse para definir nuevos métodos del Análisis y Diseño de Sistemas de Software.

En los últimos cinco años se han publicado (sin exagerar) docenas de artículos y varios libros con propuestas de diversos métodos para el análisis y diseño que presumen soportar el modelo de objetos. La variedad causa desconcierto entre los posibles usuarios. Una vez más surgen las preguntas : ¿ Cuál escoger ? ¿ Cuál aprender ? ¿ Cuál es el mejor ? Las mismas preguntas se están haciendo los productores comerciales de herramientas de software que apoyan el desarrollo de sistemas, ya que, en función del método que elijan dependerá su éxito o fracaso.

La Tecnología Orientada a Objetos (TOO), no es un tópico de investigación larga y obscura. Esta ha sido aceptada de todo corazón en principales firmas como lo son, *la Federal Express, Citibank, John Deere, Boeing, Allen Bradley* y muchas de las firmas de servicios de negocios y financieros en Wall Street.

Sin embargo, la adopción de la tecnología y del proceso pensado bajo la tecnología son dos cosas diferentes. Tómese en cuenta un punto crítico : aprender los conceptos es un reto que se piensa lejano considerando que se necesitan aprender las herramientas y los lenguajes.

Irónicamente, muchas compañías intentan emigrar a la Tecnología Orientada a Objetos para tener sus desarrollos experimentales con un primer lenguaje de Programación Orientado a Objetos. Intentado los métodos necesarios para atacar proyectos a larga escala, éstos vienen a ser secundarios del todo. Entonces surge la inquietud : ¿ Emergerá por Osmosis el entendimiento de los conceptos ? La mayoría de las organizaciones han obtenido sucesivamente el conocimiento del primer concepto y ésta es la única forma de empezar. Cualquier actividad que se disfrute toma tiempo y dinero. Aprendiendo los conceptos para empezar con el correcto direccionamiento para los problemas. Este texto tiene la intención de ayudar con una parte de estos desafíos : enseñando la clave consecuente en el Análisis y Diseño Orientado a Objetos.

El objetivo de éste capítulo es presentar los componentes más importantes de los métodos de Análisis y Diseño Orientado a Objetos (ADOO), así como dotar al lector de criterios para poder elegir un método que le convenga en una aplicación particular. Como podrá observarse, la enorme diversidad de métodos de Análisis Orientado a Objetos dificulta la labor de elección del más apropiado o quizá, el más óptimo para una aplicación dada. Sin embargo y, precisamente en función de conocer todas éstas metodologías **(Véase el Apéndice "A" Metodologías de Análisis Orientado a Objetos)**, es que, el Método que presenta Rebeca Wirfs será, sin duda alguna, el más práctico, sencillo y fácil de seguir para el lector, por tal motivo, éste representará el método base empleado.

III.1) ¿ Cuál es el ciclo de vida de un Sistema ?

La mayor parte de los autores citan el inicio de la vida de todo software con la llamada etapa de **"Especificación de Requerimientos"**, personalmente me parece que ésta etapa debería ser denominada más apropiadamente como **"Especificación y Análisis de Requerimientos"**, dado que es aquí donde comienza el Análisis del Sistema, el examen del o los objetivos

propuestos y el material que se empleará para lograrlos; posteriormente el software continua en la segunda etapa, la de **"Diseño"**, donde se muestra el cómo se realizará el sistema en función de los elementos contemplados en el análisis, según lo cual el sistema es **"Implantado"** y, finalmente probado. Para lograr que el sistema cumpla debidamente con los servicios para los cuales fue solicitado, es realmente conveniente que los usuarios finales tengan acceso a él, lo manipulen y verifiquen su funcionalidad, errores ocultos y posiblemente aportaciones al mismo, esta etapa es conocida como de **"Mantenimiento"**, una vez que los usuarios se convierten en operadores expertos (es decir, que han trabajado lo suficiente con el sistema de tal forma que sienten conocerlo aún mejor que el propio diseñador), el software puede ser extendido.

La creación de una versión posterior, indudablemente mejorada de la anterior, implica nuevamente, el paso por cada una de las fases de especificación/análisis, diseño, implantación y prueba, de ahí surge el concepto de "Ciclo", puesto que, la continua revisión de un sistema para aplicación específica propicia que las etapas que originaron el sistema sean retomadas nuevamente, lográndose un producto de mayor calidad. Sin embargo, resulta curioso por ejemplo, comparar a un "ser humano" con un "sistema", podremos notar que el primero (el hombre) inevitablemente constituye un ser orgánico finito, esto es, nace, crece y muere, por el contrario, el software constituye un ente infinito, puesto que también nace, crece y, nuevamente puede volver a nacer, por consiguiente a crecer y su utilidad jamás perece (a menos que la Compañía que lo requirió desaparezca). Por la sencilla razón que el software no es necesariamente un proceso que se ejecute una sola vez en su vida, el término ciclo viene siendo el más apropiado a emplear al hablar de sistemas, es conveniente entonces, imaginar que la vida del software es una espiral en la cual el ciclo de creación puede repetirse tantas veces como sea necesario.

III.1.1) Ciclos de Software Tradicional y Orientado a Objetos

Consideremos el proyecto medio de software, es decir, la vida del software hasta su graduación. La programación orientada a los procedimientos (programación tradicional), se enfoca primero en cómo implantar, produciendo una espiral como la mostrada en la **Fig. 3.1**

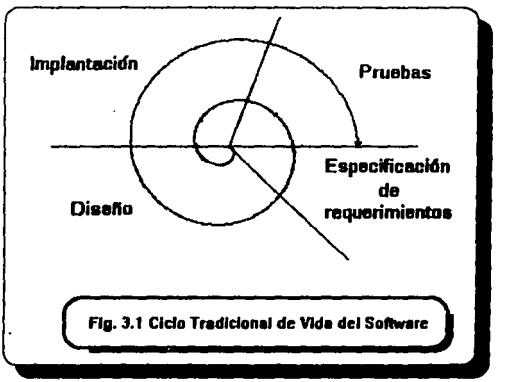


Fig. 3.1 Ciclo Tradicional de Vida del Software

En el desarrollo tradicional de un Sistema, una gran porción del tiempo de vida del proyecto es gastado en la implantación del diseño. A diferencia de ello, el propósito del diseño orientado a los objetos es tener un software robusto que pueda ser fácilmente reutilizado, refinado, probado, mantenido y, extendido, de tal modo que se asegura una madurez productiva. Esto suena bastante bien pero ¿ Qué signifca ?

El software es **reutilizado** cuando es usado como una parte de otro software para el cual no fue inicialmente diseñado. El software es **refinado** cuando es usado como la base para la definición de otro software. El software se **prueba** cuando su conducta se determina conforme a la especificación del software, o no, para aquellos casos remotos en los que la conducta del software es completamente opuesta a lo que se pretendió que hiciese en un principio. El software se **mantiene** cuando se encuentran errores y se corrigen. Y, finalmente, el software se **extiende** cuando una nueva funcionalidad es agregada a un programa ya existente.

La espiral de un proyecto de software orientado a objetos idealmente tiende a parecerse a la mostrada en la **Fig. 3.2**. Esta espiral claramente muestra que se gasta una fracción más

grande del tiempo completo en la fase de diseño; disminuyéndose el tiempo gastado una fracción más pequeña en las fases de implantación y prueba.

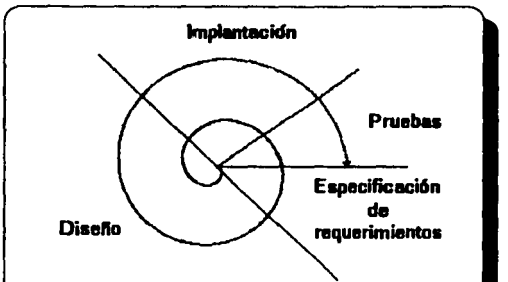


Fig. 3.2 Ciclo de Vida del Software Orientado a Objetos

Se gasta una proporción más grande del tiempo en diseñar porque el software está siendo diseñado para una fácil reutilización, mantenimiento y, modificación. Las herramientas de programación orientada a objetos por sí mismas no garantizan la reutilización, mantenimiento y, extensibilidad del software. No son, en realidad, absolutamente necesarias. En lugar de eso, las herramientas de programación orientada a objetos pueden ayudar a soportar un proyecto en que los miembros del equipo usen el tiempo explorando el problema y haciendo un cuidadoso diseño. Se requiere un gran esfuerzo para diseñar código que se pretenda reutilizar, debido entre otras cosas, a la necesidad de procurar generalización en las clases definidas, pero ese esfuerzo se paga generosamente al final. El tiempo gastado en hacer un diseño cuidadoso permite entender el problema más profundamente. La implantación es más rápida porque ya se ha aprendido mucho del problema. Por lo tanto, el tiempo total requerido para un ciclo puede permanecer sin cambios o aún disminuirse. Y porque el software ha sido diseñado desde el inicio teniendo en mente la idea de mantenimiento, extensión y, reutilización, el tiempo requerido para esfuerzos subsiguientes disminuye radicalmente.

Como vemos, los resultados con la metodología orientada a objetos son verdaderamente visibles para algunos a largo plazo, para los expertos es posible que sea visible desde el inicio, dada la posibilidad de crecimiento, la gran capacidad de reutilización de un sistema que puede ser mayormente aprovechado con usos inimaginables al momento de su creación.

III.1.2) ¿ Qué criterio seguir ?

Si la continuidad en el estudio de la Metodología Orientada a Objetos nos ha proporcionado herramientas suficientes para el inicio de todo sistema de manera más completa, entonces se han logrado mayores conocimientos y técnicas, comentarlas en el orden en el que se han ido originando no es precisamente, la forma más conveniente, es más bien, una filosofía de estudio, por lo cual, se hace énfasis en que hay que trabajar con todo el material disponible al respecto en la secuencia de creación de un sistema. Sin embargo, para el lector se encuentra abierta la opción de asumir su manera de aplicar éste estudio; concretamente : analizarlo de una u otra forma (hablando de cronología) no es mejor ni peor, son simplemente, filosofías que cada autor interpreta y emplea según le convenga. Es también muy importante recordar que no existe una metodología estricta y específica para resolver un problema (sistema), es decir, no existen *"recetas de cocina"* en las que se nos indique paso a paso que acción tomar para una situación determinada, el software es frecuentemente una herramienta muy versátil y las aplicaciones muy específicas, así pues, la mejor y mayor ayuda que se puede proporcionar con el presente trabajo es brindar diversas posibilidades de Análisis y Diseño enfocadas a la Metodología Orientada a Objetos, al final del capitulado, exactamente en el Apéndice "A", se encuentran otras filosofías de estudio que, definitivamente se ha decidido agregar con la intención de mostrar todas las posibilidades existentes, de tal forma que se elija la mejor opción que se adapte a sus necesidades.

De cualquier forma, pasemos a la desintegración de cada una de las etapas de las que hemos estado platicando en cuanto al ciclo de vida del software :

III.1.3) ¿ Qué entendemos por Análisis ?

Para comenzar diremos que el análisis nos permite desintegrar un problema en todos los elementos que lo conforman, en la búsqueda de un planteamiento apropiado a las necesidades y requerimientos futuros del mismo, por supuesto, hablamos de un "problema" que se deseé analizar en términos de computación. Además de ello, el análisis es verdaderamente útil en la especificación de los requerimientos del usuario, en la estructuración del sistema y en su funcionalidad.

Existen varios tipos de análisis, entre los más convencionales encontramos : el ***Análisis Estructurado***, en el cual se emplea el diseño ***Top-Down (Arriba-hacia-Abajo)***, partiendo del concepto más general al más particular; o bien, el diseño conocido como ***Bottom-Up (Abajo-hacia-Arriba)***. Otro tipo de análisis es aquél que se fundamenta en la descomposición funcional combinada con el análisis separado de datos.

El aspecto más importante de éste capítulo es precisamente la etapa en el ciclo de vida orientado a objetos dedicada : al **"Análisis"**, como se comentó en el anterior apartado, para muchos autores se trata de la fase de **"Especificación de Requerimientos"**. Antes de continuar, es conveniente resaltar que la intención particular en éste momento es proporcionar al lector herramientas y experiencia adquiridas en la realización de un proyecto definido **(Diseño de un Juego** en Computadora) empleando la técnica del Diseño Orientado a Objetos, para lo cual resulta muy útil explicar el planteamiento, requerimientos y demás detalles del programa ya definido, no es de extrañarse en el ambiente de la computación que se posea la tendencia a "bautizar" a cada uno de los sistemas que el programador crea a lo largo de la vida, de tal forma que, el que se abordará en éste texto se denominará ***"Citlalli (estrella)"***, el cual pretende permitir al usuario por medio de un computador personal manipular el control de la entrada de datos por medio del teclado, con las flechas del cursor y, a su vez, el ataque que figuras móviles (bichos malos) realizan al jugador. Se considera conveniente mostrar que pueden crearse proyecto del mundo real y que la metodología orientada a objetos, pese a que es nueva en nuestro país, puede ser adaptada fácilmente a nuestras necesidades. Además, es muy deseable conocer ¿ Qué sistema se pretende lograr durante este estudio ?

En cualquier caso, si no contásemos con un programa en particular para ser abordado con ésta metodología, un entendimiento del proceso de diseño nos demuestra que puede ayudarnos a generar una especificación de los requerimientos que podremos usar como una guía. Para este fin, por lo tanto, analicemos qué es necesario en la generación de un buen documento de requerimientos.

Una buena especificación de los requerimientos describe lo que el software puede hacer y lo que no puede hacer. Adicionalmente, una buena especificación de los requerimientos debe proveer alguna idea de la importancia de cada característica relativa a las otras. Finalmente, las dificultades operacionales del mundo real en el producto no tienen que ser descuidadas. Esta información representa la **Entrada** a su diseño.

Ciertamente no todas las especificaciones de requerimientos están bien escritas o completas. En realidad, es excepcional el caso de un documento de requerimientos completo y bien escrito. Sin embargo, si entendemos cuáles entradas son requeridas, podremos ser capaces de determinar en qué momento nos hace falta alguna pieza vital de información, lo cual nos habilita para cuestionar preguntas correctas y, es de esperarse que esos huecos sean llenados de la fuente apropiada o, también podemos dejar pendiente su clarificación, hasta contar con una idea más completa de nuestras dudas.

III.1.4) ¿ Qué se requiere en el Diseño ?

De la entrada que comentamos se producirá un diseño. Su salida (el diseño final) consistirá de :

- **Un sistema de objetos que cumpla los requerimientos,**
- **Una descripción de la conducta pública de esos objetos y,**
- **Los patrones de comunicación entre los objetos.**

De alguna forma, estos aspectos constituyen el centro de atención del tema de tesis elegido.

III.1.5) ¿ En qué consiste la Implantación ?

El diseño orientado a objetos logra que el trabajo de la implantación sea más fácil al mejorar la habilidad del diseñador para comunicarse con el implantador, puesto que se concentra en modelar aspectos del mundo real del sistema, un diseño orientado a objetos es un medio excelente para expresar el intento de diseño. El implantador tiene un mejor entendimiento del problema antes de empezar a codificar. Quizá suene irónico, pero tradicionalmente, en muchos casos no se tiene un conocimiento y un entendimiento completo del sistema que se pretende realizar y, no es de sorprenderse la cantidad de problemas que pueden originarse por estos motivos.

Los Lenguajes Orientados a Objetos proveen un mejor soporte para la implantación del diseño orientado a objetos, pero no son estrictamente necesarios. Las implantaciones escritas en cualquier lenguaje de programación pueden beneficiarse de éste proceso de diseño.

Idealmente, el lenguaje que usemos debe facilitarnos lo que tenemos que hacer y, dificultar las cosas que no tenemos que hacer. Es aún más agradable si el entorno de programación también soporta lo que deseamos llevar a cabo. Pero es realmente extraña la existencia de un mundo ideal y, en el mejor de los casos, podemos usar estos principios de diseño y, alguna disciplina, para beneficiarnos de cualquier diseño de software en cualquier lenguaje de programación.

El diseño además, ya ha tenido cuidado de otro problema de implantación, el cual encapsula las piezas del sistema en unidades que pueden ser implantadas sin considerar las interacciones con el resto del sistema, lo cual nos aporta dos ventajas: primero, ***dividir el trabajo a través de muchas entidades permitiendo a un líder de proyecto distribuir el trabajo de implantución entre muchos programadores de un modo natural;*** segundo, ***si, después de empezar a codificar, una interfaz entre entidades parece incorrecta por alguna razón, el sistema puede ser cambiado justo en ese punto. La implantación cambia en una área, pero otras partes del sistema (y otros miembros del equipo) no son afectados.***

III.1.6) Prueba del sistema

Para el personal encargado de probar el software (usualmente el Ing. en Sistemas), en el diseño orientado a objetos significa que las entidades del sistema pueden ser aisladas y probadas una a la vez. Un error puede ser más fácilmente rastreado en una entidad específica. Las entidades pueden parecer funciones antes de ser insertadas en el resto del sistema. Similarmente, la cuidadosa especificación de las interfaces entre las entidades permite a los verificadores destacar más fácilmente las discrepancias entre la salida de un componente y la entrada requerida por otro. Tal especificación cuidadosa de las interfaces requiere un completo entendimiento de las responsabilidades de cada componente. Huecos en el sistema, lugares donde una responsabilidad en el diseño haya sido omitida, o bien, la responsabilidad se hizo parte de la entidad equivocada, pueden más fácilmente ser distinguidos y corregidos (lo cual, bien podría representar una ventaja más para convencer a nuestro superior inmediato del empleo del paradigma orientado a objetos, pues la rapidez con la que sean detectados los errores se reflejará en los costos que implicará nuestro sistema).

III.1.7) Mantenimiento del sistema

Es muy remoto el caso en el cual los programadores son responsables de mantener su propio código. Lo que sí es muy común es que, la persona responsable de darle mantenimiento a una aplicación es seguramente alguien que nunca lo ha visto antes. ***La comprensión de una aplicación es por consiguiente de suprema importancia para mantener a los usuarios satisfechos.*** Para brindar mantenimiento a un sistema primero necesitamos entenderlo, pero si no podemos entenderlo, las oportunidades para lograrlo son precisas si no deseamos hacer "arreglos" deficientes.

Dado que un buen diseño orientado a objetos usa la encapsulación y el ocultamiento de la información, los patrones de comunicación dentro de la aplicación son rígidamente forzados. Es decir, se puede entender más fácilmente. Primero, es más fácil determinar donde alguna ramificación puede aparecer después de que se ha arreglado el problema. De esta manera, la

vigilancia contra el problema de que un error oculto introduzca otros errores es mucho más estrecha.

III.1.8) Refinamiento y Extensión del Sistema

Efectivamente, si el software ha sido diseñado con rigurosa consistencia, las interfaces pueden ser extendidas y nuevas entidades pueden ser agregadas. Los programadores pueden agregar nuevas entidades que respondan a solicitudes anteriores de maneras apropiadas al nuevo sistema del cual ahora forman parte. Si las interfaces entre entidades han sido rigurosamente controladas, nuevas porciones del sistema pueden ser creadas usando las mismas interfaces, pero haciendo cosas diferentes con ellas.

III.2) La exploración inicial

Inicialmente, el proceso del diseño orientado a objetos es exploratorio. El diseñador busca las clases, obteniendo una variedad de esquemas para obtener la más natural y razonable manera de abstraer el sistema.

Un buen proceso de ***diseño orientado a objetos*** contemplará los siguientes pasos :

1) **Encontrar las clases en el sistema.**

2) **Determinar las operaciones a desarrollar de las cuales cada clase será responsable y qué conocimiento debe manejarse.**

3) **Determinar las maneras en que los objetos colaboran con otros objetos para aliviar sus responsabilidades.**

¿ Qué se produce con ellos ? :

* ***una lista de clases para la aplicación.***

* ***una descripción del conocimiento y operaciones para las cuales***

cada clase es responsable y,

** una descripción de colaboraciones entre las clases.*

III.3) El Análisis detallado

Con ésta información, podremos finalmente, iniciar la fase analítica del proceso de diseño:

Primero, debemos examinar las relaciones de herencia entre las clases. Observar cuidadosamente el funcionamiento de cada clase, es posible que existan clases que compartan ciertas responsabilidades, en ese caso, podemos agrupar esos conjuntos de responsabilidades compartidas dentro de superclases.

Segundo, analizar las colaboraciones entre las clases tratando de agruparlas. Y podrán surgirnos una serie de dudas tales como : ¿ Existen lugares en el sistema donde el tráfico de mensajes es particularmente pesado? ¿ Existen clases que colaboran con todas? ¿ Existen clases que no colaboran con nadie? ¿ Existen grupos de clases que naturalmente pueden agruparse para trabajar más estrechamente? Si se piensa en grupos de clases de esa manera, ¿ Qué cambios implican al modelo de colaboraciones entre ellas y, entre ellas y el resto del sistema?

El análisis de su diseño preliminar consistirá de :

1) Agrupar responsabilidades comunes para construir jerarquías de clases y,

2) Clasificar las colaboraciones entre los objetos.

Se pueden transformar las responsabilidades de cada clase en entidades especificadas completamente, dotando de esta manera un conjunto de especificaciones de clases para ser implantadas.

III.3.1) Subsistemas de clases

Hemos estado hablando acerca de clases como si ellas fueran la única entidad conceptual que compone una aplicación. Pero dependiendo de la complejidad de su diseño, varios niveles de encapsulación pueden ser anidados, uno dentro del otro. Más tarde quizá sea más entendible la idea de lo que esto puede ser, cuando discutamos "grupos de clases que naturalmente pueden agruparse para trabajar más estrechamente".

Las clases son la manera de dividir y estructurar su aplicación para la reutilización. Al iniciar a descomponer la aplicación, podrá inmediatamente identificar las clases. Pero podrá encontrar también otras cosas : piezas que tienen cierta integridad lógica, pero que son en sí mismas divididas en piezas más pequeñas. Nos referiremos a esas piezas como subsistemas. Un **subsistema *es un conjunto de clases que colaboran para cumplir con un conjunto de responsabilidades.*** Y aunque los subsistemas no existan como software ejecutable, ellos son útiles entidades conceptuales. Los subsistemas pueden ser vistos desde dos puntos de vista :

* *Desde el exterior*, los subsistemas pueden ser tratados como una entidad simple. Parecen ser unidades íntegras que colaboran con otras partes de la aplicación para cumplir con sus responsabilidades. Sus colaboradores en la aplicación tratan tales subsistemas como cajas negras. Visto de esta manera, los subsistemas son todavía otro mecanismo de encapsulación.

* *Desde el interior*, los subsistemas se revelan a sí mismos la estructura compleja que tienen. Son programas en miniatura, las clases colaboran con otras para cumplir con distintas responsabilidades que contribuyen al objetivo del subsistema en conjunto : el cumplimiento de sus responsabilidades.

Por ejemplo, hablemos sobre Computadoras, donde se crea la clase llamada **Computadora_Personal (PC), PC** tiene toda la conducta general de un computador, citemos dos subclases concretas **PC_80486 y PC_Pentium.** Ambas se encuentran en un mismo Centro de Cómputo y su uso es indistinto (dado que ambas poseen las mismas características, sin embargo, la segunda es mucho más rápida que la primera), por lo cual se requiere de alguna forma de coordinación entre ellas, dependiendo de la premura con la que se requiera realizar un

trabajo dado. Esta coordinación puede ser suministrada por un controlador general de computadoras (suponiendo que nuestro "Centro de Cómputo" lo tuviera) llamado ***Servidor*** y, varias clases de computadoras comprenden un subsistema facilitando la habilidad de emplear un ordenador. Denominaremos a este subsistema el ***Subsistema de Computación.***

III.3.2) Clientes y Servidores

Anteriormente, hemos indicado que el diseño orientado al objeto busca modelar el mundo en términos de objetos que colaboren para deslindar sus responsabilidades. Estas colaboraciones son vistas como interacciones en un solo sentido: ***"Un objeto solicita un servicio de otro objeto". El objeto que hace la solicitud es el*** **cliente** ***y, el objeto que recibe la solicitud e inmediatamente después proveé el servicio, es el*** **servidor.**

Las maneras en las cuales un cliente dado puede interactuar con un servidor dado son descritas por un **contrato.** ***"Un contrato es una lista de requerimientos que un cliente puede hacer a un servidor".*** Ambos deberán cumplir el contrato: el cliente al hacer sólo esas solicitudes que el contrato especifica y el servidor, al responder apropiadamente a esas solicitudes. La relación es mostrada en la **Fig. 3.3.**

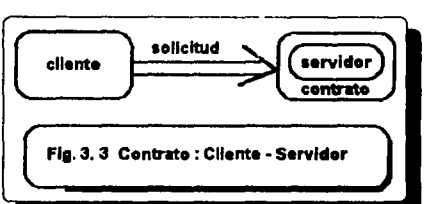


Fig. 3. 3 Contrato : Cliente - Servidor

Modele su diseño como clientes y servidores que colaboran en las maneras especificadas por los contratos. Tal visión conduce a la fase exploratoria inicial del diseño de su sistema. Aquí está el *Cómo.*

III.3.3) Encontrando los objetos

Encontrar los objetos en el sistema a crear es el verdadero problema al modelar. ***Modelar es el proceso por el cual los objetos lógicos en su espacio del problema son mapeados hacia los objetos actuales en su programa.*** Para determinar inteligentemente los objetos que requerirá el modelo de nuestro sistema, primero debemos determinar los objetivos de nuestro diseño :

* **¿ Qué debe realizar el sistema?**
* **¿ Qué funcionamiento está claramente fuera del sistema?**
* **Definición del sistema dibujando líneas de separación y determinar ¿ qué está adentro y qué está afuera ?**

Para entonces podemos preguntarnos razonablemente :

* **¿ Qué objetos son requeridos para modelar al sistema y cumplir con los objetivos.**

Y por cada objetivo debemos procurar preguntarnos :

* **¿ Qué tipo de objetos se necesitan para lograrlo?**

Necesitamos conocer qué objetos conformarán el sistema dado que, desde el punto de vista orientado a objetos, los objetos representan nuestra abstracción primaria. Esta es quizá, la manera más apropiada de iniciar la estructuración del sistema.

III.3.4) Determinando responsabilidades y colaboraciones

Cuando meditamos acerca de las responsabilidades en el sistema, surgen inquietudes como :

* **¿ Qué debe conocer cada objeto para cumplir cada objetivo relacionado con él ?**

* **¿ Qué pasos se deben ejecutar para cumplir cada objetivo del cual se es responsable ?**

Las responsabilidades son la manera en que aportamos el trabajo que el sistema como un todo debe efectuar. El *¿* **Cómo** *cada acción es ejecutada ?* es una pregunta que aplazaremos para después. *¿* **Qué** *acciones deben efectuarse ?* y, *¿* **Cuáles** *objetos las realizarán ?* son preguntas que se deben realizar al principio.

* **¿ Con quién colaborará cada objeto para cumplir con sus responsabilidades?**
* **¿ Qué otros objetos en el sistema tienen algún conocimiento que éste necesita ? o,**
* **¿ Quién sabe desarrollar alguna operación que éste requiera?**
* **¿ Cuál es la naturaleza exacta de las colaboraciones entre los objetos?**

Es realmente deseable que los objetos existentes dentro de un programa colaboren entre sí, en caso contrario, el programa consistirá únicamente de un gran objeto que hará todas las funciones requeridas.

III.3.5) El Contrato Cliente-Servidor

Finalmente tendremos :

* **Una lista de entidades para la aplicación ¡ue tendrán los papeles de clientes y servidores y,**
* **Una lista de colaboraciones entre ellos que sirven como las bases para los contratos.**

Cliente y servidor son funciones. Una función no es un atributo inherente al objeto en sí mismo, un objeto puede tomar una u otra función en cualquier momento. Después de todo, eso sucede con uno mismo, se pueden asumir muchas funciones, por ejemplo : en el curso de un día, un persona podría ser esposo, padre, empleado o estudiante. Similarmente, dentro de una aplicación, un objeto específico puede tomar las funciones de cliente y servidor en relación a diferentes objetos. Mientras el software se ejecuta, un objeto primero puede requerir un servicio de un objeto y, después proveer un servicio diferente para otro.

"Los conceptos de cliente y servidor son simples maneras de seguir la pista al tipo de relación entre dos objetos en un tiempo especifico".

- **Los objetos, entonces, pueden ser clientes y servidores dentro de un sistema.**
- **Las responsabilidades llegan a ser contratos entre ellos.**
- **Las colaboraciones son una manera de determinar cuáles clientes y cuáles servidores se relacionarán por determinados contratos.**

Cada objeto puede tomar parte en muchos diferentes contratos y, será responsable de cumplir con todos los contratos de los cuales es servidor. Las responsabilidades de una clase dada de objeto no deberán ser contradictorias, más bien deben representar una unidad coherente y lógica. Por lo cual, cada objeto deberá compartir o dividir responsabilidades entre varias clases razonablemente.

El contrato entre el cliente y el servidor es definido por el conjunto de solicitudes que un cliente puede hacer a un servidor. Para cada solicitud, un conjunto de entidades servirán como la especificación formal del contrato. Pero, ¿ Cómo el contrato cliente-servidor mejora el mantenimiento del sistema y permite la reutilización ? Veamos el próximo apartado.

III.3.6) Reutilización del software

Ya se ha comentado que los componentes y estructuras básicas de programas son los resultados de la reutilización como abstracción de sus aplicaciones al construírlas. *¿ Cómo usa el contrato cliente-servidor los componentes y las estructuras básicas de programas ?*

Recordemos que el contrato cliente-servidor divide a los objetos en dos categorías : *los **que proveen** servicios (servidores) y, aquellos que **usan** los servicios (clientes).* Los contratos especifican quien colabora con quien y, qué se espera de la colaboración. Si deseamos agregar una nueva entidad en el sistema, el contrato cliente-servidor tiene la mitad del trabajo por nosotros: dado que conocemos la finalidad del contrato así como las responsabilidades que los

otros deben cumplir. Esta es una tarea simplemente comparativa que define la clase que por otra parte del contrato debe corresponder a esos dos imperativos.

Es muy claro en esta formulación que los componentes casi siempre desempeñan la función de servidores. Los componentes por su naturaleza son piezas únicas que se pueden agregar al sistema para cumplir con las responsabilidades específicas (su propósito es proveer una pieza dada de funcionalidad, tal como una *"caja de verificación"* en una interfaz de usuario, o una lista de propiedad de ciertos datos, que otros objetos pueden usar). Raramente los componentes necesitarán requerir servicios de los sistemas donde ellos son agregados con la finalidad de cumplir con sus responsabilidades.

Las estructuras básicas de programa (frameworks), en contraste, usualmente desempeñan ambas funciones. Cuando invocan a componentes existentes o a código específico de la aplicación para que le provean de funciones, claramente actúan como clientes. Frecuentemente son insertadas en las aplicaciones para proveer una porción de la funcionalidad global. En este caso, actúan como servidores.

III.3.7) Encontrando las clases

Se inicia con la especificación de los requerimientos para buscar las clases. Leer completamente la especificación y asegurarse de que se entiende todo antes de iniciar.

Hacer una lista con las frases que contengan sustantivos, cambiando todos los plurales a singulares. En ésta lista aparecerán probablemente tres categorías de elementos: clases obvias, tonterías y, frases de las cuales no se está seguro de su significado. Con seguridad se descartarán las tonterías. De las otras dos categorías, se tomarán selectivamente las frases candidatas a clase.

No todas las clases candidatas se encontrarán en el diseño final. Otras clases aparecerán al final del diseño al hacer una revisión posterior.

Seleccionar las clases en su sistema es el primer paso en la definición de la esencia del problema. Si puede nombrar una abstracción, o mejor aún, encontrar un nombre apropiado en la especificación, se ha encontrado una clase candidata. Si se puede formular una sentencia del propósito de esa clase candidata, las oportunidades son mayores de ser incluidas en el diseño. Enfatice la importancia y, elimine lo irrelevante.

Aparte del sentido común, ¿ Qué otro criterio se puede usar que ayude a decidir cuáles frases con sustantivos son candidatas significativas para clases? Aquí se muestran unas directrices para seleccionar las clases candidatas :

* **Modele físicamente los objetos,** tales como discos o impresoras en una red.

* **Modele entidades conceptuales para formar una abstracción cohesiva,** tal como una ventana en la pantalla, o un archivo.

* **Si más de una palabra es usada para expresar el mismo concepto, seleccione la más significativa en términos del resto del sistema.** Esto es parte de construír un vocabulario común para ser usado por los integrantes del equipo de desarrollo. Seleccione cuidadosamente, la palabra parecerá lo que uno quiere que sea y, tendrá un impacto sutil en el desarrollo del sistema.

* **Sea precavido con el uso de los adjetivos.** Los adjetivos pueden ser usados de diversas maneras. Un adjetivo puede sugerir un tipo diferente de objeto, un uso diferente del mismo objeto o pudiera ser completamente irrelevante. ¿ El objeto representado por un sustantivo se comporta diferente cuando un adjetivo es aplicado a él ? Si el uso de un adjetivo señala que el comportamiento de un objeto es diferente, entonces crear una nueva clase.

* **Sea precavido con las voces pasivas, o de esos temas que no son parte del sistema.** Las sentencias escritas en voz pasiva pueden implicar ciertos temas, aunque ningún sustantivo que aluda al tema aparezca en la página. Rehacer la sentencia en voz activa. ¿ Esto oculta un sujeto que puede ser requerido por un objeto en la aplicación, o este oculta algo sin importancia

? Similarmente, muchas sentencias pueden estar en voz activa, pero sus temas son externos al sistema, por ejemplo "el usuario". ¿ La sentencia sugiere que un objeto puede necesitar ser modelado para actuar a favor del usuario ?

* **Modele categorías de clases,** tales categorías pueden llegar a ser **superclases abstractas.** Después de todo, su diseño es aún fluido, probablemente alterará la clasificación de las clases posteriormente.

* **Sepa modelar la interfaz hacia el mundo exterior,** tal como la interfaz del usuario, o interfaces a otros programas o al sistema operativo, tan completamente como su entendimiento inicial se lo permita. Estas interfaces son probablemente las estructuras más complejas a desarrollar.

* **Modele los valores de atributos de los objetos, pero no los atributos en sí mismos.** Por ejemplo, si se tiene una clase llamada Línea en su diseño, es probable que tenga un atributo llamado longitud, cuyo valor es una unidad de medida tal como un número de punto flotante. Ninguna clase "longitud" necesita ser modelada, pero la clase "flotante" es requerida. La estructura de la clase Línea tendrá una instancia de un "float", la cual será interpretada como longitud. Ningún comportamiento "longitud" necesita ser agregado a la clase float porque la clase tiene un uso específico.

El resultado de este procedimiento es la primera lista tentativa de las clases del programa. Algunas clases serán omitidas, otras serán eliminadas más tarde. Su diseño pasará por varios estados hasta que se complete y, tendrá varias oportunidades para revisiones.

Ciertas generalizaciones pueden ser hechas acerca de las primeras aproximaciones de las clases. Ellas tienden a ser un poco dispersas, a menos que el diseñador domine los conocimientos necesarios. Probablemente se omitirán más clases de las que se eliminarán.

También, todas las partes de la aplicación no son comparablemente detalladas. Diseñará mejor las partes de la aplicación que conozca más. Si usted no entiende algunos subsistemas

dentro de la aplicación, puede realizar esta complejidad sin ser capaz de descomponerlo. Y puede no querer interrumpir el proceso ahora que ha hecho la investigación necesaria. Esto es excelente, porque los subsistemas son encapsulaciones, justo como las clases, se puede tratar a un subsistema que no se comprenda bien como una caja negra. Si comprende bastante bien como para especificar la interfaz (un conjunto inicial de responsabilidades), entonces puede diseñar el resto de la aplicación y retornar a éste posteriormente, o asignando el diseño del subsistema a alguien que lo comprenda mejor.

Para apreciar los puntos recomendados en acción, se presentará una pequeña porción de una especificación de requerimientos.

III.4) Especificación de requerimientos de un Juego de Video

La intención de la creación del juego de video consiste en realizar un sistema bajo la metodología orientada a objetos. El juego trata de un elemento que será capaz de desplazarse a lo largo y ancho de la pantalla, el elemento en cuestión se denominará en lo sucesivo como el* Inocente *del juego y, será representado gráficamente por una carita en la pantalla, el carácter ASCII "001", la movilidad del Inocente tiene la finalidad de que al superponerse sobre otras figuras situadas en el mismo marco de trabajo (pantalla), sean eliminadas y su puntaje se incremente, esas figuras son conocidas como "Pastillas", las pastillas son los rombitos (carácter ASCII "004") de colores que se generarán aleatoriamente en la pantalla, son figuras fijas, su presencia servirá como aliciente para el usuario, por cierto, el usuario se desplazará por medio de las flechas del cursor. Para hacer más interesante el juego, se ha pensado en la creación de la contraparte del Inocente, los llamados "BichosMalos", es decir, los elementos que se comerán al Inocente, los cuales existen en dos categorías, en la de "Malitos" y "Voraces". Los Malitos serán los elementos capaces de comer al Inocente pero cuya movilidad está determinada, su desplazamiento se realizará en un barrido de la pantalla de izquierda a derecha, después de su aparición por primera vez, auxiliados en el uso de una función aleatoria, se desplazarán buscando al Inocente, siempre con tendencia hacia la derecha y, una vez que alcancen el límite derecho de la pantalla, se regenerarán

nuevamente en la última posición referida, pero al margen izquierdo, los Malitos se desplegarán con diversos colores y. si en su recorrido encontraron un Inocente y, su movimiento los llevó a superponerse a él, el Inocente será eliminado y el puntaje de vidas se decrementará en uno. En cambio, los Voraces (figuras móviles) son considerados como los BichosMalos cuyo desplazamiento es guiado por la posición actual del Inocente, es decir, sus coordenadas en la pantalla, de tal forma que, éste tipo de BichoMalo parece ser más inteligente pues aparentemente, cuenta con un "radar" que le permitirá ubicar al Inocente; como el desplazamiento de los Voraces se encuentra totalmente dirigido al Inocente, es más probable que muera con los Voraces que con los Malitos, se deja a juicio del usuario el desplazamiento del Inocente y, dado que la velocidad del Inocente es un poco más rápida que la de los BichosMalos, es posible en buena medida salir librado de morir. Sin embargo, para el caso en que una vida del Inocente se decremente, por conveniencia un nuevo Inocente se regenerará al centro de la pantalla y, aquél Voraz o Voraces existentes volverán a dibujarse a partir de la esquina superior izquierda de la pantalla, con la finalidad de evitar que, si el Inocente murió en el entorno de video cercano a las coordenadas centrales de la pantalla, no vuelva a ocurrir sin darle al menos, una oportunidad al usuario de salvar ésta vida.

Por otro lado, la cantidad de Pastillas existentes variará en función del Nivel en el que se encuentre el usuario, si el Inocente come todas las Pastillas y, todavía tiene al menos 1 vida, entonces el Nivel se incrementará en uno y, al momento de lograrlo, la pantalla se limpiará, incrementando la cantidad de pastillas y de BichosMalos que se dibujarán en la pantalla.

El puntaje sobre el Número de Nivel, Número de Vidas del Inocente, Número de Pastillas y Número de BichosMalos en pantalla será indicado en el primer renglón de la pantalla.

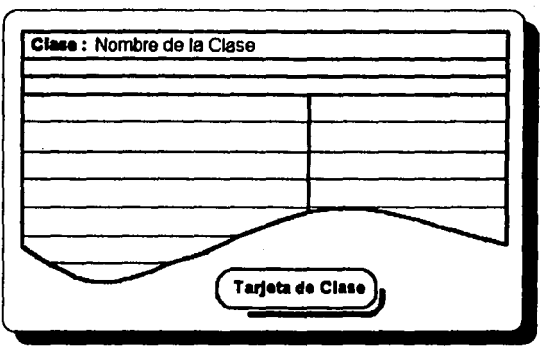
III.5) Conclusiones

Hasta éste momento se ha hecho referencia al proceso de ***Análisis y Especificación de Requerimientos*** para la creación de un juego de video, un adecuado diseño del mismo se podrá apreciar en el Capítulo IV (Diseño Orientado a Objetos), se considera conveniente no comentar en éste capítulo las descripciones tanto de la conducta de los objetos que participarán, así como los patrones de comunicación que presentarían, debido a que ello compete a la etapa de Diseño, por otro lado, antes de profundizar en el ejemplo citado, se ampliará brevemente la descripción de la metodología de Diseño en cuestión, la de **Rebeca Wirfs**.

Sin embargo, la técnica descrita a lo largo de los ***Capítulos III) Análisis Orientado a Objetos*** y, ***Capítulo IV) Diseño Orientado a Objetos***, resulta verdaderamente útil para todo Analista desde el punto de vista teórico. Sin embargo, para el lector interesado en estudiar la parte práctica del proyecto,se le recomienda referirse directamente al ***Capítulo V) Aplicaciones,*** el cual fue contemplado para almacenar todos los listados en código fuente desarrollado en Lenguaje de Programación C++, para la creación de "Citlalli" (el juego).

Recuérdese que, la metodología analizada muestra tan sólo, una alternativa entre las diversas metodologías existentes.

CAPITULO IV
DISEÑO ORIENTADO A OBJETOS



Para éstos momentos, la Tecnología Orien tada a Objetos se encuentra en el mayor interés del Software desarrollado comunitariamente para las notaciones a ser estandariza das a través de varias metodologías.

Grady Booch.

CAPITULO IV
DISEÑO ORIENTADO A OBJETOS

IV.1) ¿ Qué es el Diseño Orientado a Objetos ?

El Diseño Orientado a Objetos es el proceso por el cual los requerimientos de software son transformados en especificaciones detalladas de objetos. Esta especificación incluye una completa descripción de las respectivas funciones y responsabilidades de los objetos y cómo es que ellos se comunican con cada uno de los otros.

IV.2) ¿ Quién Diseña Software ?

Pocas aplicaciones hoy en día son diseñadas por una sola persona. La mayoría de aplicaciones de aún una moderada complejidad son diseñadas por un equipo. Concluímos entonces que el Software es diseñado por un equipo de trabajo donde cada elemento tiene definido un módulo del Sistema sobre el cual desarrollará su labor. Un propósito de la etapa del diseño entonces, puede ser ayudar al equipo a construir un vocabulario común para manejar la aplicación desde su nacimiento. Un vocabulario común puede ser una poderosa herramienta para modelar, porque describir cosas con palabras comunes es compartir ciertas ideas sobre ellas. ¿ De dónde vienen esas ideas comunes ?

El punto de vista orientado al objeto viene del entorno de nuestro mundo, "las cosas como realmente son", de un propósito de ingenua ficción, pero que conduce a conclusiones útiles. El diseño orientado a objetos anima a diseñadores y programadores a empezar pensando acerca de los aspectos del mundo real de un problema tan pronto como sea posible. Por lo tanto, por lo menos uno de los miembros del equipo debe tener pleno conocimiento del problema. Esto es, si escribe software para soportar el cómputo de dinámica de fluídos, ayudaría si fuera un físico, o bien, procurar trabajar estrechamente cerca de uno. Si escribe software que ayude a pronosticar las corrientes del viento en las cercanías de cierto aeropuerto, el conocimiento de meteorología, tanto general como específica a la localidad es una necesidad y un requerimiento

propio del sistema. O si escribe una nueva aplicación de preparación de documentos WYSIWYG, sería buen consejo descubrir lo que escritores, tipografistas y, diseñadores de documentos pueden requerir que la aplicación haga.

Este requisito puede ser evidente, pero a menudo ha sido pasado por alto en la historia del diseño de software, tomemos nota especial de esto. Lo cual resulta particularmente pertinente al diseño orientado a los objetos, porque el punto de vista orientado a los objetos explícitamente indica modelar el mundo "como éste realmente es". Esta es la función del experto que domina el conocimiento : proveer al equipo de la entrada exacta de los detalles y, particularmente abstraer e indicar cuáles detalles son pertinentes.

Sabemos que el mundo es contrario a lo que debe ser y, construimos un mundo idealizado en que cada entidad sabe lo que debe saber y lo que tiene que hacer y, que se comporta como tiene que comportarse. Los cursores parpadean, los menúes se destacan y, las ventanas alegremente despliegan en sí mismas las pantallas de los usuarios.

IV.3) El proceso del diseño

En el presente capítulo se aprenderá ¿ Cómo iniciar con el proceso de diseño ? para diseñar su software. Sin embargo, antes de comenzar, se harán dos observaciones acerca del proceso de diseño en general :

* ***El resultado de este proceso no es el producto final.*** En este sentido, ningún diseño tiene un final. Incluso después de que el software es implantado, verificado y entregado a los usuarios, éste puede pasar de una revisión a otra revisión repetidamente. Ciertamente, antes de que éste sea implantado, se retoman antiguas decisiones, volviendo a trabajar en porciones del software. Aunque por claridad se presenta un proceso lineal, la gente pocas veces trabaja así. *Siéntase libre de explorar, caer en errores y, regresar a su diseño con ideas frescas. Ningún resultado de este proceso necesita ser irreversible.*

* ***El proceso presentado no es rígido, las pautas que se ofrecen no son reglas inflexibles.*** Aunque el diseño del software requiere rigor y disciplina, es también un espacio

para el arte. Siéntase libre de usar su sensibilidad estética como guía. Si no puede justificar completamente una decisión en el diseño, pero siente que está en lo correcto, continúe - al menos hasta donde vea que esto es así. Por experiencia propia, una buena corazonada en el estilo a menudo ayuda a producir diseños elegantes, limpios -diseños que tienen una gran sensibilidad desde el punto de vista de un ingeniero.

IV.3.1) La exploración inicial

Al inicio, el proceso del diseño orientado a objetos es exploratorio. El diseñador busca las clases, obteniendo una variedad de esquemas para obtener la más natural y razonable manera de abstraer el sistema. El proceso del diseño orientado a objetos consiste de los siguientes pasos :

1) ***Encontrar las clases en el sistema.***

2) ***Determinar las operaciones a desarrollar de las cuales cada clase será responsable y, qué conocimiento debe manejarse.***

3) ***Determinar las maneras en que los objetos colaboran con otros objetos para aliviar sus responsabilidades.***

Estos pasos producen :

* ***una lista de clases para la aplicación.***

* ***una descripción del conocimiento y operaciones para las cuales cada clase es responsable y,***

* ***una descripción de colaboraciones entre las clases.***

¿ De dónde vienen los objetos ? La primera tarea en el Diseño Orientado a Objetos es la búsqueda de las clases por las cuáles se compone. Este capítulo proveé un proceso que puede usar para buscar las clases. Se otorgan un número de pautas, herramientas y, ejemplos para ayudarle a descomponer su aplicación en clases.

IV.3.2) Responsabilidades

Los cimientos del diseño comienzan con la identificación de las clases, lo cual se apreció en el anterior capítulo : "Análisis Orientado a Objetos"; ahora continuaremos más firmemente al definir el propósito de cada clase y, el papel que jugará en el sistema. Posteriormente se decidirá ¿ qué conocerán y qué harán las instancias de cada clase ?.

Se aprenderá cómo determinar las responsabilidades de su software como una totalidad y, cómo asignar sus responsabilidades a clases específicas de objetos. Pero, antes de continuar : ¿ Qué son las responsabilidades ?

Las responsabilidades incluyen dos secciones importantes :

* ***el conocimiento que un objeto mantiene y,***

* ***las acciones que un objeto puede realizar.***

Las responsabilidades sugieren en cierto sentido el propósito de un objeto y su lugar en el sistema. Las responsabilidades de un objeto son todas servicios que provee para todos los contratos que mantiene. Cuando asignamos responsabilidades a una clase, declaramos que todas y cada una de las instancias de esa clase tendrán esas responsabilidades, no importa si es sólo una o muchas.

Recordando : "Un objeto juega el papel de un ***servidor*** cuando cumple con una solicitud hecha por otro objeto".

Un ***contrato*** entre dos clases representa una lista de servicios que una instancia de una clase puede solicitar de una instancia de la otra clase. Un servicio puede consistir en la ejecución de alguna acción, o el retorno de alguna información. Todos los servicios registrados en un contrato en particular son responsabilidad del servidor de ese contrato.

IV.4) Identificando Responsabilidades

Las responsabilidades se pueden identificar de varias maneras. Al inicio, dos de las fuentes más productivas son la especificación de los requerimientos y las clases que ya ha identificado.

Las responsabilidades intentan representar solamente los servicios públicamente disponibles. Un objeto puede necesitar saber y hacer otras cosas para cumplir con sus responsabilidades públicas, pero esas cosas pueden ser consideradas como privadas al objeto propio. Se recomienda aplazar la definición de los aspectos privados de una clase porque, en esta etapa de su diseño, usted no tiene bastante información acerca de los detalles de cómo las responsabilidades deben ser asignadas. Por lo tanto, defina responsabilidades a un nivel independiente de la implantación. Recuerde : el contrato entre cliente y servidor no especifica *cómo* se obtienen las cosas, solamente *qué* se obtiene.

Si define las responsabilidades de una clase que sirve como interfaz a algo externo a su aplicación, *puede* saber precisamente cómo debe trabajar esa interfaz. Pero en general, en esta fase exploratoria en el diseño, no es necesario. Las responsabilidades son todavía generales -no necesita consideraciones específicas-, tal como los nombres de los mensajes. En esta etapa, tiene que expresar las responsabilidades en términos generales y tratar de mantener todas las responsabilidades de una clase al mismo nivel conceptual.

IV.4.1) La especificación de los Requerimientos

De nuevo, lea la especificación de requerimientos cuidadosamente (la especificación señalada en el epígrafe III.4). Aunque esta es solamente una especificación parcial, es bastante completa. Iniciemos el proceso de diseño ***seleccionando las frases con sustantivos.*** Sin embargo, no se categorizarán inmediatamente. En lugar de eso, para el propósito de mostrar el proceso, se comenzará con *todas* las frases con sustantivos y, se discutirá el razonamiento obtenido para mantenerlas o descartarlas una a la vez.

La lista de sustantivos derivada de la Especificación de Requerimientos es :

juego de video	***juego***
sistema	***elemento***
Inocente	***figuras***
pantalla	***puntaje***
Pastillas	***figuras fijas***
figuras móviles	***presencia***
Malitos	***colores***
Voraces	***barrido de pantalla***
aparición	***función aleatoria***
tendencia	***límite derecho***
margen izquierdo	***puntaje de vidas***
centro	***BichosMalos***
radar	***esquina superior izquierda***
entorno de video	***coordenadas centrales***
oportunidad	***cantidad de Pastillas***
Nivel	***incrementando***
Número de Nivel	***Número de Vidas del Inocente***
Número de Pastillas	***Número de BichosMalos***
primer renglón de la pantalla	

IV.4.2) Modelar los objetos físicos

Uno de los objetos físicos mencionados son las ***teclas del cursor (hacia arriba, hacia abajo, derecha e izquierda).*** Dependiendo del tipo de entorno de ventana para el cual están siendo empleadas, su estado probablemente es un atributo de algunos dispositivos o un evento. En ningún caso se modelarán las teclas del cursor directamente.

En segundo término tenemos, como objetos físicos, a las figuras que se ven involucradas en el juego de video, es decir, las ***Pastillas***, los ***BichosMalos*** y, por supuesto, el ***Inocente.***

juego de video	*juego*
sistema	*elemento*
Inocente	*figuras*
pantalla	*puntaje*
Pastillas	*figuras fijas*
figuras móviles	*presencia*
Malitos	*colores*
Voraces	*barrido de pantalla*
aparición	*función aleatoria*
tendencia	*límite derecho*
margen izquierdo	*puntaje de vidas*
centro	*BichosMalos*
radar	*esquina superior izquierda*
entorno de video	*coordenadas centrales*
oportunidad	*cantidad de Pastillas*
Nivel	*incrementando*
Número de Nivel	*Número de Vidas del Inocente*
Número de Pastillas	*Número de BichosMalos*
primer renglón de la pantalla	

IV.4.2) Modelar los objetos físicos

Uno de los objetos físicos mencionados son las ***teclas del cursor (hacia arriba, hacia abajo, derecha e izquierda).*** Dependiendo del tipo de entorno de ventana para el cual están siendo empleadas, su estado probablemente es un atributo de algunos dispositivos o un evento. En ningún caso se modelarán las teclas del cursor directamente.

En segundo término tenemos, como objetos físicos, a las figuras que se ven involucradas en el juego de video, es decir, las ***Pastillas***, los ***BichosMalos*** y, por supuesto, el ***Inocente.***

IV.4.3) Modelar entidades conceptuales

Parece probable que cosas como ***Inocente, BichosMalos, Pastillas, Malitos y Voraces,*** serán clases útiles a modelar para una aplicación de un Juego de video que las manipule. El juego parece ser otra probable entidad conceptual, como lo son las ***coordenadas, el nivel y, el puntaje.***

Similarmente el objeto ***nivel*** en la interfaz del usuario parece ser un buen candidato. Según parece, al incrementar el nivel, el número de las figuras participantes también se incrementará proporcionalmente.

Se deja a gusto del lector, la decisión de clasificar las figuras existentes en el juego, como ***Figuras Móviles y/o Fijas***, pensando en la posibilidad de extender el juego en versiones futuras. Esta subclasificación parece ser un candidato razonable para una clase, ya que el usuario podría manipularla. Debido a que nos encontramos en la etapa de análisis, la conservaremos en nuestra lista de clases potenciales.

Existen otros elementos en la lista previa de objetos que, representan entidades conceptuales un poco menos claras. Tal es el caso de : ***pantalla, puntaje, izquierda, derecha, radar, límite derecho, margen izquierdo, etc.*** Por lo que, inevitablemente surge la pregunta : ¿ Estas clases son útiles o no ? Para responder de manera apropiada, apliquemos otros principios.

IV.4.4) Elija una palabra para un concepto

Este punto es importante, especialmente porque nos ayuda a eliminar la redundancia que pudiese existir en nuestra lista inicial. Es decir, ¿ Existen algunas entradas en la lista anterior donde parecen repetirse ideas encontradas anteriormente ? Si es así, repasemos el listado de objetos, donde dos elementos : ***juego y juego de video,*** seguidos más adelante por ***sistema.*** Por supuesto que más de un concepto no se maneja aquí; el segundo es aparentemente una amplificación del primero y, en el último caso, una abreviatura. No se considerarán ***sistema*** y ***juego***, entonces se hará referencia a la aplicación como un ***Juego de Video.***

También se tienen a ***elemento*** y ***figuras***, ambos se refieren al mismo concepto. El término ***figuras*** es un nombre más descriptivo para entidades en un dibujo que, en éste caso, ***elemento***. Por tanto, en lo sucesivo, nos referiremos a un objeto del juego como una figura y, se dejará a elemento fuera de consideración.

Otras dos posibilidades son las entradas : ***coordenadas centrales y centro***, surgiendo la inquietud ¿ Son éstas realmente dos ideas separadas ? Recuérdese que tenemos el concepto de ***entorno de video***, entonces, eliminemos ***centro*** en favor de ***coordenadas centrales***, considéresc que, siempre es posible regresar y retomar de nuevo alguna decisión que provoque desaliento para nuevamente ponerse a consideración. De manera similar, agréguense a la palabra ***BichosMalos*** los elementos ***Malitos y Voraces***.

IV.4.5) Ser cuidadoso con los adjetivos

Tenemos muchas frases de sustantivos-adjetivos en la lista anterior. ¿ Cuántos de éstos en realidad significan un tipo diferente de objeto ?

Existe un sustantivo ***nivel*** y una frase con nivel : ***Número de Nivel***, se eliminará el término **nivel** y debido a que **Número de Nivel** es mucho más explícita, es la que se conservará.

Por otro lado, el sustantivo ***puntaje***, nos va a retornar el ***Número de nivel, Número de vidas, Número de pastillas, Número de BichosMalos***. Parece claro que se necesita una clase para controlar el puntaje, ya que la aplicación los usa de diferentes maneras. Naturalmente a esta clase se le denominará ***Clase Controladora***.

IV.4.6) Ser cuidadoso con las frases que se omiten o confunden los temas

La especificación incluye gran cantidad de frases en voz pasiva. Muchas de éstas no presentan problemas, ya que la información omitida puede ser encontrada en frases anteriores. Sin embargo, tiene el potencial de implicar nueva información.

" El juego trata de un elemento que será capaz de desplazarse a lo largo y ancho de la pantalla, el elemento en cuestión se denominará en lo sucesivo como el Inocente del juego y, será representado gráficamente por una carita en la pantalla, el carácter ASCII 001". Lo que falta a esta frase es la respuesta a las preguntas :

- ¿ Quién es responsable de estas indicaciones visuales ?
- ¿ Quién realiza el desplegado ?
- ¿ Ocurren estas acciones (desplegado y movimiento) en la pantalla del monitor ?
- ¿ Algún controlador global indica la aparición de las figuras a desplegar ?
- ¿ Son varios los objetos que colaboran para lograr éste efecto ?

Aunque no resulte muy claro en este momento, supongamos que esto es hecho por el ***Juego de Video*** y, que no necesitamos crear una clase nueva.

Un número de estas frases presume que el usuario es el sujeto. Por ejemplo, la especificación incluye lo siguiente : " *... el usuario se desplazará por medio de las flechas del cursor*".

IV.4.7) Modele categorías

Consideraremos aquí, lo que mencionamos en el apartado anterior, se hace gran énfasis en la necesidad de contar con una clase que sea capaz de controlar los atributos y movimientos de las figuras en la pantalla, por tanto, se habla de una clase controladora, cuyas funciones serán: definir por Nivel, el Número de obstáculos (si existen), Número de BichosMalos, Número de objetivos, Número de vidas, Velocidad de desplazamiento de los BichosMalos así como del Inocente.

IV.4.8) Modele interfaces del sistema

La lista de frases con sustantivos incluye pocas posibilidades de que algo no sea del interior del sistema. Probablemente no necesitamos modelar al usuario, por ejemplo, aunque sin duda tenemos que crear una interfaz de usuario.

Otra posibilidad es el cursor. Presumiblemente, esta aplicación corre en un sistema usando las teclas del cursor y, el sistema operativo maneja el movimiento del cursor automáticamente. En este caso, no hay necesidad de modelar el cursor.

IV.4.9) Modele valores de atributos, no los atributos en sí mismos

La lista de frases con sustantivos incluyen frases sospechosas, tales como margen izquierdo, esquina superior izquierda. El ***margen izquierdo*** y la ***esquina superior izquierda*** son atributos propios de la pantalla; sus valores son posiciones que pueden cambiar, pero su comportamiento no cambia en ningún sentido. El mantener los valores para cada uno de ellos puede ser responsabilidad exclusiva de la clase controladora; asumiendo que varias clases de posiciones ya existen, una clase nueva para estos atributos parece no agregar nada al sistema.

El proceso de distinguir los valores de atributos de los atributos en sí mismos no es siempre una labor sencilla. El lenguaje puede ser bastante confuso y, a veces se usa la misma palabra tanto para el atributo como para su valor. Por ejemplo, la ***velocidad (atributo)*** de un **Malito** es probablemente un **Float** (valor), pero el ***color*** (atributo) de un Malito es probablemente un Color (valor). Es recomendable considerar esto, al intentar determinar las clases que requiere el sistema.

IV.4.10) Resumen de Clases

Un análisis preliminar, entonces, produce los siguientes candidatos a clases. Puede ver que tenemos una lista aislada considerable. Estos candidatos son todavía preliminares, e indudablemente otras clases serán evidentes cuando comencemos a diseñar el sistema con más detalle. Por ahora, sin embargo, esto es lo que tenemos :

Inocente	***BichosMalos***	***coordenadas centrales***
Malitos	***Voraces***	***Figuras Móviles***
Figuras Fijas	***Número de Nivel***	***figuras***
Juego de Video	***Pastillas***	***Clase Controladora***

IV.4.11) Registre sus Clases Candidatas

Cuando ha identificado las clases candidato, escriba sus nombres en tarjetas de índice, una clase por tarjeta, como se muestra en la **Fig. 4.1**. En el reverso de la tarjeta, tiene que escribir una breve descripción del propósito completo de la clase. Tal declaración puede ayudarle a clarificar su motivación para crear esta clase y, puede servir más tarde como el núcleo de la documentación de la clase.

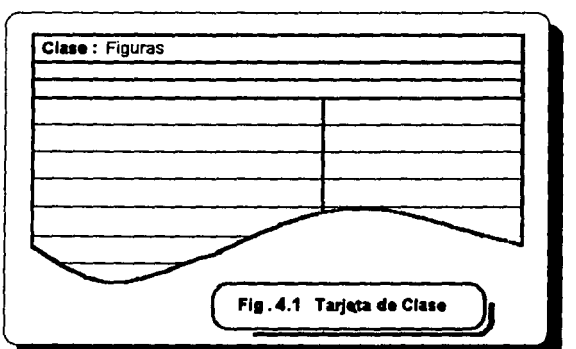


Fig . 4.1 Tarjeta de Clase

Se ha comprobado que dichas tarjetas de índice son muy útiles porque son compactas, fáciles de manipular y, fáciles de modificar o descartar. Porque usted no las creó, no las sentirá valiosas. Si la clase parece ser falsa, puede deshechar la tarjeta.

Debido a que son pequeñas y, no están en la pantalla de la computadora, puede fácilmente arreglarlas sobre una mesa y tener muchas a la vista (o quizá todas ellas) al mismo tiempo. Puede tomarlas, reorganizarlas y, colocarlas en un nuevo orden para ampliar una idea reciente. Puede dejar algunas en blanco sobre la mesa representando clases omitidas. Si descubre que erróneamente ha descartado una tarjeta de clase, es sencillo recobrarla, o hacer una nueva.

Sin embargo, es inútil insistir en tarjetas. Si trabaja mejor con hojas blancas tamaño carta u hojas amarillas, o papel gráfico, úselos para trabajar. Si desea trabajar en la computadora, entonces hágalo. En esta etapa, explorar las posibilidades de una variedad de modelos debe ser fácil y divertido y, puede descubrir problemas ocultos en un principio, en la etapa costo-efectividad. La herramienta que use tiene que realzar, no impedir, el proceso de diseño.

IV.5) Encontrando Clases Abstractas

Preparado con una lista apropiada de candidatos a clase para una aplicación, ahora debe evitar cuidar lo pequeño. Reexaminando nuestra lista identificaremos como son posibles muchas clases abstractas. Identifique estas clases para ayudar a identificar la estructura del software (se harán superclases de otras, las clases concretas) y, para ayudar a identificar las clases que se han pasado por alto.

Una clase abstracta surge de un conjunto de clases que comparten un atributo útil. Un atributo útil es uno que implica un comportamiento compartido para clases que tienen ese atributo. Si el comportamiento es compartido por varias clases, tiene que diseñar una superclase abstracta que retenga ese comportamiento compartido en un lugar. Las subclases pueden entonces heredar ese comportamiento. La meta es encontrar tantas superclases abstractas como sea posible. Busque atributos comunes en las clases, como se describió en los requerimientos. Puede revisar su lista si parece que algunos atributos no lo son, después de todo, implica comportamiento compartido.

IV.6) Grupos de Clases

Identifique candidatos para superclases abstractas al agrupar clases relacionadas. Una clase puede aparecer en más de un grupo. Cuando se ha identificado un grupo, nombre la superclase con eso que siente representa al grupo. Haga el nombre con un sustantivo singular o una frase con ese sustantivo. Por ejemplo, suponer que ha identificado un grupo de elementos :

Su atributo compartido es el que las clases son los elementos a manipular en el juego. Es posible, por lo tanto nombrar a este grupo **"Figuras"**.

Algunos de estos grupos ya pueden ser identificados por usted. Las categorías de clases que identificamos anteriormente usualmente se representarán justamente como un grupo. La mayoría de las clases creadas representan categorías y por lo tanto serán superclases abstractas de las clases en la categoría.

IV.7) Registro de Superclases

Cuando las superclases han sido identificadas, regístrelas en las tarjetas de índice, una clase por tarjeta. También escriba sus subclases en las líneas debajo del nombre. Tome sus tarjetas de clase y, registre las superclases y las subclases, si ya conoce alguna, en las líneas debajo de sus nombres. Las tarjetas de clase ahora se mirarán como la mostrada en la **Fig. 4.2.**

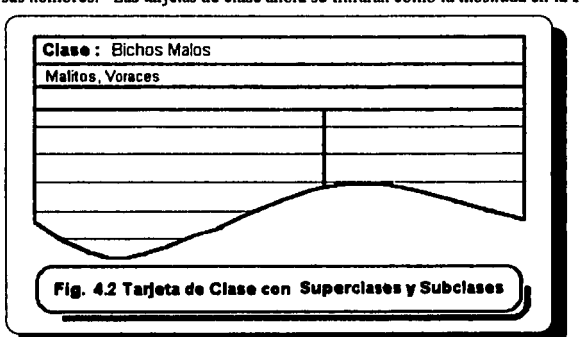


Fig. 4.2 Tarjeta de Clase con Superclases y Subclases

Si tiene problemas al nombrar un grupo que ha identificado :

- Ennumere los atributos compartidos por los elementos de la categoría y,

derive el nombre de esos atributos o,

* Divida los elementos en categorías más claramente definidas y pequeñas.

Si todavía no puede nombrarla, descarte el grupo y busque otras.

IV.8) Ejemplos de Atributos

Identificar los atributos de esta manera alienta a formar patrones útiles de comportamiento compartido con sus clases, pone la base para un uso productivo de la herencia y la reutilización del código. Muchos atributos útiles son específicos a la aplicación. Algunos atributos comunes por los cuales las clases pueden ser agrupados son :

* ***Físico vs Conceptual :*** ***Videojuego, Sistema***
* ***Activo vs Pasivo :*** ***Inocente, Pastillas***
* ***Temporal vs Permanente :*** ***Nivel, Puntaje***
* ***Genérico vs Específico :*** ***BichosMalos, Malitos y Voraces***
* ***Compartido vs No compartido :*** ***Pantalla, Coordenadas Centrales***

Descarte atributos si no lo ayudan a distinguir entre las clases.

IV.9) Identificando Clases Omitidas

Una vez que las ha identificado, extender las categorías puede ser una manera útil de identificar clases omitidas.

Las clases se pueden omitir porque no son importantes, o porque la especificación era imprecisa. Por ejemplo, el **Juego de Video** olvida una clase importante porque la especificación no era explícita.

La especificación declara que las figuras pueden ser creadas. Esto significa que los elementos del juego pueden ser modificados, agregados y, eliminados del juego. El usuario

puede cambiar de idea y puede desear eliminar temporalmente una figura del juego para colocarlo en otro sitio en el juego.

El encontrar clases omitidas no es fácil. Aún para el más experimentado diseñador es lo más fácil que le puede suceder. Para permitirle ganar alguna experiencia inmediatamente, se propone al lector realice ahora una especificación de requerimientos, por ejemplo, la de un cajero automático. Lea la especificación que realizó y, ejercite los principios que ha aprendido en este capítulo al encontrar las clases de objetos que necesitará para modelar el sistema especificado.

IV.10) Asignando Responsabilidades

También use el trabajo que tiene hasta ahora cuando identificó las clases. El hecho de identificar una clase indica que vio una necesidad y que satisfacerla por lo menos tiene una responsabilidad. El nombre que elija para esa clase sugerirá esa responsabilidad y, posiblemente otras. Esos nombres representan papeles dentro de su sistema, implican responsabilidades para los objetos que tienen que cumplir esos papeles. La declaración de propósitos para cada clase puede también acarrear responsabilidades adicionales. ¿ Qué conocimiento y acciones se implican para este propósito ? ¿ Qué responsabilidades tiene cada clase para manejar estos atributos ? Comparando y contrastando los papeles de clases puede también generar nuevas responsabilidades.

La especificación proveyó las semillas para su modelo del sistema. Ahora sus clases crecen de estas semillas. Uso que se ha desarrollado para ayudarle a razonar más sobre las responsabilidades, tanto evidentes como implícitas de sus objetos.

Liste o ***destaque todos los verbos*** y, usando su juicio determine cuáles de estos claramente representan acciones que algunos objetos pueden realizar dentro del sistema. De igual manera, la información mencionada por todas partes, anótela. La información que algún objeto dentro el sistema debe mantener y manipular también representa **responsabilidad**.

La lista de **verbos** derivada de la Especificación de Requerimientos es :

intentar	*creación*
consiste	*realizar*
trata	*desplazarse*
denominará	*representado*
movilidad	*superponerse (sobreponerse)*
eliminadas	*incremente*
conocidas	*generarán*
servirá	*hacer*
pensado	*comerán*
comer	*barrido*
aparición	*tendencia*
alcancen	*regenerarán*
cuenta	*permitirá*
ubicar	*dirigido*
muera	*deja*
juicio	*librada*
morir	*existentes*
volverán	*dibujarse*
finalidad	*evitar*
ocurrir	*salvar*
variará	*encuentre*
limpiará	*indicado*

Una de las maneras más útiles para encontrar responsabilidades es hacer un recorrido por el sistema en su totalidad. Imagine cómo el sistema será invocado, atravesando por una variedad de escenarios usando tantas capacidades del sistema como sea posible. Busque lugares donde algo tiene que ocurrir como resultado de una entrada al sistema ¿ Qué nuevas responsabilidades son implícitas para esta necesidad ?

Asigne cada responsabilidad que ha identificado a la clase o clases a la(s) que lógicamente pertenece. La fuente más clara de esta información es, por supuesto, la

especificación de requerimientos. Para la mayoría de los indicios útiles, examinar el contexto en el cual se identificó la responsabilidad. Al repasar la especificación, puede encontrar varias responsabilidades que claramente no pertenecen a ninguna clase. Para ayudarle a asignar las responsabilidades, se presentan las siguientes pautas.

- ***Distribuya uniformemente la inteligencia del sistema.***
- ***Plantee responsabilidades tan generales como sea posible.***
- ***Mantenga el comportamiento con la información relacionada, si se da el caso.***
- ***Distribuya responsabilidades entre objetos relacionados.***

IV.10.1) Distribuya uniformemente la Inteligencia del Sistema

Un sistema puede ser visto como algo que tiene cierta cantidad de inteligencia, tal inteligencia comienza con lo que el sistema sabe, las acciones que puede desarrollar y, el impacto de éste sobre otros sistemas (incluyendo a los usuarios) con los cuales interactúa. Dentro de cualquier sistema, algunas clases pueden parecer relativamente "inteligentes", mientras otras lo parecen menos. Una clase incorpora más o menos inteligencia de acuerdo a lo que sabe o puede hacer y, cuántos otros objetos puedan afectarla. Por ejemplo, colecciones de objetos tales como conjuntos o arreglos no son usualmente vistos como particularmente inteligentes : pueden almacenar y recobrar información, pero tienen relativamente poco impacto en otros objetos dentro del sistema, aún incluso en los objetos que almacenan.

En el Videojuego, se citan a elementos del tipo **BichosMalos**, los cuales parecen objetos inteligentes. Un **BichoMalo** sabe cómo dibujarse a sí mismo. Presumiblemente también puede moverse y, dimensionarse, así mismo es posible que otros objetos sepan cuando ha cambiado, llevando consigo una gran parte de la inteligencia del sistema.

Una clase aún más inteligente del mismo ejemplo, es la clase controladora, la Clase **Jueguito**, que es la clase responsable de interpretar las acciones del usuario, así como de los objetos que interactúan con él, además es responsable de dibujar en la pantalla los cambios que se generaron por algún elemento.

Cuando diseñe clases, tiene que decidir cómo distribuir la inteligencia entre ellas. ¿Cuáles clases deben incorporar cuáles aspectos de la inteligencia completa del sistema? Claramente, puede tomar dos caminos contrarios acerca de este problema.

Una manera es minimizar el número de clases inteligentes. En el caso más extremo, solamente un objeto debe saber tanto como sea posible del sistema. Todos los otros objetos serían desprovistos de tanta inteligencia como fuera posible. Este proceder tiene el efecto de centralizar control con el objeto inteligente. En realidad, tiene algo de un aspecto procedimental: el objeto inteligente servirá para el mismo propósito que el módulo de control principal de un programa procedural mientras los otros objetos se comportarán como estructuras tradicionales de datos.

Una ventaja de este proceder es que, al tomar el punto de vista del objeto inteligente, es relativamente fácil entender el flujo de control dentro de la aplicación. Sin embargo, esta táctica tiene varias desventajas. El concentrar demasiado el comportamiento del sistema dentro de una clase tiene el efecto de complicar la conducta del sistema; puede ser más difícil modificar el sistema cuando más adelante se requiera. También, entre menos inteligencia tenga un objeto y, menos comportamiento tenga; se usarán relativamente más clases no inteligentes para implantar el mismo sistema y, por lo tanto más esfuerzo.

La otra táctica es distribuir la inteligencia del sistema tan equitativamente como sea posible, de esta manera al diseñar todas las clases serán igualmente inteligentes. Mientras esta táctica tiene más del espíritu de un sistema orientado a objetos, una distribución perfecta es usualmente un ideal imposible. Al asignar sus papeles dentro del sistema, unos objetos necesitan más inteligencia que otros, justo como el papel de un salvavidas llamado para ejercer más juicio que el papel de una persona que sólo toma el sol.

Mientras esto requiere más estudio antes de obtener un entendimiento completo del sistema y, poder ver cómo el control y la información fluyen dentro, este proceder tiene varias ventajas. El distribuir la inteligencia dentro de su sistema entre una variedad de objetos permite que cada objeto sepa relativamente pocas cosas. Esto produce un sistema más flexible, más fácil

de modificar. También requerirá diseñar relativamente pocas clases para implantar un sistema con funcionalidad comparable.

Puede verificar para una distribución uniforme de inteligencia al comparar las responsabilidades para cada clase. Si una clase particular parece tener una lista indebidamente larga, deténgase y examínela. Verificar que cada una de las responsabilidades sean expresadas al mismo nivel de detalle; si es necesario, reescríbalas para que así sea. Si la lista todavía parece larga, quizás esa clase tiene demasiadas responsabilidades y, deberá ser dividida en dos o más clases.

IV.10.2) Plantee responsabilidades tan generales como sea posible

Cada tipo de elemento del Videojuego sabe cómo dibujarse a sí mismo. Es posible afirmar que un **elemento de Pastillas** sabe cómo dibujar una "Criptonita", inclusive el mismo **Inocente** (Yo), sabe cómo dibujarse. Pero en su lugar, es preferible declarar las responsabilidades de una manera general: cada tipo de **elemento del Videojuego** sabe cómo dibujarse a sí mismo. Cada subclase del **Videojuego** sabe cómo dibujarse a sí misma a su particular manera. Si declaramos responsabilidades en términos generales, podemos encontrar responsabilidades comunes compartidas entre clases más fácilmente. Y podemos encontrar que una clase descrita de esta manera puede ser más útil de lo que inicialmente se especificó.

IV.10.3) Mantenga el comportamiento con la información relacionada

En general, la responsabilidad para mantener información específica no debe ser compartida. Compartir información significa una duplicación que podría conducir a inconsistencia. Parte de hacer el software más fácil de mantener es minimizando tal duplicación.

Por ejemplo, cada **Figura** sabe qué elementos de dibujo desplegar, pero cada **elemento dibujado** sabe la localización en la cual es desplegado. Si ese **Objeto** y la **Figura** que lo contiene son mantenidos informados de la localización, un mensaje para actualizar la

información tendría que pasar fiablemente de uno al otro cuando un usuario moviera la **Figura** en cuestión.

Si más de un objeto debe conocer la misma información para ejecutar una acción, existen tres posibles soluciones al dilema.

* Un objeto nuevo (posiblemente una instancia de una nueva clase) creado esencialmente para el papel puede ser la fuente exclusiva de la información. Si otros objetos requieren la información pueden ser forzados a accesarla al mandar un mensaje a la única instancia informada.

* En cambio, puede ser que el comportamiento que requiere la información es solamente una pequeña parte del comportamiento de uno de los objetos. En ese caso, puede ser menos drástico simplemente al reasignar la responsabilidad para ese comportamiento del objeto del cual la principal responsabilidad es mantener la información. Otros objetos pueden pedir esta información al mandar mensajes al objeto con el conocimiento.

* O puede ser apropiado colapsar los diferentes objetos que requieren la misma información en un sólo objeto (que pueda involucrar clases colapsadas también). Estos medios encapsulan el comportamiento que requiere la información con la información en sí misma dentro de un objeto y, obliga a la distinción entre los objetos colapsados.

IV.10.4) Distribuya responsabilidades

Ocasionalmente, puede descubrir que una cierta responsabilidad parece ser varias responsabilidades en una, o una responsabilidad compuesta, es mejor dividir o compartirlas entre dos o más objetos.

Por ejemplo, el videojuego puede requerir que el estado actual del puntaje sea desplegado en todo momento. Esto implica una responsabilidad para desplegar el puntaje que se está originando. Pero ¿A cuál(es) clase(s) debe ser asignada esta responsabilidad?

Se prefiere visualizar esta responsabilidad como una que se completa por la cooperación de varios objetos :

- El **Juguete** sabe cuando el dibujo ha cambiado y necesita ser redibujado, sabe cuáles elementos de dibujo son desplegados.
- Cada **Figura** sabe cómo y dónde su representación visual debe ser dibujada.

Así, un mayor entendimiento de cómo el modelo debe funcionar nos conduce a determinar que la responsabilidad de despliegue de las figuras necesita ser compartida. Tenemos por lo tanto que dividir las responsabilidades en varias más pequeñas, más específicas y, asignar cada una de estas responsabilidades más pequeñas a la clase más apropiada.

IV.11) Exámen de las Relaciones entre Clases

Responsabilidades adicionales pueden ser identificadas al examinar las relaciones entre clases. Una vez que tenga una buena idea de las clases que componen su sistema, examine las relaciones entre ellas para ganar experiencia al asignar sus responsabilidades. Tres relaciones son particularmente útiles en esta consideración :

- la relación "es-tipo-de",
- la relación "es-análogo-a" y,
- la relación "es-parte-de".

Por supuesto, no todas estas relaciones serán útiles para cada diseño. En adición, habrá sin duda ciertas relaciones específicas a la aplicación que tienen que ser identificadas y analizadas. Sin embargo, una discusión de estas relaciones genéricas puede ayuda a clarificar el proceso de identificar y asignar tipos particulares de responsabilidades.

IV.11.1) La relación "Es-tipo-de"

Cuando una clase parece ser del tipo de otra clase, es frecuentemente una señal de relación subclase-superclase entre ellas. Tal relación es a menudo dejada fuera por el hecho de que dos clases comparten un atributo.

Hasta este momento ya se ha comentado acerca de la categorización de las clases y, la designación de los atributos a usar para cada categorización. Cada atributo identificado significa una responsabilidad específica, o un conjunto de responsabilidades. Los nombres de estos atributos en sí pueden generar más responsabilidades y, ayuda a asignar esas ya identificadas.

Cuando categoriza clases según atributos específicos, usted está implicando que algún comportamiento o información es común a todas esas clases. Todas esas clases que son de ***un tipo de*** alguna otra clase comparten algunas responsabilidades. ¿ Qué son estas responsabilidades ?

Encontrar tales relaciones nos permite asignar responsabilidades a las superclases. Cuando una responsabilidad implicada por un atributo es identificada, asigna ésta a la superclase de las clases que comparten el atributo.

¿ Por qué tiene que asignar una responsabilidad a una superclase ? Como una asignación puede ser bastante ventajosa, porque de esta manera la responsabilidad es asignada una sola vez y, heredada por todas las subclases. Así, de un golpe ha tomado cuidado de un número potencialmente grande de clases.

IV.11.2) La relación "Es-parte-de"

Cuando una clase parece ser parte de otra clase, no se implica la herencia del comportamiento. Justo porque un objeto es compuesto de muchas instancias de otras clases no significa que se comporte de alguna manera como esas partes. En realidad, usualmente se comportan bastante diferente.

Una clara distinción entre la totalidad y una parte puede ayudarnos a determinar donde las responsabilidades para cierta conducta deben ser, principalmente el límite de alcance de las posibilidades. La relación sin duda ayudará a identificar responsabilidades para mantener información también. Un objeto compuesto de otros objetos debe conocer siempre más estrechamente esas partes.

Las superclases pueden parecer un poco confusas al principio del proceso de diseño, e inicialmente puede tener algunas dificultades para identificar responsabilidades para una superclase en particular. Si esto pasa, no la descarte inmediatamente. En lugar de eso, póngala a un lado para más tarde. Después de que usted haya completado la fase exploratoria de su diseño y, haya empezado el análisis más cuidadoso, puede aún encontrar que después de todo esa superclase es útil.

IV.11.3) La relación "Es-análogo-a"

Puede examinar las relaciones entre las clases para encontrar las analogías también. Si la clase X tiene una relación con otra parte del sistema que es análoga a la soportada por la clase Y, entonces quizás las clases X e Y comparten las mismas (o análogas) responsabilidades. Quizás son ambas tipos de otras cosa : una superclase aún-no-identificada.

Cuando varias clases parecen ser análogas, es frecuentemente una inidicación de que comparten una superclase común. Tal relación es a menudo sugerida por el hecho de que las clases tienen algunas de las mismas responsabilidades. Encontrar tal relación puede permitirnos identificar una superclase olvidada y, asignarle las responsabilidades comunes a la superclase. Tiene que asignar una responsabilidad a una superclase abstracta, sin embargo, solamente si todas las subclases de la superclase abstracta comparten la responsabilidad.

IV.11.4) Clases Omitidas

Puede ser bueno para un diseño el agregar una nueva clase que encapsule un conjunto relacionado de responsabilidades no asignadas. Por ejemplo, después de nuestro análisis inicial de clases en el videojuego, determinamos que un usuario puede desear elegir que su representación en pantalla sea variable y, especificar que esos elementos sean manipulados como un grupo. La especificación del videojuego no incluyó una descripción detallada de la interacción entre un grupo de elementos y las operaciones del juego. En realidad, proveyó solamente una descripción de alto-nivel.

Manipular elementos como un grupo implica la responsabilidad de mantener la información específica de qué elementos han sido agrupados.

IV.11.5) Asignamiento Arbitrario

Los requerimientos del videojuego declaran que exactamente una figura es activa a un solo tiempo (aunque debido a la rapidez de ejecución, pareciera que todas las figuras se desplazan simultáneamente en pantalla). Esto significa una responsabilidad para algún objeto que sabrá que elementos se encuentran activos.

La cuestión es, cuando seleccione de entre varias clases para asignar una responsabilidad, pregúntese a sí mismo : " ¿ Cuáles son todas las posibi'idades ? Si elijo esta posibilidad, ¿ Qué implica la opción para la funcionalidad del sistema ? ¿ Para el aspecto y funcionamiento del sistema ? "

Si tiene dificultad al asignar una responsabilidad particular a una clase u otra, elija una trayectoria y recorra el sistema para ver cómo se afecta. Ahora imagine que ha decidido por la otra vía y, repita el ejercicio. ¿ Por dónde le parece más natural ? ¿ Por dónde parece hacer un uso más eficiente de recursos ? Puede también preguntar a quienes con suficiente dominio entiendan (usuarios potenciales, quizás, u otros compañeros de su equipo de diseño) y recorran el sistema también, e informen de sus percepciones.

A veces debe tomar decisiones, aún cuando las impresiones pueden no ser claras para usted, simplemente tendrá algo que probar. Tal prueba debe persuadirlo para tratar otro método,

o puede destacar beneficios imprevistos. Una de las ventajas de este proceso de diseño es que lo anima a experimentar en las etapas tempranas del diseño, cuando tal experimentación lleva una sanción pequeña. En esta etapa es una tarea sencilla dividir, agrupar, o reasignar responsabilidades a varias clases. Será más difícil hacerlo después de que el software está implantado. Por lo tanto, tome su tiempo ahora para explorar las posibilidades de su sistema.

IV.11.6) Registrando Responsabilidades

En cada tarjeta de clase que ha creado, registre cada responsabilidad asignada a esa clase. Liste las responsabilidades tan suscintamente como sea posible, se hará una frase por cada una. Las tarjetas de índice tienen la ventaja de forzarle a ser breve.

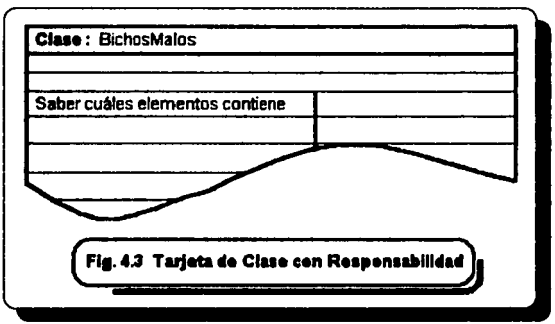


Fig. 4.3 Tarjeta de Clase con Responsabilidad

Una clase ordinariamente no debe tener un gran número de responsabilidades listadas. Si tiene una clase con demasiadas responsabilidades que no quepan en una tarjeta, puede ser una señal de que ha detallado demasiado, o que ha centralizado la inteligencia de todo su sistema dentro de una clase omnisciente y omnipotente. Es también posible que la responsabilidad ya ha sido registrada en una tarjeta perteneciente a una superclase, en tal caso no necesita registrarla otra vez en la tarjeta perteneciente a la subclase.

Si ha creado una clase que centraliza todo el conocimiento, tiene que dividirlo en varias clases que cooperen. Más tarde, en la fase de análisis, si parece que su software tiene demasiadas clases, puede descartar unas y reasignar responsabilidades. Por ahora, sin embargo, distribuir la inteligencia es la manera más apropiadas de comenzar, esto permitirá un diseño más flexible y, posteriormente un software más fácil de mantener y refinar.

IV.12) ¿ Qué son las colaboraciones ?

Ahora bien, ¿ Cómo cum0plen las clases sus responsabilidades ? Lo pueden hacer de dos maneras : al hacer el cómputo necesario ellas mismas, o por colaborar con otras clases.

Hasta este momento se ha mostrado cómo identificar las clases dentro de un diseño y, se ha determinado -al menos en un diseño preliminar-, las responsabilidades que cada clase tiene que cumplir. También se han comentado, siguiendo la Metodología empleada por Rebeca Wirfs, algunas pautas en la definición de clases iniciales de un diseño y el comportamiento por el cual cada una es responsable.

A continuación, es posible determinar cómo interactúan las clases. Al identificar las colaboraciones entre ellas, parecerá que clases que podrían haber parecido en un principio fuera disparatadas, puedan ser integradas. Puede de este modo incrementar el entendimiento de cómo se dividen las responsabilidades entre las clases que ***colaboran***.

Entendiendo entonces que, las clases además de responsabilidades deben contemplar colaboraciones, comencemos con su definición :

Las colaboraciones representan solicitudes de un cliente a un servidor en cumplimiento de una responsabilidad del cliente. Una colaboración es la personificación del contrato entre un cliente y un servidor. Un objeto puede cumplir una responsabilidad particular por sí solo, o puede requerir la ayuda de otros objetos. Decimos que un objeto colabora con otro si, para cumplir una responsabilidad, este necesita mandar a otro objeto algunos mensajes. Una simple

colaboración fluye en una dirección representando una solicitud del cliente hacia el servidor. Desde el punto de vista del cliente, cada una de estas colaboraciones son asociaciones con una responsabilidad en particular implantada por el servidor.

Aunque cada colaboración trabaja para cumplir con una responsabilidad, el cumplimiento de una responsabilidad no necesariamente requiere una colaboración. Puede implicar varias colaboraciones para cumplir completamente con una simple responsabilidad; en cambio, algunos objetos cumplirán una responsabilidad sin colaborar con cualquier otro objeto. Ejecutarán los cómputos necesarios por sí mismos, o por conocer ya la información requerida.

Entonces, ¿ Por qué las colaboraciones son tan importantes ? El modelo de colaboraciones dentro de su aplicación revela el flujo de control e información durante su ejecución. Identificando colaboraciones entre clases por lo tanto le permite tomar decisiones más sensatas e informadas sobre el diseño de su solicitud. Para identificar colaboraciones, debe identificar procedimientos de comunicación entre clases. Encontrar tales procedimientos le permitirá identificar subsistemas de clases que colaborarán, lo cual es importante para fomentar el encapsulamiento del comportamiento y conocimiento en su diseño.

El identificar colaboraciones entre las clases le obliga a determinar qué clases juegan el papel de clientes y, cuales de servidores, para cada contrato. Puede usar esta información para ayudarse a identificar responsabilidades omitidas o extraviadas. Puede a veces hallar una responsabilidad olvidada simplemente al identificar una colaboración. Si ve una colaboración sin una responsabilidad asociada, entonces sabe que ha olvidado una responsabilidad. Puede ahora volver y agregar la responsabilidad a la clase cliente. Similarmente, si ve una colaboración en la cual la clase servidor no tiene una responsabilidad para contestar, puede ahora agregar esa responsabilidad a la clase servidor.

Analizar los patrones de comunicación entre objetos en su aplicación puede también revelar cuando una responsabilidad ha sido mal asignada. Puede entonces corregir el error asignando la responsabilidad a la clase apropiada y, hacer los ajustes necesarios al resto de su diseño.

IV.12.1) Encontrando colaboraciones

Para determinar las colaboraciones entre clases, inicie analizando las interacciones de cada clase. Examine las responsabilidades para encontrar dependencias. Por ejemplo, si una clase es responsable de una acción específica, pero no posee todo el conocimiento necesario para llevar a cabo esa acción, tiene que colaborar con otra clase (o clases) que ***posean*** ese conocimiento.

Al identificar colaboraciones, haga las siguientes preguntas para cada responsabilidad de cada clase.

1) ¿ Es capaz la clase de cumplir esta responsabilidad por sí sola ?
2) ¿ Si no, qué es lo que necesita ?
3) ¿ De qué otra clase puede adquirir lo que necesita ?

Cada responsabilidad que decida compartir entre clases también representa una colaboración entre esas clases. Similarmente, para cada clase, pregunte :

1.- ¿ Qué hace o sabe esta clase ?
2.- ¿ Qué otras clases necesitan el resultado o información ?
3.- Si una clase a su vez no tiene interacciones con otras clases, podrá ser descartada. Sin embargo, antes de hacerlo es sensato hacer una rigurosa revisión de su diseño para verificar la referencia cruzada de las colaboraciones de las clases.

Se entiende que las "no interacciones", se refieren a que la clase no colabora con otra clase y, ninguna clase colabora con esta. Es importante recordar que las colaboraciones son interacciones en un solo sentido y, que ciertos tipos de clases son típicamente clientes o servidores; esto es, uno u otro acaba la colaboración. Para la mayor parte, por ejemplo, las clases que representan interfaces externas no son clientes de otros objetos en el sistema. En lugar de eso, ellas existen para ser servidores; otras clases colaborarán con ellas.

Como al identificar responsabilidades, el examinar las relaciones entre clases puede ser bastante útil al identificar las colaboraciones. Al respecto hay tres relaciones particularmente útiles :

- la relación ***"es-parte-de"***
- la relación ***"tiene-conocimiento-de"*** y,
- la relación ***"dependencia-sobre"***

De nuevo, no todas estas relaciones serán útiles para todo diseño y, ciertas relaciones específicas a la aplicación también tienen que ser identificadas y analizadas. Sin embargo, una discusión de estas relaciones puede iluminar ciertos tipos de colaboraciones que seguramente puede encontrar.

IV.12.2) La relación "Es-parte-de"

Relaciones totalidad-parte son a menudo inidicadas en la especificación por sentencias tales como "X's son compuestas de Y's". Por ejemplo, la frase "Los dibujos están compuesto de elementos" indica que las Figuras son partes del Juego.

Una relación totalidad-parte puede a veces implicar una responsabilidad para mantener información. Las relaciones totalidad-parte son de dos tipos distintos: relaciones entre clases compuestas y los objetos que la integran y, relaciones entre clases contenedoras y sus elementos.

Las clases compuestas a menudo cumplen una responsabilidad al delegar la responsabilidad a una o más de sus partes. Por ejemplo, un coche está compuesto de muchas partes, entre ellos un volante y cuatro ruedas. Al dar vuelta un coche, uno usa el volante, el cual da vuelta a las ruedas. Esto causa que el coche entero dé la vuelta. El coche tiene que colaborar con varias de sus partes para cumplir su responsabilidad de dar vuelta cuando se requiera.

Relaciones entre clases contenedoras es un arreglo. Mientras los arreglos contienen sus elementos, no necesitan mandar mensajes a sus elementos. Los elementos de los arreglos son accesados por medio de un índice y, este es otro objeto dentro del sistema que necesita ser accesado. Por lo tanto, los arreglos no tienen la responsabilidad de mantener información de sus

Como al identificar responsabilidades, el examinar las relaciones entre clases puede ser bastante útil al identificar las colaboraciones. Al respecto hay tres relaciones particularmente útiles :

- la relación *"es-parte-de"*
- la relación *"tiene-conocimiento-de"* y,
- la relación *"dependencia-sobre"*

De nuevo, no todas estas relaciones serán útiles para todo diseño y, ciertas relaciones específicas a la aplicación también tienen que ser identificadas y analizadas. Sin embargo, una discusión de estas relaciones puede iluminar ciertos tipos de colaboraciones que seguramente puede encontrar.

IV.12.2) La relación "Es-parte-de"

Relaciones totalidad-parte son a menudo inidicadas en la especificación por sentencias tales como "X's son compuestas de Y's". Por ejemplo, la frase "Los dibujos están compuesto de elementos" indica que las Figuras son partes del Juego.

Una relación totalidad-parte puede a veces implicar una responsabilidad para mantener información. Las relaciones totalidad-parte son de dos tipos distintos: relaciones entre clases compuestas y los objetos que la integran y, relaciones entre clases contenedoras y sus elementos.

Las clases compuestas a menudo cumplen una responsabilidad al delegar la responsabilidad a una o más de sus partes. Por ejemplo, un coche está compuesto de muchas partes, entre ellos un volante y cuatro ruedas. Al dar vuelta un coche, uno usa el volante, el cual da vuelta a las ruedas. Esto causa que el coche entero dé la vuelta. El coche tiene que colaborar con varias de sus partes para cumplir su responsabilidad de dar vuelta cuando se requiera.

Relaciones entre clases contenedoras es un arreglo. Mientras los arreglos contienen sus elementos, no necesitan mandar mensajes a sus elementos. Los elementos de los arreglos son accesados por medio de un índice y, este es otro objeto dentro del sistema que necesita ser accesado. Por lo tanto, los arreglos no tienen la responsabilidad de mantener información de sus

elementos (más que saber cuáles elementos son almacenados en que índice) y, no necesitan colaborar con ellos.

Otros tipos de clases contenedoras, sin embargo, pueden necesitar colaborar con sus elementos. Por ejemplo, una lista clasificada puede pedir a un elemento que se compare a sí mismo con otro elemento.

IV.12.3) La relación "Tiene-conocimiento-de"

Las clases pueden a veces saber algo de otras clases, aunque no esté compuesta de ellas. La relación "Tiene-conocimiento-de" puede ser indicada en la especificación por frases tales como "que obtiene de". Tales relaciones pueden implicar responsabilidades para conocer información y, por lo tanto significa una colaboración entre la clase que tiene el conocimiento y la clase que está conociendo (obteniendo el conocimiento).

IV.12.4) La relación "Dependencia-sobre"

Las clases son enlazadas juntas frecuentemente de otras maneras también. Las relaciones "Dependencia-sobre" son a veces indicadas en la especificación por frases tales como "cambia con". Tales relaciones pueden significar una relación "tiene-conocimiento-de", o pueden significar la existencia de una tercera parte que forma la conexión.

IV.12.5) Registro de Colaboraciones

Se asocian colaboraciones con una responsabilidad. Las colaboraciones no existen salvo para cumplir una responsabilidad. Para registrar las colaboraciones que ha encontrado, tome la tarjeta de esa clase que juega el papel de cliente. En ella escriba a la derecha de la responsabilidad la colaboración que permite cumplirla. Si una responsabilidad requiere varias colaboraciones, escriba el nombre de cada clase requerida para cumplir la responsabilidad. A la inversa, si varias responsabilidades requieren que una clase colabore con la misma otra clase,

registrar varias colaboraciones, una por cada responsabilidad. De esta manera, si más tarde decide eliminar una responsabilidad, no se confundirá al quitar una colaboración aún válida porque las otras responsabilidades la requieran.

Asegúrese de que la correspondiente responsabilidad exista para cada colaboración que registre. Sin embargo, recuerde, que si una colaboración ocurre con una subclase es proveyendo un servicio definido por una superclase, la correspondiente responsabilidad será registrada en la tarjeta de la superclase. Los servicios provistos por una clase incluyen esos listados en la tarjeta y las responsabilidades inherentes de sus superclases.

Si el cumplimiento de una responsabilidad requiere colaboración con otras instancias de la misma clase (o instancias de sus superclases), registrar esa colaboración también. En este caso, el cumplimiento de esa responsabilidad requiere comunicación entre dos distintos objetos, es un hecho que debe registrarse.

Las tarjetas de clase ahora se ven como se muestra en la siguiente Figura :

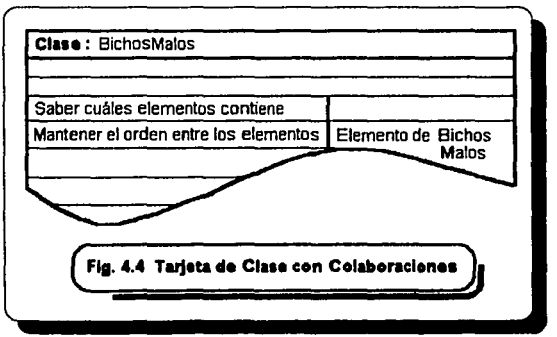


Fig. 4.4 Tarjeta de Clase con Colaboraciones

IV.13) RESUMEN

Dado que se ha cumplido la fase exploratoria del diseño de la aplicación presentada. Es posible sintetizar los pasos seguidos, en los cuales se produjo :

1) Clases (incluso algunas superclases),
2) Responsabilidades y,
3) Colaboraciones

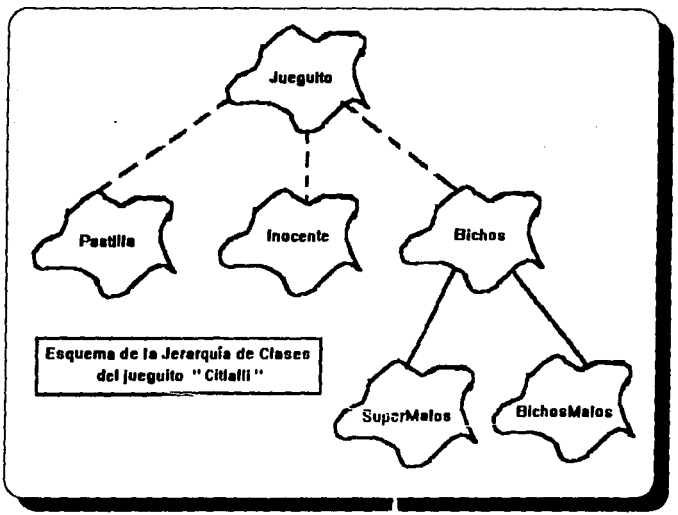
Antes de seguir, es recomendable hacer algunas observaciones del proceso de diseño a seguir y, la dirección a seguir.

Es claro que diferentes diseños pueden solucionar el mismo problema. De ninguna manera, algunos de estos diseños pueden solucionar el problema igualmente bien. En todas las decisiones de diseño hay alternativas y, puede no ser muy clara la superioridad entre varios diseños que compiten. Por lo tanto, siéntase libre de seguir una corazonada o recurra a lo antiestético para justificar una decisión de diseño. Como el nombre sugiere, este es el intento de la fase exploratoria del diseño orientado a objetos para animarlo efectivamente a realizar tal exploración. En esta fase del diseño se le invita a revisitar decisiones anteriores, cambiar de idea y, tratar varios métodos, lo cual tiene pocas penalidades. En lugar de ello, cada decisión modificada le enseña algo de las implicaciones de, por ejemplo, reasignar una responsabilidad o agregar una nueva colaboración. Ha invertido relativamente poco tiempo aún en lo específico y, nada en la implantación.

Este diseño preliminar nos ha familiarizado con el sistema y proveyó una estructura fundamental de clases y responsabilidades. Ahora tenemos la base para crear un buen diseño.

En el Capítulo V se presentan los listados que conforman la aplicación mencionada a lo largo de las Etapas de Análisis y Diseño : Un juego de Video, compaginando así, la teoría mostrada hasta este momento, con la práctica.

CAPITULO V
APLICACIONES : UN JUEGO DE VIDEO



Esquema de la Jerarquía de Clases del jueguito "Citlalli"

Ley de Murphy :

" Si algo puede salir mal, sin duda alguna, saldrá mal ".

Ley de Murphy de la dinámica activa :

" No es mucho lo que puede salir mal con la computadora, siempre y cuando ésta se encuentre apagada ... "

La ley de Murphy para las PC, Gene Weisskopf

CAPITULO V
APLICACIONES : UN JUEGO DE VIDEO

Ya en los Capítulos III y IV, Análisis Orientado a Objetos y Diseño Orientado a Objetos respectivamente, se citó la creación de un Videojuego, sin embargo, pensando en el lector interesado en referirse de manera directa a la aplicación que se presenta en ésta tesis, se cita nuevamente, la Especificación de Requerimientos del mismo (para el lector familiarizado con ésta, se recomienda pasar al listado de preferencia) :

V.1) Especificación de requerimientos de un Juego de Video

La intención de la creación del juego de video consiste en realizar un sistema bajo la metodología orientada a objetos. El juego trata de un elemento que será capaz de desplazarse a lo largo y ancho de la pantalla, el elemento en cuestión se denominará en lo sucesivo como el Inocente del juego y, será representado gráficamente por una carita en la pantalla, el caracter ASCII "001", la movilidad del Inocente tiene la finalidad de que al superponerse sobre otras figuras situadas en el mismo marco de trabajo (pantalla), sean eliminadas y su puntaje se incremente, esas figuras son conocidas como "Pastillas", las pastillas son los rombitos (caracter ASCII "004") de colores que se generarán aleatoriamente en la pantalla, son figuras fijas, su presencia servirá como aliciente para el usuario, por cierto, el usuario se desplazará por medio de las flechas del cursor. Para hacer más interesante el juego, se ha pensado en la creación de la contraparte del Inocente, los llamados "BichosMalos", es decir, los elementos que se comerán al Inocente, los cuales existen en dos categorías, en la de "Malitos" y "Voraces". Los Malitos serán los elementos capaces de comer al Inocente pero cuya movilidad está determinada, su desplazamiento se realizará en un barrido de la pantalla de izquierda a derecha, después de su aparición por primera vez, auxiliados en el uso de una función aleatoria, se desplazarán buscando al Inocente, siempre con tendencia hacia la derecha y, una vez que alcancen el límite derecho de la pantalla, se regenerarán nuevamente en la última posición referida, pero al margen izquierdo, los Malitos se desplegarán con diversos colores y. si en su recorrido encontraron un Inocente y, su movimiento los llevó a superponerse a él, el Inocente será eliminado y el puntaje

de vidas se decrementará en uno. En cambio, los Voraces (figuras móviles) son considerados como los BichosMalos cuyo desplazamiento es guiado por la posición actual del Inocente, es decir, sus coordenadas en la pantalla, de tal forma que, éste tipo de BichoMalo parece ser más inteligente pues aparentemente, cuenta con un "radar" que le permitirá ubicar al Inocente; como el desplazamiento de los Voraces se encuentra totalmente dirigido al Inocente, es más probable que muera con los Voraces que con los Malitos, se deja a juicio del usuario el desplazamiento del Inocente y, dado que la velocidad del Inocente es un poco más rápida que la de los BichosMalos, es posible en buena medida salir librado de morir. Sin embargo, para el caso en que una vida del Inocente se decremente, por conveniencia un nuevo Inocente se regenerará al centro de la pantalla y, aquél Voraz o Voraces existentes volverán a dibujarse a partir de la esquina superior izquierda de la pantalla, con la finalidad de evitar que, si el Inocente murió en el entorno de video cercano a las coordenadas centrales de la pantalla, no vuelva a ocurrir sin darle al menos, una oportunidad al usuario de salvar ésta vida.

Por otro lado, la cantidad de Pastillas existentes variará en función del Nivel en el que se encuentre el usuario, si el Inocente come todas las Pastillas y, todavía tiene al menos 1 vida, entonces el Nivel se incrementará en uno y, al momento de lograrlo, la pantalla se limpiará, incrementando la cantidad de pastillas y de BichosMalos que se dibujarán en la pantalla.

El puntaje sobre el Número de Nivel, Número de Vidas del Inocente, Número de Pastillas y Número de BichosMalos en pantalla será indicado en el primer renglón de la pantalla.

Programa principal

En el programa principal, archivo **"MAINJUEG.CPP"**, se define la clase controladora ***JUEGITO***, la más importante para el control en la interacción de todos los elementos (objetos) participantes, JUEGITO es del tipo PanPrincipal. Se incluye también, la pantalla inicial de presentación, en la cual se citan a los objetos denominados bueno, malos, malísimos y pastillas, su valor numérico representativo, es decir el puntaje acumulativo al que se puede hacer acreedor el usuario, así como el nivel en el que se encuentra.

Una vez desplegada la pantalla de presentación y, hasta que se digite cualquier tecla, el programa principal continuará, haciendo referencia a las funciones PintaPantalla y Jugar.

MAINJUEG.CPP

```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
#include <time.h>
#include "logico.h"
#include "juego.h"

void main(void)
{
 JUEGUITO PanPrincipal;
 time_t t;
 srand((unsigned) time(&t));
 textbackground(BLUE);
 clrscr();
 cprintf("\n          C I T L A L L I                    \n");
 cprintf("\n                                              ");
 cprintf("\n    Protagonistas:                            ");
 cprintf("\n     El bueno          %c                    ",0x01);
 cprintf("\n     Los malos         %c                    ",0x09);
 cprintf("\n     Los malisimos     %c                    ",0x02);
 cprintf("\n     Las Pastillas     %c                    ",0x04);
 cprintf("\n                                              ");
 cprintf("\n    Cada pastilla da  10 puntos          ");
 cprintf("\n    Cada 100 puntos da  una vida              ");
cprintf("\n    Cada nivel da  una vida                   ");

 getch();
 clrscr();
 PanPrincipal.PintaPantalla();
 PanPrincipal.Jugar();

}
```

El archivo *"JUEGO.H"*, almacena la definición de la clase JUEGITO, renombrando los objetos, ahora las pastillas (elementos que deberán ser eliminados por la superposición del bueno) serán conocidas en lo sucesivo como criptonitas y, cada vez que el Inocente (el bueno) se coma a una de ellas, el puntaje se incrementará en 10 puntos, los objetos que se encargarán de

eliminar al Inocente (los malos y los malísimos) existen en dos categorías : los Bichos del tipo intruso y los MuyMalos.

Los métodos considerados en la clase Juegito son : el constructor de la clase, el destructor de la misma, Jugar, PintaPantalla, Puntos y NuevoNivel, como se podrá apreciar en el siguiente listado :

JUEGO.H

```
#include <conio.h>
#include "bichosm.h"
#include "superm.h"
#include "inocente.h"
#include "pastilla.h"
#include "efectos.h"

class JUEGUITO
{
      private:
            Pastilla      *criptonita;
            Inocente      Yo;
            Bichos        *intruso[30];
            int iNivel;
            int iNumPastillas;
            int iPuntos;
            int iPastillasActivas;
            int NUM_BICHOS;
            int iMuyMalos;

      public:
            JUEGUITO(void);
            ~JUEGUITO(void);
            void Jugar(void);
            void PintaPantalla(void);
            void Puntos(void);
            void NuevoNivel(void);
};

// Constructor del Juegito
JUEGUITO::JUEGUITO(void)
{

 clrscr();
```

```
// Variables principales
iNivel              =1;
iPuntos             =0;     // Puntos ganados por el jugador
NUM_BICHOS          =3;     // Bichos benignos
iMuyMalos           =1;     // Uy que miedo!


iNumPastillas       = NUM_BICHOS*iNivel;        // Pastillas en juego
iPastillasActivas   = iNumPastillas;            // Todas activas
criptonita          = new Pastilla[iNumPastillas];  // crea las pastillas

// Los malos

//intruso = new (Bichos**)[NUM_BICHOS+iMuyMalos];
for (int i=0;i<NUM_BICHOS;i++)
        intruso[i]= new BichosMalos[1];
// El muy Malo
intruso[NUM_BICHOS]=new SuperMalos[1];
}

// Destructor del Juegito
JUEGUITO::~JUEGUITO(void)
{
        clrscr();
        Puntos();
        delete criptonita;
        for (int i=0;i<30;i++)
                delete intruso[i];
}
```

Asigna NuevoNivel, si y sólo si, el número de criptonitas es nulo y el Inocente tiene por lo menos, una vida.

```
void JUEGUITO::NuevoNivel(void)
{
        int i;
        Bichos *BMaux, *SMaux;

        iNivel++;                                   // Incrementa el nivel

        // Crea mas pastillas
        iNumPastillas = NUM_BICHOS+iNivel*2;        // Calcula el nuevo numero de pastilas
        iPastillasActivas = iNumPastillas;
        delete criptonita;                          // Borra las pastillas que tenias
        criptonita  = new Pastilla[iNumPastillas];  // Crea las pastillas
```

```
        // Crea mas malos
        NUM_BICHOS++;                       // Numero de bichos
        if ((int)(iNivel/2)*2 == iNivel)    // Si es nivel par, mas supermalos
              iMuyMalos++;

        // crea los objetos para los malos
        for(i=0;i<NUM_BICHOS;i++)
        {
              BMaux= intruso[i];
              delete BMaux;
              BMaux= new BichosMalos[1];          // Borra el objeto anterior
              intruso[i]= BMaux;                  // Crea un nuevo objeto
              BMaux=NULL;
              intruso[i]->Habilita();
        }
        if (iMuyMalos > 0)
        {
              for (i=NUM_BICHOS; i<NUM_BICHOS+iMuyMalos;i++)
              {
                    SMaux=intruso[i];
                    delete SMaux;
                    SMaux = new SuperMalos[1];
                    intruso[i]=SMaux;
                    SMaux=NULL;
                    intruso[i]->Habilita();
              }
        }

        // Ya que estan creados reinicia el nivel
        clrscr();
        gotoxy(30,12);
        cprintf(" N I V E L   %i",iNivel);
        efecto();
        sleep(2);                           // Se da oportunidad de leer el mensaje

        Yo.NumVidas(Yo.PideVidas()+1);      // Incrementa mis vidas
        Yo.Habilita();                      // Si me mataron, habilita
        Yo.EstableceXY(40,12);              // Y me colocas al centro

        clrscr();
        PintaPantalla();

}
```

Acumula el puntaje de : Número de Vidas del Inocente, de la cantidad de criptonitas en existencia, del Nivel de juego y del Número total de puntos adquiridos durante el juego.

```
void JUEGUITO::Puntos(void)
{
        gotoxy(1,1);
        textcolor(CYAN);
        cprintf("Vidas: %2d",Yo.PideVidas());
        gotoxy(60,1);
        cprintf("Puntaje: %d",iPuntos);
        gotoxy(15,1);
        cprintf("Nivel: %2d",iNivel);
        gotoxy(30,1);
        cprintf(" Numero de Pastillas: %2d",iPastillasActivas);
}
```

Se encarga de re-dibujar la pantalla en caso de un incremento de Nivel, se pintan nuevamente todos los elementos que participan en el juego, el Inocente al centro de la pantalla, las nuevas criptonitas generadas y, por supuesto, los bichos malos, la cantidad de objetos (pastillas y malos) se consideran en función del Nivel de Juego actual.

```
void JUEGUITO::PintaPantalla(void)
{
        int i;

        Yo.Dibuja();
        Puntos();
        for(i=0; i<NUM_BICHOS+iMuyMalos; i++)
                intruso[i]->Dibuja();

        for(i=0; i<iNumPastillas;i++)
        {
                if (i !=(int)BLACK)
                        textcolor(i);
                else
                        textcolor(7);
                criptonita[i].Dibuja();
        }
}
```

Con la acción de Jugar, se hace referencia a poner en movimiento a los objetos, toda vez que el usuario (Inocente) ejecute el primer movimiento en pantalla, auxiliado por las teclas del cursor, su desplazamiento puede ser : vertical, horizontal, hacia la derecha y, hacia la izquierda.

Se encarga también de actualizar el puntaje en caso de que un BichoMalo elimine al Inocente, le decrementa una de las vidas disponibles.

```
void JUEGUITO::Jugar(void)
{
 int i;
 int j;
 int iMalosos;

 iMalosos=NUM_BICHOS+iMuyMalos;
 _setcursortype(_NOCURSOR);
 while (bioskey(0)==0);
 Yo.Habilita();
 for (i=0;i <iMalosos;i++)
        intruso[i]->Habilita();

 while ( Yo.EstoyVivo() )
 {
              iMalosos=NUM_BICHOS+iMuyMalos;          // Malosos en juego
              if (iPastillasActivas < 1)              // si no  pastillas
              {                                       // Incrementa el nivel
                    NuevoNivel();
              }

              #ifdef DEPURA
               gotoxy(1,1);
               cout<<endl<<Yo.PideY()<<" , "<<Yo.PideX();
               for (i=0; i<NUM_BICHOS;i++)
                     cout<<endl<<intruso[i].PideY()<<" , "<<intruso[i].PideX();
              #endif

              textcolor(15);
              Yo.Mueve();
              Yo.Dibuja();
              for (i=0;i<iNumPastillas;i++)
              {
              if ( criptonita[i].EstaHabilitado())

               if( Yo.ComePastillas(criptonita[i].PideX(), criptonita[i].PideY()) )
               {
                     iPuntos+=10;
                     if ((int)(iPuntos/100)*100 == iPuntos)
                     {
                                 Yo.NumVidas(Yo.PideVidas()+1);
                                 nivel();
                     }
```

```
                    iPastillasActivas--;
                    Puntos();
                    criptonita[ij].Deshabilita();
                    Yo.Dibuja();
                }
                for (i=0; i<iMalosos ; i++)
                {
                 textcolor(15-i);
                 intruso[ij]->OnTaInocente(Yo.PideX(),Yo.PideY());
                 intruso[ij]->Desplaza();
                 if( intruso[ij]->ComeInocente(Yo.PideX(), Yo.PideY()) )
                 {
                        Yo.RestaUnaVida();
                        Yo.Borra();
                        Yo.EstableceXY(40,12);
                        Yo.Dibuja();
                        Puntos();
                 }
                }
        j=1;

        for(i=0; i<iNumPastillas;i++)
        {
   if (criptonita[i].EstaHabilitado())
   {
                        textcolor(15-j);
     j++;
   }
                criptonita[ij].Dibuja();
        }

 }

  _setcursortype(_NORMALCURSOR);
}
```

Los BichosMalos se subdividen a su vez en dos tipos de Bichos : los SuperMalos y los Malitos, en el listado ***SUPERM.H*** se dan las pautas para la definición de los Bichos más temidos en el Juego, aquellos que además de desplazarse y poder comer al Inocente, tienen la capacidad de "seguir" a su victima con la finalidad de eliminarla, de ahí que sean considerados como BichosMalos peligrosos.

```
SUPERM.H
#include "logico.h"
#include "bichos.h"
#include "efectos.h"
#define CARACTER_MALOSO 0X01

/* Muestra prototipo de Herencia, la nueva clase SuperMalos es heredera de los atributos de
la clase Bichos */
class SuperMalos : public Bichos

{
        private:
                int icxI,icyI;

                public:
                        SuperMalos(void);
                        virtual void Velocidad(int);
                        virtual void OnTaInocente(int icxX,int icyY);
                        virtual void Desplaza(void);
};

// Constructor de los SuperMalos
SuperMalos::SuperMalos(void):Bichos()
{
        // Coordenadas del bicho
        lquieto=VERDADERO;
        lvivo=VERDADERO;
        cFigura=CARACTER_MALOSO;
        fRetrazo=50.0;
        gettime(&tHora);
        // Coordenadas del bicho
        icxBm=(int) random(78);
        if (icxBm==0)
                icxBm=1;

        icyBm=(int) random(23);
        if (icyBm<=1)
                icyBm=2;
        icxI=12;
        icyI=40;
}
```

```
// Para identificar la ubicación del Inocente en la Pantalla, coordenadas X, Y
void SuperMalos::OnTaInocente(int icxx,int icyy)
{
        icxI=icxx;
        icyI=icyy;
}

/* Constructor con el que se define el movimiento que habilita a los SuperMalos para
desplazarse */
void SuperMalos::Desplaza(void)
{
        int itDiff;
        struct time tAux;

        gettime(&tAux);

        itDiff=abs(tAux.ti_hund-tHora.ti_hund);

        if ((float)itDiff>=fRetrazo)
        {
                tHora=tAux;
        Borra();

        if (icxBm != icxI)
                // Si esta arriba el Inocente
                if ((icxBm-icxI) > 0)
                        icxBm--;
                else
                        icxBm++;
        if (icyBm != icyI)
                // Si esta arriba el Inocente
                if ((icyBm-icyI) > 0)
                        icyBm--;
                else
                        icyBm++;
        Dibuja();
        }
}

/* Se asigna un tiempo de retraso para la velocidad a la que serán acreedores los SuperMalos
en función del Nivel en el que el usuario se encuentre */
void SuperMalos::Velocidad(int iNivel)
{
        fRetrazo-=iNivel*3;
}
```

Con el archivo ***LOGICO.H*** se asignan - en casi de no existir - las banderas del tipo logico : Verdadero y Falso, es un archivo auxiliar en la manipulación del Videojuego, cuando se requiere es incluído en el módulo que lo mande a llamar.

LOGICO.H

```
#ifndef _boolean.h
typedef unsigned LOGICO;
const  unsigned FALSO=0;
const  unsigned VERDADERO=!FALSO;
#define _boolean_h
#endif
```

El archivo **BICHOS.H,** define la clase Bichos, se encuentra incluído dentro del archivo SUPERM.H, contempla todas las capacidades que poseen los BichosMalos, tales como Habilitarlos, Dibujarlos, Borrarlos, Detenerlos, Matar, Desplazarse, Comer al Inocente, Solicitar sus coordenadas : PideX y PideY, Establecerse en las coordenadas XY obtenidas e, identificar la posición actual del Inocente : OntaInocente.

BICHOS.H

```
#ifndef BICHOS_H
#define BICHOS_H
class Bichos
{
 protected:
        int icxBm,icyBm;
        char cFigura;
        LOGICO lquieto;
        LOGICO lvivo;
        float fRetrazo;
        struct time tHora;
 public:
        virtual void Velocidad(int iNivel)=0;
        void Habilita(void);
        void Dibuja(void);
        void Borra(void);
        void Deten(void);
        void Mata(void);
        virtual void Desplaza(void)=0;
        int ComeInocente(int icxIX,int icyIY);
        int PideX(void);
        int PideY(void);
        void EstableceXY(int X,int Y);
```

```
        virtual void OnTaInocente(int,int)=0;
};

// Inicia el movimiento de los bichos
void Bichos::Habilita(void)
{
            lvivo=VERDADERO;
            lquieto=FALSO;

}

// Pinta las figuras representativas de los BichosMalos
void Bichos::Dibuja(void)
{
    if( lvivo)
    {
        gotoxy(icxBm,icyBm);
        cprintf("%1c",cFigura);
    }
}

// Elimina en la pantalla un BichoMalo dado
void Bichos::Borra(void)
{
    if( lvivo)
    {
        gotoxy(icxBm,icyBm);
        cprintf(" ");
    }
}

// Habilita la bandera quieto, deteniendo al BichoMalo
void Bichos::Deten(void)
{
     lquieto=VERDADERO;
}

// Deshabilita la bandera vivo del BichoMalo
void Bichos::Mata(void)
{
    lquieto=VERDADERO;
    lvivo=FALSO;
}
```

```
// Si el BichoMalo se encuentra en las coordenadas del Inocente, elimina al Inocente de la
pantalla, le decrementa una vida y se posiciona al Bicho en (1,1)
int Bichos::ComeInocente(int icxIX,int icyIY)
{
        int iVidasQuitar=0;
        if (icxIX == icxBm && icyIY == icyBm)
        {
                iVidasQuitar=1;
                icxBm=1;
                icyBm=1;
                efecto1();
        }
        return (iVidasQuitar);
}

// Solicita la coordenada en X
int Bichos::PideX(void)
{
        return(icxBm);
}

// Solicita la coordenada en Y
int Bichos::PideY(void)
{
        return (icyBm);
}

// Posiciona en las coordenada X,Y
void Bichos::EstableceXY(int X,int Y)
{
 icxBm=X;
 icyBm=Y;
}
#endif
```

Sin duda alguna, uno de los ingredientes que vuelven muy atractivo a un juego de video lo es, el sonido, la combinación de imágen y sonido produce al jugador mayor expectación y, también le ayuda a asimilar los eventos que ocurren aún cuando su atención se encuentre concentrada en una zona diferente en la pantalla de donde se originó el sonido, por ejemplo, cuando el Inocente muere debido a la superposición de un BichoMalo, o bién cuando ha eliminado todas las pastillas existentes en ese nivel, el ascenso de Nivel activa precisamente los sonidos que se han contemplado en el archivo ***EFECTOS.C***

EFECTOS.C

```
#include <stdlib.h>
#include <dos.h>
#define RETARDO 64000
#define RETARDO1 10000
#define RAZON 100
void nivel()
{
        sonido(440);
        sonido(500);
        sonido(440);
        sonido(480);
        sonido(440);
}

void efecto()
{
 int i,frec;
 for(i=1;i<15;i++)
 {
   do{
     frec = rand();
   }while(frec>5000);
   sonido(frec);
 }
}

void efecto1()
{
 int i;
 for(i=0;i<10;i++)
  sirena();
}

void sonido(int frec)
{
 unsigned i;
 union
 {
   long divisor;
   unsigned char c[2];
 }
 cuenta;
 unsigned char p;
 cuenta.divisor = 1193280/frec;
 outportb(67,182);
```

```
  outportb(66,cuenta.c[0]);
  outportb(66,cuenta.c[1]);
  p=inportb(97);
  outportb(97,p|3);
  for(i=0;i<RETARDO;++i);
  outportb(97,p);
}


void sirena()
{
  unsigned i,frec;
  union
  {
    long divisor;
    unsigned char c[2];
  }

  cuenta;
  unsigned char p;

  p = inportb(97);
  outportb(97,p|3);
  for(frec=1000;frec<3000;frec+=RAZON)
  {
  cuenta.divisor = 1193280/frec;
  outportb(67,182);
  outportb(66,cuenta.c[0]);
  outportb(66,cuenta.c[1]);
  for(i=0;i<RETARDO1;++i);
  }

  for(;frec>1000;frec-=RAZON){
    cuenta.divisor = 1193280/frec;
    outportb(67,182);
    outportb(66,cuenta.c[0]);
    outportb(66,cuenta.c[1]);

    for(i=0;i<RETARDO1;++i);
  }
  outportb(97,p);
}
```

El archivo ***EFECTOS.H*** es el encargado de habilitar o deshabilitar según sea el caso, las frecuencias producidas ya sea en un cambio de Nivel o en el decremento de una vida del Inocente en el Videojuego

EFECTOS.H

```
#ifndef EFECTOS_H
        #define EFECTOS_H 1
        void nivel(void);
        void efecto(void);
        void efecto1(void);
        void sonido(int frec);
        void sirena(void);
        #include "efectos.c"
#endif
```

En el archivo ***BICHOSM.H*** se definen las pautas para la clase BichosMalos, su Desplazamiento y la Velocidad que poseerán, también se encarga de "monitorear" la ubicación del Inocente por medio del constructor OnTaInocente.

BICHOSM.H

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <iostream.h>
#include <dos.h>
#include <time.h>
#include "logico.h"
#include "efectos.h"
#include "bichos.h"
#ifndef BICHOSM_H
#define BICHOSM_H 1
#define CARACTER_BICHO 0x09

class BichosMalos:public Bichos
{
 public:
        BichosMalos(void);
        virtual void Desplaza(void);
        virtual void Velocidad(int);
        virtual void OnTaInocente(int,int){};
};
```

```
// Constructor del bicho
BichosMalos::BichosMalos(void)
{
        // Coordenadas del bicho
        icxBm=1;
        icyBm=20;
        iquieto=VERDADERO;
        ivivo=VERDADERO;
        cFigura=CARACTER_BICHO;
        fRetrazo=0.3;
        gettime(&tHora);
}

// Asigna movimiento a los BichosMalos
void BichosMalos::Desplaza(void)
{
        int iDesplaza,itDiff;
        struct time tAux;

        gettime(&tAux);

        itDiff=abs(tHora.ti_hund-tAux.ti_hund);

        if (itDiff>=fRetrazo)
        {
                tHora=tAux;
                iDesplaza=(int) random(3);
                // a la derecha
                Borra();
                if (iDesplaza)
                {
                        icxBm++;
                        if (icxBm > 78)
                                icxBm=2;
                }
                else
                {
                        icxBm--;
                        if (icxBm<2)
                                icxBm=78;
                }
                iDesplaza=(int) random(2);
```

```
            // Para arriba
            if (iDesplaza)
            {
                  icyBm++;
                  if (icyBm > 23)
                        icyBm=2;
            }
            else
            {
                  icyBm--;
                  if (icyBm < 2)
                        icyBm=23;
            }

            Dibuja0;
      }

}

void BichosMalos::Velocidad(int iNivel)
{
      fRetrazo+=iNivel*2.0;
}
#endif
```

A continuación se presenta la definición de la clase ***PASTILLA,*** que representan el puntaje que irá adquiriendo el usuario toda vez que las elimine, su representación en pantalla se realiza por medio de rombitos de colores, los cuales presentan el efecto de modificar su coloración cuando una sóla pastilla a la vez haya sido eliminada por el Inocente, los BichosMalos no las afectan en absoluto, pueden inclusive superponerse a ellas y no se muestra cambio alguno al respecto, las pastillas son figuras fijas, la virtud del Videojuego se presenta en que, cuando se incrementa el Nivel de juego, la cantidad de pastillas, así como de BichosMalos se incrementa proporcionalmente y, el reto es aún mayor al intentar eliminar una mayor cantidad de pastillas con muchos más enemigos por esquivar, aún incluso, aquellos que persiguen al Inocente.

PASTILLA.H

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <iostream.h>
#include <dos.h>
#include <time.h>
#include "logico.h"
#include "efectos.h"
#define CARACTER_PASTILLA 0x04

class Pastilla
{
 private:
        int icxX,icyY;
        char cFigura;
        LOGICO lHabilitado;
 public:
         Pastilla();
         void Deshabilita(void);
         void Dibuja(void);
         void Borra(void);
         void Mata(void);
         int PideX(void);
         int PideY(void);
         LOGICO EstaHabilitado();
   void EstableceXY(int X,int Y);
};


// Constructor de la pastilla
Pastilla::Pastilla(void)
{
        // Coordenadas del bicho
        icxX=(int) random(78);
        if (icxX==0)
                icxX=1;

        icyY=(int) random(23);
        if (icyY<=1)
                icyY=2;
        lHabilitado=VERDADERO;
        cFigura=CARACTER_PASTILLA;
}
```

```
// Inicia el movimiento de los bichos
void Pastilla::Deshabilita(void)
{
        lHabilitado=FALSO;
}

// Pinta en pantalla el simbolo de una pastilla (un rombo)
void Pastilla::Dibuja(void)
{
    if( lHabilitado)
    {
        gotoxy(icxX,icyY);
        cprintf("%1c",cFigura);
    }
}
void Pastilla::Borra(void)
{
    if( lHabilitado)
    {
        gotoxy(icxX,icyY);
        cprintf(" ");
    }
}

/*  Deshabilita las pastillas que hayan sido eliminadas, sólamente al cambiar de Nivel la
pantalla será redibujada, el efecto de borrado se logra al deshabilitar la pastilla en cuestión */
void Pastilla::Mata(void)
{
    efecto();
    Deshabilita();
}
// Solicita la coordenada X de la pastilla
int Pastilla::PideX(void)
{
    return(icxX);
}

// Solicita la coordenada Y de la pastilla
int Pastilla::PideY(void)
{
    return (icyY);
}
```

```
// Ubica la pastilla en las coordenadas X,Y dadas
void Pastilla::EstableceXY(int X,int Y)
{
  icxX=X;
  icyY=Y;
}

// Asigna un valor booleano a la pastilla en función de su habilitación
LOGICO Pastilla::EstaHabilitado(void)
{
        return lHabilitado;
}
```

El archivo *INOCENTE.H* fue empleado para definir la clase Inocente, que es la representativa del usuario en el Videojuego, sus funciones principales deberán ser : eliminación de la mayor cantidad de pastillas posibles y, evasión de cualquier tipo de BichoMalo.

INOCENTE.H

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <bios.h>
#include "logico.h"
#include "efectos.h"
#define CARACTER_INOCENTE 0x02

class Inocente
{
 private:
        int  icxX,icyY;
        char cFigura;
        LOGICO lQuieto;
        LOGICO lVivo;
        int iNumVidas;

 public:
        Inocente();
        void Dibuja(void);
        void Borra(void);
        void Habilita(void);
        int ComePastillas(int X,int Y);
        void NumVidas(int iNumeroVidas);
        void RestaUnaVida(void);
        void Mueve(void);
        int EstoyVivo(void);
```

```
        int PideX(void);
        int PideY(void);
        int PideVidas();
        void EstableceXY(int X,int Y);
};

// Constructor del Inocente
Inocente::Inocente()
{
 icxX=40;
 icyY=12;
 lQuieto=FALSO;
 lVivo=VERDADERO;
 NumVidas(9);
 cFigura=CARACTER_INOCENTE;
}

// Pinta al Inocente en pantalla, representado por una carita color blanco
void Inocente::Dibuja(void)
{
        if (lVivo)
        {
                gotoxy(icxX,icyY);
                cprintf("%1c",cFigura);
        }
}

// Borra al Inocente de la pantalla al sustituirlo por un espacio en blanco
void Inocente::Borra(void)
{
        if (lVivo)
        {
                gotoxy(icxX,icyY);
                printf(" ");
        }
}
// Activa al Inocente
void Inocente::Habilita(void)
{
   lVivo=VERDADERO;
        lQuieto=FALSO;
}
```

```
// Al superponerse a una pastilla, la deshabilita y se incrementa su puntaje
int Inocente::ComePastillas(int X,int Y)
{
        LOGICO lsi_come;

        lsi_come=(icxX==X && icyY == Y);
        if (lsi_come)
                nivel();
        return lsi_come;
}

// Cuenta el número de vidas disponibles por el Inocente
void Inocente::NumVidas(int iNumeroVidas)
{
 iNumVidas=iNumeroVidas;
}

// Decrementa una vida al Inocente, en caso de que un BichoMalo se superponga a él
void Inocente::RestaUnaVida(void)
{
        iNumVidas--;
}

// Desplaza al Inocente
void Inocente::Mueve(void)
{
        register int iTecla,imodd;
        /* la funcion 1 retorna 0 hasta que una tecla sea presionada */
        if (bioskey(1) != 0)
        {
                /* la funcion 0 retorna la tecla que esta en el buffer */
                iTecla = bioskey(0);
                imodd= (iTecla & 0xFF00) >> 8 ;
                iTecla = iTecla & 0x00FF;

                Borra();
                switch(imodd)
                {
                        // hacia abajo
                        case 80:
                                icyY++;
                                if (icyY >23)
                                        icyY--;
                                break;
                        // hacia arriba
                        case 72:
```

```
                        icyY--;
                        if (icyY<2)
                                icyY++;
                        break;
                // hacia derecha
                case 77:
                        icxX++;
                        if (icxX >78)
                                icxX--;
                        break;
                // hacia izquierda
                case 75:
                        icxX--;
                        if (icxX<1)
                                icxX++;
                        break;
        }
        Dibuja();

    }
}

// El Inocente se reconoce a sí mismo activo en el Videojuego
int Inocente::EstoyVivo(void)
{
    if (iNumVidas > 0)
        return(1);
    else
        return(0);
}

// Solicita la coordenada X del Inocente
int Inocente::PideX(void)
{
    return(icxX);
}

// Solicita la coordena Y del Inocente
int Inocente::PideY(void)
{
    return(icyY);
}
```

```
// Solicita el Número de Vidas disponibles por el Inocente
int Inocente::PideVidas(void)
{
        return iNumVidas;
}

// Se establece al Inocente en las coordenadas X,Y dadas
void Inocente::EstableceXY(int X,int Y)
{
  icxX=X;
  icyY=Y;
}
```

En el archivo ***BOOLEAN.H*** se definen las constantes ***FALSE*** y ***TRUE*** como 0 y 1 respectivamente, se emplea en diversas ocasiones por algunas clases que lo incluyen, permite manejar con mayor libertad los valores booleanos de Falso y Verdadero.

```
BOOLEAN.H
#ifndef_boolean.h
typedef unsigned BOOLEAN;
const  unsigned FALSE=0;
const  unsigned TRUE=1;
#define _boolean_h
#endif
```

El archivo ***PANTA.C*** se emplea para borrar la pantalla, asignarle color, posicionarse en las coordenas (5,5) y finalmente reasignarle color.

```
PANTA.C
#include <stdio.h>
// Elimina la pantalla
void borrapantalla(void)
{
  printf("\x1b[2J");
}

// Ubica en las coordenas x, y dadas
void ve_xy(int col,int ren)
{
  printf("\x1b[%d;%dH",ren,col);
}
// Asina color
void EstableceColor(int color)
{
  printf("\x1b[%dm",color);
}
```

```
// Programa principal del archivo Panta.C
void main(void)
{
  borrapantalla();
  EstableceColor(31);
  we_xy(5,5);
  printf("ya");
  EstableceColor(0);
}
```

Finalizaremos este capítulo con una frase muy adecuada al tema en cuestión :

" Todo proyecto es perfectible "

y; por lo tanto, se invita al lector a adicionar, reformar y/o en su caso, a incrementar las capacidades de la aplicación mostrada. Con fines didácticos, el Videojuego es un ejemplo sencillo, nada complicado y, precisamente por la simplicidad de su diseño, resultó idóneo para mostrar la Metodología Orientada a Objetos con suficiente amplitud que, por cierto era el objetivo por alcanzar.

CAPITULO VI
CONCLUSIONES

En un principio, cuando se creaban sistemas no existía formalmente alguna herramienta en la cual apoyarse para su generación, así pues, se creaban a "prueba y error", posteriormente, el Análisis Estructurado fue concebido en los años 60's, cuyo objetivo consistía en la descomposición funcional de un sistema; posteriormente surgió el llamado Análisis de Suceso-Respuesta, del mismo tipo que el Análisis Estructurado, el cual se centraba en los sucesos externos y posteriormente, en la deducción de los procesos del sistema.

A finales de los 80's, casi 20 años después, se crea el llamado Análisis Orientado a Objetos y la correspondiente etapa de Diseño Orientado a Objetos, para éstas fechas el Análisis Estructurado no sólo se había propagado entre los conocedores de ésta rama, además adquirió sólidas raíces y su empleo (¿salvo las muy contadas excepciones?) se había generalizado.

Hoy, aproximadamente 7 años después de la comercialización del Análisis y el Diseño Orientado a Objetos, todavía no existen reglas, normas, ni procedimietno alguno que pueda considerarse el prototipo a seguir en el empleo de éste Paradigma.

Personalmente, estoy convencida de que el hecho de contar con el antecedente : Análisis Estructurado, no implica que la nueva Metodología Orientada a Objetos, obstaculice o bién, coloque barreras en su aprendizaje y/o entendimiento.

A lo largo de la Historia, no recuerdo (al menos en éste momento), que la evolución de un conocimiento y lo que ésta implica (nacimiento, creación, desarrollo, mantenimiento, etc) haya entorpecido la obtención del conocimiento como un producto mejorado, por el contrario, dado que existió un antecedente, es que hoy en día contamos con un consecuente. Confirmo que quizá resulte difícil la idea de renovar hipótesis, costumbres y tradiciones en la labor de generación de un sistema, pero ésto no se debe a que el conocimiento del Análisis Estructurado lo complique, más bién se debe a la natural tendencia humana de rechazar un nuevo conocimiento hasta que quede satisfactoriamente probado y comprobado y, en otros casos, la sóla idea de migrar a algo nuevo, aún a sabiendas de que representa mejoras y menor esfuerzo alentan dicho cambio.

Por otro lado, entender el Paradigma Orientado a Objetos no es sencillo, de hecho poder alcanzar un nivel de madurez aceptable para ser capaces de dirigir más enfoques sobre ésta metodología implica un largo tiempo, no tan extenso como para que jamás se comprenda pero sí, lo suficientemente amplio para que nuestra mente asimile lo que intenta transmitirle y, posteriormente sea capaz de asociarlo con el Mundo Real (otra de las virtudes de la orientada a objetos).

Estas son las razones por las que considero que el Análisis Estructurado y el Análisis Orientado a Objetos no son áreas que deban considerarse por separado o contrapunteadas mutuamente.

El Diseño Orientado a Objetos permite entre otras cosas, reutilizar código generado para aplicaciones específicas y, entre mayor grado de abstracción alcance el Diseñador, mejores Bibliotecas de Objetos existirán, lo fascinante de la MOO es su pretensión de modelar al Mundo Real, logrando que el abismo entre lo real (mundo real) y lo virtual (Sistema : problema modelado en una computadora) se vea disminuído en forma considerable.

Sabido es que no existe metodología única o "mejor" para diseñar un sistema orientado a objetos, sin embargo, la lectura efectuada nos proporciona diferentes procedimientos elaborados por diversos autores expertos en el área que, si bién no resuelven del todo nuestro problema de unificación de Metodologías, sí nos orientas con mayor certeza en cuál camino elegir y qué herramientas y caminos ya han sido seguidos con dicha tendencia.

El Análisis y el Diseño son las primeras fases en la producción de un sistema de software en las cuales se trata de definir con precisión cuál es el problema y cómo lo vamos a solucionar. **En particular, el Análisis, como su nombre lo indica, significa analizar el problema del mundo real y crear su modelo abstracto, el cual se conoce como la representación del dominio del problema. La fase de Diseño busca la definición de la solución.** Por lo general, ésta fase utiliza la representación abstracta del problema y la extiende para lograr la representación de la solución, misma que se conoce como la **representación del dominio de la solución** que incluye, por lo general, la representación del dominio del problema sumándole los conceptos indispensables para lograr la solución pero no del problema mismo.

Si a los procesos de Análisis y Diseño les añadimos el término Orientado a Objetos, lo que estamos diciendo es que las representaciones de los dominios del problema y de la solución están expresadas en los términos del modelo de objetos.

La Fase del Análisis Orientado a Objetos (AOO) es la que se encarga de descubrir los objetos semánticos dentro del Dominio del problema, es decir, las abstracciones que representan conceptos que tienen un significado claro en la descripción del problema, como por ejemplo, cliente, cajero, producto, servicio,etc.

El proceso de Análisis orientado a objetos es una técnica general de definición de los objetos semánticos del dominio del problema. Esta técnica se divide en tres actividades principales :

1.- Identificación de las clases de objetos semánticos, los atributos y las operaciones que describen el comportamiento de los objetos.

2.- Colocación de los atributos y las operaciones dentro de las clases, así como la definición de las relaciones de generalización, agregación y asociación entre las clases.

3.- Especificación del comportamiento dinámico de los objetos.

Estas actividades no tienen por qué realizarse secuencialmente, es decir, después de identificar algunas clases y atributos en el dominio del problema podemos pasar a la Fase de colocación para ordenar, en cierta forma, los conceptos identificados, y luego se puede regresar nuevamente a la fase de identificación para buscar nuevos candidatos para los objetos semánticos. Normalmente, el proceso de análisis es iterativo, donde las actividades de identificación, colocación y especificación del comportamiento se repiten e intercalan de manera aleatoria.

Durante la fase de Diseño Orientado a Objetos (DOO) se descubren otros objetos o abstracciones útiles para llegar a la solución. En el ejemplo mencionado se puede añadir la clase persona como la superclase que generaliza las características de un cliente y un cajero, o unas clases que ofrecen el manejo de ventanas para una interfaz amigable del sistema.

El proceso de Diseño Orientado a Objetos, cuyo objetivo es hacer el modelado del dominio de la solución, incluye todas las actividades citadas con anterioridad en el proceso de Análisis, pero en

esta ocasión dedicadas a la identificación, colocación y especificación de las clases de objetos de interfaz, aplicación y utilería. **Una fase nueva es la de optimización.**

4.- Optimización de clases, la cual puede consistir en la restructuración de las jerarquías y relaciones entre las clases debido al descubrimiento de ciertas posibilidades de mejorar el diseño, como son, por ejemplo, la reutilización de clases ya existentes, la generalización introduciendo clases abstractas, la partición de las clases demasiado grandes, etc.

Como se puede observar, es difícil decir cuando acaba el proceso de Análisis y empieza el de Diseño. Muchos autores de nuevas propuestas las clasifican como de Análisis o de Diseño según su propio sentir. En general, los analistas se concentran más en la estructura de los objetos, mientras que los diseñadores se preocupan más por el comportamiento dinámico.

La metdología más sencilla, accesible y de fácil comprensión (de entre todas las estudiadas), es sin duda, la de ***Rebeca Wirfs-Brock*** **introduce el concepto de un convenio:** ***un Subsistema describe estas partes exportadas en términos de contratos y pueden tener un número de contratos. Esto permite múltiples niveles de visibilidad a ser definidos, junto con la de Grady Booch (1983),*** la que señala que *:*

El inicio del DOO consiste en identificar objetos, encontrarlos es el desafío del ADOO. Booch creó el "Método Gramatical", el cual le sugiere al diseñador comience con una descripción en prosa del sistema deseado, donde los nombres sean visualizados representen identificadores potenciales de las clases de objetos. Donde los verbos identifican a los métodos :

* ***Definición del problema***
* ***Descripción de la solución : subrayando a los sustantivos como posibles objetos.***
* ***Asociación de atributos con cada objeto.***
* ***Identificación de posibles métodos subrayando a los verbos.***

El problema con la Metodología de Objetos radica en que oculta parte de la complejidad de la definición de clases. Además, es difícil lograr abstracciones útiles a partir de la definición del problema. Debido a que el ADOO se encuentran en sus primeras etapas, se dispone de pocas herramientas automatizadas.

Para el diseño de clases se sugiere :

* ***Modificar con clases las entidades que ocurren de forma natural en el problema.***
* ***Diseñar métodos de finalidad única.***
* ***Diseñar un nuevo método al encontrar una oportunidad de ampliar uno existente.***
* ***Evitar métodos extensos.***
* ***Guardar como variables instancia (modelo) los datos necesitados por más de un método o por una subclase.***
* ***El diseñador debe trabjar para la biblioteca de clases, no sólo para él mismo, ni para su aplicación actual.***

En general, se considera que el Diseño Orientada a Objetos consta de :

1) Identificación y definición de objetos y clases.
2) Organización de relaciones entre las clases.
3) Extracción de estructuras en una jerarquía de clases.
4) Construcción de bibliotecas de clases y marcos estructurales de aplicación reutilizables.

Los diseñadores experimentados también sugieren que una buena biblioteca de clases debería ser profunda y estrecha, con varios niveles de subclasificación. Las reglas siguientes sugieren formas de construír clases abstractas:

1) Identificar mensajes y métodos comunes y trasladarlos a una superclase.
2) Eliminar de una superclase aquellos métodos que son ignorados frecuentemente, en lugar de que los hereden sus subclases.
3) Acceder a todas las variables solamente mediante el envío de mensajes.
4) Reelaborar las subclases para construir especializaciones. Una subclase es una especialización si hereda todos los métodos de superclase y agrega nuevos métodos propios.

En un futuro que se considera no muy lejano el empleo de la POO constituira una práctica aceptada de manera general, lo cual es bastante lógico si se compara con la evolución de la Computación como resultado de una búsqueda continua de la elevación del nivel de expresividad y de estructuración de las herramientas computacionales. El mundo real es un modelo abstracto en términos de computación, de aquí se infiere la dificultad a la que se presenta el programador al tratar de representarlo, para facilitarle esta labor se han generado abstracciones computacionales en las cuales se fundamenta el programador y su tarea es más sencilla. Es por ello que se explica la existencia de los lenguajes de programación tales como : Lenguaje Ensamblador, Lenguajes de Alto Nivel, el concepto de Subrutina, los Sistemas de Módulos y finalmente la POO, con los conceptos de objetos, clases y herencia.

Ninguno de los Métodos Orientados a Objetos son perfectos, así que es importante que los desarrolladores se dén cuenta de que la construcción de los métodos es frecuentemente necesaria. Sin embargo, un buen entendimiento de ambos, del problema y del método propuesto son necesarios. Además es importante que ésta disciplina esté basada en conceptos fundamentales más disciplinados que los particulares de un sólo método. Por encima de todo, existe plena seguridad en que en los próximos cinco años de tiempo, ninguno de los métodos será el mismo.

Los elementos que proporciona la Programación Orientada a Objetos permiten la elaboración de sistemas estructurados modularmente, fáciles de extender y mantener. El poder y la flexibilidad de las herramientas provistas en los lenguajes de programación de este paradigma varían.

Una de las preocupaciones actuales más urgentes de la industria de la computación es la de crear software y sistemas corporativos más pronto y de más bajo costo. Para hacer un buen uso del poder cada vez mayor de las computadoras, necesitamos un software de mayor complejidad y mucho más confiable.

La necesidad de crear un mejor software se aplica tanto en el interior de la propia industria del software como dentro de las empresas de todo tipo que crean sus propias aplicaciones de computación. Si el desarrollo de aplicaciones tarda de dos a tres años, con un retraso de creación de varios años, las empresas no pueden crear más aplicaciones ni reaccionar ante la competencia de manera rápida. Se pierde la capacidad vital para el cambio dinámico.

A través de la historia de la ingeniería, parece surgir un principio : la gran ingeniería es la ingeniería sencilla. Las ideas que se tornan muy rebuscadas, inflexibles y problemáticas tienden a ser reemplazadas por otras más nuevas y claras desde el punto de vista conceptual y con sencillez estética.

Los componentes serán cada vez más complejos desde el punto de vista interno, pero será más sencillo interactuar con ellos. Serán *cajas negras* donde no podremos mirar al interior.

Contínuamente se están documentando otras estrategias para ayudar a los diseñadores a pensar en términos orientados a objetos. Un ejemplo son las tarjetas ***Clase, Responsabilidad y Colaboración*** (CRC), las cuales se usan principalmente como herramienta didáctica y como metodología para estudiar la conducta de los diseñadores orientados a objetos; son también un recordatorio y ayuda a los programadores experimentados y noveles a comunicarse entre sí acerca de la modelación del entorno con objetos.

No existen metodologías formales para la identificación de objetos en lo que se conoce comúnmente como la fase de análisis. Se suele utilizar el método de Booch. La clasificación de objetos en la fase de diseño es posible en diferentes metodologías y sistemas notacionales.

A estas altura de la vida del Paradigma Orientado a Objetos, se espera lograr un producto estable y una herramienta más refinada del mismo, quizá surga de la completa fusión del Análisis Estructurado y el Análisis Orientado a Objetos, o bién de las "Mil y un historias Orientadas a Objetos" (dado que cada autor al profundizarse en ésta metodología tiende a crear su propia versión orientada a objetos, cosa que concluyo en función de tantas metodologías y además por experiencia propia). Su futuro es incierto, pero de cualquier forma, la Metodología Orientada a Objetos actualmente tiene mucho que darnos y se espera aún más de ella.

En SOMA las clases consisten de un identificador, los apuntadores a superclases, apuntadores a componentes de clases, atributos, métodos y reglas. Los atributos pueden ser simples retenciones para objetos o pueden denotar relaciones con multiplicidad apropiada y modalidad de información. Los atributos son taquigrafiados para su estandarización en la obtención y colocación de métodos. Los métodos pueden especificar aserciones de tres tipos : precondiciones, postcondiciones y condiciones de invariancia. Los atributos, métodos y reglas pueden ser heredados. Las reglas pueden ser de seis tipos, como siguen :

(1) ***Reglas que relacionan atributos a atributos.*** Ejemplo :

if Servicio > 5 then

Hoyos = Hoyos + 1

Esto podría ser expresado como una postcondición en "Coloca.Servicio" causando "Coloca.Hoyos (Hoyos+1)" o una precondición en "Dame.Hoyos"

(2) ***Reglas que relacionan métodos a métodos.***

Esos son naturalmente expresados como aserciones de preferencia que reglas de cualquier manera.

(3) ***Reglas que relacionan atributos a métodos.*** Ejemplo :

Ellas pueden ser expresados como

if_needed(pre.get) o if_cambia demonios(post.put) o como "pre-" o "post-condiciones"

(4) ***Control de reglas para atributos.*** Ejemplo :

El comportamiento bajo conflictos de herencia múltiple y omisiones (precondiciones en gets).

(5) ***Control de reglas para métodos***. Ejemplo :

El comportamiento bajo conflictos de herencia múltiple (postcondiciones en gets).

(6) ***Reglas de excepción de manejo.*** Ejemplo :

Sensor recalentado (invariancia o postcondición en temperatura).

Iah Graham afirma que hay exactamente esos seis tipos de reglas y que existe una forma mecánica para convertir la forma de regla a forma de aserción. La investigación es bajo la forma de proveer esta aserción y definir un lenguaje formal para la expresión de las reglas (un estilo BNF, de preferencia) y aserciones (Z o VDM).

Obsérvese que las aserciones atañen a cada método como heredados y, que integran las reglas como parte de los datos semánticos.

Caso de Estudio : SACIS

Para un ejemplo más concreto y realista, examinaremos algunos objetos en SACIS con sus reglas. El escenario involucra un tipo particular de evento : una venta. Una venta involucra a un cliente, quien puede ser un adulto o un niño y, una lista de artículos, los cuales pueden ser de varios tipos. Existe una lectura de código de barras que lee y actualiza las existencias del sistema y un sistema cuya tarea es garantizar que los artículos sean vendidos como convenientes para los propósitos del cliente. El término "conveniente" abarca varios criterios de "propiedades para el propósito", una de las cuales es la seguridad. Asumamos que un cliente en particular está intentando comprar un tubo de pegamento. El modelo convencional de datos se encuentra en el contorno mostrado en la **Fig. A.55**. La decisión como a cualquiera de las relaciones mostradas como diamantes en la figura deberían ser objetos o no encontrarse postergados para el tiempo de existencia.

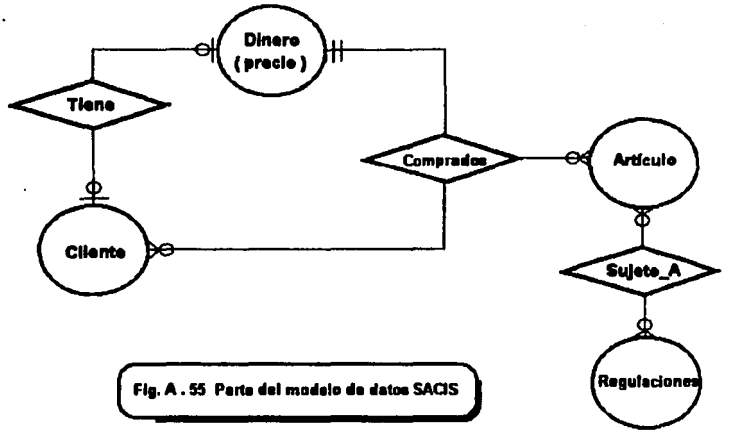


Fig. A . 55 Parte del modelo de datos SACIS

Nótese que hay un número de estructuras de herencia múltiple involucradas : específicamente esas empleadas para "pegar" y cliente mostradas en la **Fig. A.56**, la cual también ilustra que en algunas ocasiones conviene diseñar herencia múltiple disyuntiva cuando una subclase no es una A o una B. La convención es que una barra doble es usada encima del icono AKO, lo cual se evade de ser posible.

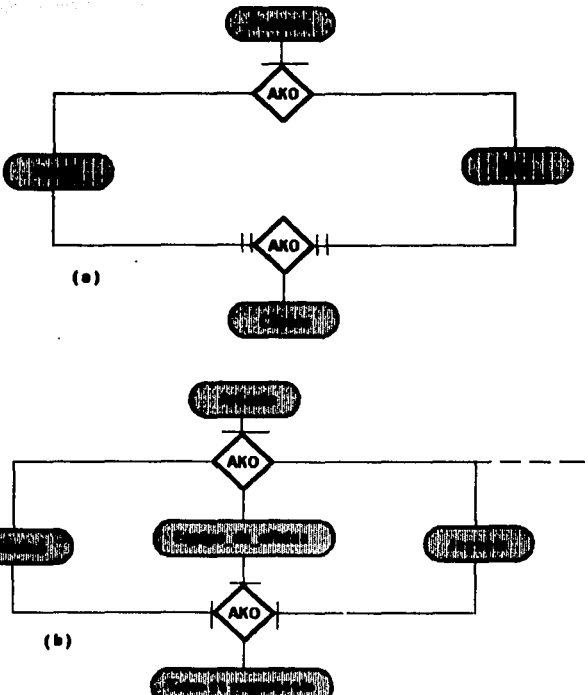


Fig. A . 56 Redes de herencia múltiple en SACIS

La pregunta es ¿ Cómo modelar la derivación de seguridad para ese tubo de pegamento ? Desde que "pegar" hereda a partir de los artículos de oficina y juguetes no deberá ser un problema seguro.

Sin embargo, las estructuras ilustradas muestran que hemos modelado el hecho de que el "pegar" pueda ser abusado como una droga introducida. La mayoría de las tiendas se niega actualmente a vender "pegamento" a los niños por ésta razón; el modelamiento estructural de este hecho representa una regla de negocio para cada tienda. Sin embargo, es posible que exista una auténtica razón para un niño comprador de pegamento, quizá si un modelo a escala que debe ser "pegado" ha sido comprado. En cada caso el sistema tampoco debe permitir la venta o, más estrictamente quizás, negar la venta del pegamento.

Al implantarlo, el sistema debe debilitar el valor de la seguridad del pegamento como bajo y examinar la entera transacción para observar a los compradores del pegamento-relacionado. La **Fig. A.57** muestra dos de los objetos que deben ser registrados durante el análisis de éste problema, a convertir el contorno del modelo de datos a la forma orientada a objetos.

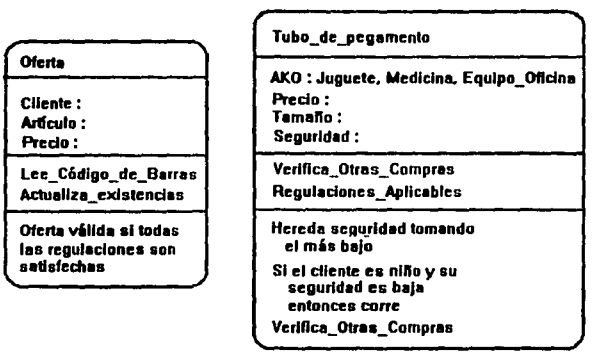


Fig. A . 57 Parte del Análisis Orientado a objetos de SACIS

En síntesis, SOMA extiende los mnemónicos de Coad y Yourdon SOSAS a lemas más comprensivos, es decir, los analistas deben proceder por la identificación de :

- ***Etiquetas***
- ***Objetos***
- ***Estructuras***

- ***Uso de estructuras***
- ***Clasificación de estructuras***
- ***Composición de estructuras***

* ***Asociaciones y sus datos semánticos***
* ***Atributos con notación para representar tipo, validación, seguridad y omisiones.***
* ***Métodos con notación para representar la excepción del mantenimiento, aserciones y, paso de parámetros.***
* ***Reglas***
  - ***Control de reglas***
  - ***Reglas de negocios***
  - ***Puesta en funcionamiento***

El método recomendado es seguir los pasos en el orden dado, aún cuando la etiquetación es una actividad sujeta a revisión en la mayoría de los estados. El modelo puede ser convenido como una cascada con iteración o, preferiblemente, Graham la concibe como una espiral. El proceso es un riesgo-manejado, como con la mayoría de los modelos de espiral.

Ahora daremos un breve repaso al estado actual de las herramientas CASE y a los modelos de ciclo de vida en relación a la orientada a objetos.

Las Herramientas CASE y los Modelos de Ciclo de Vida

La ***Ingeniería de Software en Auxilio a la Computación (CASE)*** viene a estar de moda durante los años 80's junto con la Programación Orientada a Objetos. Paralelamente hubo un incremento en el énfasis de métodos desarrollados estructuradamente, tales como la ***Ingeniería de Software (IE)*** y ***Análisis de Sistemas Estructurados (SSADM)***, con incorporación a series de técnicas notacionales y de diseño y, reglas para su uso. Los beneficios confirmados para los métodos estructurados son bién conocidos. Están diseñados para evadir la producción caótica de los indocumentados, pobremente costeados y justificados, sistemas inmantenibles. Por otro lado existen a menudo sistemas burocráticos e inflexibles y, con tendencias por debajo de los aspectos creativos de la producción de software, ocasionando problemas cuando ocurre lo inesperado -como usualmente. En efecto, una inspección de 600 proyectos por la Compañía Butler (1990) encontró que los proyectos que usaban técnicas estructuradas (cerca de 1/6 parte del total inspeccionado) tiene un 15% más baja productividad que el promedio y, un error del 40% por encima del promedio durante la revisión y el primer mes de operación.

Los proyectos que emplean productos modernos de software 4GL son lejanamente mejores, cón "screen painters" dando una mejoría superior al 40% en tiempo de desarrollo. Algunos desarrolladores de métodos, tales como el CCTA quien define SSADM, han intentado actualizar sus métodos e incorporar 4GL y técnicas prototipo, pero debemos preguntarle a cualquier método las ideas de incorporación orientada a objetos que tiene un "rol" a efectuar en la restauración del balance entre disciplina y creatividad. Ciertamente, la orientación a objetos ofrece una filosofía en el desarrollo de sistemas que puede ser descrita por un método. En la visión de Graham, las técnicas orientadas a objetos se abordarán los métodos gradualmente. La Ingeniería de Software (IE) y Análisis de Sistemas Estructurados (SSADM) se encuentran bajo presión para ofrecer algo de cada extensión. ***Mientras tanto nos encontramos -al menos- familiarizados con muchísimas técnicas orientadas a objetos, las cuales representan la construcción de bloques de métodos.***

Los sistemas para la verificación completamente automática y la consistencia de un diagrama que empleé jerarquía a un diseño son claramente útiles a ambos : al inexperto y al diseñador consumado. El único peligro es que la herramienta podría imponer un método o notación también estrictamente para permitir el ejercicio de la creatividad, o inhibirla con un orden diferente de aplicación, donde las reglas podrían no ser apropiadas.

Muchísimos beneficios que algunas herramientas CASE liberan se deben a su habilidad para generar código automáticamente. Esto significa que el líder del proyecto tiene que poseer la confianza para destinar muy poco tiempo a la etapa de codificación. Si alguien se equivoca en la etapa de codificación entonces no existe tiempo para corregirlo. Más distante, la generación de código sin la re-ingeniería es muy peligrosa. La obligación del programador que efectúa un parche urgente a media noche origina la improbabilidad de que vaya y mezcle todos los diseños y análisis de diagramas, aún cuando él manipule los controles. Actualmente la generación de herramientas CASE sólamente soporta una muy limitada forma de re-ingeniería. Los Sistemas que capturan la semántica son requeridos para hacer la re-ingeniería posible. Esto sugiere que es posible que las BDOO's *(Bases de Datos Orientadas a Objetos)* con versiones y diagramación orientada a objetos "frontera-final" puedan ser construidas por completo, tales como la Tecnología Index, cuyos próximos productos se encuentran en la cima de las BDOO's.

Al menos para pequeños, por ejemplo la decisión de apoyo, los sistemas probablemente con Bases de Datos Orientadas a Objetos suplantarán ambas métodos convencionales RDBM's y herramientas CASE. En el presente existen solamente unas cuantas herramientas aunque la situación es probable que cambie rápidamente. Hasta que ésto suceda los desarrolladores de sistemas tendrán que construir sus propias herramientas de diagramación. Las plataformas para tales desarrollos probablemente serán : Smalltalk e HyperCard. Sin embargo, como existe un

número de herramientas emergiendo, nosotros atacaremos una breve descripción y evaluación de ellas próximamente. El uso de gráficas y herramientas CASE es recomendado para incrementar la productividad. Alguna herramienta empleada, sin embargo, no debe implicar un compromiso al lenguaje de implantación a menos que, sea conocido con completa certaza cuál será el ambiente destino y que nunca cambiará. Desde que los métodos están experimentados, la evolución ha sido muy rápida, en el presente podría ser sabio usar herramientas de muy bajo costo que puedan ser reemplazadas o adoptar un método meta-CASE. El más reciente es más caro pero más adaptable y las herramientas deben ser menos probables de necesitar reemplazo, quienes proporcionan ventajas en términos del entrenamiento y la continuidad de largos proyectos.

Lar Arquitectura del los Sistemas (de Popkin Software & Systems Inc.) es una comprehensiva herramienta CASE corriendo bajo Microsoft Windows que soporta muchísimas técnicas tales como Entidad-Relación, Transición-Estado y Diagramas de Flujo. Como una herramienta opcional extra, soporta una técnica de diagramación basada en Booch's (1991) DOO *(Diseño orientado a objetos)* y, existe alguna flexibilidad en la que los iconos pueden ser diseñados por el usuario. Coad/Yourdon es también soportado en esta herramienta de bajo-costo.

ROSE es una herramienta dedicada al soporte de los métodos Booch única y completamente y es sustituida por la propia compañía de Booch. Corre racionalmente, bajo UNIX.

HOOD es soportado por un creciente número de herramientas, incluyendo IPSYS's HOOD Toolset, VSF's HOOD Software Factory, el equipo de trabajo Cadre's e IDE's Software a través de imágenes, las cuales también soportan OOSD.

La herramienta IPSYS HOOD kit surgió del trabajo en Ciencias de Software en PCTE (Portable Common Tools Environment) y la herramienta ECLIPSE CASE. Consiste de 8 herramientas : un editor gráfico, editores informales y texto estructurado, como un verificador de diseño, un código generador para C, C++ y Ada, un documento generador, una facilidad de ventana y una herramienta acostumbrada. El Contexto-sensitivo de ayuda para ambos HOOD y las herramientas es provisto.

HOOD-SF es una herramienta conjunta escrita en Visual Software Factory (VSF), disponible de una compañía llamada "VSF Limited". La última es una herramienta para la construcción de herramientas CASE: una "herramienta meta-CASE", la cual usa un lenguaje formal de descripción como Prolog llamado Cantor para describir las reglas metodológicas y estructurales. La Diagramación y las reglas son definidas dentro de un editor gráfico. Las restricciones y los datos semánticos pueden también ser definidos. De ésta manera, la verificación sintáctica, como no es

permite conexión inmediata entre los datos almacenados en un diagrama de flujo de datos, o verificación semántica, tal como no permite diagramas de flujo de datos en bajo-nivel los cuales no han sido definidos en un alto nivel, pueden ser impuestos. Cantor adicionalmente permite la definición de generadores de código para 4GLs y otros lenguajes. Seguido fácilmente entre análisis, diseño y estados de código es también soportado, ayudado con métodos potenciales de re-ingeniería. El depósito datos/diseño está implementado como una red semántica invertida, la cual dá una ejecución de degradación áspera lineal con tamaño, diferente del método relacional dentro del Depósito de IBM y productos similares, donde una curva exponencial puede ser esperada. Este método superior sufre somanete de una propiedad natural y además la dificultad de aprendizaje de Cantor. El diseñador comercial podría, sospecho, haber considerado la dificultad con la sintaxis inusual de Cantor. VSF ha sido usado para construir herramientas CASE para muchísimos métodos, otros como HOOD, incluyendo SSADM. La versión HOOD provee soporte para Issue 3.0 ó HOOD. A gusto de la notación empleada puede estar acumulada en la Fig. A.58.

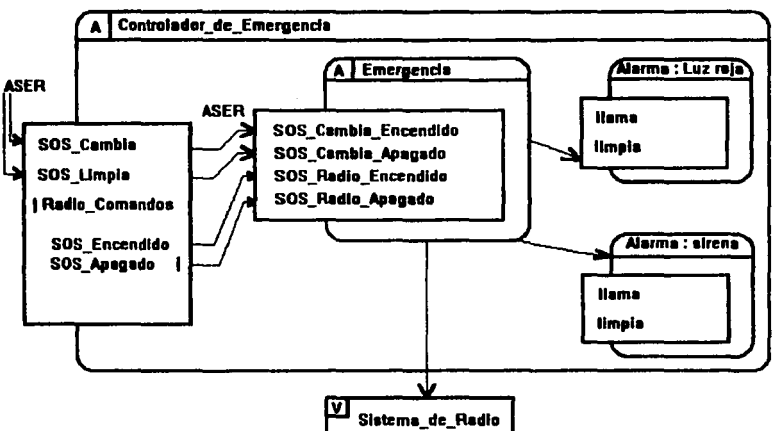


Fig. A.58 Parte del diseño HOOD para una simulación de radio. ASER significa Requerimiento de Ejecución Asíncrono, las líneas sólidas muestran el uso de jerarquía y las líneas punteadas muestran las relaciones " implantadas por ".

La más reciente adición al método VSF's soporta al momento de escritura que el método Texel describió anteriormente y una variación de Ptech.

El "Object Maker" es una herramienta meta/CASE que corre en muchísimas plataformas, incluyendo PC's, con el soporte para un enorme rango de métodos orientados a objetos y convencionales incluyendo Booch, Coad/Yourdon, TMO, Firesmith's ADM3, CRC, Shlaer/Mellor y otros. El Paradigma Plus de Protosoft es un producto similar meta-CASE y soporta muchos métodos orientados a objetos.

SELECT, de Select Software Limited, es otra herramienta meta-CASE pero tiene la ventaja del costo de corrimiento en PC's. SELECT soporta muchísimos métodos en tiempo-real tales como Ward/Mellor, Hatley y MASCOT y el apoyo de SSADM es proyectado. Este soporta HOOD Issue 3.0 y TMO, y los autores se propusieron proveer soporte para otros métodos de Análisis y Diseño Orientado a Objetos. La Base de datos fundamental es Btrieve, así que la interface con otros sistemas, incluyendo sistemas expoertos, es posible. El aspecto más notable que Graham encuentra sobre SELECT es que tiene un precio muy bajo y la plataforma PC.

Software por medio de Imágenes (StP, Software through Pictures), de Interactive Development Environment (IDE) también ofrece soporte para HOOD y otros métodos de diseño orientado a objetos, incluyendo CRC y TMO, tan bién como métodos convencionales. Aunque este no tiene la configurabilidad de VSF, este permite notaciones a objetos las cuales podrían en principio ser usados para adicionar reglas, etc. IDE están encomendados a hacer ésto mucho más que una meta-herramienta. Sin embargo, StP proveé a los usuarios con la capacidad de configurar su ambiente, características deseables, generación de código y, buena interacción con otras herramientas. Se encomendó a IDE para hacer a StP gradualmente más como una herramienta meta-CASE. La filosofía implícita por el nombre, escribiendo software completamente por medio de dibujos es muy seductor. Las notaciones gráficas están bajo el control del usuario y la documentación automática es posible. OOSD es una notación muy detallada situable para diseñadores de preferencia que analistas y no soporta directamente el registro de las cosas como reglas de negocios, etc. Segundamente, la principal fortaleza de Stp es la capacidad de generación de código y soporte específico para el lenguaje C++. Lo cual viola el principio señalado a evadir que el análisis debe tener implantación independiente.

El trabajo en equipo de Cadre Tecnhonologies de herramientas de análisis parcialmente soportadas por Shlaer/Mellor y provee soporte para HOOD. Esto es también cierto quen nada del uso de programación orientada a objetos en el reciente desarrollo del producto. Como Cognos, este es reportador que Cadre encontrado algunos problemas cercanamente adicionales con el método.

Ptech es una herramienta tan buena como un método. Ha sido descrita en el precedente, pero es cierto remencionando como una herramienta completa pero poderosa herramienta CASE y

generador de código en su propio correcto. Tempranamente las versiones generadas en C; el fin del producto con la Versión 2 son las llamadas : C++ y Ontos. De especial interés son las características las cuales soportan objetos empaquetados, así que la existencia de código y el código parcheado en el por ingenieros de Software puede ser separado fuera del código generado. El desarrollo original fue por el Hardware Silicon Graphics pero una versión SparcStation existe ahora. El Generador de código Ptech está incorporado en el Diseño DEC, una herramienta meta-CASE la cual al tiempo de escritura, soporta Ptech y Coad/Yourdon.

El AOO de Coad/Yourdon es soportado por una Herramienta de AOO de Object International. GE tiene una herramienta UNIX soportando TMO.

Como se ha dicho que existen pocas herramientas comerciales CASE soportando un método genuinamente orientado a objetos, podemos ya especificar una colocación de requerimientos mínimos para cada herramienta. Estas incluyen el orden de flexibilidad provisto por Object Maker, Paradigm Plus, VSF e IPSYS, el método de "usuario-definido" empleando reglas y gráficas, buena interfaz externa a bases de datos, hojas de cálculo, procesadores de texto y herramientas de sistemas expertos, soporta multi-usuario y un depósito abierto. Los sistemas como Ptech y VSF, los cuales son esencialmente herramientas para 1-usuario, están severamente limitadas a causa de ésto. Las herramientas CASE sin soporte a multi-usuario, el control de la versión y alguna noción del trabajo de empaque son virtualmente inútiles excepto por el proyecto más pequeño. El soporte a Multi-usuario es esencial para algunos proyectos largos y, especialmente cuando los equipos de desarrollo se encuentran geográficamente distribuídos. Esto es enfatizado por la prototipificación y los problemas particulares de sistemas extensiblemente evolucionados.

La herramientas de diagramación son esenciales para el AOO. La flexibilidad y las reglas y gráficas de usuario definidas son un auxilio obvio a las compañías no dispuestas a empatarse por sí mismas a los métodos los cuales pueden cambiar o llegar a ser inapropiados. La buena interfaz externa a bases de datos, hojas de cálculo y procesadores de texto permiten a los sistemas existentes y a los profesionales estar acomodados y dejar a todos los contribuidores a un proyecto de trabajo en la forma que ellos encuentren más confortable y productiva.

El significado de los anunciones de IBM's AD/Cycle y Repository Manager no son subestimados debido a que el compromiso de muchos usuarios a esos productos cuando ellos emergieron y se estabilizaron. Las herramientas CASE, si ellas soportan a los métodos orientados a objetos o no, tendrán que tolerar interfaz con un depósito aunque si ellos soporten más datos eficientemente y la representación del conocimiento.

La interfaz a herramientas de sistemas expertos permite deficiencias en la forma de reglas metodológicas que son implantadas para ser corregidas. Lo último será importante cuando en métodos caseros sean desarrollados y exista una necesidad de imponer nueva consistencia o reglas de seguir-métodos en la herramienta CASE. La alternativa es fijar lenguajes basados en reglas en las herramientas. Los usuarios podrían definir sistemas expertos los cuales podrían aplicar varios cequeos y limitantes a sus diseños. Esta posibilidad enfatiza la necesidad prar un depósito abierto, así que el sistema experto puede accesar esto directamente por medio de una interfaz conocida. Ejemplos de herramientas CASE con una arquitectura abierta son DesignAid y SELECT, ambos usan Btrieve como la base de datos primaria para el texto y gráficas. Un sistema experto es (caparazón) con una interfaz Btrieve, tales como Leonardo, podría ser usado, en principio, para implantar algunas reglas adicionales que un diseñador de sistemas desearia especificar. A menudo un depósito mainframe es requerido para garantizar la consistencia y la amplia disponibilidad, lo cual militariza las herramientas en contra sin arquitecturas abiertas. Las ventajas de un mainframe sobre una micro o una estación de trabajo en red es que son evaluadas en contra de la fuerza, flexibilidad y el costo efectivo de lo último.

La independencia del Método es una cuestión controversial. Por un lado es dicho que, ser de valor real, una herramienta CASE tiene que ser usada sin el contexto de un método estructurado y modelo de ciclo de vida al cual la organización es totalmente encomendada. Además, la mayoría de los usuarios y suministros están de acuerdo con que un experto en el método particular empleado podría ser atribuido al proyecto al menos de tiempo completo para garantizar el éxito. Por otro lado, como ha sido señalado, adhesión rigida a métodos es a menudo burocrática, la inovación sofoca, no es aplicable igualmente a todos los tipos de proyectos y puede incrementar el costo para una cantidad grande. Los analistas quiene siguen cieg imente reglas formales no siempre hacen desarrollo con la cualidad de ser capaces de marcar sucesivamente con lo inesperado. Así que será una necesidad en algunos proyectos adaptar y alterar aunque el mejor de los métodos. Esto indica un fuerte requerimiento para herramientas las cuales son útiles independientemente do algún método particular : las herramientas donde algún tipo de diagrama apropiado o estilo de documentación puede ser usado. Una alternativa es desarrollar herramientas, tales como Object Maker y VSF, las cuales pueden ser adaptadas a diferentes métodos. Sin embargo, esto no es lo mismo como el método mix-y-match lo cual es algunas veces encontrado a ser eficaz. También, el costo de cada adaptación puede ser significativa e inapropiada para una menor diversión de un método sin un proyecto el cual es mayoritariamente conforme a ésto. Así que, para muchos proyectos las herramientas CASE "denominadas "vertical" y "lower" pueden ser preferidas a algunas las cuales incorporan un modelo de ciclo de vida completo y método. Aquí la productividad es incrementada por unas pderosas herramientas de diagramación y cuequeo de consistencia, pero la creatividad es menor restringida. El "downside" es que los ingenieros de sistemas tiene que ser

capaces de seguir un método, cualquiera, formal o en sus cabezas, sin la guía y soporte de la herramienta. La respuesta correcta, según Ian Graham, es tomar la decisión en el punto de vista de las peculiaridades de cada proyecto individual.

Un requerimiento adicional para una herramienta CASE para proyectos orientados a objetos es la provisión de las herramientas de navegación clase-biblioteca. Tales herramientas implican la necesidad para un bibliotecа análoga al administrador de datos en toda organización de software.

No existen herramientas CASE que cumplan con todos esos requerimientos. En el presente Ian Graham podría recomienda seleccionar una herramienta de bajo costo que puede ser abandonada y reemplazada conforme el campo madure o, si éste es inaceptable, investigando una herramienta meta-CASE.

Los métodos de análisis de negocios los cuales mapean dentro de diseños orientados a objetos fácilmente son raros. Los métodos tales como IE son datos manejados pero permanecen separados del diseño fucnional en el, a menudo inútil, la esperanza de que el proyecto en equipo encontrará el tiempo de cosntruir al ente de las historias de vida o matrices de procesos/entidad para completar la combinación de función y datos. BIS ha intentado incorporar métodos de análisis y diseño orientado a objetos sin SSADM. IE y otros métodos principales. Muchos de otros consultores están buscando ésta ruta activamente. Es ciertamente a ser recomendado que algunos métodos orientados a objetos no deberían descuidar las lecciones aprendidas de los últimos 30 años de experiencia en desarrollo de sistemas. Es importante que las organizaciones de desarrollo sean capaces de construir en su existencia modelamiento de datos, DFD y otras habilidades.

Booch (1991) piensa que la tecnología de los sistemas expertos puede ser empleado para ayudar a los diseñadores a inventar nuevas clases para simplificar una estructura clase y con creatividad similar diseñar "actores", aunque él opina que este es no característico de ser posible en el futuro próximo debido al problema de la incorporación del conocimiento del sentido/común. Ian Graham no piensa que ésto sea posible del todo, parcialmente por razones "ontological" y parcialmente porque él cree que la inteligencia es un fenómeno social y además no replicable para las máquinas, pero éste no es el lugar para dar los argumento en detalle (Véase Langham (1993)).

Con el desarrollo de los productos de base de datos orientados a objetos y extensiones similares a RDBMS-basados en 4GL's, una larga parte de la funcionalidad de las herramientas CASE puede llegar a ser redundante. La razón para ésto es que mucha de la justificación para técnicas de diagramación se reducen a dos cosas : la independencia en la implantación y la dificultad de lectura de la estructura de un sistema de éste código o de acuerdo con usuarios no-

computarizados-literalmente. La independencia de implantación ha sido un tópico importante en diseño de sistemas durante algún tiempo. Una representación de implantación independiente, dice en la forma de Inglés estructurado y/o diagramas de flujo de datos, permite el análisis y diseño a ser transmitido fuera del "prior" a la elección de lenguaje y hardware de implantación y puede actuar como un medio para la transportación de aplicaciones de una máquina a otra. Así que, aunque los prototipos son a menudo útilmente convertidos dentro de "modelos de papel", como lo ilustró por ejemplo por el estudios-de-case mencionado por Bodkin y Graham (1989). Ahora, un lenguaje de alto nivel es precisamente uno donde la discrepancia entre la concepción y la implantación es narrada y si nosotros podemos generar código automáticamente de dibujos -y después todos lo que es precisamente lo que los dibujante de pantalla y generadores de código en las herramientas CASE a menudo hacen- entonces nosotros podemos, en efecto, el código en la representación intermedia. En un 4GL basado en una base de datos por ejemplo, la generación del software pictórico está ya disponible para algunas extensiones con reportes de formas-base y diseñadores de pantalla. Como los 4GL's de éste tipo tienden a cerrarse a la idea del no-procedimiento, la programación pictórica la distinción entre análisis, diseño y programación se nublaron (empañaron). En algunas aplicaciones de alto-performance, éste nunca será un método tolerable por encima del performance inevitable hacia arriba, pero en muchos de los grandes sistemas comerciales el costo extra de la fuerza de la computadora (obteniendo mayor economía) será superado por el crecimiento en el cosoto del desarrollo y mantenimiento el cual depende de la habilidad humana, de la fuerza laboral. Para éstos tiempos, sin embargo, los métodos permanecerán como un tópico importante en el desarrollo de software. Además, las bases de datos orientadas a objetos ahora automatizan las consecuencias como el control de la versión el cual podría ser añadido a productos convencionales.

En el presente, Graham piensa que CASE es una tecnología inmadura que evolucionará rápidamente bajo la influencia de las ideas y métodos orientados a objetos.

El prototipo y el uso de métodos formales y pruebas de corrección ambas contribuyen significativamente a la calidad y confiabilidad de los sistemas. La discusión detallada de éstos importantes secuencias en el ciclo de vida es comentado en el próximo apartado.

El Modelo Objeto Abstracto (OMG) y el Modelo de Referencia de AOO / DOO

El Grupo de Manipulación de Objeto ha definido un modelo de referencia de alto-nivel para el análisis y diseño orientado a objetos. Su propósito es cubrir todas las secuencias imaginables en la Ingeniería de Software Orientada a Objetos pero no es un método en sí mismo. Esquemáticamente el modelo de referencia es descrito en la Fig. A.59.

El modelamiento de objeto proveé una colocación de términos y los conceptos para la representación del mismo sin la amplitud del análisis y diseño como un objeto. Este define los términos estándares. SOMA, es concerniente con el modelamiento de objeto.

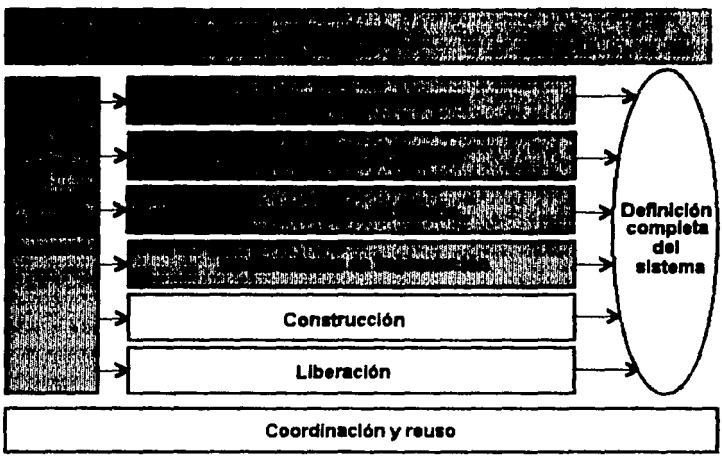


Fig. A . 59 El modelo de referencia OMG. Sólamente las áreas sombreadas están en el ámbito del modelo.

El modelamiento estratégico cubre la iniciativa y el modelamiento de objetos. Los requerimientos de captura y la planeación del desarrollo. ***El modelamiento de análisis*** cubre el proceso de obtención de una descripción del campo del problema. ***El modelamiento del diseño*** (es decir, diseño lógico) consiste en adicionar información no pública a las especificaciones de clase y a la producción de una solución para algún problema particular, incluyendo objetos del sistema. ***El modelamiento de implantación*** es un diseño físico e implica diseñamiento de módulos, la estrategia de distribución y el conteo tomado del software y hardware a ser empleados.

Existe una estructura normativa mínima en el modelo de referencia. Se podría usar una cascada o prototipificación de método. Podemos comenzar en algún punto de la estrategia del modelamiento de negocios al código. El grupo OMG, desarrolló el modelo de referencia considerado sobre 30 métodos y ninguno de ellos fue excluido por el resultado. Interesantemente, los aspectos de los modelos funcionales SIG se consideraron con el agrupamiento y aspectos de

preferencia que como una parte del modelo de análisis. La referencia el modelo aprovecha el modelo de objeto OMG para el análisis y diseño orientado a objetos el cual es ahora bastante comprensivo, además de total y extensamente entendido. La necesidad de elevación para el modelo objeto se debe a la amplia variación en el significado dado al término como "objeto" y "método" en la literatura. El modelo objeto declara ahora que el tipo de objeto será el término correcto para la clase. Los Métodos están definidos como la implantación de operaciones. Estos apuntes han empleado la terminología estándar preferentemente. La Tabla A.I coloca la terminología del modelo objeto y las diversas relaciones de especialización entre los términos.

Término	Especialización
Valor	Objeto / No-objeto / Concepto
No-objeto	Relación
Concepto	Concepto de modelamiento Concepto del Modelo Objeto
Concepto de Modelamiento	Concepto de modelamiento estratégico Concepto de modelamiento de análisis Concepto de modelamiento de implantación Liberación / Tipo de actividad / Técnica
Concepto del Modelo Objeto	Concepto de regla Concepto del comportamiento del objeto Concepto de grupo y aspecto Concepto de la estructura del objeto
Concepto de regla	Límite / Excepción
Concepto del comportamiento del objeto	Mensaje / Evento / Estado / Transición
Mensaje	Requerimiento
Concepto de grupo y aspecto	Diagrama / Esquema Concepto de calidad Conceptos del modelo estratégico G & V Conceptos del modelo de análisis G & V
Concepto de estructura de objeto	Tipo de atributo, de objeto, de relación

Tabla A. I El concepto de jerarquía del modelo objeto abstracto OMG

Uno de los beneficios lado-efecto de las actividades del análisis orientado a objetos de los OMG's, es que el SIG enfoca a los metodologistas a extender sus métodos dentro de más áreas del modelo de referencia.

Aplicando el AOO al modelamiento de negocios - un estudio-case

El Análisis Orientado a Objetos puede ser aplicado no sólamente al análisis de los sistemas de cómputo pero, inicialmente, al análisis de algún "sistema". Lo cual se ilustrará con una estudio-case mostrando cómo el método es aplicado al modelamiento de la estructura de una organización y la información fluye sin ésto, con el propósito de proveer un modelo de los negocios que podría ser usado para simular diferentes estrategias organizacionales. Este estudio-case está basado en uno llevado por el autor para el oficio de servicios del departamento de un oficio de vida garantizado. El proyecto usó una adaptación del método Coad/Yourdon. Aunque las reglas de negocios no tuvieron que ser usadas por la relatividad del nivel-alto natural del ejercicio, un estudio más extensivo podría haber sido usado por ellos.

Uno de los problemas del departamento fue la opción de mover la "publicidad electrónica" al estado de "completo". Este corrió una operación de impresión sofisticada pero la colocación de tipos fue llevada en una composición de estaciones de trabajo de una moda atrasada y su montada fue hecha manualmente. La **Fig. A.60** muestra la estructura de alto nivel del departamento y sus interacciones con otros departamentos como siete capas. Cada capa es mostrada como un objeto y las ligas representan posible paso de mensajes.

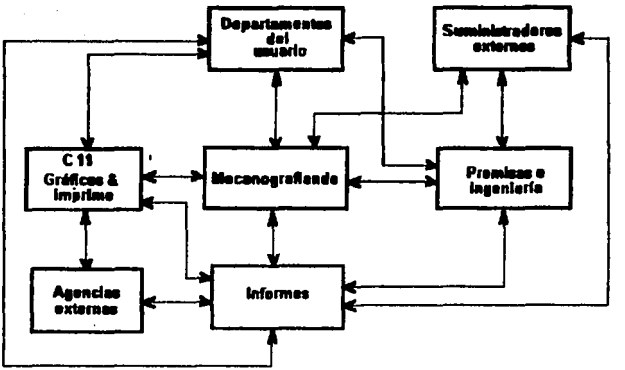
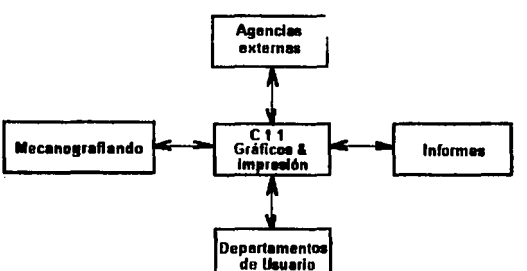


Fig. A . 60 Representación etiquetada de servicios de oficina

Cada capa puede ser vista externamente o desde un punto interno de vista, como es ilustrado para la impresión y la división gráfica en la Fig. A.61.



Aspecto externo
(a)

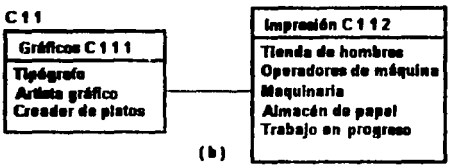


Fig. A .61 (a) Aspecto externo ; (b) Aspecto interno

El aspecto externo define el paso de mensajes permisible entre el objeto departamental y otras organizaciones. El aspecto interno indica que existen dos subdivisiones, cada una de las cuales es representada como un objeto. El área Gráfica tiene un gerente, un artista gráfico, etc (ésta composición de estructura) y las funciones requeridas están parcialmente listadas en la Fig. A.61(b). Esto comunica con el printshop quien estructura esto parcialmente expuesto por la Fig. A.61(b) para, por ejemplo, enviando negativas completas para impresión. El detalle interno de las estructuras no son mostradas. Porque la introducción del equipo de publicidad fue contemplada, se dibuja la estructura de paso de mensajes de Gráficos antes y después de esta introducción.

Debe hacerse hincapie en que, mientras ésta notación resulta muy expresiva y útil para la mayoría de las áreas de negocios, fue algunas veces más natural, usar diagramas de flujo de datos y éste fue hecho en pocas ocasiones, se refleja que los métodos rígidos sofocan las innovación e inhiben la productividad. El uso de diagramas de flujo de datos en la planeación de la estrategia de los sistemas de información es discutida por Eastlake (1987). El método de AOO auxilió en la clarificación de los problemas, soluciones y la comunicación, el resultado a la gerencia quien, según encontró Ian Graham, entendió la notación con facilidad con poca explicación.

En éste caso el modelamiento fue hecho completamente en el papel con tecnología no más sofisticada que un procesador de palabras. Esto mostró una dificultad, referida tempranamente, teniendo constantemente una cajas redibujadas conteniendo más atributos y operaciones. Estas características de los modelos de negocios y problemas comerciales y una herramienta de dibujo adaptable podría haber sido útil.

Síntesis

Los beneficios de la programación orientada a objetos aplicada igualmente al análisis y diseño orientado a objetos, son aún amplificados. Las especificaciones orientadas a objetos y los diseños tiene el potencial de ser reusable y extensible como los programas orientados a objetos. Los beneficios son potencialmente más grandes en el nivel de análisis.

El Diseño Orientado a Objetos y el Análisis coinciden más que en el diseño y análisis en los métodos convencionales, debido al hecho de que ambos intentan modelar objetos del mundo real. Lo cual es especialmente verdadero de la distinción entre el análisis y el diseño lógico. Se ayuda a cerrar el intervalo de seguimiento y está estrechamente relacionado con un método de prototipificación a los sistemas de construcción.

El AOO existe en muchísimas formas, existen cerca de 50 publicaciones sugeridas. Los métodos son temarios o unarios, cubren diferentes fragmentos del ciclo de vida y variaciones acordadas al desacuerdo del énfasis colocado en los modelos de estado. La mayoría son débiles en la emisión tales como reglas de negocios. Esto añade reglas, fuzziness, capas con semántica clara y permite a los diseñadores convertir reglas en aserciones, usando siete actividades:

* Capas
* Objetos
* Estructuras :
    - Uso de estructuras
    - Clasificación de estructuras
    - Composición de estructuras
* Asociaciones y Datos Semánticos
    - Modalidad y multiplicidad
* Atributos
    - Tipos, omisiones, validación, seguridad
* Métodos u operaciones
    - Parámetros y sus tipos, aserciones
* Reglas
    - Reglas de Control
    - Reglas de Negocios
    - Apuntadores

Otra semántica primitiva cuya especialización y composición puede ser requerida para ciertas aplicaciones. El énfasis en los atributos explícitos se eleva más en los sistemas comerciales que en el campo técnico. El énfasis en la semántica y en los modelos de datos se eleva de

consideraciones similares. El método de mantenimiento de excepciones y reglas de negocios presentado en éste texto es esencialmente no procedimental y ello contrasta con el mantenimiento de excepción procedimental de HOOD y OOSD. Las actividades enumeradas evaden la necesidad de ser convenidas como expuestos a iteración. La especificación misma debe ser prototipificada.

El AOO debe incluir técnicas y notaciones que sean semánticamente ricas, es decir, ésto debe direccionar no sólamente objetos idénticos, abstracción y herencia, sino también, las semánticas declarativas de objetos. Graham ha sugerido que las técnicas basadas-en-reglas son apropiadas para éste propósito y muestra como pueden ser notacionalmente incluidos.

Los muchos métodos orientados a objetos examinados representan opiniones, sugerencias, no se considera que representen métodos maduros. Los métodos híbridos basados en ideas sintetizadas de esas sugerencias, métodos convencionales. El modelamiento semántico están emergiendo. Se espera que una racionalización de la posición actual ocurra pronto con una pequeña emersión de guías señaladas. No existe un candidato real que sirva para señalar guias todavía. Como un ejemplo de la confusión existente, considere las muchas notaciones para la clasificación de estructuras en oferta. Sólo unas cuantas de ellas son ilustradas en la **Fig. A.62**

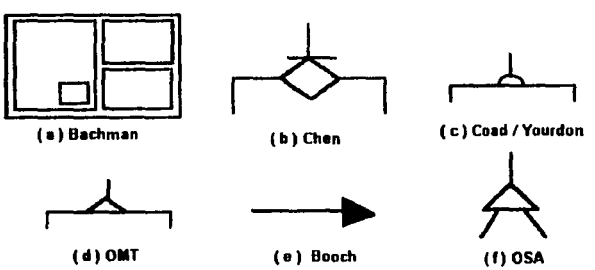


Fig. A. 62 Algunas de las notaciones de clasificación de estructura sugeridas

Los prospectos para el AOO lucen brillantes en el tiempo de escritura. Los beneficios del reuso y extensibilidad son una fuerte fuerza de manejo para generalizar éstas adopciones. Existen actualmente al menos dos distintos estilos de notación, uno derivado del tradicional Booch y representado por HOOD y OOSD y, notaciones del tipo descrito en éste apéndice derivado del modelamiento de datos. Será probable que trabajos futuros hagan una convergencia entre esos

dos estilos como un gran equipo de diseñadores, más que como sistemas comercialmente orientados y analistas de negocios en muchos de los problemas involucrando multimedia, concurrencia y operaciones en tiempo-real. Las herramientas CASE apoyan las notaciones y el método es una necesidad para habilitar a los analistas a usar los métodos efecitiva y eficientemente.

Las herramientas CASE capaces del manejo de los conceptos orientados a objetos están emergiendo. Las bases de datos orientadas a objetos y 4GL's pueden suplantar las herramientas CASE para algunos propósitos, incluyendo la manipulación de configuración.

El AOO puede ser aplicado a estrategias organizacionales tanto como a sistemas de desarrollo.

APENDICE "B"
" DISEÑO ORIENTADO A OBJETOS"

En éste apartado y en el anterior se propone dar un paso hacia atrás a las herramientas, lenguajes y emisiones de implantación en el orden para determinar cualquier método orientado a objetos que tenga un lugar genuino en el análisis y diseño de los sistemas que pueden eventualmente ser escritos convencionalmente, lenguajes de valor-orientado, lenguajes orientados a objetos o híbridos. También se buscará la utilidad y los defectos de muchísimos diseños orientados a objetos y métodos de análisis que han sido sugeridos. Ian Graham opina que debe existir una colocación de requerimientos para una técnica de análisis práctica. Siguiendo el tratamiento histórico Graham lo ha adoptado, este capítulo concentra un diseño orientado a objetos en la medida en que éste puede ser separado del análisis orientado a objetos porque el interés en el diseño precede al interés en el análisis.

Este es un exámen de los muchísimos métodos de diseño orientado a objetos y las notaciones antes del movimiento a un nivel alto de abstracción y marcado con el análisis en el apéndice previo. Estos dos apéndices : el "A" y el "B", son la parte central de ésta tesis. Su propósito es unir muchos tratados que han sido escritos por medio de muchos otros capítulos. Esos temas -de programación orientada a objetos y sus beneficios, sistemas expertos y conocimiento de la ingeniería y del modelamiento de datos y manejo de bases de datos- están unidos en los requerimientos para Ingeniería de Software Orientada a Objetos práctica. La estrategia del problema que posee por la maduración de las tecnologías de programación y bases de datos orientadas a objetos está direccionada por un método de análisis y diseño el cual debe permitir la fase gradual de explotación de los beneficios de la orientación de objetos en el nivel de análisis, así como la firma de fundaciones de capas para la explotación de la maduración del lenguaje y el manejo de las tecnologías de objetos.

En el mundo de la orientación a objetos, las técnicas desarrolladas para el diseño son a menudo útiles para análisis y viceversa. Así que, el lector interesado en algún camino no omitirá estudiar algún otro autor. Además, decidir algún método cualquiera, el cual contribuye a ambas áreas, debe ser señalado bajo análisis o bajo diseño, siendo algunas veces difícil lograrlo y, el resultado de preferencia es arbitrario. Además estos dos apéndicas deben ser leídos juntos.

A.1) ¿ Programación, Diseño o Análisis ?

El desarrollo de la Ciencia de la Computación como un todo ha procedido de un interés principal hacia la programación sola, a través del incremento del interés en el diseño, a un interés actual con los métodos de análisis los cuales empujan su interés en lenguajes dentro del respaldo. Reflexionando, quizá el interés en la orientación a objetos comenzó, históricamente, con los lenguajes desarrollados. El cual es más reciente que cuando los métodos de diseño orientado a objetos emergieron. Los métodos de análisis orientado a objetos están también emergiendo pero no han sido asentados todavía.

Los métodos orientados a objetos cubren los métodos para diseño y métodos para análisis. Algunas veces existe una sobrecarga y, es realmente sólo una idealización decir que existen actividades completamente separadas. Hodgson (1990) argumentó que los procesos de los sistemas desarrollados es uno de comprensión, invención y realización, donde un campo del problema es el primero agarrado o comprendido como fenómeno, conceptos, entidades, actividades, roles y aserciones. Este es comprensión y corresponde enteramente al análisis. Sin embargo, entendiendo el campo del problema también implica simultáneamente la comprensión de estructuras, componentes, modelos computacionales y otros constructores mentales los cuales toman en cuenta campos de solución factibles. Esta actividad inventiva corresponde al proceso de diseño. Por supuesto, la mayoría del pensamiento convencional en la ingeniería de software se escandalizará de que Ian Graham sugiera que el entendimiento de la respuesta precede, a alguna extensión, entendiendo el problema, pero que es precisamente lo que está diciendo Graham. Todos los otros procesos cognitivos proceden en ésta forma; Graham no encuentra la razón por la que la Ingeniería de Software deba ser diferente. Esas consideraciones también entran dentro del proceso de realización donde esas estructuras y componentes de arquitectura son mapeados dentro de compiladores y hardware. Los defensores de la prototipificación cuentan con largos argumentos para fundamentar que es benéfico no hacer una separación rígida entre el análisis, el diseño y la implantación. Por otro lado, las consideraciones directivas y de ejecución dejan serias preguntas acerca de la advisibilidad de la prototipificación en ambientes comerciales. Graham (1991) sugiere un número de formas en las que la prototipificación puede ser explotada pero controlada. En la raíz

de éste debate existen posiciones ontológicas y epistemológicas concernientes a los objetos : ¿ qué son y cómo pueden ser comprendidos ?

Las varias escuelas de métodos estructurados han intentado apropiadamente cualquier convenio como útil de la orientación a objetos y procuran incorporarlo sin una estructura existente. Posiblemente un método sonado en el que pueda encontrarse una transición razonablemente tranquila para las organizaciones ya comprometidas a algún método u otro, pero se aprecia claramente que existen peligros y, algunos campos de métodos han andado ásperos caminos sobre las contribuciones básicas de la orientación a objetos en su ahínco de ser capaces de usar el buzz-word. Algunas firmas consultoras están siguiendo la ruta de incorporación de orientación a objetos dentro de métodos de corriente principal con un aspecto en la obtención de los beneficios de una transición tranquila sin perder la visión de los objetivos fundamentales de la orientada a objetos.

Biggerstaff y Ritcher (1989) han sugerido que al menos la mitad de un sistema típico puede ser construido de componentes de software reutilizable y, que solamente la forma de obtener más avances significativos en la productividad y calidad es elevar el nivel de abstracción de los componentes. Los productos de Análisis o especificaciones son más abstractos que los diseños. Los Diseños son más abstractos que el código. Los artefactos abstractos tienen menos detalles y menos confianza en el Hardware y otras limitantes de implantación. Así que los beneficios del reuso pueden ser obtenidos tempranamente en un proyecto, cuando ellos están comprometidos a tener el más grande impacto. Sin embargo, el menor detalle en un objeto es el menos significativo. Esto nos deja con una pregunta, si el AOO y las técnicas de diseño existen los cuales pueden lograr esos beneficios ahora, dependiendo de la apariencia de mayor madurez, ¿ Puede existir más programación estable de lenguajes orientados a objetos ? Graham lo cree posible. Sin embargo, la pregunta subsidiada de ¿ cuáles métodos de diseño o análisis deberíamos usar ?, es más difícil. Debemos examinar muchísimos métodos propuestos de diseño y análisis criticamente, y sintetizar un método que no servirá sólamente a aplicaciones GUI (Guide User Interface, por sus siglas en inglés) o Sistemas Ada para un amplio rango de proyectos comerciales.

Las casas y consultorías de Software deben estar particularmente interesadas en las especificaciones de reusabilidad y extensibilidad. La razón es pecuniaria, ¿ Qué hace la gente empleada por las casas y consultorías de software para ganarse la vida ? Trabajar con clientes, entender sus negocios y sus requerimientos y ayudarlos a producir soluciones de software a sus problemas. Habiendo logrado con todo ésto valuable experiencia, los consultores entonces van con el próximo cliente y le venden lo que ellos han aprendido, además con honorarios más altos, justificados por un conocimiento extra. Algunas firmas van aún más lejos, entonces intentan encapsular su experiencia en especificaciones funcionales a gusto del cliente. Los Sistemas de

Información BIS, por ejemplo, tienen un producto llamado el "Modelo Hipoteca", el cual es una especificación funcional de una aplicación hipoteca, basada en un número de cada proyecto y, es posible hacerlo a la medida de las necesidades de un cliente particular. El problema es, para BIS al menos, que el modelo hipoteca no puede ser vendido a vendedores de verduras o a fábricas de máquinas de lavado, aún alguna de las entidades, tales como cuentas, pueden aplicarse a todos esos negocios. ¿ Se requiere una colocación de componentes de especificación reutilizables que puedan ser ensamblados dentro de una especificación funcional situable para *algunos* negocios ? El AOO y, una menor extensión en el Diseño, promete liberarlo como una capacidad.

Mezclando terminología, permítanos comenzar con un dibujo vastamente sobresimplificado de los procesos de desarrollo de software o ciclo de vida. Acorde a éste modelo simplificado, el desarrollo comienza con la elicitación de los requerimientos, el conocimiento del campo y finaliza con el exámen y subsecuente mantenimiento. Entre esos extremos ocurren tres actividades mayores : ***especificación y modelamiento lógico (análisis), modelamiento de arquitectura (diseño) e implantación (codificación y exámen).*** Por supuesto éste modelo permite iteración, prototipificación y otras desviaciones, pero necesitamos no considerarlas en éste estado. En la vida real, no obstante de lo que los libros de texto nos dicen, la especificación y el diseño coinciden considerablemente. Lo cual parece ser especialmente verdadero para el AOO y el DOO porque las abstracciones de ambos están modeladas en las abstracciones de la aplicación, preferentemente a las abstracciones apropiadas al mundo de los procesadores y discos. El Diseño puede ser dividido en diseño lógico y diseño físico. En el DOO el estado lógico es a menudo indistinguible de las partes de AOO. Uno de los mayores problemas encontrados con el análisis estructurado y métodos de diseño estructurado es la escasez de sobrecarga o transición tranquila entre los dos. Esto a menudo conduce a dificultades en el trazado de productos de respaldo de diseño a requerimientos originales de uso o productos de análisis. Como se vió en el Apéndice "A", el método adoptado en el AOO y DOO tiende a mezclar los análisis de sistemas con el proceso de diseño lógico, aunque existe todavía una distinción entre la elicitación de requerimientos y análisis y entre el diseño lógico y físico. No obstante, el análisis, diseño y aún la programación orientada a objetos, la filosofía trabajada consistentemente con un modelo conceptual uniforme de objetos de la filosofía fuera del ciclo de vida, *promete* -al menos- superar algunos de los problemas fáciles de seguir asociados con los sistemas desarrollados. Una de las principales razones para ello es la continua representación de la ingeniería de software orientada a objetos al desplazarse del análisis a través del diseño a la programación. Los Analistas, diseñadores y programadores pueden todos usar la misma representación, notación y metáfora de preferencia a tener que usar diagramas de flujo de datos en un estado, capítulos de estructura en el próximo y etc.

A.2) Métodos de Diseño Orientado a Objetos

Los beneficios del DOO son quizá obvios al lector quien ha absorto los señalamientos dados, pero específicamente incluye :

- ***El Diseño requiere cambios que están localizados e interacciones inesperadas con otros módulos de programa.***
- ***El Diseño basado en Objetos es situable para implantación distribuida, paralela o secuencial.***
- ***Las áreas de datos compartidas son eliminadas, reduciendo la posibilidad de modificaciones inesperadas u otras anomalías actualizadas.***

Los métodos de diseño orientado a objetos comparten los siguientes pasos básicos de diseño aunque los detalles varíen de poco a mucho :

- ***Los objetos identificados, sus atributos y nombres de método.***
- ***Establecen la visibilidad de cada objeto en relación a otros objetos.***
- ***Establecen la interfaz de cada objeto y el mantenimiento excepcional.***
- ***Implantan y examinan los objetos.***

Existen pocos métodos de DOO emergiendo, el más viejo de los cuales es probablemente el realizado por Grady Booch, quien en 1986 colocó fuera lo que es esencialmente un método para usar alguna de las características de Ada en un estilo orientado a objetos. Desde éste punto de vista el DOO está basado en el principio de información oculta, no tanto en la composición de funciones (como con los métodos tradicionales) o un método completamente orientado a objetos aprovechando la herencia. Esta encapsulación puede ser implantada usando Ada o módulos de Modula-2, en cualquier área la herencia y el polimorfismo son constructores no naturales en esos lenguajes estructurados jerárquicamente. Sin embargo, el paso de mensajes puede ser, fácilmente simulado usando funciones o llamadas de procedimientos, las cuales implantan el constructor y acceden operaciones de un objeto. Recientemente, Booch ha publicado un método mucho más extendido y mejorado que no se encuentra vinculado a Ada.

El método original de Booch comienza con un análisis de flujo de datos, el cual es usado para ayudar a identificar objetos en la búsqueda de objetos : concretos y abstractos dentro del espacio del problema que definido por los procesos de las burbujas y almacenes de datos en el DFD *(Diagrama de Flujo de Datos)*. Después, los métodos son obtenidos de los procesos de burbuja de los DFD. Un método alternativo pero complementario, primeramente sugerido por Abbott (1983), es extraer los objetos y métodos de una descripción textual del problema. Los objetos corresponden a sustantivos y los métodos a verbos. Los Verbos y Sustantivos pueden ser además subclasificados. Por ejemplo, existen sustantivos propios e impropios y, verbos cuya terminación contempla : haciendo, siendo y teniendo. ***Los verbos haciendo usualmente elevan a los métodos, los verbos siendo clasifican estructuras y los verbos teniendo la composición de estructuras.*** *Los verbos transitivos generalmente corresponden a métodos, pero algunos intransitivos pueden referirse a excepciones o eventos que dependen del tiempo* : por ejemplo, en una frase tal como "la tienda cierra". Este proceso es una guía útil pero no puede ser convenido como algún método corto o formal. La intuición es todavía, lograr mantener el mejor de los diseños. Esta técnica puede ser automatizada, como alguna herramienta de HOOD descrita anteriormente y Saeki et al. (1989) lo muestra.

Por ejemplo, una declaración de requerimientos SACIS transcrita podría contener el siguiente fragmento:

> If un cliente entra a una *tienda* con la intención de comprar un *juguete* para un *niño*, entonces se *aconseja* que debe estar disponible sin un *tiempo* razonable concerniente a la situabilidad del juguete para el niño. Esto dependerá del *rango de edad* del niño y de los *atributos* del juguete, Si el juguete es un *artículo peligroso*, entonces este es insituable.

La mayoría de los métodos descienden del trabajo de Booch el cual usa alguna forma de análisis textual de éste orden. La notación de Booch representa objetos como masas uniformes, como se muestra en la Fig. B.1 Las puntas de flecha indican qué objetos usan los servicios (o métodos) de otros objetos, así como la definición de las relaciones cliente/servidor y del manejo de mensajes.

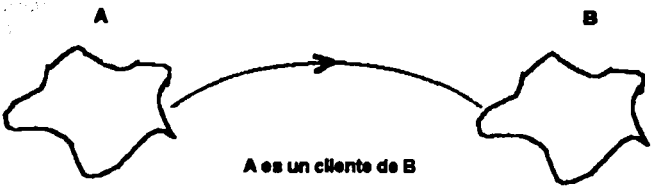
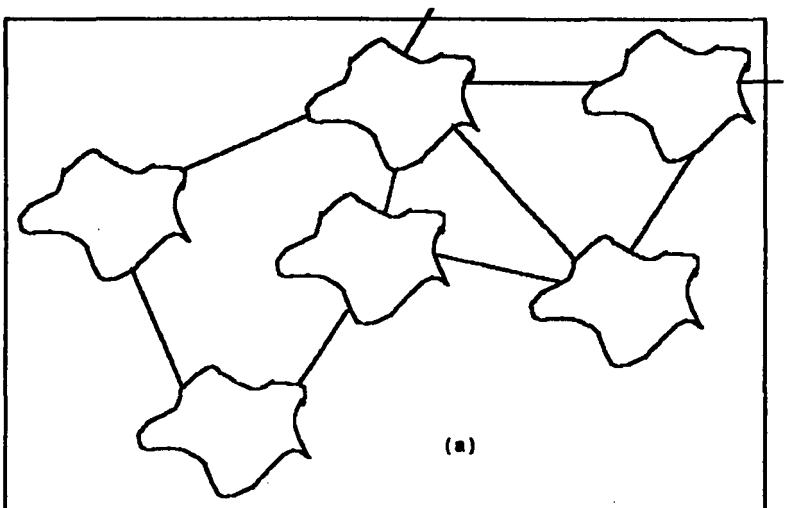
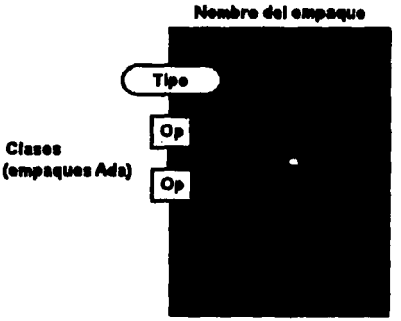


Fig. B.1 Notación Booch para la relación cliente/servidor entre objetos.

Los grupos de objetos son particionados dentro de "subsistemas" de tamaño manipulable correspondiente a los niveles de capas de un DFD, como se dibuja en la **Fig. B.2(a).** Las Clases son identificadas con empaques Ada y dibujados como lo muestra la **Fig. B.2(b).** Booch no encuentra necesidad en denotar los atributos en su notación, dado que todo acceso a las interfaces de objetos es vía un método, u operación aunque Ada por sí misma permite accesar a las estructuras de datos sin una especificación empacada. El área sombreada representa la implantación del empaque o cuerpo de empaque y, un rectángulo no sombreado (o paralelogramo para generalizar) denota la especificación. Los óvalos no sombreados representan un nombre de empaque o tipo de declaración y los rectángulos no sombreados denotan (cada uno) un método.

Los iconos mostrados en las Fig.'s B.2(b) y B.3 se conocen como **Gradygrams.** Booch distingue entre dos tipos de relación cliente/servidor entre objetos como lo ilustra en la Fig. B.3. Si las puntas de flecha comienzan del rectángulo sombreado que representa el cuerpo del objeto. Entonces la implantación de "A" depende de los servicios ofrecidos por "B". Si las puntas de flecha se originan desde el tipo de la declaración oval entonces solamente la especificación (o interfaz) de "A" depende de "B". Esto refleja preferentemente, el interés en el diseño que en el análisis.





(b)

Fig. B. 2 (a) Etiquetas en notación Booch;
(b) Las clases son identifacadas con empaques Ada en notación Booch y sólamente el tipo y los métodos son mostrados.

Sommerville (1989) emplea a ADA como un lenguaje de diseño de todo-propósito. Este es un método común para los autores de generadores de código de un método-base, quien usa un programa Ada como lenguaje de descripción (PDL) como un estado intermedio entre el dibujo o el diseño textual y el código actual. En Ada, las clases son identificadas con tipos de datos abstractos y corresponden a empaques que, en el argot de los programadores de Ada, exporta tipos privados limitados o tipos privados. Las instancias corresponden a instancias de tipos privados o tipos privados limitados o empaques que sirven como una máquina de estado abstracto. Los Métodos corresponden a subprogramas exportados de un empaque de especificación. Como una regla de torpes, si una sola instancia de una clase es requerida esta debe ser definida directamente como un empaque, pero si un número de instanciaciones son necesitadas éste debe ser definido como una clase. Si el estado del objeto necesita ser accesado por otros objetos, debe estar definido como una instanciación de una clase. La herencia es difícil en Ada pero una forma limitada puede ser implantada por medio de tipos derivados o empaques genéricos.

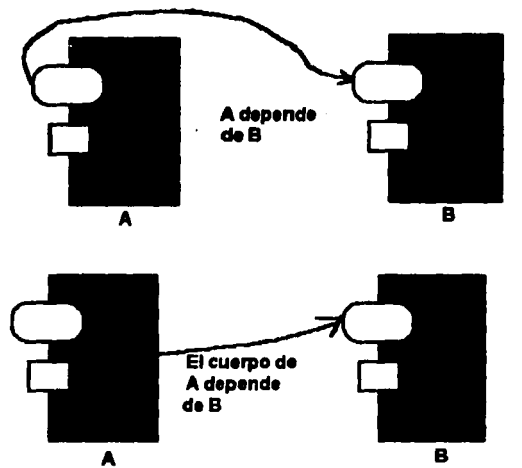


Fig. B. 3 La notación Booch distingue entre clientes que usan la especificación o interfaz de servidores y aquellos que usan su implantación.

Booch parece haber experimentado un tipo de conversión desde los inicios de éste trabajo. Su trabajo más reciente (1991) muestra una perspectiva ensanchada en el DOO, no más largamente vinculadas a algún lenguaje específico tal como Ada. Su notación revisada y método soportan implantación en Smalltalk, CLOS, C++, Ada o la mayoría de los lenguajes basados en objetos u orientados a objetos. Booch también evolucionó un tanto las pautas en el reconocimiento de objeto, el manejo del DOO y los proyectos de programación.

Siguiendo el trabajo pionero de Booch un número de métodos específicamente orientados a objetos han emergido, los cuales son cercanamente similares a sistemas desarrollados que emplean el lenguaje Ada, tal como GOOD, HOOD y MOOD. Veremos dos de ellos. Muchísimos otros métodos de diseño y convenciones notacionales, las cuales no están ligadas a Ada, han seguido de cerca los métodos de Ada. Se verán someramente dos métodos : OOSD y el trabajo de Booch más reciente. Otros tres métodos son discutidos brevemente : OODLE, JSD y un método en la frontera

del Análisis/Diseño, designado por Graham como CRC. Otros métodos y notaciones son discutidos, incluyen un amplio uso en uno hecho por J.A.Buhr (1984), usado en el equipo de trabajo/Ada CASE tool y en el método Texel y métodos influenciados tales como OODLE.

B.3) GOOD

El Método de Diseño General Orientado a Objetos (GOOD) fue desarrollado en la NASA por Seidewitz y Stark (1986). Cubre ambas especificaciones de requerimientos y diseño de los proyectos de ADA. Los procesos de los métodos son como los de Booch, de un capeado preliminarmente de colocación de diagramas de flujo de datos a una identificación de los objetos involucrados. Una mirada en los diagramas de flujo de datos, almacenes de datos, control de interfaz y control de almacenamiento. Las clases son descubiertas por el exámen del flujo de datos y control. El exámen de los procesos principales permite la obtención de un modelo abstracto de la función del sistema y, trazando los flujos de entrada y salida atribuidos a esos procesos, una colocación de capeado de los diagramas puede ser construída. Los diagramas enfatizan el control y las relaciones de datos entre las entidades. Así que las entidades y el agrupamiento de entidad viene a hacer que los objetos y las transformaciones originales de datos vengan a ser sus métodos. Como con el método de Booch y otros métodos derivados de Ada, éste método refuerza una precedencia jerárquica Top-down entre objetos, basado en cómo los objetos usan a cada uno.

B.4) HOOD

La noción de una precedencia jerárquica es tomada de otro método, HOOD. La ***"H"*** en HOOD en efecto, significa ***"Jerárquico"*** (Hierarchical en inglés) -el lector será capaz de inferir el signifcado de OOD- es decir, Diseño Orientado a Objetos, por sus siglas en inglés. Este método es mucho más directo en desarrollo Ada y fue creado en la European Space Agency. Fue directamente influenciado por GOOD y también los dibujos en el método de Máquina Abstracta de Matra Espace y las técnicas originadas en CISI Ingéniere, ambas compañías francesas. Como GOOD éste énfasis composicional (parte-de) de jerarquías, pero no tiene nada que decir acerca de la clasificación de jerarquías (herencia). En HOOD, los objetos son "pasivos" o "activos". *Los objetos pasivos pueden sólamente usar los servicios de otros objetos pasivos pero activa algunos que pueden usar algunos servicios de los objetos.* HOOD es un método Top-down que procede por la descomposición a un objeto de alto-nivel y además descomponiendo los objetos resultantes. En efecto, ***existen dos jerarquías en HOOD : la jerarquía composicional y el uso de la jerarquía.*** El uso de la jerarquía puede ser una red.

Diagramaticalmente, HOOD sigue un estilo Booch, la notación Gradygram. Una notación típica es mostrada en la Fig. B.4, la cual indica cómo la inclusión o composición jerárquica es denotada por una inclusión de icono.

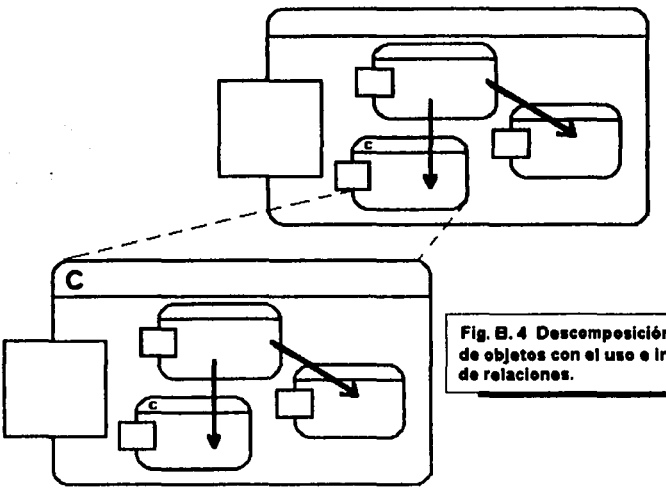
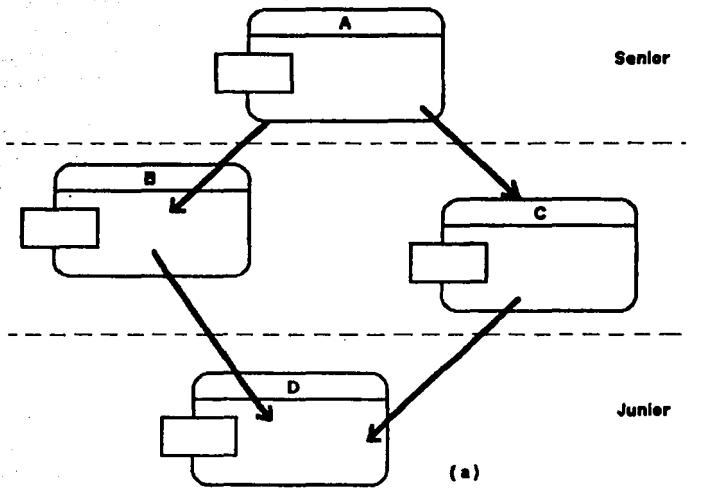


Fig. B. 4 Descomposición HOOD de objetos con el uso e inclusión de relaciones.

La precedencia o uso de relaciones son mostradas por flechas cuyas puntas de flecha apuntan en la parte más junior como en la Fig. B.5(a). La Fig. B.5(b) dá una notación para el "implantado por" liga, lo cual se muestra como una línea punteada. La operación padre "comienza" está implantada por la operación "comienza" del objeto hijo A, el cual delega toda la funcionalidad esperada de "comienza". La liga "implantada por" es una técnica de descomposición muy útil.



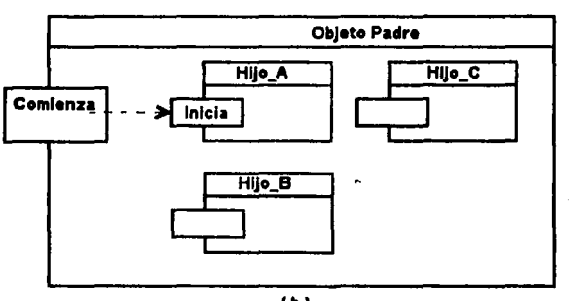


Fig. B.6 HOOD emplea relaciones e implantado_por ligas : (a) La jerarquía con relaciones; (b) El método padre Comienza es empiado para "implantado_por" El método Comienza del Hijo_A el cual suple la funcionalidad de Comienza.

El método HOOD usa un número de pasos para descomponer cada objeto, comenzando con un paso de diseño básico con el cual podría involucrar técnicas de diagramación de otros métodos de análisis y diseño estructurado. Los conceptos así derivados son mapeados dentro de objetos e interfaces externas. Este paso sale del esquema dentro del estado del problema, usualmente como texto, analizando y estructurando los datos. La idea Abbot de usar sujetos y verbos para identificar los objetos y los métodos, es soportada. Como un ejemplo de cómo las técnicas convencionales pueden ayudar en la identificación de los objetos, considerando cómo mapear los conceptos del método de diseño estructurado de Yourdon. Los diagramas de Contexto dan al hardware objetos e interfaz externa. Los DFD's dan tipos de datos abstractos y bancos de datos. El modelo de información da tipos de datos. Los diagramas de transición de estado dan objetos activos y estructuras de control basadas-en-objetos.

El próximo paso es producir una estrategia informal de solución. Esta se descompone en un número de tareas determinadas : un lenguaje natural fuera de línea, ásperos diagramas HOOD y una descripción del nivel actual de abstracción. El tercer paso es formalizar la estrategia de solución. Esto es hecho en los estados para identificación y descripción de los objetos y sus operaciones, agrupándolos y produciendo diagramas HOOD mostrando las relaciones padre-hijo y las operaciones (la composición jerárquica), uso de jerarquías (cliente/servidor o precedencia), ligas "implantados por", flujo de datos y excepciones. Ultimamente, algunas decisiones de diseño pueden ser justificadas. Es también prudente registrar ligas a las especificaciones originales de requerimientos para diseños futuros.

Una nota en la terminología es necesaria en éste momento. HOOD llama ***"operaciones" de métodos y ligas "implantados por"***, a la necesidad de las operaciones en objetos padres para llamar operaciones en un objeto hijo.

Ahora la estrategia de solución está formalizada, el próximo paso es formalizar la solución misma. Lo cual consiste en el refinamiento de la definición del bosquejo del objeto (ODS, por sus siglas en inglés) de los objetos padre para definición de los tipos, constantes y datos, refinando el ODS de objetos temimales para la definición de sus internos y la estructura del control de operación (OPCS, por sus siglas en inglés). verificando las inconsistencias, produciendo la documentación y generando y examinando el código Ada.

Como se ha visto, el uso de relaciones se indica por flechas. Las excepciones son denotadas por un barra a través de la flecha, como lo muestra la **Fig. B.6**, pero no son detalladas explícitamente. Esto es, la dirección del flujo de la excepción no es mostrada (está implícitamente al reverso del uso de la flecha) y la excepción es detallada en relación al objeto como un todo,

preferentemente que atribuida a un parámetro o flujo de datos. En éste ejemplo, si la temperatura excede algún límite, o un descenso en el sensor de temperatura ocurre, la excepción "recalentado" puede ser elevada.

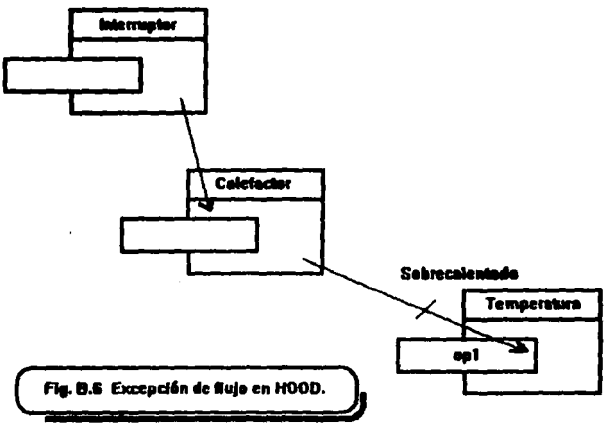


Fig. B.6 Excepción de flujo en HOOD.

Los flujos de datos pueden también ser registrados usando la notación ilustrada en la Fig. B.7, la cual está basada en estructura de capítulos (Yourdon y Constantine, 1979).

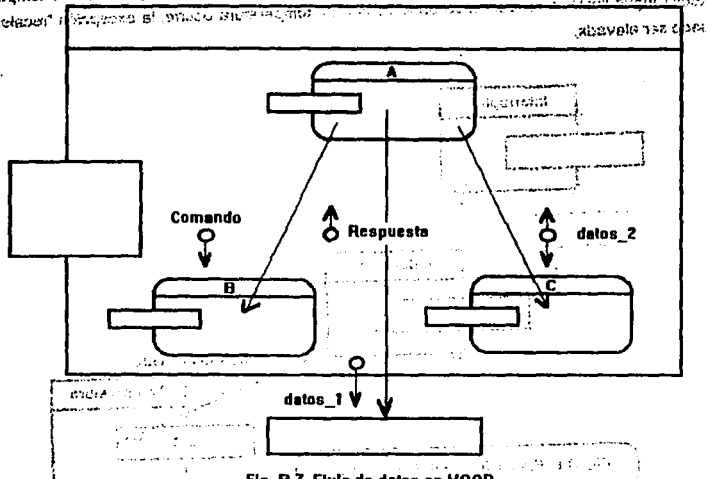


Fig. B.7 Flujo de datos en HOOD

Esta ha sido una descripción muy breve del método HOOD. Una de las mayores atracciones en HOOD es que es un método publicado y soportado en el campo público.

Los criticismos comunes de HOOD están identificados por Hodgson (1990) como sigue :

* No existe soporte para la genericidad, herencia y polimorfismo. Los genéricos son comúnmente usados por los programadores Ada, de tal modo que constituye una seria deficiencia aún para un método basado en objetos.

* Existe insuficiente separación de la definición de un objeto de éste uso y poco soporte para reusabilidad.

* El formalismo gráfico es : incompleto e inconsistente. Los diagramas HOOD no muestran explícitamente cuáles operaciones son usadas por qué objeto, qué flujos son definidos en cada operación, cómo las excepciones están propagadas y qué datos están encapsulados por los objetos. La notación representa subprogramas como objetos y control de estructura, mostrando

como un subprograma que pasa el control a la tarea. El objeto padre es representado con la misma notación como un objeto hijo.

* Existen dificultades con la precedencia jerárquica estricta padre-hijo. En los sistemas de tiempo-real, los eventos y excepciones pueden afectar la ejecución de un sub-objeto encajado profundamente y este es ineficiente al paso de control por medio de muchos objetos encerrados.

Adicionalmente, mucha gente siente que HOOD, resulta apropiado para muchas aplicaciones militares en tiempo-real, pero es menos comercial que algunos. Además de la falta de soporte para la herencia hace que este objeto-base sea visto más bién como un objeto-orientado. Así que, el reuso es soportado pero no extensivamente. El rol de la estructura de datos (atributos entidad) es tocado en favor de la concentración en abstracción funcional. En el próximo apartado, examinaremos una notación diferente, la cual encuentra parcialmente algunos de esos atajos.

B.5) OOSD

Algunas de las críticas listadas anteriormente son señaladas con otro método de diseño. El Diseño Estructurado Orientado a Objetos (OOSD, por sus siglas en inglés), introducido por Wasserman, Pircher y Muller (1990). OOSD es, estrictamente hablando, una notación a la cual las reglas metodológicas pueden ser añadidas. Esta notación es probablemente la más cercana de todas a los métodos encontrados tan lejanos al espíritu de la orientación a objetos, en el caso de que soporte herencia tan bién como la abstracción. Aunque es mucho más cercano a los principios orientados a objetos, todavía produce las marcas de la influencia de Booch y las ideas de Ada. También provee una transición más gradual para los desarrolladores familiarizados con el diseño estructurado, en el cual éste se encuentra parcialmente basado.

OOSD es una notación no-propietaria para el diseño de arquitecturas que combina el diseño estructurado top-down y el diseño orientado a objetos. Se encuentra propuesto en las metas generales orientadas a objetos tales como el reuso, modularidad, extensibilidad y la representación de la herencia y la abstracción. También propone representación de soporte visual de interfaz entre los componentes del diseño, la generación del código, la independiencia del lenguaje, la comunicación entre diseñadores y usuarios y, una variedad de métodos. ! Ambiciosas propuestas !

OOSD una vez más usa una notación derivada de la de Booch pero también influenciada por los capítulos estructurados (Yourdon y Constantine, 1979) y el soporte de concurrencia se basa en la noción de monitores de Hoare (1974). Las clases son mostradas como rectángulos con pequeños

rectángulos superimpuestos denotando sus métodos. Los atributos no son explícitamente mencionados en la literatura OOSD pero la ostensibilidad privada del almacenamiento de bancos podría ser usada por éste propósito notacional. Los atributos se encuentran implícitos en los flujos de datos, los cuales están indicados por flechas originadas en círculos no rellenados. Esas flechas están enumeradas explícitamente como lo muestra la Fig. B.8, en una notación similar a la estructura de los trazos. En COSD los paréntesis cuadrados denotan indirección o parámetros como los usuales.

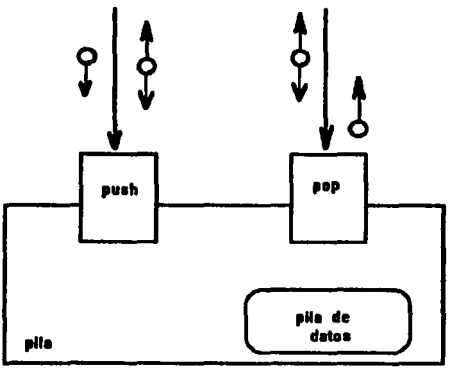


Fig. B.8 Un objeto OOSD representando una pila

Las relaciones cliente/servidor son mostradas por unas flechas gruesas entre los objetos. Si una de esas flechas tiene una salida de flujo de datos como en la Fig. B.9, éstas indican que el objeto cliente (guarda_estado) instancía al objeto.

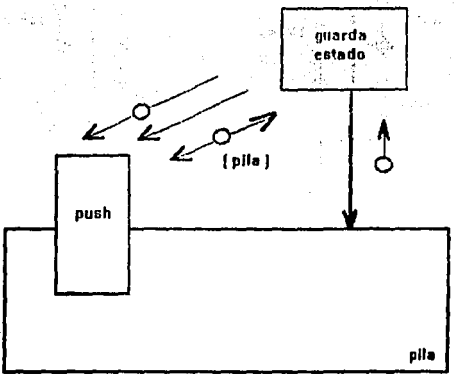


Fig. B.9 La clase guarda_estado instancia 103 objetos de tipo pila y puede usar éstos métodos

HOOD y OOSD probablemente hacen escasas declaraciones explícitas de excepciones . La convención notacional está en áreas en forma de diamante sombreadas y superpuestas al cuerpo del objeto. La excepción del paso de parámetros es mostrada en la misma notación como la que se usa para el flujo de datos pero con diamantes rellenos en la flecha colindante. **(Fig. B.10).**

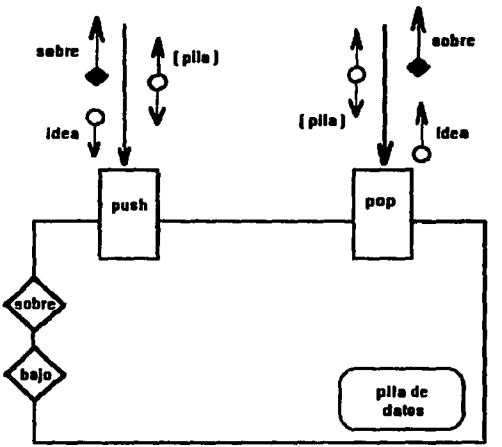


Fig. B.10 Excepción, manejo de una pila

Las operaciones ocultas se muestran en la Fig. B.11, donde "*está completo*" es usado por el método de *empuje* visible externamente.

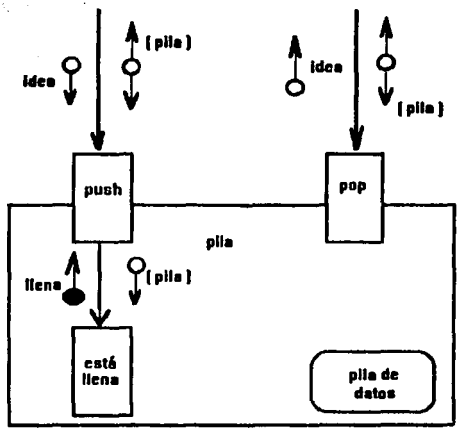


Fig. B.11 Mostrando operaciones ocultas

Las clases genéricas son señaladas en la misma forma, pero el cuerpo de las clases rectangulares fuera de línea es representado como una línea punteada. La herencia entre clases es indicada por una flecha punteada y la herencia múltiple es permitida. Los procesos concurrentes o asíncronos son satisfechos por monitores, los cuales son mostrados con paralelogramos como en la **Fig. B.12.**

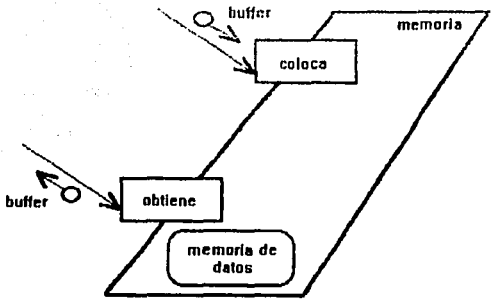


Fig. B.12 Un monitor de memoria

Esta noción de un monitor es como la de una clase, pero los datos son compartidos entre varios métodos. En el caso ilustrado, más que un proceso (como con los interruptores de red) puede colocar u obtener datos. Esas operaciones necesitan acceso exclusivo al buffer de datos oculto.

Si el lector requiere de un profundo entendimiento puede consultar el periódico Wasserman et al. (1990), del cual éste resumen fue ampliamente tomado.

OOSD no es tanto un método, en general es una notación que soporta métodos de diseño orientado a objetos. El usuario de OOSD puede añadir al diseño reglas de acuerdo al método particular en uso. Donde esas reglas de diseño son explícitas, pero ello podría ser fácilmente formalizado en donde las reglas son vagas, tal como la insistencia de HOOD de que los objetos deben tener "un bajo fan-out", se encuentra involucrado el juicio de los diseñadores, quien es requerido dependiendo no obstante del uso de la tecnología de sistemas expertos para éste propósito. OOSD está intencionado a ser un lenguaje-independiente y no está relacionado a Ada o algún otro lenguaje. Esto significa que alguna de las nociones de OOSD no será explícitamente soportada por el lenguaje destino. Las clases implantadas son mucho más fáciles en C++ que en FORTRAN, aunque es posible tenerlas en ambos. De nuevo, los lenguajes tales como FORTRAN que no soportan la concurrencia hecha, es difícil implantar diseños que incluyan monitores. Igualmente, alguno de los detalles de lenguajes no son capturados por OOSD. Por ejemplo, no existe equivalente a tipos privados limitados de Ada o funciones virtuales de C++ o amigos. Otro

punto importante a notar es que OOSD es más bién una notación para diseño de arquitecturas que un diseño físico detallado . Esto significa que ofrece poca ayuda con el agrupamiento físico de archivos en discos, el hardware más situable, encontrando los niveles de servicio o el uso óptimo de memoria.

Otros puntos notables concernientes a OOSD son que ésta listo para que los desarrolladores lo acepten, los ya familiarizados con el diseño estructurado y es situable para los sistemas en tiempo-real del concepto monitor. Las flechas en un diagrama OOSD, por ejemplo, son interpretadas exactamente como en un trazo estructura. Lo cual hace una distinción entre la definición de un objeto y más tarde referencia a éste por otros objetos. Es decir, significa que la notación no requiere todos los detalles de un objeto a ser desplegado cada vez que éste es usado. El protocolo para los métodos impuestos en la herencia múltiple es también significativo. Si un método puede ser heredado en dos formas, debe tener claro qué camino es preferido o impuesto. Por ejemplo, en el campo de la Geometría, la clase de objetos geométricos puede tener subclases "figuras regulares" y "sólidos". Un sólido regular está en ambas clases. Ahora, todos los objetos geométricos tienen un método llamado "rotación invariante", pero para los sólidos generales es un método poco menos complicado que para las figuras regulares. Así que los sólidos regulares deben heredar un método designado como "rotación invariante de figura regular" o impuesto con su propio método, de acuerdo a los requerimientos del problema. En OOSD, los métodos pueden ser añadidos a una subclase o impuestos, no pueden ser eliminados.

OOSD es uno de los híbridos más aventajados, cuyas notaciones de diseño orientado a objetos son de bajo-nivel. Pareciendo ésto poco prometedor de ser extensible a una notación de análisis consistente, debido a la dificultad de señalamiento con números grandes de métodos y la ausencia de una forma de señalamiento con estructuras de datos y atributos muy complejas . Por otro lado, es más situable para la arquitectura o diseño lógico que para el diseño lógico.

B.6) JSD y OOJSD

El Diseño Estructurado de Jackson (JSD) (Jackson, 1983) es un objeto-base más que un método completamente orientado a objetos. Los modelos JSD son descompuestos en términos de eventos o acciones y sus dependencias en tiempo. Sin esos eventos el método JSD primero define los objetos. El próximo paso es construir una especificación en los términos de la comunicación de los procesos secuenciales que pueden accesar cada estado de otro -violando así el principio de información oculta. El método puede ser útil si el diseño orientado a objetos es implantado en un lenguaje convencional.

Un número de similitudes entre los métodos JSD y diseño orientado a objetos puede ser identificado. JSD contiene técnicas útiles para la entidad e identificación de métodos en su etapa de modelamiento. También, las técnicas de análisis ordenando-tiempo de JSD pueden ser convenidas como un significado de clases documentadas. JSD y el DOO usan el concepto de objetos similarmente por medio de sus terminologías que son diferentes. JSD puede dar guías en qué objetos y operaciones son relevantes a un problema. Especialmente donde el tiempo de ordenamiento de los eventos es importante. De otra manera, las ligas son lejanamente ténues.

Jackson ha declarado el interés que tiene con el problema de que la herencia viola la encapsulación y además la reusabilidad. Uno podría esperar además que JSD involucrará un objeto base más completamente, no por medio del estilo orientado a objetos. Las ideas de Jackson han sido extremadamente influenciadas en el desarrollo de otros métodos orientados a objetos.

Más recientemente, las versiones orientadas a objetos de JSD han comenzado a emerger. En una forma similar, los intentos han sido hechos para crear extensiones orientadas a objetos de métodos convencionales que usan estilo-JSD técnicas de vida historia-entidad (ELH) tales como SSADM donde, por ejemplo, los diagramas correspondencia efecto son sugeridos como un significado de visibilidad mostrada. La sugerencia más extrema tiene aún el reclamo que la herencia no es nada más que una relación uno-a-muchos y puede así ser acomodada sin las técnicas del modelamiento de datos SSADM.

Todas las marcas de los métodos son extremadamente débiles en el área del modelamiento semántico de datos y, ninguna de las técnicas de diagramación permite la descripción detallada de los atributos de los objetos. Para corregir ésto, primero es necesario examinar el último pensamiento de Grady Booch.

B.7) Booch '91

El método de diseño revisado de Booch (1991) y la notación consisten de cuatro actividades mayores y seis notaciones. El primer paso señala con los aspectos estáticos de los sistemas, ambos lógico y físico. Los dinámicos están señalados con el uso de técnicas existentes de transición de estado y diagramas de tiempo. Esquemáticamente, puede ser mostrado como :

Estructura lógica

Diagramas clase

Diagramas objeto

Estructura física

Diagramas módulo

Diagramas proceso

Dinámica de clases

Diagramas de transición de estado

Dinámica de instancias

Diagramas de tiempo

Booch sugiere que la interpretación lingüística del estilo-Abbot, técnicas convencionales de análisis estructurado o análisis orientado a objetos son del todo precursores situales al diseño orientado a objetos.

Ambas clases e instancias son mostradas como formas menos uniformes, pero las clases tiene una línea punteada. Si esos formas tienen sombras, como en la Fig. B.1, ésto indica que ellos denotan subprogramas libres, los cuales algunos lenguajes permiten.

Una notación lejanamente rica de las relaciones entre las clases es añadida, la cual usa diferentes tipos de líneas para indicar el uso, herencia y otras relaciones. La **Fig. B.13** muestra esos símbolos. Booch recomienda una forma de apeado, así que las clases son organizadas dentro de "categorías" conteniendo muchísimas clases relacionadas. Esas capas son mostradas como rectángulos. Las flechas entre esos rectángulos denotan visibilidad o uso de relaciones.

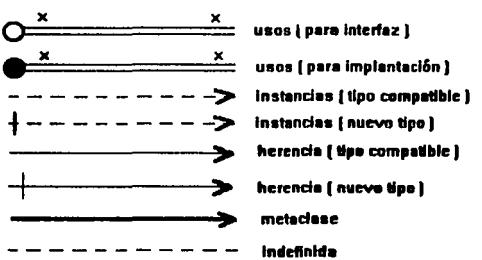


Fig. B.13 Diagrama relacional de iconos, clase de Booch

Cada clase, en el método Booch, es descrito llenando un comportamiento standard el cual incluye identidad, atributos, métodos y la siguiente información extra diseño.

Documentación (texto)
Visibilidad (exportada/privada/importada)
Multiplicidad (0/1/n)
Superclases
Clases usadas
Parámetros genéricos
Implantación de interfaz (pública/protegida/privada)
Diagrama de transición de estado
Concurrencia (secuencial/blocking/activa)
Persistencia (estática/dinámica)
Complejidad de espacio (texto)

Nótese que existen algunos datos semánticos presentes pero no procesos semánticos, aunque el diagrama de transición de estado asociado con cada clase puede representar ésta información. Algunos items, tales como concurrencia y complejidad de espacio, son relatados con poco diseño físico bajo-nivel y, sus detalles no son relevantes a los argumentos presentados aquí.

Los diagramas de clase y objeto y su comportamiento asociado describen el diseño lógico y estático de un sistema. El diseño físico puede diferir del lógico. Por ejemplo, los datos pueden ser agrupados para acceso eficiente, o los procesos agrupados de acuerdo a su uso relativo para prevenir caer en discusiones prolongadas. Para enforzar ésta distinción, Booch distingue entre clases y módulos, los módulos correspondientes a segmentos de programa, podrían estar compilando funciones separadamente en C++ o empaques complejos de Ada. La notación para módulos está basada en la notación temprana de Gradygram mostrada en la **Fig. B.2 (b)**. Los diagramas de procesos son realmente simples diagramas de bloque, muestran las relaciones de comunicación entre los dispositivos físicos y los procesadores.

Lo dinámico de un sistema también debe ser descrito, se acompaña en dos formas. Los diagramas de transición de estado muestran los dinámicos de las clases, lo cual es una técnica compartida con muchísimos de los métodos orientados a objetos discutidos en el próximo apartado. Los dinámicos nivel-instancia son mostrados por diagramas de tiempo prestado del campo del diseño de hardware. Esos diagramas de tiempo muestran los métodos de cada instancia iniciando y terminando en relación a cada otro, como se observa en la **Fig. B.14.**

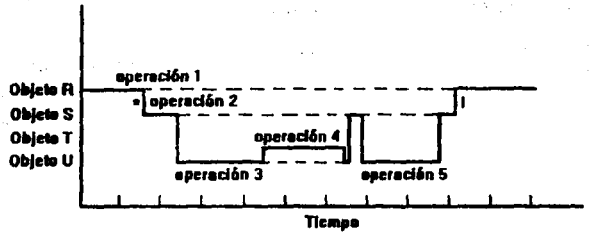


Fig. B.14 Un diagrama de tiempo usado por Booch. El asterisco indica la creación de una instancia y el signo de admiración, su destrucción.

El problema con los diagramas de transición de estado es que, mientras ellos pueden ser finos para los sistemas con un número pequeño de estados -como con los controladores- ellos son imposibles para los sistemas con largos números de estados, o aún estados continuos. Un objeto con "n" atributos Booleanos pueden tener "2n" estados. La mayoría de los sistemas de objetos comerciales tienen muchísimos atributos no-Booleanos. Por ésta razón, es necesario enfocarse en estados que otorgen logros a los cambios de datos significantes a los negocios. Esto significa que los estados y sus cambios relatados deben ser aprendidos al mismo tiempo. El trabajo de BIS, presenta la capacidad de estado de transición y diagramas de entidad de historia de vida, o el cambio en la notación de estado, es preferible porque los habilita tanto a los estados como a los eventos a ser percibidos una vez.

Los diagramas de tiempo, por razones similares, pueden solamente ser manipulados para sistemas razonablemente pequeños, dependiendo de la complejidad de la interacción entre los procesos. El método *Objectory* ofrece un método similar para el mantenimiento de lo dinámico de las instancias. Un método diferente es descrito en la subsección correspondiente a Ptech.

El método Booch es uno de los mejores métodos de diseño trabajados y, es superior a GOOD y HOOD, no siendo vinculado a Ada y teniendo una noción mucho más general de estructura. De nuevo la sabiduría convencional, esas rica propiedades semánticas del método significan que sobrecarga el territorio del analista tan bién como el del diseñador. Se justifica por el método incremental a la ingeniería de software, enfocada por Booch para una prototipificación, el análisis y el diseño se encuentran frecuentemente entretejidos. La principal debilidad del método de Booch y, este tiene ésto en común con otros métodos de diseño orientado a objetos, es que las

dinámicas globales son identificados. Además, no existen serios atentados a señalar con las reglas de negocios y otro tal proceso-semántico aspectos de especificación.

Uno de los problemas centrales de la Ingeniería de Sotware es la copia con la complejidad. Los métodos de diseño orientado a objetos direccionan precisamente ésta emisión. Los modelos Waterfally otros métodos burocráticos o "libros de cocina" de la construcción de sistemas caen en el reconocimiento de que la manipulación de la complejidad requiere gran flexibilidad. Los Risk-driven y los métodos incrementales reconocen la necesidad del sonido para métodos adaptativos. Booch recomienda lo que él llama "round-trip gestalt design", que es un proceso de diseños completos incrementales en diferentes niveles de abstracción y refinamiento, procediendo estrictamente de arriba-hacia-abajo o de abajo-hacia-arriba. Estos procesos proceden confusamente en el siguiente orden :

- ***Identificación de clases e instancias.***
- ***Define sus semánticas.***
- ***Encuentra relaciones entre ellos.***
- ***Implanta el diseño como un prototipo.***
- ***Examina el sistema para cohesión y consistencia.***
- ***Redefine clases, instancias, semánticas y estructuras en base a lo que ha sido aprendido.***

Los procesos se detienen cuando se cree que todas las claves de abstracciones y funciones han sido definidas y, es altamente no-lineal. Definiendo unos métodos de objetos e interfaz pueden afectar el protocolo de otro objeto o una clasificación de estructura. Igualmente, el descubrimiento de una estructura puede dejar a la percepción de nuevos objetos. Ultimamente, hemos notado lo obvio : ***nunca existe un mejor u óptimo diseño. Los diferentes diseñadores producirán diferentes modelos, los cuales pueden ser igualmente situables para su propósito.***

Este método hace incapié en que la identificación de los objetos y sus clases es una decisión de diseño lógico independiente de su organización dentro de módulos, la cual es una decisión física de diseño.

El método de Booch distingue clienta/servidor, antiguedad, o relaciones de uso y contención o relaciones padre-hijo. En el nivel de la clase, la contención tiene dos sentidos : clasificación (el concepto está contenido) y composición (la cosa está contenida). El método de diseño Booch hace la distinción pero no otorga ninguna guía en cómo esos dos tipos de estructura están interrelacionados.

El uso de las relaciones define el control de la estructura de un sistema, o su topología en el paso de mensajes. Por ejemplo, el significado de haciendo es un paso adelante para los métodos de diseño orientado a objetos, pero la práctica de explicación de todo un mensaje no es una buena notación la cual sintetice la estructura de control, o flujo de control, en una forma simple y suficiente para aprensión directa de la complejidad de los diseños y fácil comparación entre más o menos diseños complicados.

Booch distingue tres roles para los objetos. Los actores u objetos activos pueden inicializar su comportamiento en otros objetos pero no actúan sobre y, los servidores pueden sólamente estar operados sobre otros objetos. Los Agentes hacen ambas cosas. Desde el punto de vista de Graham, la relación cliente/servidor es mejor visualizada como una noción relativa. Un objeto puede ser un servidor en un aspecto pero un actor en otros.

En éste método y, en general, existe un trueque entre el uso y la contención de estructuras. La contención de un objeto (es decir, teniendo éste como un valor de atributo) reduce el número de objetos que son visibles al encierramiento del objeto. Observando, la contención restringe el fácil uso de un objeto por otras partes del sistema y puede comprometer su reuso potencial. Esta observación apunta a una importante decisión de diseño . La granularidad de los objetos es una de las claves de decisiones de diseño y, como se ha indicado, los principiantes tienden a obtenerla en forma incorrecta, también tienden a hacer la granularidad fina.

El método Booch es experimentado, además su desarrollo y una versión mejorada son anticipados para 1993 y ansiosamente esperados por muchos.

B.8) OODLE y el Diseño Recursivo

El método Shlaer/Mellor de Análisis y Diseño Orientado a Objetos es descrito en Shlaer y Mellor (1992). En éste apartado examinaremos dos de estos componentes de diseño orientado.

OODLE (Object Oriented Design LanguegE) es un componente de diseño específico del método Shlaer/Mellor de quien el método de AOO es descrito en el Apéndice "A". Prescribe cuatro tipos de diagrama interrelacionados por un esquema etiquetado para ayudar con la documentación y potencial automatizado. La notación es vista como un lenguaje-independiente, Graham piensa que existe una pequeña sugerencia de Ada en la notación y el soporte para los amigos es reminiscente de C++. Los tipos de diagramas son como siguen :

* **Diagramas de Dependencia** muestran el uso (cliente/servidor) y las relaciones amigables entre las clases.

* **Diagramas de Clase** muestran el aspecto externo de una clase en una manera similar a Buhr (1984).

* **Trazos de la estructura Clase** muestran la estructura de los métodos de una clase y el flujo de datos y el control.

* **Diagramas de Herencia** representan herencia.

La **Fig. B.15** muestra las tres notaciones. En el diagrama de dependencia de flechas sólas denota mensajes mientras la línea doble muestra las violaciones de encapsulación. En el diagrama de Clase, la caja hasta arriba es el nombre de la clase y los otros rectángulos son éstos métodos. Los hexágonos dibujan el icono y conectan a un método, además muestran el paso de parámetros. El dibujo dentro del icono representa estructuras de datos ocultas. En los trazos de estructura de clase, los rectángulos denotan métodos y los hexágonos el paso de parámetros entre ellos en muchas de las formas de OOSD. OODLE es una notación más rica que la de los diagramas simplificados, con el soporte para la representación de cada cosa como polimorfismo, las excepciones y etc.

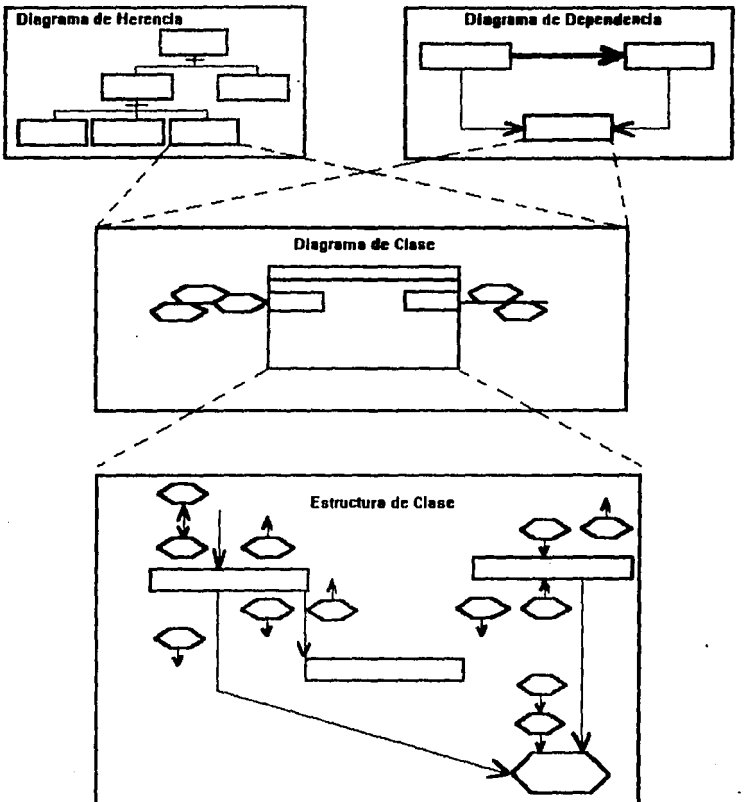


Fig. B. 15 Diagramas OODLE simplificados y sus relaciones

DISEÑO RECURSIVO

Muchísimos problemas han motivado la idea del diseño recursivo. A menudo las clases no se conservan juntas bién. También, se considera el hecho de que el reuso está basado sólamente en interfaces externas, los diferentes estilos de implantación a través de diferentes objetos dejan las dificultades de mantenimiento. ***El Diseño debe ser recursivo no tanto iterativo.*** Debe ser realizado usando un método con la semántica precisa cuando sea posible. El diseño recursivo es un diseño abstracto, un principio de diseño general, no un proceso de diseño en sí mismo.

La idea básica, es que la ejecución y mantenimiento de un sistema debe ser direccionado por la aplicación de un conjunto general de reglas a través de todos los módulos de código. Existe cordura en ésto y contrasta con la mucha existencia práctica en los proyectos orientados a objetos. Sin embargo, podrían ser exactamente casos especiales. El desarrollo recursivo requiere que la aplicación sea separada de varios campos reutilizables en diferentes niveles y que las técnicas formales sean constructibles entre los campos en diferentes niveles. Por ejemplo, nosotros podríamos construír una técnica estandar entre el bajo-nivel de diseños IS y en un sistema de operación particular. El campo clave es la "arquitectura de campo" la cual aisla los campos de aplicación de implantaciones particulares, como lo ilustra en la **Fig. B.16.** Curiosamente ésto es exactamente lo que los diseñadores de NeXStep y la Máquina Taligent están intentando hacer a nivel de máquina. HOOD ofrece un puente, pero sólamente a una arquitectura, que Ada ejecuta. Esto no es general ni suficiente para la mayoría de los propósitos.

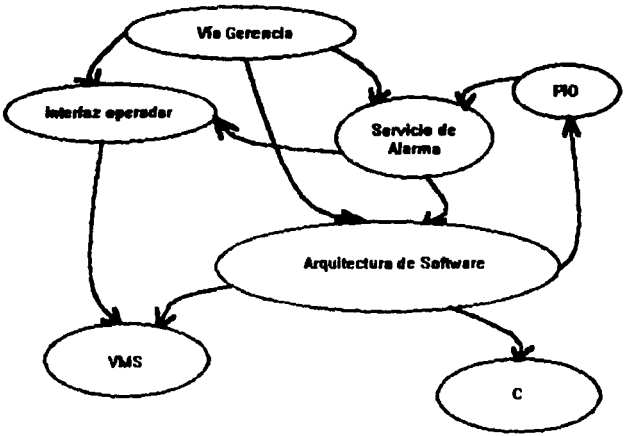


Fig. B. 16 Un típico Shlaer/Mellor

Ultimamente, el desarrollo recursivo puede ser contrastado con desarrollo iterativo, como lo muestra en la **Tabla B.1**

Iterativo	Recursivo
HAZ PARA CADA requerimiento	HAZ PARA CADA requerimiento
análisis	análisis
diseño	ejecuta modelo de análisis (exámen)
código	FIN HAZ
ejecuta código (exámen)	define reglas de diseño
FIN HAZ	exámen reglas
integra software	aplica reglas al código "generado"

Tabla B. 1 Diseño Iterativo vs Diseño Recursivo

De ésto puede ser deducido que el tiempo para todos los estados de desarrollo iterativo es proporcional a la complejidad de los requerimientos, donde el diseño de desarrollo recursivo y los

costos del código son fijados y, después una inversión inicialmente alta reusa los beneficios que se acumulan para cada proyecto sucesivo.

El término "generado" aquí podría referirse al código generado de las reglas. Además y, en la opinión de Ian Graham, los principios de diseño recursivo podrían justamente estar listos para ser aplicados al desarrollo convencional. El método Shlaer/Mellor enfatiza la evolución de los profesionales actuales sobre los revolucionarios métodos orientados a objetos.

B.9) CRC y RDD

Un método responsablemente-manejado de análisis y diseño usando tarjetas de clase, responsabilidad y colaboracion (CRC), de Beck y Cunningham (1989) y descritos completamente en Wirfs-Brock et al. (1990) es útil para la documentación de los diseños orientados a objetos y también para la enseñanza de los conceptos básicos. Esta técnica es a menudo conocida como RDD (Diseño de Responsabilidad-Manejada). Tiene la singular ventaja de usar nada más expansivo que una caja de tarjetas índice como una herramienta CASE, aunque originalmente la idea fue usar un sistema hypertexto. La tarjeta se encontró más efectiva en la práctica.

El método CRC asume la existencia de una especificación de requerimientos escritos y procedidos usando un análisis textual para nombrar los objetos clave. Para cada objeto representando una clase, una tarjeta está preparada conteniendo el nombre de la clase y listas de superclases y miembros. Después, un análisis textual encuentra las "responsabilidades" o métodos requeridos por la clase. Esos métodos son además refinados por la examinación de tres tipos de estructura : clasificación, composición y, analogía de estructuras. El uso de la estructura está determinado como una lista de clases con la indicación cada clase "colabora" con y, son adicionadas a las tarjetas. Una vez más la composición de estructuras es examinada para identificar el uso, por delante con las relaciones "tiene conocimiento de" y "depende de". La colaboración ayuda con la granularidad. Las estructuras son ahora analizadas para diferenciar entre las clases abstractas y concretas y una notación teórica es usada para dibujar los métodos compartidos. Las clases son también organizadas dentro de etiquetas, o subsistemas, los cuales son asignados a relaciones contractuales con otros subsistemas y listas de tareas delegadas. Finalmente, todos los componentes -clases, métodos, subsistemas y contractos- son designadas en detalle.

Este método enfatiza las responsabilidades, o métodos, sobre los atributos o datos, para ayudar a deferir como muchas decisiones de implantación son posibles, pero todavía reconoce la

existencia de los atributos. Esto es usable en ambos niveles : de diseño y de análisis y, la notación aside, puede ser convenida como una colocación valuable de guía de líneas, las cuales pueden ser adicionadas a otros métodos de análisis orientado a objetos. Sólo ayuda en la identificación de objetos o postcondiciones, las cuales ayudan con algo del control de estructura, pero existe poco soporte para las reglas de negocios y semáticas funcionales.

Formalmente, las responsabilidades están divididas dentro atributos que representan responsabilidades para el conocimiento, o el estado del objeto, y responsabilidades para haciendo, o las operaciones al objeto pueden ejecutar. Las colaboraciones no son bién mantenidas para un servidor, ayudar a un cliente con la responsabilidad y en la constitución de la visibilidad o uso de relaciones en un nivel detallado. Las asociaciones gnerales no son bién mantenidas por CRC. Un contracto es la colocación de mensajes que un cliente puede enviar a un servidor y constituír el uso de relación de alto nivel. Los contratos son algunas veces usados para grupos relacionando responsabilidades.

La **Fig. B.17** muestra un tarjeta típica CRC y un tarjeta de un subsistema. Listar los miembros o subclases no es una buena idea desde el compromiso de encapsulación y el reuso.

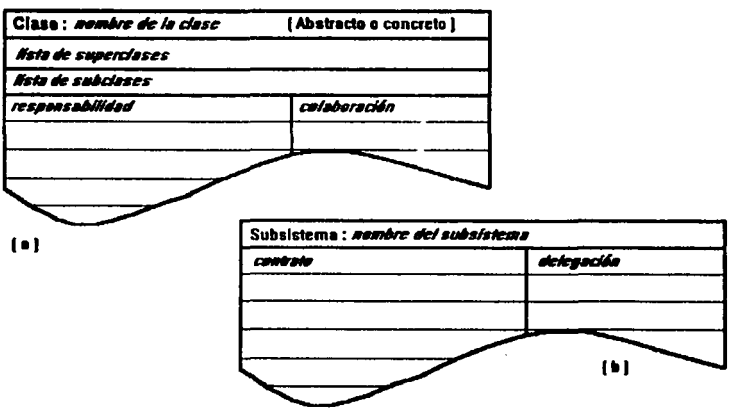


Fig. B. 17 [a] Tarjeta de clase CRC ; [b] Tarjeta de subsistema CRC

Las colaboraciones pueden ser mostradas gráficamente usando una gráfica de colaboración, empleando la notación mostrada en la Fig. B.18. El método usa esas gráficas para extraer subsistemas coherentes basados en contenciones y agrupaciones contractuales. Para sistemas triviales esas gráficas vienen a ser muy ciertas y, mientras la idea es buena no es fácil aplicarla en la práctica. Otros métodos de capas y agrupamiento deben ser encontrados.

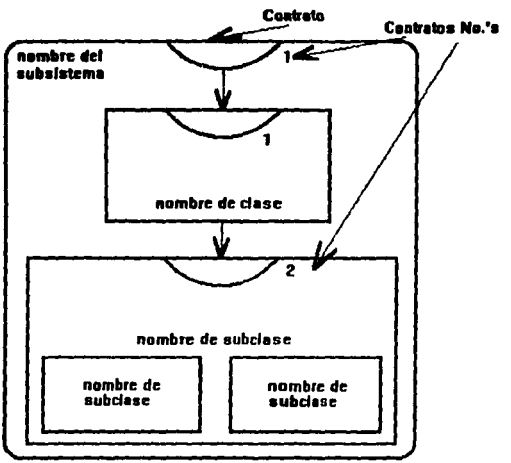


Fig. B.18 Un ejemplo gráfico de colaboración

El método requiere una declaración de requerimientos escritos como una entrada y procede por medio de los siguientes pasos :

(1) Identificar objetos (de los sujetos en la especificación) y organizarlos dentro de las estructuras de clasificación.

(2) Encontrar responsabilidades (de los verbos en la especificación).

(3) Asignar responsabilidades a las clases.

(4) Examinar las estructuras para definir responsabilidades.

(5) Encontrar colaboraciones.

(6) Distinguir clases sin colaboraciones.

(7) Refinar las estructuras.

(8) Grupos de responsabilidades dentro de contratos.

(9) Usar colaboraciones para definir subsistemas.

(10) Llenar en los detalles.

Muchos usuarios de CRC usan atajos para validar su modelo y descubrir las dinámicas globales del sistema : algo que muy pocos métodos hacen bién, incidentalmente. Esto a menudo hecho por la organización de un equipo de trabajo donde la gente juega-el-rol de los objetos, lo cual es al menos muy divertido y a menudo altamente relevante.

CRC es un método simplista pero muy práctico, usualmente empleado como un precursor del uso de otras notaciones. Graham lo usa en el estado de captura de los requerimientos y como una herramienta pedagógica, dado que la gente la obtiene para entender el énfasis en el comportamiento de la tecnología de objetos tranquila y rápidamente.

B.10) SINTESIS

El AOO y el DOO no pueden ser separados con claridad. El Diseño es el modelamiento de arquitectura. Adiciona el detalle, precisión e implantación-dependiente de las características de los modelos de análisis. El Diseño puede estar dividido den ro del diseño lógico y el diseño físico. El Grupo de Manipulación de Objeto llama diseño físico a la "implantación del modelamiento" y define la modelación del diseño como proveniente de "rigurosas especificaciones de la interfaz provista por un conjunto de tipos de objetos". El modelamiento de la implantación es entonces llevado para desarrollar un solución implantable incluyendo sus características de distribución. Este el modelamiento lógico que mezcla la mayoría con el Análisis Orientado a Objetos.

El método original de diseño orientado a objetos fue creado por Booch (1986). Los principales métodos de Diseño Orientado a Objetos, Basados en Objetos y notaciones en el uso actual son provenientes del método de Booch (1991), OODLE, HOOD y OOSD. Muchos de ellos usan análisis textual como un punto de arranque. Los STD's son ampliamente usados para representar lo dinámico de los objetos. Booch' 86 y HOOD son real y sólamente situables para desarrollos en Ada. OOSD no es propiamente un método, pero sí una notación a quien las reglas de un método pueden ser adicionadas. Es es lenguaje independiente, como es Booch ' 91. JSD

está desplazándose a versiones basadas en objetos y orientadas a objetos que han aparecido. Un método refinado de Booch enfatizará el análisis y el diseño y es simple, profesional y una buena guía de enseñanza, pero está incompleto en algunas áreas, tales como la carencia de señalamiento en la composición de estructuras.

APENDICE " A "
" ANALISIS ORIENTADO A OBJETOS "

A.1) ¿ Qué entendemos por Análisis ?

El análisis nos permite desintegrar un problema en todos los elementos que lo conforman en la búsqueda de un planteamiento apropiado a las necesidades y requerimientos del futuro sistema, por supuesto, hablamos de un "problema" que se deseé analizar en términos de computación. Además de ello, el análisis es verdaderamente útil en la especificación de los requerimientos del usuario, en la estructuración del sistema y funcionalidad.

Existen varios tipos de análisis, entre los más convencionales encontramos : el *Análisis Estructurado* quien emplea frecuentemente dos modalidades : el diseño Top-Down (Arriba-hacia-Abajo), partiendo del concepto más general al más particular, o bién, el diseño Bottom-Up (Abajo-hacia-Arriba). Otro tipo de análisis es aquél que se fundamenta en la descomposición funcional combinada con el análisis separado de datos.

A.2) ¿ Existe una gran diferencia si hablamos del Análisis Orientado a Objetos ?

En realidad no se trata de diferencias sino más bién de complementaciones, en el Análisis Orientado a Objetos se pretende que el sistema sea descrito en los mismos términos empleados en el mundo real, es decir, en el mundo de los negocios, donde existen muchas abstracciones, las mismas que se emplearán en el sistema. Recordando un poco, en el Capítulo I hablamos sobre el concepto *"abstracción"*, llegando a la conclusión de que éste representa uno de los pilares del Paradigma Orientado a Objetos.

Quizá fuese muy conveniente que el *AOO (Análisis Orientado a Objetos)* contemplase un elemento de "síntesis", cuando abstraemos los requerimientos de los usuarios e identificamos los objetos escenciales, es decir, trascendentales del sistema, para después ensamblarlos dentro de estructuras de tal forma que éstas soporten diseño físico en una etapa posterior, ahí radica la importancia del elemento de "síntesis". Al analizar un sistema, en realidad lo que estamos realizando es la imposición de una estructura en el dominio del problema.

Un sistema posee tres aspectos primarios :

a) datos, objetos o conceptos y su estructura
b) arquitectura o procesos atemporales
c) dinamismo o comportamiento del sistema

a lo largo de éste apartado se hará referencia de ellos como datos, procesos y control o dinamismo respectivamente. La Tecnología Orientada a Objetos combina a los datos y a los procesos mediante el encapsulamiento local del comportamiento con los datos. De ésta forma, el AOO podría ser considerado como una forma de silogismo en movimiento desde lo particular (clases) a través de lo individual (instancias), hasta lo universal (control).

Con el tiempo la conceptualización del AOO ha ido mejorando, lográndose un método interesante : el método *SOMA (SOMA, por sus siglas en inglés)*. SOMA es un filtro para las ideas actuales dentro de otros métodos convencionales y orientados a objetos, la notación que se empleará es del todo neutral en cuanto a figura y diseño de iconos, con ello logramos mayor flexibilidad propiciando que el lector sea quien eliga la notación que deseé emplear.

Pero, de cualquier forma, si el lector desea profundizar en la notación de SOMA, podrá referirse a Graham Ian, quien en una parte de su libro *"Migrating to Object Technology"* trata, precisamente al método y a su ciclo de vida.

A.3) La participación de la Ingeniería de Software dentro del contexto del AOO

Como lo acabamos de mencionar en el apartado anterior , un sistema se compone de tres aspectos primarios, lo afirmamos categóricamente dada la sabiduría convencional que nos revela la Ingeniería de Software, la cual describe a un sistema en forma tridimensional (datos, procesos y control), como se aprecia en la **Fig. A.1** :

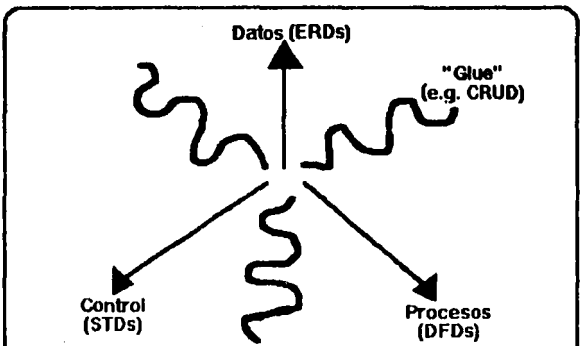


Fig. A.1 Las tres dimensiones de la Ingeniería de Software.

La dimensión de los datos corresponde a los ***Diagramas de Relación-Entidad (ERD's, por sus siglas en inglés)*** es decir, a los modelos lógicos de datos. Los procesos de los modelos se encuentran representados por un flujo de datos de un orden a otro. Los dinámicos son descritos por un estado de transición.

Hablando en términos exclusivamente de documentación, éste es un método increíblemente potencial. Los métodos centrados en datos en la Ingeniería de Software comienzan con el Modelo de Datos, mientras que, **los métodos de los procesos orientados a objetos inician con los Diagramas de Flujo de Datos (DFD's)**. Por otro lado, los métodos en tiempo real empiezan con las **máquinas de estado finitas o STD's** (por sus siglas en inglés), aún cuando son un tanto inusuales para los desarrolladores de sistemas comerciales. Los datos son más estables que las funciones y, es por ello que los métodos centrados en datos son preferidos en la mayoría de los casos.

Los métodos de AOO y DOO suelen caer dentro de dos tipos básicos : ***el ternario***, es aquél en el cual se simula la existencia de métodos estructurados, contando con tres notaciones separadas para los datos, dinamismo y procesos; y ***el unario***, donde se asume que algunos objetos heredan procesos (métodos) combinados y datos, requiriendo solamente una notación. Por simplicidad hemos asumido la clasificación propuesta por Graham (ternarios y unarios) de las diversas metodologías orientadas a objetos.

Visualizando los métodos desde afuera, los métodos ternarios tienen gran aceptación y cuentan con la ventaja de que existirá familiaridad con los métodos estructurados existentes así como con su notación y filosofía. Por otro lado, los métodos unarios presentan dos ventajas : *(1) son más consistentes con la metáfora orientada a objetos y (2) son más fáciles de aprender desde el inicio.*

Como ejemplos de los métodos ternarios tenemos a : OMT (Rumbaugh,1991), las modificaciones Ptech de Martin y Odell(1992) y OSA(Embley, 1992). De los métodos unarios uno de los más conocidos es el CRC (Wirfs-Brock,1990) y Coad/Yourdon (Coad y Yourdon, 1990-1991) se tratarán en apartados posteriores. Existen inclusive, un número creciente de métodos híbridos, es decir, de ambos tipos, tales como el de Henderson-Sellers (1992) y SOMA.

De todos los métodos de AOO disponibles, existen aspectos que son notables : de Coad/Yourdon son simples pero de escaso soporte para la descripción de sistemas dinámicos. Algunos como el OMT de Rumbaugh y el de Shlaer/Mellor son muy útiles pero de gran dificultad en su aprendizaje pues son complejos, pero en SOMA encontramos que se combinan la notación unaria para AOO y el conocimiento basado en el estilo de reglas de sistemas para la descripción de limitantes, reglas de negocios, sistema de control global, bases de datos y cuantificación sobre relaciones. Además SOMA es el único que soporta la clasificación confusa, de gran importancia para la especificación de requerimientos en algunos ámbitos, tales como modelación y el control de procesos.

Aunque SOMA se perfila como el único método de Análisis que combina objetos y reglas, hay muchísimos productos de software que también lo hacen, tales como Nexpert Object, Kappa, KBMS, ADS, XShell y ObjectIQ/ESKernel y algunos productos de bases de datos como Ingres y Generis.

La extensión de que el modelo objeto encapsule no solamente atributos y métodos, sino también reglas, es un gran paso, lo cual habilita a la descripción OO (Orientada a Objetos) de sistemas basados en reglas, avanzadas bases de datos y modelos iniciativos con los beneficios de la reutilización y la extensibilidad, también se incrementa la riqueza semántica de las descripciones de sistemas convencionales.

A.4) ¿ Qué métodos existen en el Análisis Orientado a Objetos ?

Recientemente muchísimos de los reconocidos métodos desarrollados de software han producido extensiones orientadas a objetos de métodos de Diseño y Análisis de sistemas

convencionales, notablemente se citan a Constantine, Jackson, Mellor, Page-Jones y Yourdon. Otros no tan conocidos también están contribuyendo en éste sentido, es conveniente comentar que esos métodos se encuentran más o menos incompletos y, que más que métodos parecen sugerencias de métodos, en éste apartado nos enfocaremos a examinar a muchos de ellos y algunos híbridos.

Es un tanto difícil diferenciar entre los métodos de AOO y DOO, debido a la continuidad de la representación y a la filosofía del Análisis hasta el Diseño Lógico, es por ello que resulta confuso indicar donde comienza uno y termina el otro o indicar si existe alguna diferencia.

El método CRC de DOO podría ser equivalente a una técnica de análisis. De cualquier forma, es indudable que los diversos métodos citados podrán aportarnos conclusiones propias si procedemos a estudiarlos con mayor detenimiento, comencemos entonces con el próximo apartado derivado del estudio sobre AOO y sus "métodos" :

A.4.1) Shlaer/Mellor : Análisis de Sistemas Orientados a Objetos (OOSA)

Uno de los primeros ejemplos del AOO lo realizaron Shlaer y Mellor (1988), aunque éste método podría no ser considerado como Orientado a Objetos (OO) por muchas razones, incluyendo la total ausencia de alguna noción de herencia; más bién se trata de algo más que una extensión OO del modelamiento de datos, para 1991 Shlaer y Mellor introdujeron la herencia y la idea de que los métodos podían ser descubiertos por el modelamiento de los ciclos de vida de las entidades con máquinas de estados finitos. Su postura ante la identidad del objeto fue convenida en la identificación de éstos como atributos o claves. Tiempo después, la normalización de las reglas (reglas de atribución) son aplicadas a los objetos, donde ellos concordaron un poco más fue en las tablas relacionales (aunque no son del todo recomendables pues podrían resultar peligrosas e indeseables).

El primer paso en el Método de Shlaer y Mellor es la definición de objetos y sus atributos. El modelado de la notación es descendente (según la notación Ward/Mellor). El énfasis se situa próximo a la definición de la historia de vida del objeto, empleando los diagramas de transición de estado "Moore Style" y sus correspondientes tablas, utilizados para definir las operaciones. La notación para los modelos de estado es lejana a la estándar. La dinámica global es mantenida por la coordinación de los ciclos de vida del objeto anotados en los modelos de comunicación del objeto, el cual muestra el paso de eventos (es decir, mensajes) entre entidades. Las relaciones son también indicadas por los modelos del ciclo de vida.

Una idea brillante aportada por ellos fue la definición del dominio reutilizable, tal como lo muestra la Fig. A.2, la cual nos auxilia en la imposición de un método etiquetado por la Ingeniería de Software y fortalece el reuso.

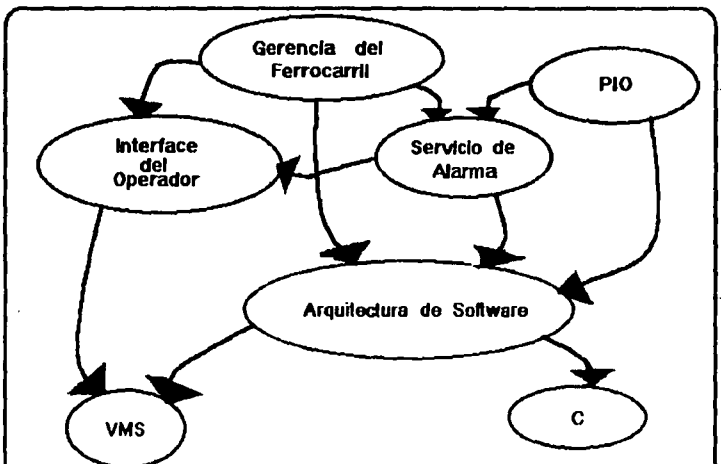


Fig. A.2 Un trazo típico en el ambiente dominante de Shlaer / Mellor.

Este método podría ser clasificado como temario y procedente a la creación de un modelo de información (o datos) mostrando objetos, atributos y relaciones. Después, un modelo de estado describirá los estados de los objetos y las transiciones entre los estados. Finalmente un ***DFD (Diagrama de Flujo de Datos)*** define el proceso del modelo. El método es fuertemente influenciado por el diseño relacional : los objetos se encuentran en principio en forma normal y la identidad del objeto no es una característica natural de los diseños. Los usuarios del método tienden a ser usuarios que han migrado del método Ward/Mellor y las aplicaciones derivadas parecen tener un tiempo real o control de procesos, aunque podría ser un accidente histórico.

Este es un método complejo que abarca tanto análisis como diseño, además ha tenido una historia controversial y actualmente tiene un amplio uso. Este método es particularmente soportado por el equipo de trabajo de la herramienta CASE.

A.4.2) Coad / Yourdon

Un método que emerge del campo de Yourdon, es sintetizado en Coad y Yourdon (1990-1991) y fue esencialmente interesante como el primer acontecimiento publicado de un método de AOO, práctico, razonablemente completo y soportando notación situable para proyectos comerciales.

Coad/Yourdon introdujeron una notación menos incómoda de la que encontró Booch, Shlaer/Mellor o la mayoría de los métodos de DOO. Ellos enfatizaron el Análisis como opuesto al Diseño, su método es remarcable por simplicidad aunque incompleto en algunas áreas.

Sin embargo, sus ideas son lo suficientemente útiles para formar las bases de un método que, siempre que se encuentre desarrollado sobre tiempo, pueda deliberar los beneficios reales en términos de especificaciones reutilizables y extensibles.

Una de las características más importantes de las notaciones de Shlaer/Mellor y Coad/Yourdon es la explicitud o claridad de los atributos. Los atributos de hecho pueden ser identificados con métodos estándares que accesen su estado, como con *"Coloca_Salario"* y *"Obten_Salario"*. Para Ian Graham, lo más importante de un sistema comercial es la necesidad de crear atributos explícitos.

Coad y Yourdon hicieron dos publicaciones, el tiempo de diferencia entre éstas fue de aproximadamente un año, en ellas se expone su pensamiento del método, notaciones y terminología, siguiendo el trabajo presentado en el libro *"Object Oriented Methods"*, se compaginan ambos textos de la manera más correcta y útil encontrada por Graham, logrando considerar el procedimiento del Análisis en sus cinco estados :

- **Sujetos.**- El área del problema es descompuesta dentro de *"sujetos"*, los cuales corresponden a la notación de **"niveles" o "etiquetas"** en diagramas de flujo de datos y en subsistemas de Booch o categorías de clase. Esos sujetos deberán tener un tamaño manipulable en el que se contemplen solamente 5 ó 9 objetos.

- **Objetos.**- Posteriormente, los objetos son identificados en detalle, utilizando el método de investigación para las entidades de negocios de la misma manera como un Análisis de Datos podría ser ejecutado. Coad y Yourdon otorgan un poco de ayuda adicional respecto a cómo ejecutar ésta tarea.

- **Estructuras.-** Dos estructuras completamente diferentes son identificadas : **la clasificación de estructuras y la composición de estructuras.** Es aquí en donde la herencia es repartida, la clasificación de estructuras son meramente árboles de especialización y generalización. Coad y Yourdon tratan muy poco acerca de cómo esos árboles son procesados.

- **Atributos.-** Como en el Análisis de Datos convencional, **los atributos son detallados y modelados**, las relaciones multiplicadas especifican una versión de Análisis Relacional Extendido (ERA). Aquí vemos que existe una desviación de algunos otros métodos de DOO, en los cuales sólamente se especifican los métodos.

- **Servicios.-** Esta es la forma en la que Coad/Yourdon denominan a las operaciones. Cada tipo de objeto debe ser equipado con métodos para la creación y borrado de instancias, obteniendo y colocando los valores con métodos de caracterización especial del comportamiento de los objetos.

Coad/Yourdon ofrecen el mnemónico "SOSAS" a la remembranza de estos pasos.

El Diseño Orientado a Objetos de Coad (1991) añade cuatro componentes a esas cinco actividades (SOSAS, Sujetos, Objetos, Estructuras, Atributos y Servicios), principalmente su DOO consiste en el refinamiento de los productos del AOO dentro del cual llaman al componente del campo del problema al diseño (añadiendo nuevas soluciones al espacio de clases) y adiciona además, tres nuevos componentes llamados **"Interacción Humana" (HIC)**, estos componentes permiten especificar el diseño de ideas que serán incluídas en los diagramas del AOO.

La Fig. A.3 muestra a un tipo de objeto como un icono triple, según la notación definida por **Coad y Yourdon** :

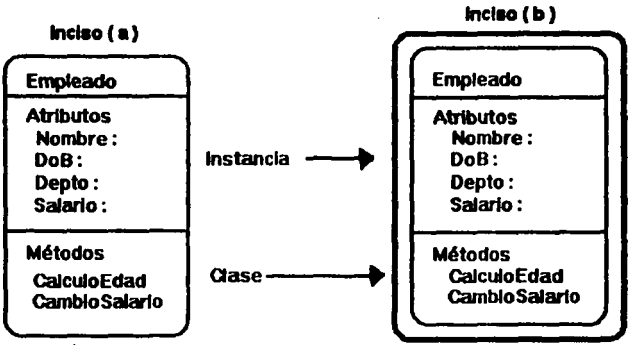


Fig. A. 3 **(a) La notación de Coad / Yourdon para un objeto general; (b) La notación revisada dibuja una línea gris fuera de línea alrededor de los objetos que pueden ser instancias.**

Las dos cajas correspondientes a los atributos y a los métodos son útiles si pensamos que es muy difícil escribir todo a detalle. Para soportar esta notación, se creó una herramienta de bajo costo llamado **AOOTool** ***Herramientas de Análisis Orientado a Objetos)*** producido por la compañía Coad.

El paso de mensajes se muestra tal como Buoch lo emplea, por medio de flechas. Coad/Yourdon permiten la ennumeración de los mensajes. Al principio ésto parece muy útil, pero no lo es tanto cuando el sistema se vuelve complejo, se convierte en un verdadero "spaghetti". La **Fig. A.4** muestra cómo puede ser indicado el paso de un mensaje.

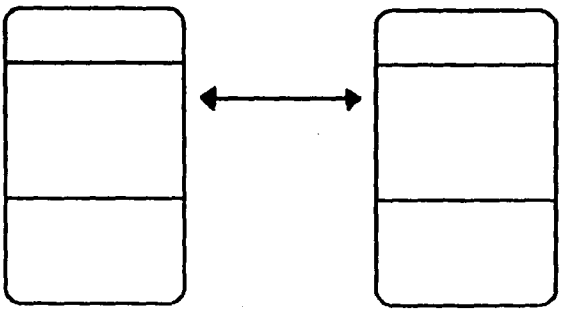


Fig. A . 4 La posibilidad del paso de mensajes entre objetos es indicada por las puntas de flecha.

Al etiquetar a los "sujetos" como un proceso que toma lugar en varios estados del análisis y pueden ser útiles para una descomposición inicial al comienzo o para organizar el modelo después de que los objetos han sido identificados o refinados.

Un proceso similar es asociado usualmente con DFD's. Las etiquetas o sujetos son representados por cajas y el paso de mensajes por flechas, como lo muestra la **Fig. A.5**.

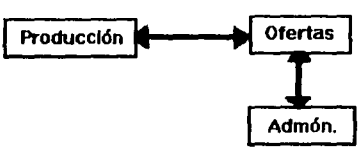
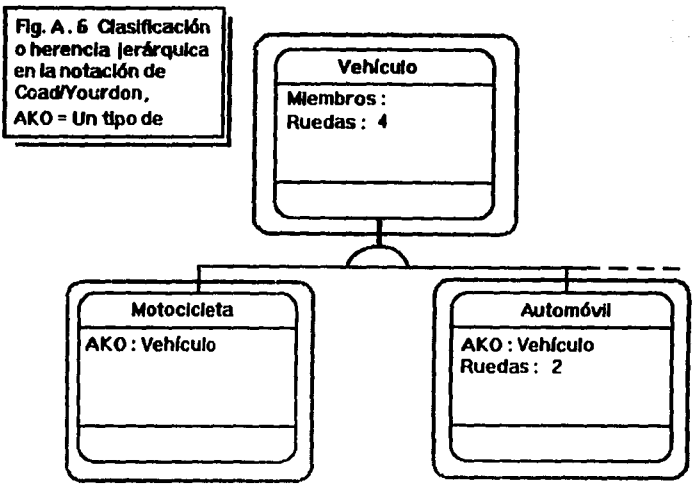
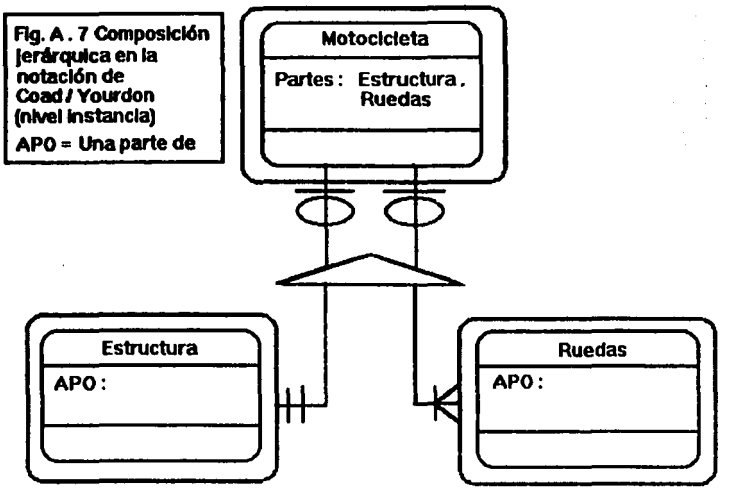


Fig. A. 5 Las etiquetas son grupos de idealmente, entre 5 y 9 objetos o posiblemente clasificación coherente o composición de estructuras.

En la **Fig. A.6** Coad/Yourdon propone la herencia jerárquica y,



en la **Fig. A.7** proponen la composición, las puntas de cabeza denotan la composición, las otras notaciones en las ligas se refieren a los datos semánticos.



Conviniendo que esas estructuras sean vistas como sujetos, los mensajes enviados al sujeto son normalmente convenidos como el inicio del envio a la parte superior de una estructura compuesta y a la parte inferior de una clasificación de estructura.

Los mensajes y las relaciones son mostradas solamente entre instancias.

Cuando definimos los atributos, exactamente las mismas consideraciones son aplicadas como en algunos ejercicios de modelamiento de datos. El analista es capaz de tomar ventaja de la herencia y asume que los elementos no especificados son heredados donde tiene sentido hacerlo. La herencia múltiple es notacionalmente permitida, pero no existe nada que nos ayude con la solución del conflicto de la anotación de todo atributo solo y método con cada instrucción y, puede ser necesario. De cualquier forma, el método revisado no soporta genuinamente la herencia múltiple.

Método o servicio, la definición es el estado donde los aspectos funcionales del sistema son especificados. Un método es un procedimiento que altera el estado de un objeto (es decir, el valor de sus atributos) o provoca al objeto el envio de mensajes. Los métodos

definen exactamente ¿ qué mensajes puede procesar un objeto ?. Así que, como se ha visto, es raramente necesario o útil etiquetar a todas las flechas con "mensajes" en los diagramas, como los mensajes legales pueden leer desde la ventana de los métodos. Es necesariamente en el estado del registro de mantenimiento y paso de mensajes del comportamiento del objeto. Este debe ser textual pero no existe notación que soporte las excepciones.

Una útil sugerencia emitida por Coad y Yourdon concierne al uso de la descripción de un patrón estandar de un objeto y método. Su método está basado en la especificación formal del lenguaje, pero algún estilo del programa patrón ayudará a reforzar un estilo comunicable de documentación y asistirá en la creación de especificaciones completas para el objeto y el método. El patrón debe nombrar al objeto y listar los nombres de los atributos y métodos en la interfaz, exactamente como en la notación diagramática. Sin embargo, deberá indicar características de los atributos, anotando el objeto con las intenciones del diseñador y remarcando su seguimiento, tamaño, etc. Para cada método el patrón especifica el propósito de la forma en Español o las declaraciones estructuradas en Español (o en el idioma en cuestión). Por supuesto es posible añadir diagramas al texto cuando se considere útil. Fundamentalmente, cada método es especificado en la misma forma en la que los analistas han especificado siempre los componentes del sistema, la única diferencia es que los detalles serán ocultados con el objeto y solamente accesados por sus interfaces.

La parte más débil del método definida por Coad/Yourdon viene a ser la especificación del dinamismo del sistema, pero un método de transición de estado se sugiere para los objetos dinámicos aunque su notación no se encuentre aún definida. No existe soporte explícito para la especificación de reglas de negocios.

Solamente la definición de los datos semánticos no es direccionada por los métodos de DOO como el HOOD y el OOSD. Esto refleja que las abstracciones de programación en bajo nivel en aplicaciones de tiempo real fueron problemas complejos de manipulación de datos. Además, los LPOO's *(Lenguajes de Programación Orientada a Objetos)* no han sido identificados para la manipulación de objetos persistentes y, algunas autoridades (por ejemplo, Wegner, 1987) van más lejos al decir que los objetos contradicen el gran espíritu de la Orientada a Objetos, mientras que otros (como Meyer) insisten en que es fundamental al método.

La notación de los *Análisis Relacionales Extendidos (ERA)* es similar a la recomendada por Coad/Yourdon (1990) y es probablemente tan buena como cualquier otra. ***La notación ayuda a especificar dos aspectos de relaciones : su multiplicidad y modalidad. La multiplicidad en cualquier relación es : uno a uno, uno a varios o varios a varios y, la modalidad se refiere a***

que sean necesarios o posibles (opcional) y se encuentra estrechamente conectado a la idea integramente referencial.

La **Fig. A.8** define una relación de varios a uno y el nivel de instancia. Las relaciones de éste tipo no son mostradas en los primeros objetos en la notación de Coad/Yourdon. Es útil en algunas aplicaciones, aún cuando no es frecuentemente utilizada, por ejemplo al convenir una relación de ***"unión"***, como la de un objeto con sus propiedades, tales como la fecha y los métodos.

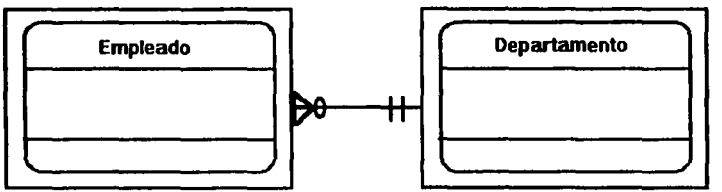
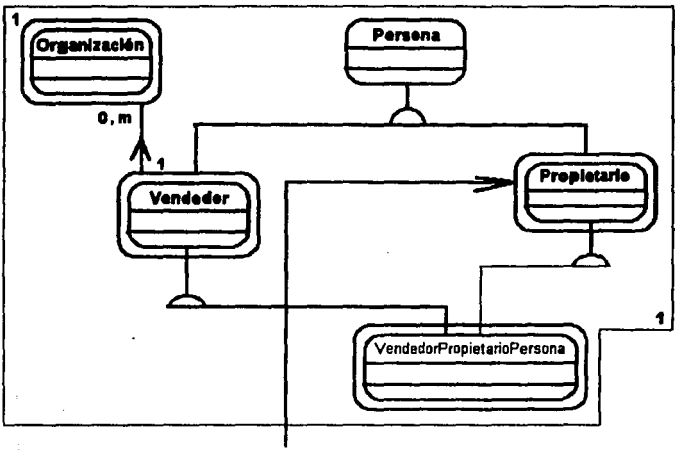


Fig. A . 8 Datos semánticos. Nota que los números pueden ser instanciados de los símbolos "crow's feet" usados aquí.

Ciertamente hay otros defectos en el método de Coad/Yourdon, dada la influencia de las bases de datos relacionales, ellos insisten en que los atributos deber ser "atómicos", lo cual es incorrecto, la Metodología Orientada a Objetos se refiere al modelamiento de sistemas complejos los cuales necesitan no ser atómicos.

La **Fig. A.9** muestra la mayoría de las características de la notación de Coad/Yourdon, exactamente como se muestra en la segunda edición de su libro de ***"Análisis"***.


1
Organización
Persona
0, m
1
Vendedor
Propietario
1
VendedorPropietarioPersona

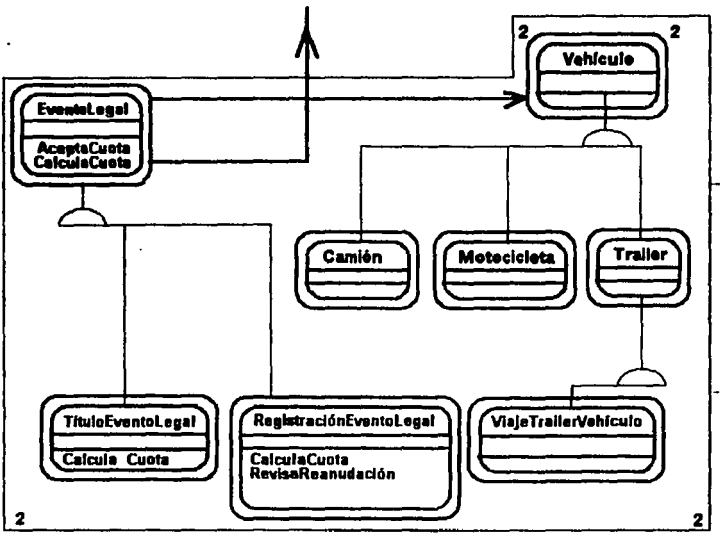


Fig. A.9 Notación de Análisis Orientado a Objetos de Coad Yourdon.

Este diagrama representa el Análisis del registro de un vehículo, existen dos etiquetas, una estructura compuesta y muchísimas clasificaciones de estructuras. Las líneas más oscuras representan el paso de mensajes. El método no es perfecto pero es simple y unario. El contraste ahora lo será la notación temaria de considerable riqueza y complejidad : *TMO (Técnica del Modelamiento de Objetos).*

A.4.3) Rumbaugh y la Técnica del Modelamiento de Objetos (TMO)

La técnica del Modelamiento de Objetos (TMO) es ampliamente convenida por la mayoría de los Sistemas de Análisis Orientados a Objetos que han sido publicados. Este ***es un método de tres puntas con raíces fuertes en los métodos tradicionales estructurados y ofrece una notación extremadamente rica pero espantosamente complicada y detallada.*** El soporte

automatizado es además admisible y un número de herramientas soportan la notación. La complejidad es particularmente pagada por la habilidad de algunas de esas herramientas para generar código automáticamente. El método es relativamente independiente del lenguaje aunque es frecuentemente asociado con C++ y Smalltalk.

La TMO cae dentro de tres fases principales o actividades : Análisis, Diseño del sistema y Diseño del objeto. El *Análisis* asume que una especificación de los requerimientos existe y procede construyendo tres modelos separados empleando tres notaciones diferentes. La primera en ser construída es el *Modelo Objeto (MO)*, el cual consiste de diagramas similares a los de Coad/Yourdon y un diccionario de datos. La notación es fundamentalmente la del modelamiento de Estructuras Relacionales con operaciones añadidas a la identidad de los iconos.

Después, todo ***Objeto Dinámico (OD)*** se construye consistiendo de STD's dibujados en una notación extendida de Harel y diagramas globales del flujo de los eventos. El tercer paso no es comúnmente utilizado por todo profesional, solamente en el más alto nivel de abstracción de la mayoría de quien lo hace, este es, el ***Modelo Funcional (FM)***; el Modelo Funcional se distingue de un Diagrama de Flujo de Datos en todo intento y propósitos. Las operaciones descubiertas en ambos (DM y FM) son entonces añadidas al Modelo Objeto, después de que el análisis reporta la inclusión de todo lo anterior, la liberación es producida.

La notación básica del modelamiento de objetos se observa en las Fig. A.10, A.11 y A.12 y es mucho más rica que la de Coad/Yourdon. En la Fig. A.10 los atributos especifican su tipo y las operaciones dadas listan los argumentos y regresan sus tipos y las instancias se muestran en rectángulos con bordes circulares y nombres entre paréntesis. Aún cuando los analistas se interesan más en las clases que en las instancias, la TMO permite a cada diagrama mostrar sus instancias.

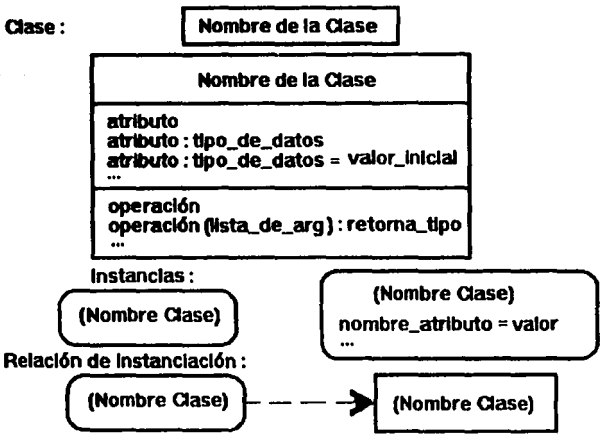


Fig. A . 10 Clases e instancias en la Técnica del Modelamiento de Objeto (TMO).

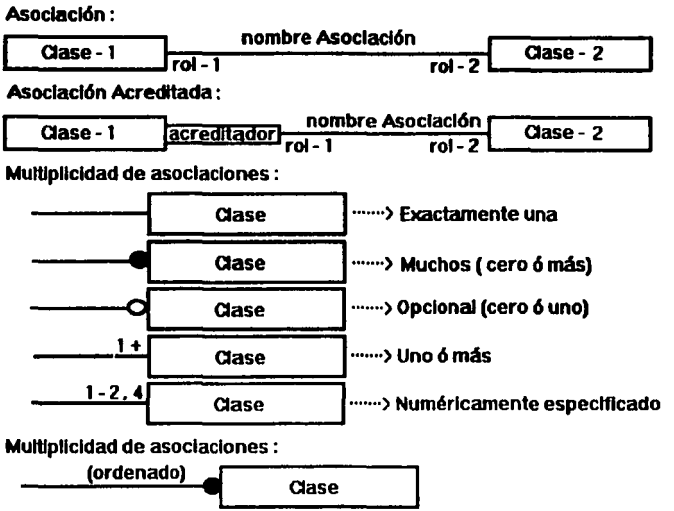


Fig. A.11 Asociaciones en TMO (Técnica del Modelamiento de Objetos)

La **Fig. A.12 (a)** muestra que las asociaciones ternarias son permitidas, pero las asociaciones con atributos son a menudo expandidas dentro de los primeros objetos de la clase, como en la **Fig. A.12 (b)** y **A.13.** Las asociaciones son señaladas en el papel y pueden tener un calificativo, éste es ocasionalmente muy útil; por ejemplo, si la Clase_1 es Directorio y la Clase_2 es Archivo y el calificativo es el nombre_del_archivo que es el único de un archivo dentro de un directorio.

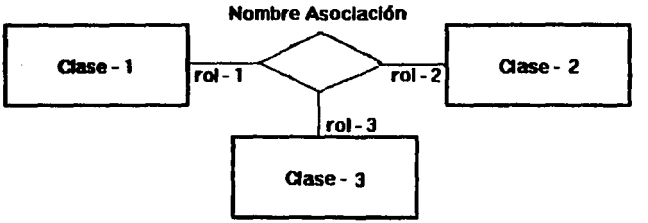


Fig. A . 12 (a) Asociaciones ternarias

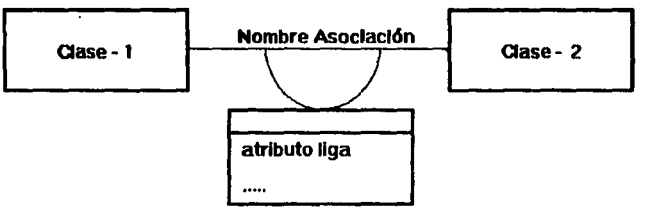


Fig. A . 12 (b) Asociaciones con atributos en TMO.

La relación uno a varios entre Directorio y Archivo es de ésta forma reducida, efectivamente, en una relación de uno a uno.

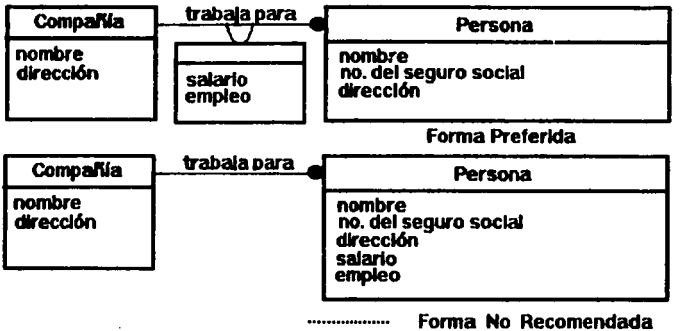
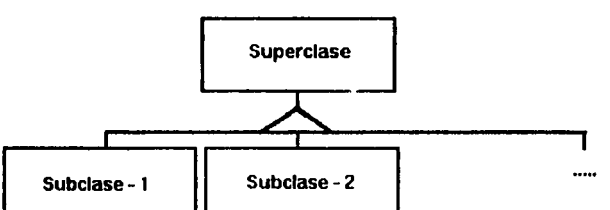
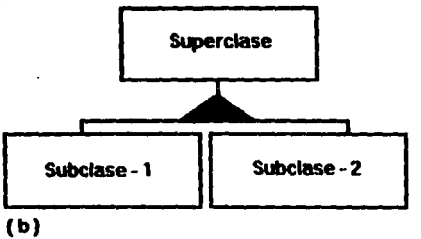


Fig. A. 13 Las asociaciones deben ser expandidas dentro de objetos, de preferencia añadiendo los atributos a los objetos en existencia.

La clasificación de estructuras es mostrada con la **Fig. A.14,** en la cual puede observarse que la notación es más completa que la de Coad con la expresión de exclusividad y opcionalidad .



(a)



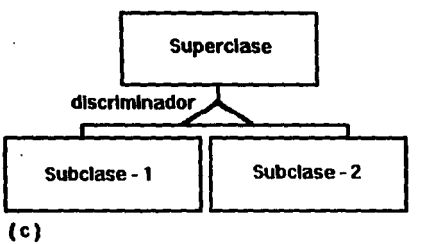


Fig. A. 14 La clasificación en TMO
(a) La mayoría de las subclases existen ;
(b) Las subclases tienen superpuestos sus integrantes (no-separados)
(c) El discriminador es un atributo cuyo valor se diferencia entre subclases

La composición de estructuras es mostrada en la Fig. A .15.

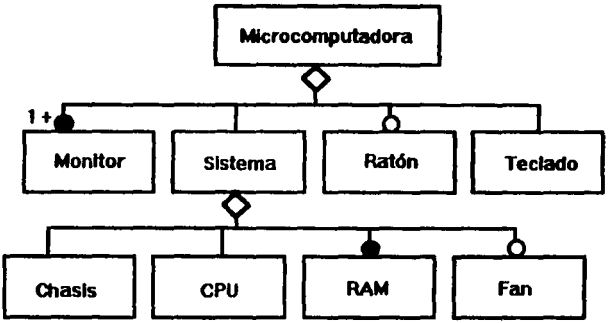


Fig. A. 15 Una composición (agregación) de estructura TMO para ensamblar PC.

Una característica fuerte de la TMO es su habilidad de representar recursividad compuesta, como en la Fig. A.16.

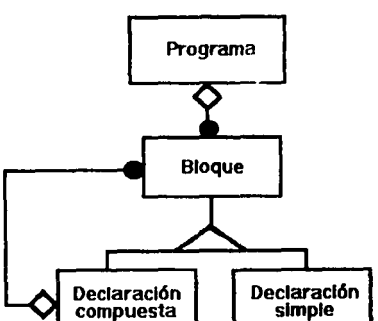


Fig. A. 16 Estructura de la composición recursiva mostrando cómo se construyen los programas hacia arriba.

La Fig. A.17 proveé una idea dada de cómo exactamente se completa -y complica- la notación de la Técnica del Modelamiento de Objeto (TMO).

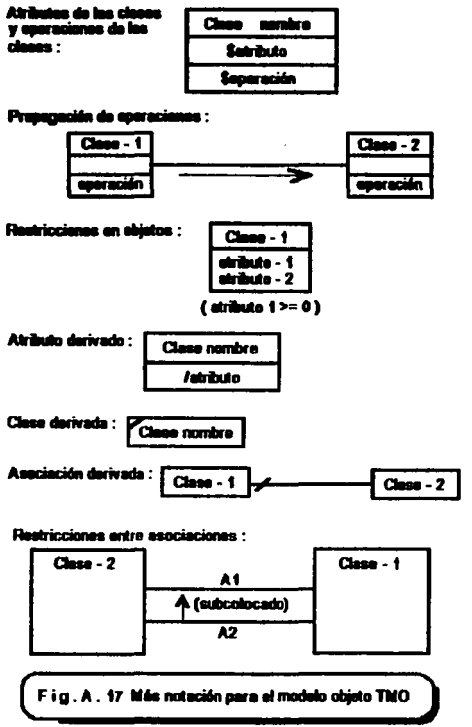


Fig. A. 17 Más notación para el modelo objeto TMO

La Fig. A.18 muestra un trazo de un evento global para las actividades que implica una llamada telefónica. Un objeto de interés : *la línea telefónica*, es colocado en el centro y la secuencia de eventos ocurridos es mostrada.

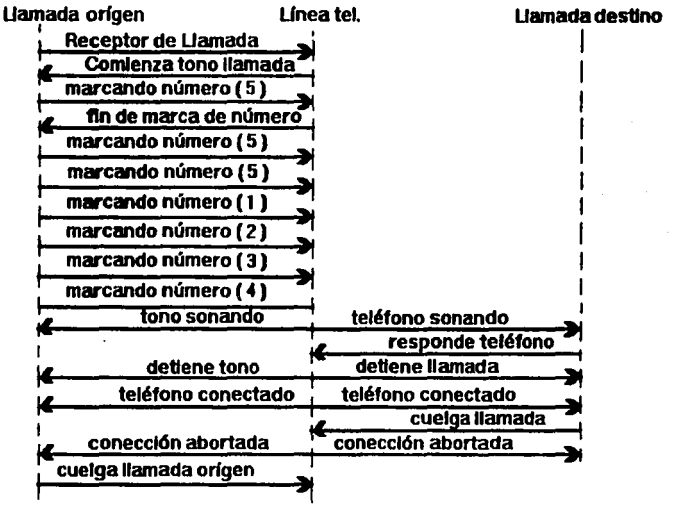


Fig. A. 18 Un diagrama de trazo de evento

Desde el análisis de su comportamiento externo un modelo del estado interno de la línea telefónica es construido como lo muestra la Fig. A.19.

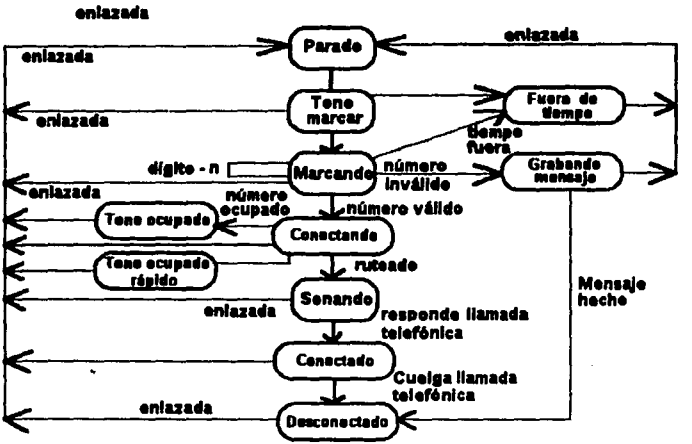


Fig. A.19 Modelo de estado TMO para el objeto Llamada en línea.

La notación empleada se basa en las cartas de estado Harel y se explica en la **Fig. A.20.** Los eventos pueden acontecer con atributos, provocando acciones que puedan ser garantizadas por declaraciones condicionales (precondiciones); algo complejo de la acción del objeto se introduce con el estado destino.

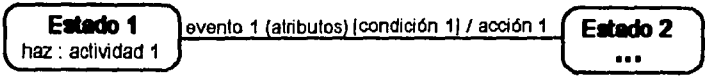
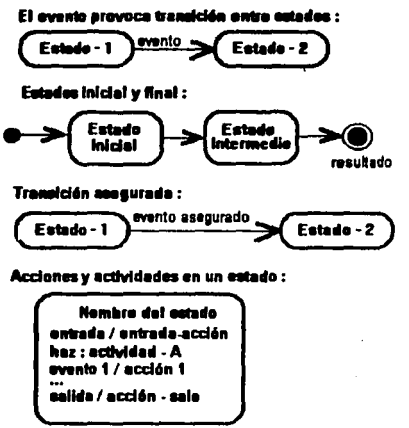


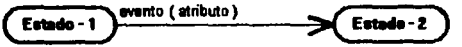
Fig. A. 20 Notación del Diagrama de transición de estado en la TMO.

Cuando un estado es introducido, la actividad *"hacer"* se ejecuta, las acciones son instantáneas pero las actividades toman tiempo. Cuando el sistema sale del estado la actividad cesa. Con la notación Harel, los estados pueden tener subestados y existen concurrentemente.

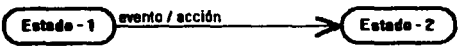
La notación DM completa es del todo compleja, lo cual se observa en la Fig. A.21.



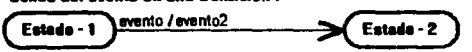
Evento con atributo :



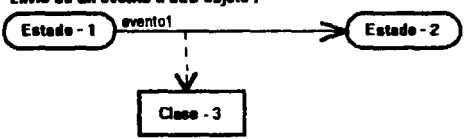
Acción en una transición :



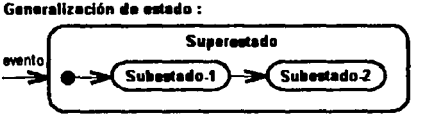
Salida del evento en una transición :



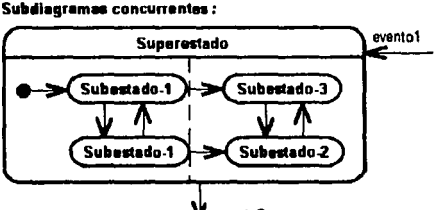
Envío de un evento a otro objeto :



Generalización de estado :



Subdiagramas concurrentes :



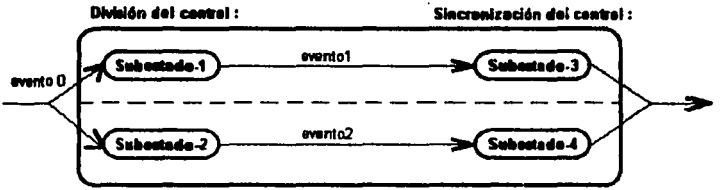
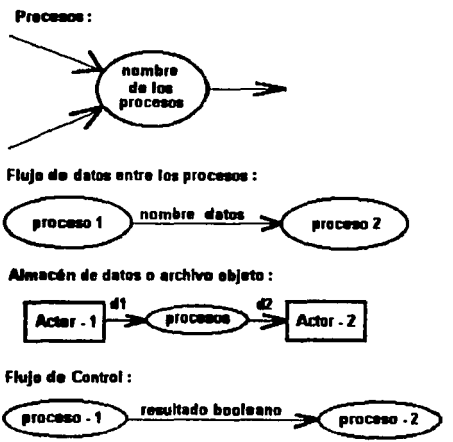


Fig. A . 21 Notación completa del modelo dinámico de la Técnica del Modelamiento de Objetos.

El estado de modelamiento final es la producción del MF *(Modelo Funcional)*. Este no siempre se utiliza y la mayoría de los profesionales lo ven como uso de alto nivel, tal como un analista podría emplear contexto en los diagramas *(Nivel Cero en Diagramas de Flujo de Datos)*. La Fig. A.22 muestra la gran diferencia con respecto a DFD's. Siendo completados los tres (ó dos) modelos, las operaciones descubiertas son copiadas al Modelo Objeto (OM) y podemos movernos al *sistema y al diseño del objeto*.



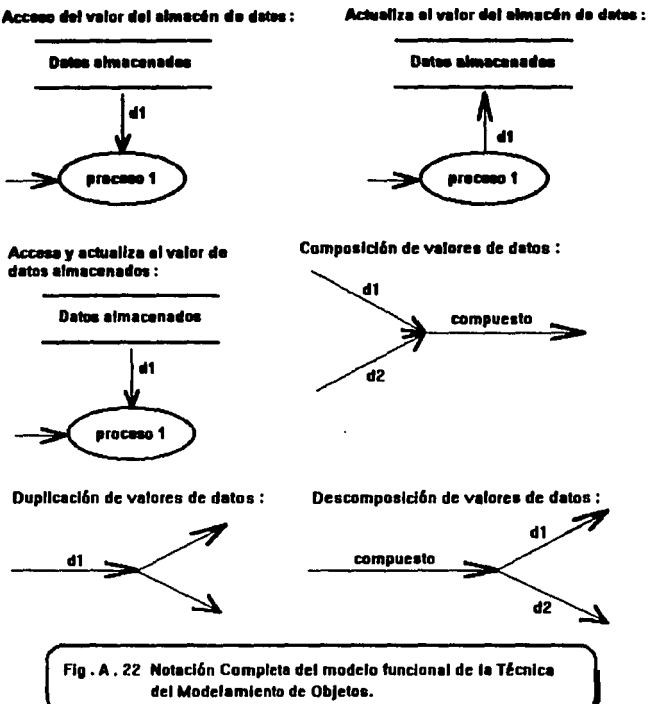


Fig . A . 22 Notación Completa del modelo funcional de la Técnica del Modelamiento de Objetos.

El Diseño del Sistema se apreciará en el Apéndice B.

A.4.4) Martin / Odell y el Método Ptech

Ptech (de la Tecnología de Diseño Asociativo), es propietario de una serie de métodos y herramientas que cubren tanto Análisis como Diseño. Tiene suficientes características en común

con los MOO's *(Métodos Orientados a Objetos)* como para citarlos en éste apartado. La herramienta CASE de Ptech también genera código para C++ y Ontos.

Las ideas de Ptech se basan en la metáfora de los procesos de Ingeniería como la producción de sistemas por ensamble de componentes reutilizables, un método que por supuesto separa el análisis del diseño lógico para implantación. El énfasis además se enfoca a : **"en qué y cómo está hecho".** En éste sentido Ptech es un proceso orientado preferencialmente a objetos, al Diseño Orientado a Objetos y algunas ideas de la Teoría e Inteligencia Artificial.

Edwards (1989) autor de Ptech, explica la necesidad de una base formal para algunos Métodos Orientados a Objetos y critica lo que él llama : métodos *"nativos"* de AOO, tales como Coad/Yourdon o aún la TMO. El énfasis en Ptech es un prototipo. Edwards piensa que las especificaciones ejecutables son prerequisitos para la Ingeniería inversa y mantenimiento, lo cual se comentará en el Apéndice "B".

James Martin y Jim Odell (1992) describen a Ptech en uno de sus libros, el sistema notacional Ptech consiste en tres tipos de diagramas : ***Diagramas de Concepto***, llamados ***Esquema de Objeto o Esquema Concepto***; ***los Diagramas de Evento o Esquemas*** (parecidos a los diagramas de transición de estado) y, ***Diagramas de Actividad/Función*** (son muchos de los propósitos de los DFD's). Las primeras dos notaciones de diagramación son soportadas por lenguajes formales, la clase *"Calculos"* y el evento *"Calculos"* en la **Fig. A.23** sintetizan los componentes de la arquitectura principal del método.

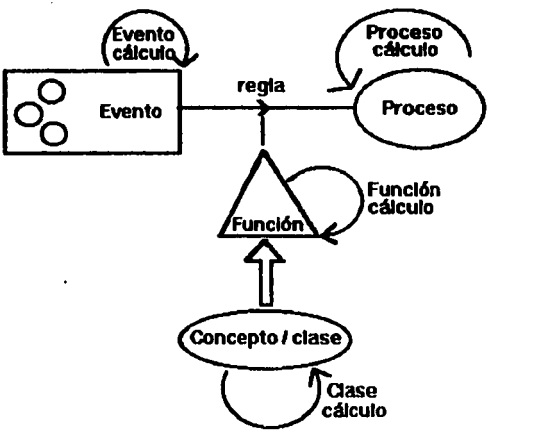


Fig. A. 23 El proceso de metáfora-ingeniería Ptech.

Los diagramas concepto muestran los aspectos estáticos de los procesos y aún su dinamismo. Es posible ilustrar cómo las instancias son creadas y terminadas y cómo se mueven adentro y afuera de las clases empleando un concepto combinado y el diagrama evento. Las actividades de los diagramas dan un alto nivel funcional visto análogamente a un diagrama de flujo de datos dado. Los conceptos se refieren a una colocación y la colocación de la teoría es utilizada para describir los conceptos usando operaciones simples como una unión, intersección y diferencia.

Las Fig's A.24 y A.25 se refieren a fragmentos del diseño de una máquina vendedora. Los eventos son conceptos de *n-lugares* -es decir, objetos con "n" atributos- los cuales tienen definidas "pre" y "post" condiciones. Las postcondiciones implican que todos los procesos tienen una meta definitiva. Con los eventos, la teórica fijación en la clase *"calculos"* es paralela a una función *"calculos"* basada en el cálculo lambda. Esta permite la definición de poner en funcionamiento reglas de negocios.

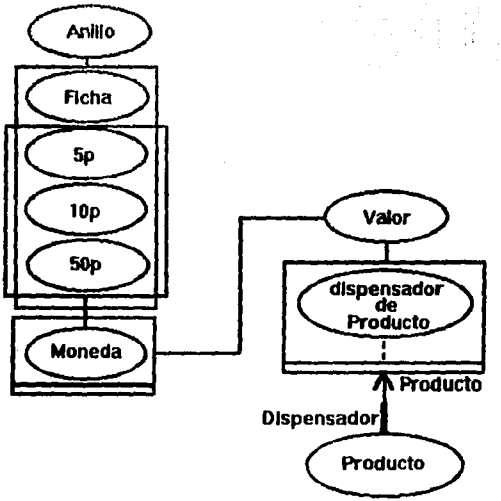


Fig. A. 24 Esquema del concepto Ptech mostrando herencia y particionando herencia .

La **Fig. A.25** ilustra un diagrama del diseño de la máquina vendedora. Los rectángulos representan eventos y flechas terminando en círculos que representan reglas o precondiciones. Los triángulos representan postcondiciones o decisiones de procesos en los cuales se evalúa verdadero o falso, dependiendo del resultado de los procesos precedentes.

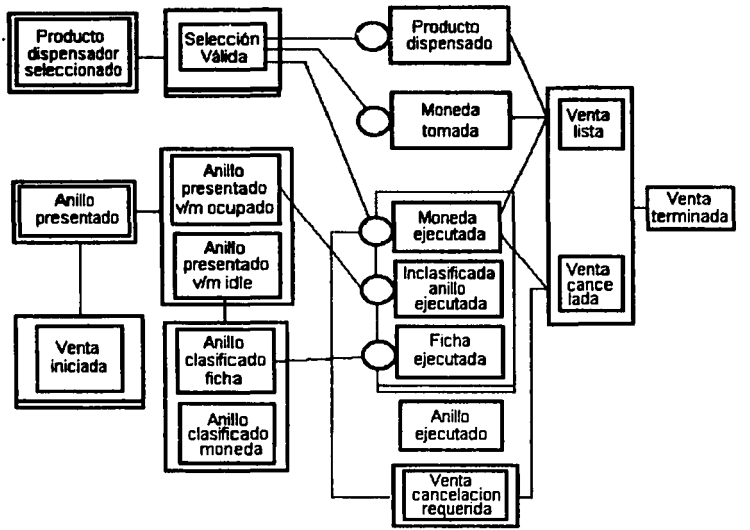
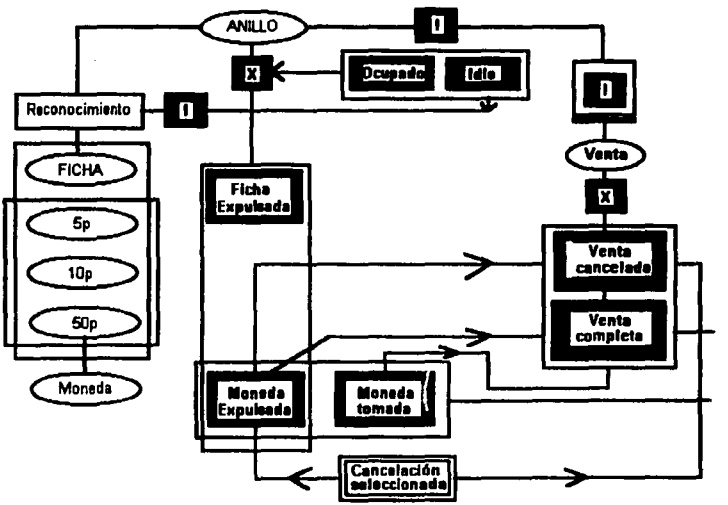


Fig. A . 25 Esquema del Evento Ptech.

La notación del modelamiento de Ptech es -plenamente hablando-, una máquina de estado finito con semántica enriquecida para señalar las restricciones, precondiciones, etc. Esta es indudablemente una notación útil, pero la notación recomendada para convertir esos diagramas al modelo objeto (lo cual involucra dibujar el bosquejo de todos los iconos y como cada icono de la clase en un diagrama objeto es separado con flechas), ésto resulta extremadamente desordenado. El principal imperfecto en el método es asumido por : ***"La operación de reusabilidad es un factor clave en la Especificación Orientada a Objetos e Implantación".*** Ian Graham se encuentra en completo desacuerdo con la anterior idea y, de hecho piensa que la reusabilidad de la clase es la clave y esto se debe a que la integración del objeto y el comportamiento de los modelos es crítico. Los diagramas del evento Ptech son especialmente útiles en el avaricioso control de estructura de un sistema. Sin embargo, los conceptos convenidos violan el espíritu de la Orientada a Objetos, porque los conceptos con métodos son mucho más que eso: son al menos Algebra. *Ptech se*

enfoca mucho más a eventos que a objetos. Por otro lado, los eventos en Ptech son objetos también y ellos pueden participar en la clasificación de estructuras. El modelo Ptech es fácil de comprender pero no es tan detallado como el TMO.

El concepto Ptech y los diagramas evento pueden ser combinados como lo muestra la Fig. A.26. Aquí las cajas dobles representan eventos externos al sistema. *Un "I" significa creación y una "X" significa terminación.*



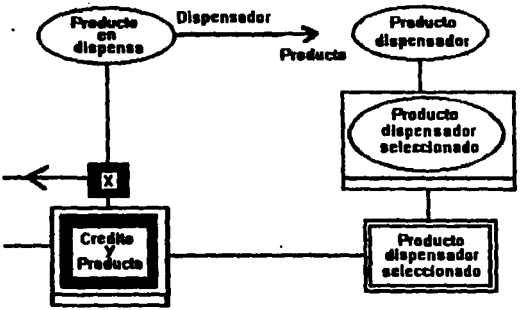


Fig. A.26 Concepto y esquema evento combinados para el ejemplo de la maquina vendedora.

El tipo final de diagrama en Ptech es el esquema de actividad/funcionamiento, espero que pocos profesionales hagan gran uso de ésta técnica como proveer nada más que una descomposición modular Top-down y, un método más de nivelamiento Orientado a Objetos, como el encontrado en HOOD o SOMA, que son más apropiados.

La **Fig. A.27** muestra como un esquema describe el flujo de un proceso de control en un restaurante de comida rápida. Los círculos son procesos y los rectángulos son los productos de esos procesos. Las flechas representan el flujo de datos o dependencias.

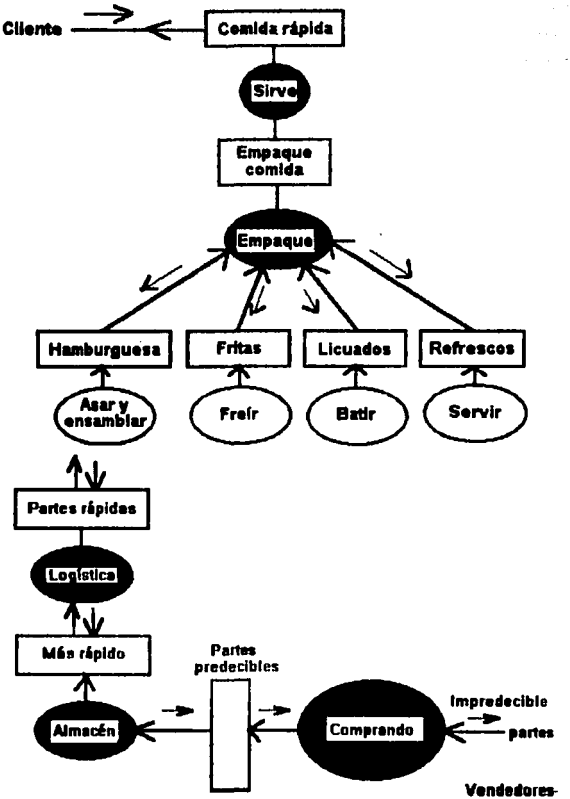


Fig. A . 27 Diagrama de actividad / función para una aplicación de comida rápida.

En términos metodológicos, ***Ptech consiste de cuatro actividades principales : Análisis de la estructura del objeto, el Análisis del comportamiento del objeto, el Diseño de la estructura del objeto y el Diseño del comportamiento del objeto :***

- ***El Análisis de la Estructura del objeto*** *(OSA, por sus siglas en inglés) involucra la construcción del concepto esquema mostrando las clases, asociaciones, clasificación y composición.*

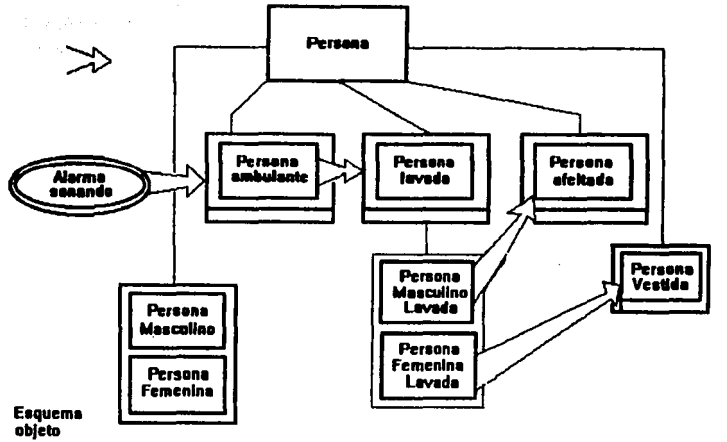
- ***El Análisis del Comportamiento del Objeto*** *usa el evento esquema y muestra las transiciones de estado, tipos de evento, reglas, condiciones de control y operaciones.*

- ***El Diseño de Estructura de Objeto*** *añade la implantación dependiendo de los aspectos de OSA y,*

- ***El Diseño del Comportamiento del Objeto*** *adiciona los detalles de los métodos.*

- ***El Comportamiento del Modelo*** *tiene suficiente fuerza expresiva para denotar precondiciones pero no así, invariancia de condiciones.*

Las reglas de negocios pueden ser expresadas como precondiciones y una sintaxis declarativa es permitida aunque sólo permita de preferencia reglas, a reglas completas. Esto podría requerir la especificación de un proceso de procedimientos. La convención notacional para mostrar el comportamiento de los aspectos a uno estructural, los cuales según Martin y Odell son "muy importantes", sin embargo resulta muy incómodo como lo muestra la **Fig. A.28.**



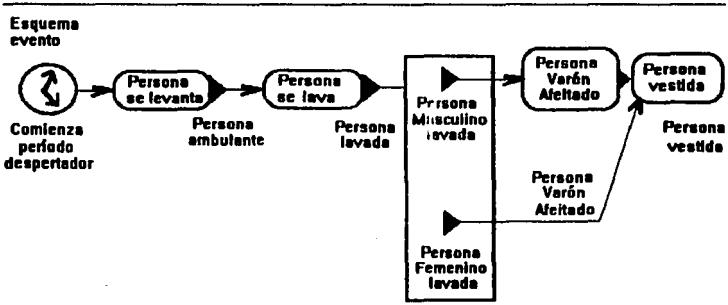


Fig. A. 28 Cruce referenciaddo del evento Ptech y el esquema de datos del objeto.

El método Ptech es sin duda, una notación muy rica y expresiva, lo mostrado representa solamente una breve descripción de éste método, el inconveniente de Ptech es que es una teoría y

una herramienta horriblemente complicada. Para un experto real su uso es satisfactorio, de lo contrario es incómodo y potencialmente burocrático en la mayoría de los métodos estructurados convencionales. Algunas de las ideas que contiene -especialmente- el evento esquema, son invaluables y deberían encontrar su camino dentro de herramientas serias de la Ingeniería de Software.

Si el lector está interesado en profundizar en el método Ptech le recomendamos referirse a Martin y Odell (1992). Por lo visto con éste método, la tecnología de objetos debe soportar negociaciones adaptables.

A.4.4.1) Estudio de CASE

El método Ptech ha sido exitosamente empleado en la UK National Health Service para el modelamiento en cuidados de la salud clínicos, aún cuando se han realizado modificaciones y extensiones (Fowler y Capey, 1991). La aplicación no es un procesamiento tradicional de datos pero concierne al dominio técnico de almacenamiento y recuperación de la información en tratamientos de pacientes. Al inicio del proyecto muchísimas piezas del sistema alimenticio existían ya en el campo, tales como la cirugía de transplantes, diabetes, etc. Esto fue deseado para construir un modelo proceso a una unidad de sistemas separados. Los métodos de respaldo fueron el SSADM y después el SSADM y después IE ha sido usado.

Los equipos fueron pequeños y aún los más interesados consistieron típicamente de uno ó dos analistas y uno ó dos clínicos (doctores, enfermeras, etc). Los especialistas clínicos fueron dando entrenamiento en la técnica de análisis y, con el tiempo, vino a ser el mejor análisis. Esto fue exclusivamente posible de lograr porque el método orientado a objetos hace modelos directos del mundo real en el campo médico.

De los tres enfoques de Ptech, el enfoque de arquitectura (el equivalente al flujo de datos) fue útil solamente en el nivel más alto. La mayoría de los Métodos Orientados a Objetos que tienen modelos de estado fueron bién descritos en la dinámica interna de los objetos. El evento esquema de Ptech fue encontrado para dar un mejor enfoque de la dinámica global.

Los esquemas combinan a los procesos con los datos de forma conjunta, dando una herramienta ideal para el modelamiento de la total iniciativa en una manera manejada de eventos. El uso de la Orientada a Objetos enfatizaba el conocimiento clinico y operacional de las reglas de negocios y su separación.

Lo más importante de Ptech es el cómo proceder desde un análisis completo a pantallas actuales, etc. Esto deja a la introducción de clases vistas que dan un dibujo simplificado y concreto de una aplicación particular basada en el núcleo del modelo, el cual es altamente abstracto. La presentación de las clases además separa el procesamiento visto de la interface y permite diferentes GUI's *(Guide User Interface)* para usar los mismos enfoques.

La Versión 1.0 de Ptech fue abandonada tempranamente en éste proyecto, subsecuentemente una herramienta dibujada de Macintosh fue utilizada exitosamente. La evaluación de la nueva versión de la herramienta Ptech se espera sea mejor que su primera versión.

A.4.5) ¿ En qué consisten:
**** Fábrica de Objetos (Objectory) e,***
**** Ingeniería de Software Orientada a Objetos (OOSE) ?***

La Ingeniería de Software Orientada a Objetos (OOSE) (Jacobson et al., 1992) es un método para el Análisis y el Diseño Orientado a Objetos, derivado de la Objectory de Jacobson ***(Object Factory ó Fábrica de Objetos)***. La Objectory es un método propietario y, la OOSE es una versión simplificada, de ésta última se dice es inadecuada para emplearse en tiempo de producción, de donde :

"Necesitarás la descripción completa..., excluyendo largos ejemplos, aproximadamente o quizá más de 1200 páginas".

Probablemente también sea necesario adquirir la herramienta CASE OrySE de la compañía de Jacobson. Este método está lleno de poderosas y útiles ideas. Jacobson es uno de los más experimentados profesionales en el AOO, la Objectory es inusual entre los MOO's ***(Métodos Orientados a Objetos)***, fue derivado originalmente de la experiencia en la construcción de sistemas telefónicos en Ericsson utilizando técnicas de bloque de diseño y es uno de los más viejos MOO's.

La interpretación funcional de la filosofía consiste en que las *herramientas* proveen soporte para actividades en tres categorías : *Arquitectura, Método y Proceso.* *La **Arquitectura** significa la elección de técnicas a ser utilizadas, el **Método** hace explícito el procedimiento a ser*

seguido y el ***Proceso*** *provee la escala del método.* El desarrollo es progresivo, involucrando ciclos iterativos de análisis, construcción y examinamiento. Cada iteración es convenida como un cambio a un sistema existente involucrando la versión y documentación en el control en el módulo y niveles de sistema. En términos de ciclo de vida, la OOSE *(Ingeniería de Software Orientada a Objetos, por sus siglas en inglés)* recomienda estados de análisis, construcción y examinamiento; la construcción está además dividida entre Diseño e Implantación. *Los requerimientos que preceden al Análisis se encuentran fuera del OOSE, los Análisis resultan en Requerimientos y Modelos de Análisis, la Construcción en Diseño e Implantación de Modelos y Examinación de ellos en un "Modelo Exámen".*

Muchas de las ideas de la OOSE son similares a aquellas de otros MOO's, pero una idea original permanece: *el uso de Case.* Usar Case's y descripciones de cómo los usuarios interactúan con el sistema, la ventaja de éste método es que los requerimientos pueden ser trazados justo como el ciclo de vida y ésto es precisamente, lo que la OOSE enfatiza. También es directa al generar escritos o guiones del uso de Case's. Otra característica del método es que es mejor que una forma usual de modelos de organización dentro de subsistemas.

El empleo de Case's primeramente en apariencia dentro de los requerimientos del modelo se realiza con los llamados ***actores***. *Los actores son entidades externas con las cuales los sistemas interactúan.* Por ejemplo, un actor podría ser un cierto rol que los usuarios juegan tales como "depositador de efectivo". El uso del modelo Case especifica el comportamiento funcional completo del sistema. Este método puede ser comparado con el agrupamiento técnico de SOMA, donde las operaciones visibles corresponden al uso de cases.

Los actores son usados para encontrar el uso de Case's y son instanciados en las iteraciones actuales con los usuarios. Otro punto a considerar es que el mismo evento puede iniciar diferentes usos de Case dependiendo del estado del sistema (modo) y el propósito y comportamiento subsecuente del usuario. Los usuarios necesitan saber cuál Case usar para ejecutar la salida. Este es un método útil alcanzando un alto nivel del sistema, lo cual es mejor que emplear diagrams de flujo de datos, particularmente porque requiere que nos preguntemos las expectativas del sistema, del usuario y, viceversa. La OOSE no es realmente el uso de la idea case.

La Fig. A.29 muestra un uso en alto nivel del diagrama Case.

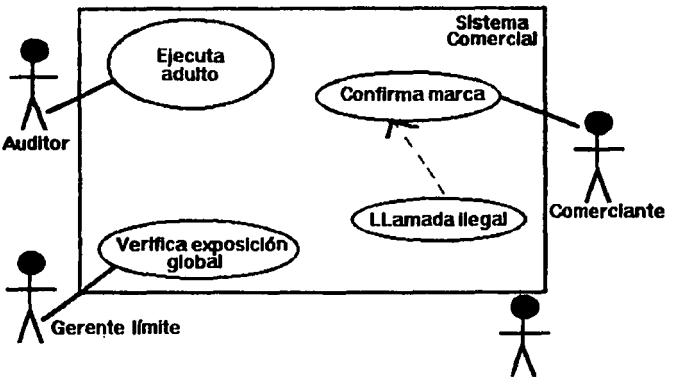


Fig. A.29 El nivel más alto en el uso del Diagrama Case para un sistema comercial financiero mostrando un subuso case para señalar la falla a confirmar el incumplimiento de las regulaciones.

El uso del modelo Case es empleado para generar un campo del modelo objeto con objetos dibujados desde las entidades de los negocios. Dicho modelo es entonces convertido a un modelo de análisis por clasificación de los objetos del campo dentro de tres tipos: ***interface de objetos, la entidad de objetos y control de objetos.*** Esas entidades de objetos enfatizan el almacenamiento y son estables, si consideramos que su estabilidad consiste en que son normalmente persistentes.

El control de los objetos enfatiza su comportamiento. Ian Graham piensa que es una buena idea estrechamente relacionada a las ideas clásicas de la independencia de datos, si todo objeto tiende a ser una entidad del "mundo real" entonces el control y la presentación del comportamiento podría ser dividirlo a través de muchísimos objetos, haciendo así el cambio del comportamiento difícil conforme los requerimientos evolucionen. Esta es una buena idea y es empleada solamente para el comportamiento verdaderamente excepcional en un Case que no pertenece a un objeto de negocios. En la práctica muchísimos Case's usados pueden tener objetos en común como lo muestra la Fig. A.30.

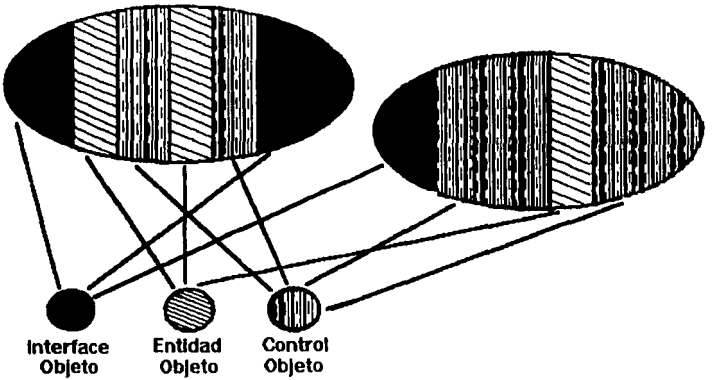


Fig. A. 30 Mapeando el uso de case al dominio de los objetos en la OOSE.

El análisis del modelo soporta herencia y asociaciones de instancia que son utilizadas también para registrar agregación. Las asociaciones son arcos dirigidos, los cuales ayudan a preservar la encapsulación.

El análisis del modelo es entonces convertido a un diseño del modelo, expresado en términos de ***"bloque"*** que son : ***implantaciones de uno ó más objetos.*** Los bloques pueden ser señalados como entidad, interface y control, nuevos bloques introducidos representan la implantación de los conceptos del campo. Los bloques son posteriormente convertidos a módulos de los lenguajes destino. Esos bloques están conectados usando diagramas de interacción. Esta es una técnica interesante, similar a los diagramas de tiempo de Booch, los cuales direccionan una de las más débiles en la mayoría de otros métodos : su habilidad expresa dinámica global. El próximo paso es generar el código origen anotado como la implantación del modelo y entonces procede al examinamiento. Un peligro grande con la OOSE es asumir que todas las secuencias pueden ser expresadas en el uso de Case's y aquí de nuevo en diagramas de interacción. Para muchos sistemas complejos y al menos todos los sistemas expertos ésto no puede ser hecho. En efecto si esto podría ser implicar un modelo de la interface de usuario y, Thimblebly (1990) ha mostrado de manera verdadera y profesional cómo ésto podría ser en general no deseable . Sin embargo, el diagrama de interacción es útil y se añade a nuestro conocimiento de la dinámica

global, aunque no exhaustivamente. Lo cual nos lleva de nuevo al uso de Case's de los requerimientos de estado directamente.

La dinámica interna de bloques es descrita con Diagramas de Estructura de Datos en una manera similar a Booch ' 91. Este tiene la longitud de habilitación simple de código generador a ser escrito pero no ayuda con los objetos del estado, lo cual no significa que su conducta podría ser inimaginable para objetos con muchos estados.

Los objetos son organizados dentro de subsistemas en ambos modelos : análisis y diseño. En el nivel más bajo son considerados como *"servicio de empaques"* que tampoco resultan deseables del todo.

La implantación del modelo convierte bloques o módulos para un lenguaje específico (C++ clases, ADA, etc.) y consiste principalmente de anotación de código origen. El modelo exámen consiste de una especificación de exámen derivado de Case's y un resultado de exámen y reporte. La OOSE tiene adaptadas muchas ideas útiles del examinamiento convencional de literatura (particularmente, Myers, 1979) a Orientada a Objetos. Las clases, bloques y subsistemas (paquetes de servicio) son primero unidad examinada y entonces la integración "examina" es ejecutada en todas las clases.

Ambas técnicas, blanco y negro son aplicadas a una unidad "examina". La Ingeniería de Software Orientada a Objetos (OOSE) puede ser adaptada a ambos sistemas de base de datos en tiempo real y relacional. Como con la Técnica del Modelamiento de Objetos (TMO), la jerarquía será dada al asistir en la translación de clases a tablas, marcada con herencia, etc.

La OOSE tiene una buena notación simple con sólo 9 nodos y 6 símbolos de ligas. Representa : intensidad y debilidad, comparado a un método como la TMO. No resulta muy claro que la OOSE soporte el prototipo como un método excepto por el UI, aunque el desarrollo incremental se fortalezca, como en el caso de sonido, riesgo, análisis y la colección de métrica. En el Case de desarrollo incremental una caja de tiempo entre 3 y 6 meses es normal y podría típicamente implantar entre 5 y 20 Case's.

A.4.6) OORASS (Rol del Análisis Orientado a Objetos, Síntesis y Estructuración)

El OORASS (Rol del Análisis Orientado a Objetos, Síntesis y Estructuración) fue desarrollado por Reenskaug (1989, 1990) y otros en Taskon AS en Norway, así como propietarios

remanentes. Cubre tanto análisis como diseño y es inusual en el enfatizamiento de roles jugados por los objetos y, direccionando muchísimas áreas en un ciclo de vida evolucionario.

El análisis comienza con el descubrimiento de *áreas de interés*, las cuales son, efectivamente funciones de negocio de alto nivel y, la unidad para la formación coherente de subsistemas. Posteriormente cada área es modelada usando agentes de colaboración y objetos los cuales pueden caer en varios roles. Los roles se obtienen de todos los objetos con la misma posición, sin la estructura de un área de interés. Los roles son interpretados en una forma similar de un sistema de preferencia que restringe a los agentes externos. Los diferentes objetos pueden adoptar diferentes roles en diferentes contextos. Los roles han reoriginado ***requerimientos : referencias a servidores, competencia : cosas que ellos conocen, deberes y derechos : operaciones.*** El rol "X" puede "saber acerca" del rol Y, quien contiene una referencia habilitando mensajes a ser enviados.

Esas ligas "cliente/servidor" son refinadas por parte de símbolos como lo muestra la **Fig. A.32**. El círculo -sólo parte de él- indica que el rol "X" no sabe nada acerca de una instancia de "X". Ninguna parte del símbolo indica que el rol al fin de la liga no sabe acerca del otro rol. Lo cual introduce la multiplicidad semántica del modelamiento de entidad dentro de los diagramas y puede ser útil para la captura de requerimientos de alto nivel en una forma similar a la manera en la cual los Case's en la OOSE son usados con herencia.

Fig. A .32 Rol del diagrama de OORASS .

Cada parte del símbolo puede haber sido asociada a una colocación de operaciones denominadas *un contrato*, quien define lo visible del rol que puede ser capaz de ofrecer. **OORASS** es neutral en la notación a ser empleada para la representación de objetos.

El análisis es convenido como uno Top-down, pero no jerárquico, los procesos son analogías con el método de SOMA en éste respecto, aunque los modelos del rol no son descompuestos como

tales. El modelamiento se considera finalizado cuando los analistas entienden por completo el área en cuestión y el modelo es estable. Si no, o si el área fue descompuesta, el analista debe redefinir el área y repetir la actividad.

Los tipos o tipos de objetos, son sintetizados desde los roles y éstos definen el comportamiento visible de algunos objetos. *Las síntesis* crean nuevos objetos debido al comportamiento de la herencia de algunos objetos simples. Terminológicamente, OORASS define clases como : *la implantación de tipos y permite redes de tipos a diferencia de clases de redes.* La herencia múltiple es permitida. La estructuración usa un meta-modelo para especificar el comportamiento en tiempo-de-corrida de los objetos, dado que están obligados con otro durante la iniciación. Esta capacidad es notoriamente ausente en la mayoría de los MOO's. Más formalmente, la síntesis comienza con un área de interés y busca los eventos y respuestas. Este mapea dentro de mensajes y operaciones. Los objetos candidatos son identificados, aunque las técnicas no formales son propuestas con OORASS y, entonces las tarjetas CRC son empleadas para identificar y asignar operaciones y colaboraciones. Después, los diagramas de sencuencia de mensajes son dibujados seguidos por los diagramas de rol. Por último, las secuencias de mensajes son usadas para identificar el contrato con restricciones e invariantes.

La ventaja de éste método es que a los analistas se les permite construír simples modelos de roles representando la interacción de objetos y entonces combinarla en una manera uniforme para crear modelos largos. Esto es reclamado para incrementar la reusabilidad de los productos de análisis y diseño y es similar a la filosofía del uso del método del CASE (Objectory).

OORASS es soportado por una herramienta CASE llamada OORAM. El método es ideal en la construcción de sistemas que enfaticen el paso de mensajes y la distribución.

A.4.7) Desfray y el Método "Clase-Relación"

Desfray (1992) describe la relación de clase y método para Análisis y Diseño Orientado a Objetos actualmente señalado por una compañía francesa, Softeam, en el contexto de los proyectos de C++ para el cual está específicamente diseñado. El método es notable por el uso de una notación derivada de los modelos de relación de entidad Chen y además potencialmente compatibles con Merise, el método estructurado es ampliamente usado en Francia. Es notable para la atención prestada a métodos formales a través del uso de las aserciones como Eiffel. Una herramienta CASE, la Objecteering, se encuentra disponible y generará el código C++.

El método relación-clase es un método de "tres puntos" en el que existen tres modelos separados para cada sistema; un modelo entidad/objeto, un modelo de transición de estado y un modelo de flujo de datos. Como con otros métodos de tres puntas tales como TMO, es a menudo difícil ver la utilidad del modelo de flujo de datos y los modelos de paso de mensajes tan ricos como con DFD's. Esta es una respetable forma de hacer AOO y permanecerá fácilmente la comparación con TMO. Algunas mejoras, tales como un refinamiento adicional de la noción de *"relación abstracta"* son posibles y, el modelo de flujo de datos especifica el ligado de procesos y contiene una noción de eventos y su ordenamiento. La relación-entidad y las notaciones de transición-de-estado son simples y más naturales y, la habilidad de usar aserciones direcciona una deficiencia fundamental de la mayoría de otros métodos (las excepciones son : BON, Ptech y SOMA). La secuencia de técnicas es similar a la TMO : construir el modelo, datos/clase, encontrar las aserciones, usar el modelo dinámico a marcar con eventos y la secuencia de métodos y finalmente usar los DFD's requeridos. El concepto de encapsulación es enfatizado lejanamente, más que en el TMO. Desfray sugiere que el modelo de transición de estado podría ser mejorado con la introducción de estados específicos y un mecanismo para síntesis de estado. En los términos que el método cubre, los modelos de relación-entidad incluyen soporte completo de herencia en adición a todos los modelamientos familiares de datos. Mientras las relaciones pueden ser convenidas como objetos.

Desfray hace una distribución importante entre clases y relaciones. "Las relaciones nunca poseen métodos. La única función es conectar clases". Ayuda con la distinción entre atributos y clases. Una relación es una liga dirigida entre dos clases que permiten a una clase saber acerca de las instancias de la clase. Las relaciones bidireccionales son impedidas. El piensa que *"una relación es parte de la definición del concepto que una clase representa".* La próxima innovación es la introducción de la clase heredable invariantes correspondiendo a una forma limitada de reglas de negocio. Las clases son convenidas siendo formadas por una composición o agregación de sus atributos, convenidos como tipos de objetos. *Una regla general en el modelamiento de la entidad es que solamente los atributos autorizados son quienes pertenecen a una clase base.* Esta ayuda a evadir la confusión originada por los campos mezclados o herencia múltiple: no se permite que un elemento sea parte de dos clases en éste método. Las clases son organizadas dentro de esquemas de datos correspondientes a áreas sujetas Coad/Yourdon, pero en común con el método de ese esquema aparece por falta de la semántica de clases, como un mensaje no puede ser enviado a ellos *"Un esquema es la representación de un campo".* Este se encuentra compuesto por un grupo de clases. Constituye la especificación de un campo particular. Una clase pertenece a uno y solamente un esquema. Una aplicación es así particionada dentro de esquemas de datos: "El método Relación-Clase permite (mutuamente) usar relaciones (paso de mensajes) entre clases pero lo prohibe entre esquemas de datos" (las clases son parte del mismo

esquema). Mejorando ésta situación, ciertas clases pueden ser declaradas como "interface de clases" para el esquema. El esquema de datos también posee invariantes definidos como la unión de todas las invariancias de clases contenidas más "algunas cláusulas generales". Para aclarar esto, el uso mutuo es definido como sigue :

Si las clases C1 y C2 pueden usar cada una otros servicios : la relación es mutua. Esto último solamente puede ocurrir con un esquema. Sin embargo, si el esquema S2 puede usar S1 entonces para S1 está prohibido emplear a S2.

Existen tres categorías para usar la relación entre clases:

(1) Uso operacional : C1 usa C2 solamente para permitir un método de C1 ejecute la terminación.

(2) Uso contenido : C1 usa a C2 como un parámetro de un método de C1.

(3) Uso característica: C1 usa a C2 cuando C1 hereda de C2. El esquema S1 tiene una liga empleada con otro esquema S2 cuando al menos una de las clases de S1 utiliza al menos una clase de S2. Las clases de S1 pueden accesar solamente la interface de clases de S2 y no es parte del "cuerpo" de las clases. En adición, el esquema de datos puede tener ligas que se hereden entre ellas.

La inovación más importante es el uso de aserc ones, aunque se encuentren restringidas a precondiciones y postcondiciones. Las condiciones de invariancia no son mencionadas, las cuales solamente afectan la aplicabilidad del método a sistemas donde el paralelismo es importante. Las clases invariantes son, sin embargo, soportadas.

A.4.8) OSA (Análisis de Sistemas Orientados a Objetos)

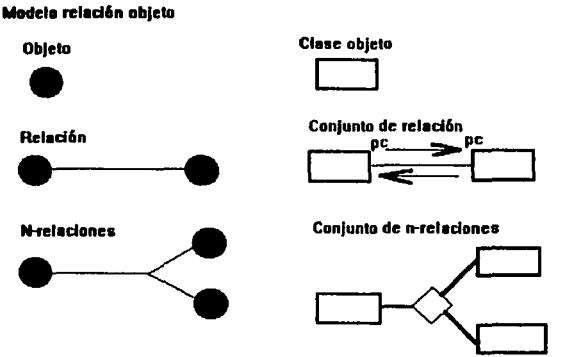
OSA (Análisis de Sistemas Orientados a Objetos) es un método desarrollado en Hewlett-Packard y es similar a la TMO de Rumbaugh en el que el método sigue la división convencional de análisis dentro de tres actividades separadas (pero relacionadas) con una notación para cada uno. Difiere en ser una notación escasamente simple y en un número de otras formas.

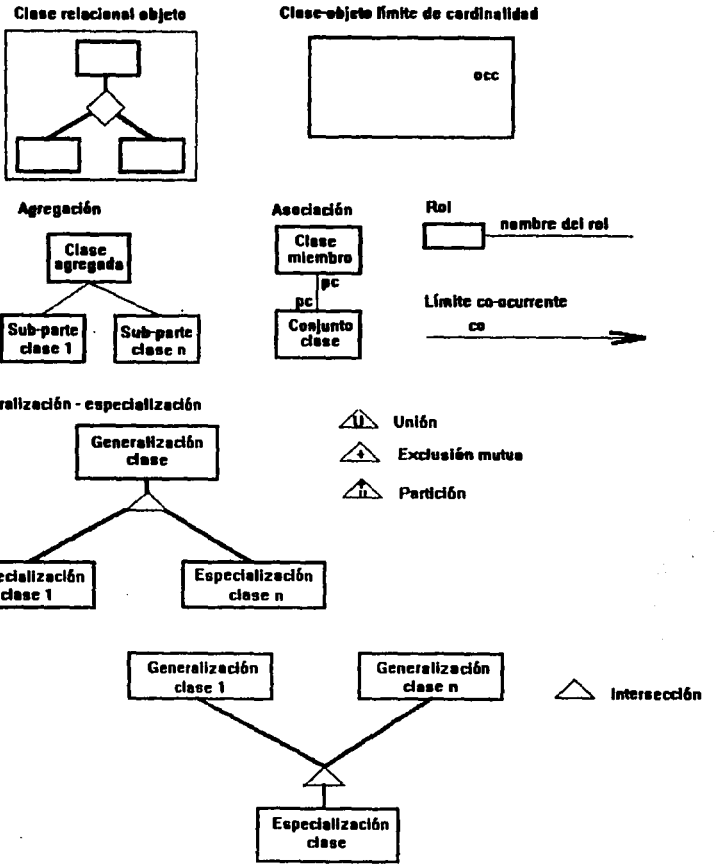
OSA comienza con una notación de "entidad-relación" *-el objeto relaciona al modelo u ORM-* el cual le permite la descripción de los atributos, clasificación y agregación de estructuras y datos semánticos en la forma de asociaciones. Las limitaciones más generales son meramente escritas

en los diagramas en forma de notas próximas al objeto al que se refieren y así las limitaciones no tienen relación para heredar. La ventaja confirmada de éste método es que no limita al analista. Sin embargo, Embley et al (1992) dijo que :

"Desde que las notas no limitan a las clases de objetos o a sus relaciones, un diagrama ORM tiene el mismo significado con, o sin ellas: así, parece ser que la información se pierde en el modelo final".

La colocación completa de notaciones disponibles con el ORM se muestra en la **Fig. A.33.**




Clase relacional objeto
Clase-objeto límite de cardinalidad
occ
Agregación
Clase agregada
Sub-parte clase 1
Sub-parte clase n
Asociación
Clase miembro
pc
pc
Conjunto clase
Rol
nombre del rol
Límite co-ocurrente
co
Generalización - especialización
Generalización clase
Especialización clase 1
Especialización clase n
Unión
Exclusión mutua
Partición
Generalización clase 1
Generalización clase n
Intersección
Especialización clase

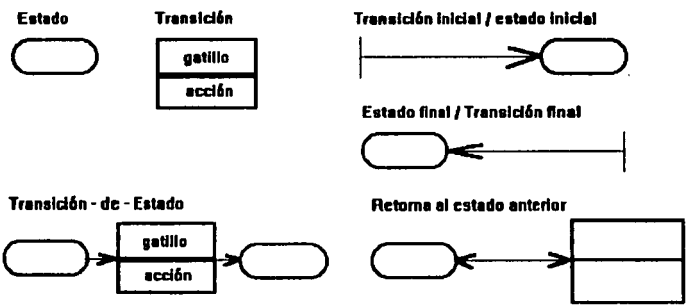
pc = límite de participación

co = límite co-ocurrencia

occ = límite de cardinalidad objeto-clase

Fig. A . 33 Modelo de relación objeto OSA

Una notación de transición de estado permite al analista describir el comportamiento de cada objeto. Este ***Modelo de Comportamiento del Objeto (OBM, por sus siglas en inglés)*** define los métodos para el objeto, pero el ***Modelo de Relación Objeto (ORM, por sus siglas en inglés)*** no puede ser extendido para incluirlos. Las limitaciones en tiempo-real, las excepciones y los eventos son todos mantenidos con el Modelo de Comportamiento del Objeto y es aquí donde la similaridad con la Técnica del Modelamiento de Objetos es más aparente, aunque el énfasis está en el modelamiento del comportamiento interno de los objetos de preferencia que las interacciones entre objetos. La Fig. A.34 ilustra las notaciones del Modelo de Comportamiento del Objeto.



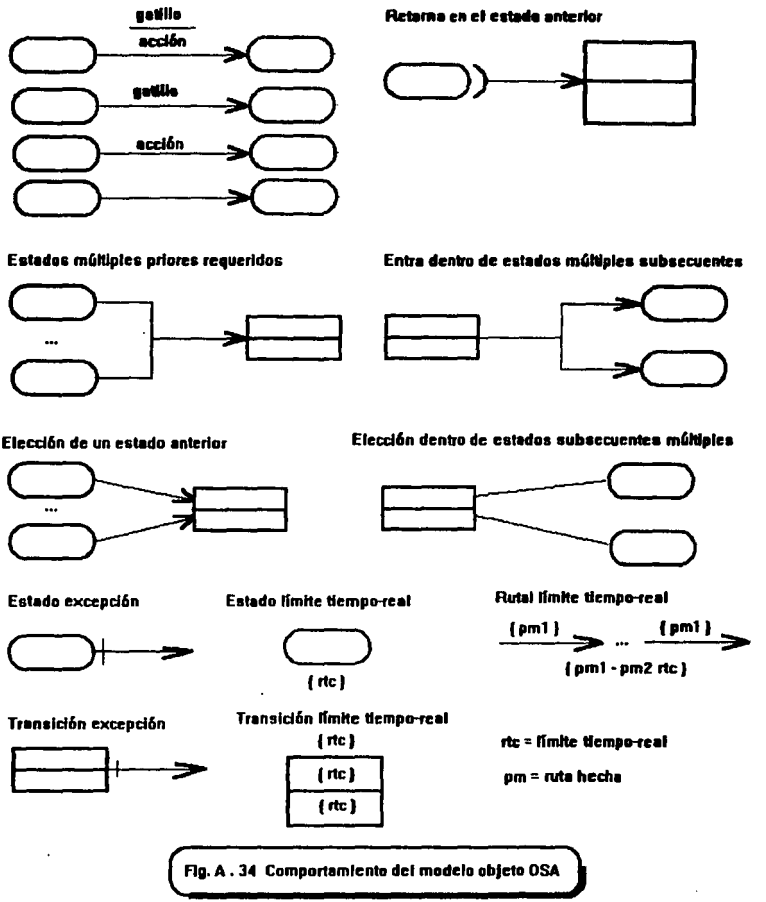


Fig. A . 34 Comportamiento del modelo objeto OSA

Finalmente, el paso de mensajes es descrito en un diagrama de flujo de datos como el ***Modelo de Interacción del Objeto (OIM).*** La ventaja sobre la Técnica del Modelamiento de Objetos es que éste modelo señala más evidentemente con el paso de mensajes que el flujo de datos es atemporal, lo cual podría ser argumentado por quienes soportan cada aspecto.

El buen nivelamiento facilita o permite al analista moverse entre los niveles alto y bajo de propuestas de un sistema. Las propuestas son clasificadas como dominantes o independientes de acuerdo a cuando el nivel alto propuesto es el nombre de una clase existente o una con identidad separada respectivamente. La descomposición de las reglas está dada por ambos : ***Modelo de Relación Objeto y Modelo de Comportamiento del Objeto*** y, ésto confirma que la habilidad de estructurar todo modelamiento construye consistentemente con una propuesta de alto nivel es una característica única del Análisis de Sistemas Orientados a Objetos. Gran parte de ésta notación parece innecesariamente compleja desde algunos puntos de vista, pero las ideas son indudablemente útiles. Cada uno de los ORM, OBM y OIM puede ser nivelado en ésta forma. Embley detalla los diagramas que contienen una gran cantidad de anotaciones suplementarias de cada cosa como limitaciones, excepciones y transición de prioridades.

La notación del Modelo de Interacción del Objeto es quizá la más útil y difiere de otros métodos de AOO. Sin embargo, podría no ser fácilmente incluído dentro de otros métodos, en los que el analista sea capaz de ligar mensajes a estados internos y transiciones de cada objeto. Algunas de las cosas que uno podría hacer en el Modelo Dinámico de la Técnica del Modelamiento de Objetos están hechas aquí, en el método de Análisis de Sistemas Orientados a Objetos *(OSA, por sus siglas en inglés).*

OSA es un "modelo manejable" en el sentido de que un modelo es construído de preferencia preguntando al analista para seguir una serie de pasos mezclados -caracterizados como un *"método manejable"*. Este evidentemente mapea bién dentro de la mayoría de la comunidad profesional Orientada a Objetos. Como con Ptech y con la Técnica del Modelamiento de Objetos (TMO), éste permanece no nuclear, como los tres resultados de análisis independiente no son combinados dentro de un sólo modelo objeto, aunque alguna atención es presentada para consistencia y conformancia entre diagramas. Está confirmado que el OSA no hace distinción entre clases y atributos y que ayuda a reforzar la consistencia. Esta es una confirmación difícil de aceptar dada la inmensa filosofía significativa del problema objeto/atributo. OSA permite herencia múltiple notacionalmente, pero en común con Coad/Yourdon, omite algún tratamiento de solución de conflicto.

Una de las simplicidades de la Orientada a Objetos es : la habilidad de expresar estructuras complejas directamente en términos de alto orden.

OSA es un método temario razonablemente completo, el cual desde el punto de vista de Graham, posee mejor notación. Sin embargo, la popularidad de OMT y la escasez de herramientas CASE OSA pueden prevenirlo del gran éxito.

A.5) Ingeniería de Sistemas Orientados a Objetos

La Ingeniería de Sistemas Orientada a Objetos o bién ***SEOO*** *-por sus siglas en inglés-* es el método soportado por la compañía UK LBMS y, tiene 4 aspectos :

(1) Trabajo que rompe estructuras y técnicas

(2) Un método de modelamiento compartido de objeto.

(3) Especifica técnicas de diseño GUI.

(4) Ligado de bases de datos relacionales.

Un método es visualizado como un proceso modelo, además de las técnicas con sus reglas y los productos finales. El propósito del ciclo de vida es un desarrollo estándar rápido y el énfasis se coloca al final de los estados con algunos ejecutables que pueden ser evaluados por los usuarios.

Una de las grandes ventajas de un manejo de base de datos convencional del sistema es la separación de los procesos y los datos, lo cual dá una noción de independencia de datos y beneficios de flexibilidad e insulación de cambio, porque la interface compartida de datos es estable. El modelo de datos es convenido como un modelo de la estática de la aplicación. Con la Orientación a Objetos, los procesos y los datos son integrados. Significa esto que tenemos que abandonar los beneficios de la independencia de datos? Ya en bases de datos relacionales cliente/servidor hemos visto un paso en la misma dirección, con la puesta en funcionamiento de las bases de datos y la integridad de las reglas almacenadas en el servidor con los datos. Con un MOO para la manipulación datos parece razonable adoptar la propuesta de que hay dos tipos de objeto, a los que Ian Graham denominó ***objetos del dominio y aplicaciones de los objetos.*** ***Los objetos del dominio*** *representan esos aspectos del sistema que son relativamente estables o genéricos.* ***Las aplicaciones de los objetos*** *son aquellas quienes pueden ser esperadas que varíen de instalación a instalación o de tiempo en tiempo rápidamente.* Este método resucita la noción de independencia de datos en un esfuerzo de la forma Orientada a Objetos.

***Un modelo convencional de datos** es una propuesta del dominio de los objetos, el cual incluye límites, reglas y dinamismo (transición de estado, etc).* Lo significativo es hacer la interface a ésta parte del modelo como estable y posible. Los objetos del campo forman el modelo objeto compartido. La mayoría de las interacciones entre componentes es vía éste modelo y por ésta razón la SEOO los designa como **objetos compartidos.**

Cameron (1992) argumentó que los métodos estructurados para los sistemas de procesamiento de datos, es una técnica altamente efectiva para el Análisis y el Diseño. Sin embargo, la mayoría de la riqueza semántica disponible para el analista se pierde en la representación. Además, él afirma que en la separación rechazando datos-procesos muchos métodos de AOO pierden los beneficios del modelamiento de datos y él propone la técnica del modelamiento compartido de objetos como un sucesor directo del modelamiento de datos. La técnica está intencionada a ser Orientada a Objetos, todavía retiene las ventajas del modelamiento de datos y se encuentra bién situada dentro de las aplicaciones comerciales del procesamiento de datos.

El modelamiento de datos es exitoso porque :

* ***El análisis de datos*** es empleado como una herramienta del modelamiento. El análisis usa modelos de datos como una base para el entendimiento de la funcionalidad. El modelo de datos es relativamente estable a la información requerida que soporta.

* ***Los componentes del procesamiento***, no subsistemas o transacciones individuales, interactúan vía los datos compartidos descritos por el modelo de datos y mantenidos en la base de datos.

* ***Hay una ruta directa*** desde un modelo de datos a un diseño preliminar de base de datos.

En esta propuesta, el modelamiento de datos debe ser generalizado a una forma de modelamiento de objeto y la división entre persistencia de datos y todos los procesos reemplazados por una división entre objetos compartidos y otros objetos. La Orientada a Objetos proveé un poderoso medio de modelamiento que datos. Los objetos compartidos similarmente forman las interfaces importantes entre subsistemas.

Las instancias -análogamente los datos- que son compartidos, no sólo las clases -análogamente las definiciones de datos-. Compartiendo, posiblemente a través de muchos sistemas, imponen una disciplina en el rango de métodos que un objeto compartido debe soportar.

La especificación del procesamiento a una aplicación particular debe ser excluída. De otra manera la definición de clase del uso amplio de objetos, debe encontrarse continuamente bajo revisión según las aplicaciones sean añadidas y modificadas. Crear subtipos separados de cliente para cada aplicación no ayuda, porque hace a los objetos que son compartidos : individuales y, no pueden cambiar su tipo para cada aplicación.

La funcionalidad específica de la aplicación es modelada fuera del modelo objeto compartido, en términos de otros objetos (por ejemplo, usar interface de objetos) con el uso de servicios provistos por los objetos compartidos.

Un beneficio de un sistema desarrollado acerca de un modelo objeto compartido es que las aplicaciones pueden ser relativamente independientes, es exactamente la forma en la que muchísimos programas pueden interactuar independientemente con una base de datos. Una aplicación puede ser extendida con nuevos o modificados objetos, o tener objetos borrados, con efectos mínimos en otras aplicaciones, provisto que las interfaces de objeto compartidos no son cambiadas.

Pocos de los ampliamente conocidos métodos de AOO y DOO hacen o dan importancia a la distinción entre compartido y otros objetos. Todavía parece un pre-requisito para el éxito de un método de AOO en los negocios de la computación.

El diagrama en la **Fig. A.37** describe la técnica del modelamiento compartido de objetos con más detalle. El alto nivel de integración es importante. Las partes individuales tienen equivalentes en la mayoría de los métodos estructurados. Un mode o de datos es equivalente a una propuesta del modelo objeto. Esas descripciones son desarrolladas como propuestas de una integridad completa de preferencia que como modelos separados a ser integrados después. La integración distingue el modelamiento compartido de objetos de los métodos estructurados.

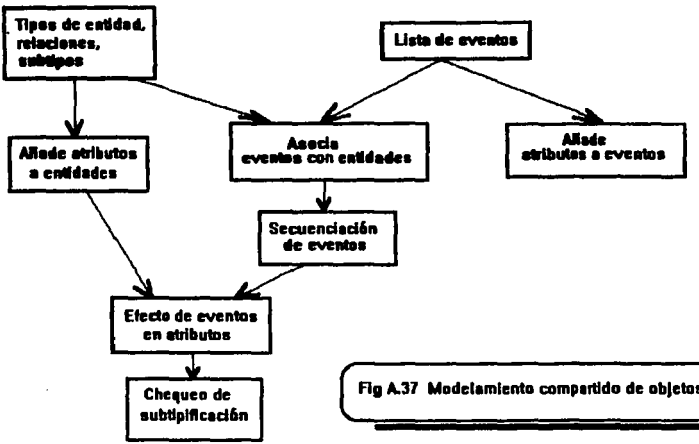


Fig A.37 Modelamiento compartido de objetos

El modelamiento GUI es convenido como un tipo especial de actividad de análisis. El contexto de los diagramas y algo de preferencia similar al uso de Case's son empleados para definir tareas y salida, además los principios de diseño son aplicados. El diseño de la ventaja es separado del uso del diseño objeto, resultando en un modelo de tres capas en el cual existen ventanas empleando los servicios del uso de objetos en la capa próxima en el uso de servicios de objetos compartidos.

Para interfaces de base de datos relacionales un mecanismo parecido a propuestas de base de datos es usado para superar los problemas asociados con compartir: *"muchos objetos de corta vida implantan un objeto de larga vida".*

La Ingeniería de Sistemas Orientados a Objetos es un método híbrido el cual es importante para la contribución de capas, el modelo objeto compartido y el énfasis en la migración evolucionaria de las tecnologías existentes.

A.6) Texel

Texel es un método de AOO y DOO, con fuertes raíces en el desarrollo mundial de Ada, aunque se afirma que los desarrollos en C++ pueden también ser soportados y la herencia no es descuidada como en HOOD. El método es soportado por PPTexel y la Compañía de New Jersey.

Las actividades con Texel son las siguientes :

Primero, nominar tipos de objeto candidatos (llamados clases de objeto), atributos y etc. Esto podría considerarse como la entrada de una especificación escrita, los documentos del sistema (tales como DFD's y diccionarios de datos) o entrevistas con el experto en el área. No especifica que la entrevista o las técnicas de identificación de objeto sean recomendadas. ***"La disposición de claves"*** son añadidas para cada frase mostrada, aunque la entrada es una clase, atributo, duplicado, descendiente, procesos, etc. Estos procesos convierten a la ***"lista de candidatos del objeto clase"*** a la ***"disposición de la lista"***. El siguiente producto es la ***"lista de la línea base del objeto clase" (BOCL,*** por sus siglas e inglés*)* quien representa la lista de clases en éste estado.

Las máquinas de estado finito son usadas para el comportamiento posterior que se encuentra, después de la producción de la síntesis del ***Diagrama de Análisis de la Clase Objeto (OOCAD)*** y los ***derivativos diagramas de análisis de la clase objeto (OCAD)***. Dos documentos, la ***especificación de la clase objeto (OCSD)*** y la ***especificación de relación (RSD)***, son los producidos, al final de la fase estática de análisis. Los primeros documentos los ligan en este diagrama descriptivamente. Estos son parcialmente ilustrados en las **Fig. A.39 y A.40** usando el mismo ejemplo de monitor de ambiente sellado.

4. Alarma_Audible

Alarma_Audible (Pseudo_Id.Estado)
Descripción : La Alarma_Audible es un dispositivo que notificará al Operador HCC cuando el valor de una de las condiciones ambientales se desvía desde el valor normal por el 3 % ó más .

Existe solamente una Alarma_Audible asociada con el HCC SEM .

La Alarma_Audible está encendido en SEM y apagado por el Operador .

4.1 Alarma_Audible.Pseudo_ID

Descripción : .....

Fig. A . 39 Parte de una Alarma_Audible - OCSD .

R5. Sensor (APUNTAS)Alarma_Audible (Mc:1)
Alarma_Audible (ES APUNTADA POR) Sensor

La severidad de la desviación entre el valor actual leído por el sensor y éste complemento en cualquier dictado Nominal_DB o no la alarma debe sonar.

Condición : Si la desviación entre el valor leído por un Sensor y el valor nominal en el Nominal_DB es 3% o mayor, entonces y sólo entonces, la Alarma_Audible sonará.

R6. Operadores (MONITORES)Alarma_Audible (1:1)
...

Fig. A . 40 Parte de un RSD .

La fase uno del método Texel es ilustrada esquemáticamente en la **Fig. A.41.**

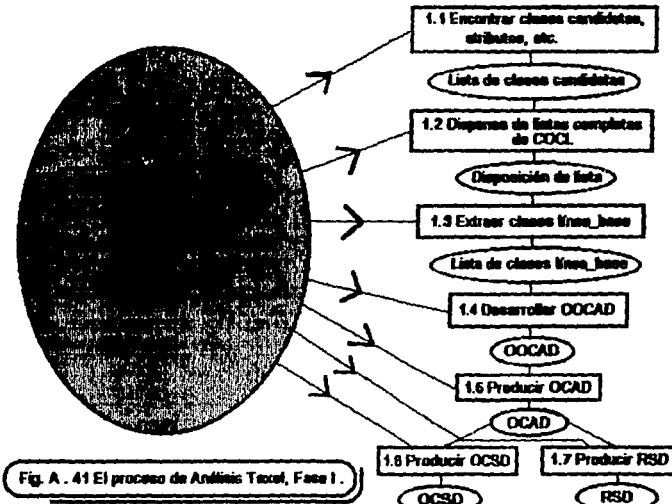


Fig. A . 41 El proceso de Análisis Texel, Fase I .

La Fase 2 del análisis resulta en modelos de estado de cada clase. Esta actividad toma un método similar al de Shlaer y Mellor (1991). El propósito es identificar los eventos y métodos para cada objeto. Esto, por supuesto no ayuda mucho para los objetos sin el estado significante, el cual ocurre comúnmente en los modelos de negocio, ésta es una de las razones por las que los métodos semejan Fusion con los modelos de estado. La solución Texel es producir un estado falso para los objetos ofendidos. Una de las mejores características del método es que el ***Modelo de Comunicación de la Clase Objeto (OCCM)*** producida en éste estado permite mensajes y flujo de eventos a ser analizados. Los mensajes son mostrados explícitamente, sin embargo, la notación puede no escalar realmente a sistemas comerciales complejos. La Fig. A.42 muestra el modelo de comunicación de la clase objeto para el ejemplo del monitor de ambiente sellado.

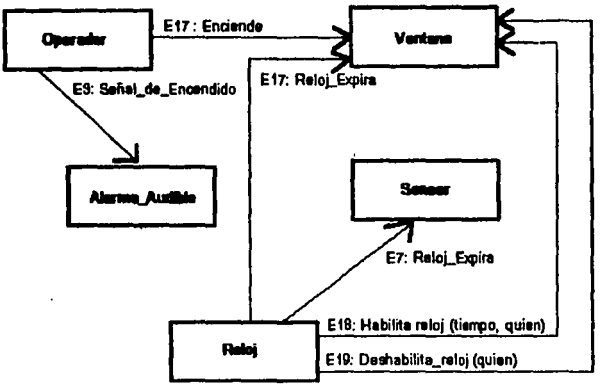


Fig. A . 42 Ejemplo de OCCM

El paso final del análisis es la producción de los modelos de estado para cada clase como lo ilustra la **Fig. A.43 (a)**. La **Fig. A.43 (b)** muestra cuán complejos pueden ser esos modelos de estado justo para cosas directas como ventanas.

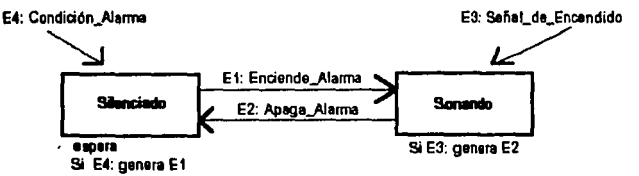


Fig. A . 43 (a) El modelo de estado Texel para una Alarma_Audible;
(b) El modelo de estado Texel para una Ventana.

La herencia múltiple está prohibida. Las limitaciones complejas y las reglas no son representables en ningún procedimiento y las aserciones no son parte del estado de diseño. El Diseño procede vía una estrategia informal similar a la de Booch. Las clases que serán los objetos del diseño están definidas y ligadas por un algoritmo principal fuera de línea. La herencia es derribada usando variación discriminada de tipos de registro, de preferencia genéricos.

Aunque el gran énfasis del proceso se enfoca a la producción del trazado de matrices, ligando requerimientos a los productos de análisis y diseño. Graham encuentra ésto distorsionado dado que el punto completo del AOO y el DOO es seguramente que : *"el trazado es incremental y, los trazos deben ser reconstruidos dinámicamente por los pasos ligados"*.

Texel está clasificado como un sistema de modelamiento temario con modelos de flujo de mensaje separado. Sus raíces en tiempo real son evidentes por la centralidad del uso de STD's, donde normalmente no es permitido definir operaciones priores en la obtención de un modelo de estado.

Como la mayoría de los métodos, no cubre todos los aspectos del modelo OMG y no existe nada acerca de la identificación del objeto, manipulación de componente. El grupo y los conceptos vistos son muy limitados excepto por un concepto similar a los sujetos de Coad. El modelo proceso está latente y no especifica iteración o técnicas prototipo de manipulación que sean visibles en la documentación.

El principal problema con éste método es que es difícil ver como podría ser útil en el diseñamiento por un lenguaje similar a Ada tal como Smalltalk, Eiffel o a un Lenguaje de 4ta. Generación Orientado a Objetos.

El segundo problema se aplica a los sistemas de negocios, donde uno sospecha que los métodos derivados de campos tales como telecomunicaciones, tiempo-real, militar y son menos situables que aquellos derivados del tradicional modelamiento de datos semánticos.

La posición de SOMA es que los modelos son finos cuando ayudan, pero son solamente una herramienta subsidiaria. La importante dificultad es conseguir mantener de los métodos y encontrar una forma de representar la dinámica global del sistema.

Texel está provisto con herramienta CASE para soportar en la forma de una sóla herramienta de usuario llamada Texel-SF e implantada en el sistema VSF meta-CASE.

A.7) Bon-Nerson

BON (La Mejor Notación de Objetos) es descrita en Nerson (1992) y es uno de los muy pocos métodos que toman la importancia de las reglas de negocios e invariancia de clases seriamente. Se deriva de la escuela de Eiffel, además la inclusión de aserciones es menos sorprendente. Fue desarrollado bajo el espíritu europeo encontrando en él influencia de Booch, Coad/Yourdon, Page-Jones y Constantine, Shlaer/Mellor, OMT, OOSO, CRC y trabaja en Z Orientado a Objetos, lo cual hace a BON verdaderamente híbrido. El énfasis se coloca en ideas como la continuidad de análisis y diseño, escalabilidad, reversatibilidad, trazabilidad, modelos estático y dinámico y manipulación de componente.

Las actividades en BON son :

(1) ***Definir la característica del sistema.*** Esto involucra la identificación de eventos externos.

(2) ***Identificar clases candidatas.*** Las clases en BON son definidas por atributos, operaciones, limitantes y relaciones con otras clases. No especifican la identificación de las técnicas de objetos que son incluídas.

(3) ***Clases de grupos dentro de agrupamientos.*** Los agrupamientos son subsistemas que no limpian la semántica pero son definitivamente no convenidas como objetos. Ellos son usados para agrupar la cohesión de clases.

(4) ***Define clases candidatas en términos de preguntas, comandos y limitaciones.*** El analista debe preguntar : "Qué datos pueden ser preguntados ?" Las preguntas vienen a ser atributos o "funciones" Eiffel. Los comandos son operaciones y limitaciones pero no son aserciones o clases invariantes. Las clases invariantes pueden ser pensadas de tales reglas y todas las limitaciones son como descripción del conocimiento mantenido por los objetos. Las asociaciones son registradas en forma textual.

(5) ***Define el comportamiento de cada clase en términos de eventos, comunicación de objetos y creación de objetos.*** Los eventos pueden ser externos o internos. Los eventos internos están relacionados usualmente con el tiempo y permiten el descubrimiento de protocolos de comunicación y aserciones relacionadas al control global del sistema. La notación para la descripción de clases incluye símbolos especiales para

las clases invariantes son escritas en una notación lógica formal. La creación de trazos de objeto muestra cuáles clases crean instancias de otras clases.

(6) ***Define características de clase, invariancia y relaciones contra-actuales.*** Las características de clase pueden corresponder a eventos internos. Las relaciones estáticas y dinámicas son mostradas en trazos separados usando una inusual y lejanamente idiosincrática notación.

(7) ***Refina las descripciones de clase.*** Aquí la reutilización de oportunidades es buscada.

(8) ***Desarrolla clasificación de estructuras.***

(9) ***Completa y revisa la arquitectura.***

La clase indexada es usada para auxiliar futuras recuperaciones de bibliotecas usando un encabezado estándar. BON tiene una gran intensidad (fuerza) no encontrada en muchos otros métodos tales como el énfasis en el manejo de componentes y reglas. Sin embargo, Graham la notación inapelable. BON es cercano en espíritu a SOMA, lo cual se describirá en un futuro apartado. Un ambiente CASE basado en un depósito PCTE-obediente y llamado EiffelCASE está desarrollado bajo Eiffel. Existen dos herramientas, una herramienta de dibujo y un componente manipulador del sistema. Algunas de las notaciones de BON se ilustran en la **Fig. A.44.**

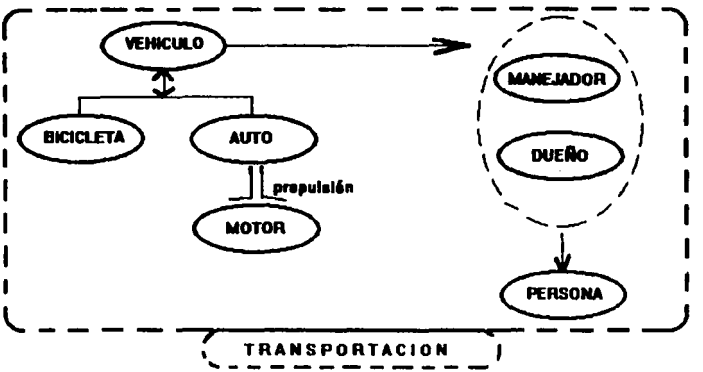


Fig. A. 44 Fragmentos de la Notación BON

A.8) Fusion - Coleman

El Método Fusion (Coleman y Hayes, 1991) es un método híbrido desarrollado por un equipo: Derek Coleman y los Laboratorios Hewlett-Packard en UK. HP creó una serie de métodos en el uso con HP y generalmente extintos en 1990, resultando en una serie de requerimientos para un Método de Análisis y Diseño Orientado a Objetos. El principal requerimiento identificado fue para un modelo de uso con una notación simple. Ningún método cubrió todas las necesidades de HP. La Técnica del Modelamiento de Objetos (TMO) le dió un proceso para Análisis pero sólamente jerarquía y diseño. Así pues, Fusion fue construído tomando prestadas las ideas de otros métodos.

Aunque permanece en desarrollo todavía. La principal influencia ha sido la notación de TMO, interactuando con el modelamiento de CRC, ideas de Booch'91 en la visibilidad y algunas ideas de Z y VDM de métodos formales. El uso de la idea case de la Objectory está siendo incorporado también. El modelo del proceso de Fusion se ilustra en la Fig. A.45.

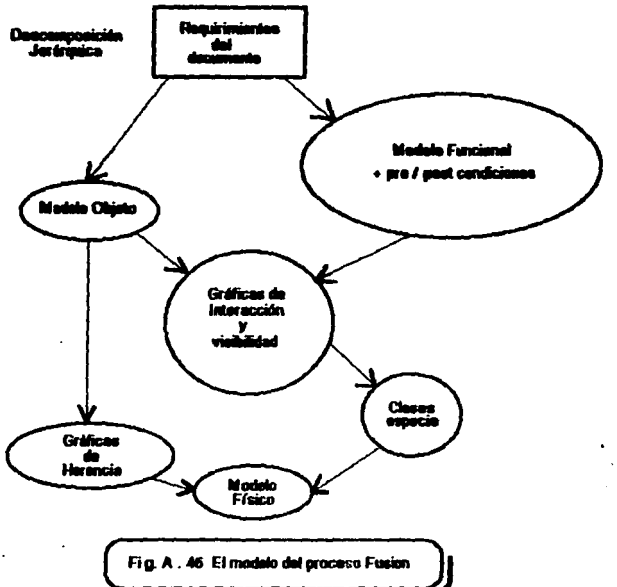


Fig. A . 46 El modelo del proceso Fusion

El modelo dinámico de TMO no fue encontrado por ser útil en la práctica y no es usado. Este es un alentador notable encontrado comparado a la sabiduría convencional, el cual según Ian Graham, lo hace sospechar que el excesivo énfasis en modelos de estado es el resultado de telecomunicaciones y respaldo en tiempo-real. Las "pre" y "post" condiciones piensa, son también usadas aunque, personalmente son difíciles de entender para los usuarios. Los métodos no son introducidos hasta el estado de diseño. La concurrencia no es soportada y así no existen condiciones de invariancia en el método todavía. Porque esas aserciones no están adheridas a las clases, por lo tanto ellas no pueden ser heredadas.

Un problema delicado para algunos profesionales del AOO es conocer cuándo detenerse haciendo el análisis.

Esto confirma que Fusion ayuda con éste problema. La visibilidad es introducida en el tiempo de diseño. Las interacciones definen cuáles objetos necesitan qué métodos -de nuevo en tiempo de diseño. La tarjeta CRC es usada por el modelo de interacción aunque los contratos no son usados.

A.9) Otros métodos

Análisis del Comportamiento del Objeto (OBA) es descrito por Rubin y Goldberg (1992). OBA es un método que inicia por sonsacamiento de guiones de entrevistas o documentos (los cuales son similares a los case's usados por Objectory) y desarrolla un modelo contexto. Después, los participantes y sus responsabilidades son identificadas e incian las distinciones. Las tarjetas CRC modificadas son usadas para registrar los detalles de los objetos y descubrir su clasificación, usan estructuras y otras asociaciones aunque la agregación no es enfatizada. Finalmente los modelos de estado de transición de Harel son desarrollados por cada objeto. OBA tiene más que decir acerca de los requerimientos de captura que la mayoría de los métodos pero no es particularmente notable en otras áreas.

Frame-Object Analysis (FOA) (Andleig y Gretzinger, 1992) es un método derivado del modelamiento conceptual en base de datos avanzada y en trabajos de inteligencia artificial, se enfatiza el uso de la semántica neta. Cubre totalmente una gran parte del ciclo de vida de los requerimientos a examinar y es inusual en el énfasis de reglas de negocios. En éste método una estructura (frame) es un componente mejor de una información del sistema identificada con una semántica neta consistente de objetos y asociaciones. Efectivamente las estructuras son subsistemas o etiquetas y son colecciones de objetos. Cada objeto tiene identidad, atributos y operaciones. Los diagramas de estructura son usadas para representar contención, clasificación y composición o transiciones de estado mediante diagramas de flujo. La fuerza de FOA radica en el énfasis en las reglas de negocios y en el uso de limitadas bases de datos y sus representaciones, aunque existe poca notación adicional que lo soporte. El método es más situable para la construcción de sistema de base de datos, donde C++ es empleado como el lenguaje de desarrollo y un avanzado sistema relacional como el manipulador de almacenamiento.

CGI Yourdon es el nombre a menudo otorgado a la moderna versión de los métodos originales de la familia de Diagramas de Flujo de Datos (DFD) -ahora mantenido por CGI. Ellos propusieron una extensión orientada a objetos a los métodos estructurados de Yourdon (Marden 1990) y ofrecen entrenamiento del mismo. Un método muy similar es descrito por Sully (1993). El método delega mucho énfasis al uso de notación DFD para describir el comportamiento interno de los objetos. El método deriva objetos de DFD's, ERD's o STD's donde cada modelo convencional ha

sido construído. Este es útil para las personas avanzadas en la filosofía de Yourdon poco común en la mayoría de los Métodos Orientados a Objetos que emplean el significado del DFD y, ningún otro método emplea diagrama de flujo para registrar la descripción de los objetos mismos. El aspecto externo de un objeto es representado con una notación Gradygram mostrando la identidad del objeto y las operaciones visibles. La herencia y las asociaciones son representadas con diagrama Chen-style y usan -dependencia de objeto- diagramas, que se dibujan donde es apropiado, con un estilo muy similar a los diagramas HOOD. Internamente, la estructura del objeto es representada como un tradicional DFD, con líneas punteadas mostrando el control del flujo. Donde hay transformaciones de control, las cuales son representadas empleando STD's, Tablas de decisión o Tablas de estado. En contraste con la mayoría de los bién conocidos métodos de AOO, éste ayuda a distinguir los objetos con estados complejos significantes pero Graham no se encuentra convencido de que el énfasis en DFD's sea de otra manera muy útil. Otros diagramas tienden a utilizar solamente DFDs para modelamiento del contexto y, aunque entonces él encontró esto necesario para expander el concepto de un diagrama de flujo para al incluír una notación de retorno de valores esperados.

Page-Jones y Weiss (1989) desarrolló el método Síntesis a Análisis y Diseño Orientado a Objetos. Más recientemente Page-Jones, Constantine y Weiss (1990) desarrollaron lo que ellos denominaron una Notación Uniforme de Objeto en el contexto de un método general de Análisis y Diseño Orientado a Objetos, con mucho en común con la convencional Ing. de Software en practica. La notación es similar a muchísimos otras notaciones de Análisis y Diseño Orientado a Objetos ya discutidas con clara influencia de Booch y Diseño Estructurado. Aún más recientemente Page-Jones (1992) ha generalizado la notación clásica de Constantine, en la que el buen diseño minimiza, acopla y maximiza la cohesión por ***"connascence" (literalmente, connascence significa "nacido juntos")*** y tres tipos diferentes de encapsulación : ***El Nivel de encapsulación 0 representa la idea de que una línea de código encapsula una cierta abstracción. El Nivel 1 es la encapsulación del procedimiento dentro de módulos y el Nivel 2 es la encapsulación de la Programación Orientada a Objetos.*** Dos elementos de un sistema son connascentes si ellos comparten la misma historia y futuro o, más exactamente, si cambia a uno, puede cambiar a otros. El principio connascence nos dice que la herencia debe ser restringida a características visibles o que debe tener dos jerarquías separadas para implantar herencia e interface. Esto podría ocasionar desacuerdo con el uso de los amigos de C++. Page-Jones clasificó muchísimos tipos de connascence como nombre, tipo, valor, posición, algoritmo, significado y polimorfismo. El connascence polimórfico es particularmente interesante para el DOO y está estrechamente relacionado a los problemas de la lógica no-monotónica. Por ejemplo, si *VUELA* es una operación de *PAJARO* entonces *VUELA* puede algunas veces fallar y otras ocurrir. Esto provoca problemas de mantenimiento pues el sistema debe ser cambiado. Se cree que las reglas pueden ser usadas para evadir éste problema. El principio de

connascence es una profunda contribución y debe ser adoptada por todo Diseñador Orientado a Objetos.

Henderson-Sellar dá su nombre al método fuera de línea en su libro (1992) y otras publicaciones. La notación es influenciada por la Notación Uniforme de Objeto (referida en el párrafo previo) pero es simple. El método es híbrido para Análisis Orientado a Objetos basado en un trabajo tempranamente publicado de muchos autores y enfatiza la necesidad de incorporar la existencia de métodos estructurados donde sea posible. Graham piensa que ésto es indicativo del tren a través de métodos híbridos. Como otros métodos hay escasa ayuda en la identificación de objetos. La notación es provista por todas las características estructurales importantes de un Sistema Orientado a Objetos. Las reglas no son soportadas. La más importante y reciente contribución es el *"surtidor"* del modelo del ciclo de vida. Mostrando la clasificación y otras asociaciones en el mismo diagrama, como otro método lo hace, es -correctamente- discutido. El método incluye guías de línea para implantación en varios Lenguajes Orientados a Objetos. Este método ha sido recientemente extendido y refinado como MOSES.

MOSES (Henderson-Sellers y Edwards, 1993) permanece como Metodología para la Ingeniería de Software Orientada a Objetos de Sistemas y está basado en una extensión de la Notación Uniforme de Objeto, aunque ésto confirma que otras notaciones son permitidas. Este proveé un modelo comprensible de ciclo de vida y enfatiza la continuación de la representación, las guías de línea manipuladas del proyecto y extensibilidad.

ADM3 (Firesmith, 1993) es una extensión de un método temprano Ada-O, ASTS, con un énfasis en desarrollo de sistemas en tiempo real. Consiste de notaciones para modelamiento de estados, modelamiento de control y modelamiento de tiempo y usa ideas de semántica de redes. Es un método temario reminiscente en algunos aspectos de Booch ' 91 y TMO.

Berard (1993) ofrece un modelo razonablemente comprensivo del Ciclo de Vida Orientado a Objetos, evocador en algunos referencias de MOSES. De nuevo una influencia semántica neta está presente en la forma del objeto de Berar y especificaciones de clase. Consiste de una *"precisa y concisa descripción"*, varias representaciones gráficas incluyendo semántica de redes y diagramas de transición de estado, listas de requerimientos y operaciones sufridas, constantes y excepciones. La notación es detallada y, se concibe mejor pensada en el papel de diseñador que en el de analista. No existe provisión para reglas.

Syntropy es un método propietario desarrollado por Steve Cook, John Daniels y sus colegas. El método confirma tener notación-independiente pero usa la popular notación TMO por

connascence es una profunda contribución y debe ser adoptada por todo Diseñador Orientado a Objetos.

Henderson-Seller dá su nombre al método fuera de línea en su libro (1992) y otras publicaciones. La notación es influenciada por la Notación Uniforme de Objeto (referida en el párrafo previo) pero es simple. El método es híbrido para Análisis Orientado a Objetos basado en un trabajo tempranamente publicado de muchos autores y enfatiza la necesidad de incorporar la existencia de métodos estructurados donde sea posible. Graham piensa que ésto es indicativo del tren a través de métodos híbridos. Como otros métodos hay escasa ayuda en la identificación de objetos. La notación es provista por todas las características estructurales importantes de un Sistema Orientado a Objetos. Las reglas no son soportadas. La más importante y reciente contribución es el *"surtidor"* del modelo del ciclo de vida. Mostrando la clasificación y otras asociaciones en el mismo diagrama, como otro método lo hace, es -correctamente- discutido. El método incluye guías de línea para implantación en varios Lenguajes Orientados a Objetos. Este método ha sido recientemente extendido y refinado como MOSES.

MOSES (Henderson-Sellers y Edwards, 1993) permanece como Metodología para la Ingeniería de Software Orientada a Objetos de Sistemas y está basado en una extensión de la Notación Uniforme de Objeto, aunque ésto confirma que otras notaciones son permitidas. Este proveé un modelo comprensible de ciclo de vida y enfatiza la continuación de la representación, las guías de línea manipuladas del proyecto y extensibilidad.

ADM3 (Firesmith, 1993) es una extensión de un método temprano Ada-O, ASTS, con un énfasis en desarrollo de sistemas en tiempo real. Consiste de notaciones para modelamiento de estados, modelamiento de control y modelamiento de tiempo y usa ideas de semántica de redes. Es un método temario reminiscente en algunos aspectos de Booch ' 91 y TMO.

Berard (1993) ofrece un modelo razonablemente comprensivo del Ciclo de Vida Orientado a Objetos, evocador en algunos referencias de MOSES. De nuevo una influencia semántica neta está presente en la forma del objeto de Berar y especificaciones de clase. Consiste de una *"precisa y concisa descripción"*, varias representaciones gráficas incluyendo semántica de redes y diagramas de transición de estado, listas de requerimientos y operaciones sufridas, constantes y excepciones. La notación es detallada y, se concibe mejor pensada en el papel de diseñador que en el de analista. No existe provisión para reglas.

Syntropy es un método propietario desarrollado por Steve Cook, John Daniels y sus colegas. El método confirma tener notación-independiente pero usa la popular notación TMO por

omisión. El énfasis es un método de comportamiento-orientado creando mucha de la noción de la especialización del comportamiento (clasificación). Algunos de los mecanismos de Syntropy son tomados de Booch ' 91 y podríamos clasificar al método seguramente como uno híbrido.

Otros métodos **:** ***COOSD*** un método desarrollado por Aksit quien se basó en la Universidad de Twente en Holanda. Se basa en la composición en el estilo del modelo funcional de datos. Las ideas noveles son de quienes participaron en el rol y tipos de comunicación de tipos abstractos. ***ORCA (Orientado a Objetos, Requerimientos, Captura y Análisis)*** usa Frameworks y NO diagramas. Lo que se puede apreciar de todos estos métodos es la gran tendencia en la construcción de métodos híbridos. *Otros* ***métodos son : ALEX-OBJ, MOOD, OSDL, OSMOSYS y SYS_P_O,*** siendo imposible cubrirlos a detalle. Existen cerca de 150 métodos completos orientados a objetos. Es claro ver que especialmente ninguno de ellos es completo y, muchos meramente representan opiniones de lo que consideran es importante en el Ciclo de Vida de un Sistema.

En el apartado inmediato posterior se presenta una breve síntesis del método : SOMA, quien tiene grandes posibilidades de empleo dentro del enfoque orientado a objetos, dada su importancia es que lo comentamos como un subtema adicional en el ámbito de los Métodos Orientados a Objetos.

A.10) SOMA : Método semánticamente rico para Análisis Orientado a Objetos

Entre los métodos descritos a lo largo del presente apéndice hay una variación considerable. Se observa un amplio rango de notaciones complejas y difíciles : TMO, Ptech y Shlaer/Mellor a unas simples: CRC en Coad/Yourdon, de un énfasis en procesos a un énfasis en representación y de dependencia de lenguajes a mareadas alturas de abstracción. Se encontraron algunos métodos híbridos. Ninguno de ellos es completo en el sentido de que todas las ideas de desarrollo del ciclo de vida del software son dirigidos o, que todo sistema concebible puede ser fácilmente descrito.

Este apartado presenta un método, el cual inicia su vida como una extensión del método y la notación avocada por Coad y Yourdon, pero no incorpora tantas ideas de otros Métodos de Análisis y Diseño Orientado a Objetos más que la conección con Coad/Yourdon, la cual es totalmete tenue. En principio otras notaciones base, tales como el TMO, podrían ser extendidas con SOMA, haciendo a SOMA más que un método, un filtro de métodos. Incorpora ideas y técnicas de los campos de Inteligencia Artificial y semántica del modelamiento de datos.

SOMA está intencionado para ser un método semánticamente rico para AOO. En forma básica, se intenta retener la simplicidad de la notación del método Coad/Yourdon pero añade

actividades extras y refina la semántica de sujetos -llamados etiquetas en SOMA-, modifica la notación dada dentro de la conformidad con el modelamiento convencional Chen-style "entidad-relación". Como Coad/Yourdon, la clasificación y la composición de estructuras son soportadas con asociaciones generales. Las condiciones de "pre", "post" e "invariancia" son soportadas. Ultimamente, añade reglas sistema-estilo-perfecto a objetos para aumentar la riqueza semántica de los modelos de análisis en todos los casos y ayuda al modelo el análisis de base de datos avanzada y el conocimiento base de sistemas. En tiempo de diseño esas reglas son convertidas a asociaciones lógicas. SOMA es único en la provisión de soporte para objetos confusos y herencia.

Las 7 actividades con SOMA son las siguientes :

- ***Identifica etiquetas.***
- ***Identifica objetos.***
- ***Identifica uso, clasificación y composición de estructuras.***
- ***Define datos semánticos y asociaciones.***
- ***Añade atributos a objetos.***
- ***Añade operaciones a objetos.***
- ***Añade las semánticas declarativas de los objetos.***

Primero, vamos a establecer algunos puntos notacionales. Los objetos en SOMA son desplegados en la forma de la **Fig. A.46 (a)**. Si el icono tiene esquinas cuadradas representa una instancia, si es redondo una clase. ***Un objeto tiene un identificador y tres listas : atributos, nombres, nombres de método y reglas de método.*** Cada uno de ellos tiene información adicional. Rodeando un icono clase con una caja como en la **Fig. A.46 (b)** dice que la clase no es una clase abstracta, en la misma manera en la que Coad emplea un contorno gris. En otras palabras, la clase puede tener instancias concretas que podrían no ser clases. Esto quiere decimos que algunas subclases que existen no forman parte de una lista exhaustiva de subclases posible.

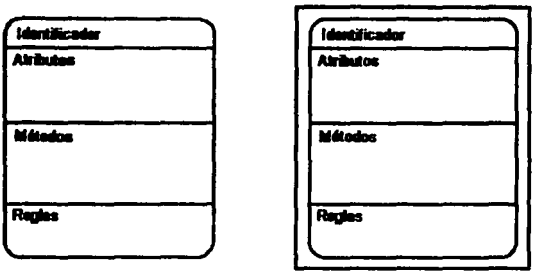


Fig. A . 46 (a) Una clase en SOMA; (b) Una clase con instancias en SOMA.

El tratamiento explícito de atributos viola el principio del ocultamiento de información. Sin embargo, la complejidad de las estructuras de datos presente en la mayoría de los sistemas comerciales comparado a una simple programación de abstracciones como ventanas o pilas, deja la necesidad de crear atributos explícitos y visibles. En éste método retomamos los atributos ventana. Para retener algunos Métodos Orientados a Objetos, convenimos cada atributo en esta ventana como taquigrafía para los métodos estándares. Por ejemplo, el atributo "*Nombre*" : es corto para los métodos "*Obtén_Nombre*" y "*Coloca_Nombre*". Las versiones especiales de esos métodos pueden ser incluídos en los métodos ventana en cuyo caso impone los métodos estándar. Cada método es convenido como conteniente de seguridad estándar. Las condiciones de ventilación deben ser especificadas para cada atributo.

Un icono objeto debe desplegar -al menos- : identidad, nombres atributo y nombres método. Las reglas no siempre existen, cada parte del icono es llamado una ventana, el mismo icono debe ser empleado para clases o instancias, pero la identidad de la ventana debe hacer la distinción clara para el nombre o para el anexamiento de la carta en corchetes. El uso de la convención de las esquinas redondeadas denotan clases, mientras unas esquinas puntiagudas señalan las instancias.

El nombre de la operación puede contener sólo con los detalles de la función del método, parámetros y tipo de información -"invariancia", "pre" y "post" condiciones- que se deben mantener cuando el método esté corriendo. En esta forma, una parte del control de la estructura es encapsulado en los métodos del objeto.

A.10.1) Capas

Capas en SOMA no son sólamente una forma conveniente para descomponer el problema del campo como en la mayoría de otros métodos, ellas : ***son auténticos objetos en su propia forma, con las semánticas de objetos.*** Las ***"capas"*** difieren de otros objetos en dos formas :

(a) existen en lo más alto (cima) de una composición de estructura;
(b) cada uno de sus métodos debe ser implantado por un método de algún objeto con esa estructura.

Esto significa que las capas pueden ser diseñadas en Top-down, como en Coad/Yourdon, o Bottom-up durante la actividad de las estructuras. Además, del requerimiento que todo método debe tener : una implantación por liga propiamente terminada, la cual asiste el chequeo completo durante el Diseño Top-down. Igualmente, el analista podría preguntarse durante un Análisis Bottom-up : ¿ Qué implanta esté método para las capas ?

Notacionalmente, las capas pueden ser mostradas en la notación de la Fig. A.46 pero una notación "Gradygram" como en la **Fig. A.47** es preferible ya que es más fácil de mostrar la implantación por ligas en ésta forma.

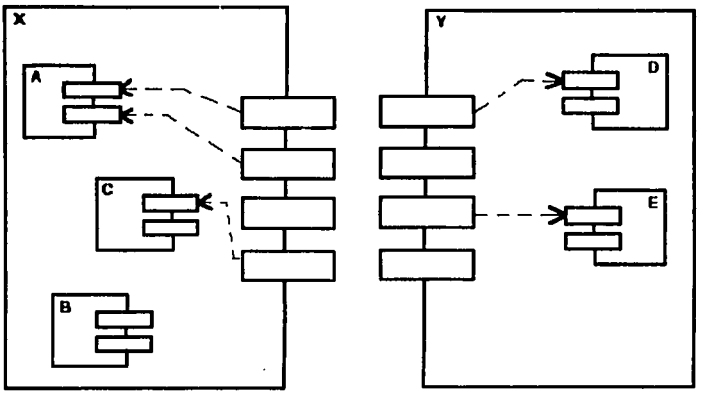


Fig. A. 47 Capas y ligas implantadas en SOMA

Por ejemplo : un sistema comercial podría tener las capas : Mercado, Ofertas, Cuentas y Producción. La capa ***"Cuentas"*** es un objeto representando las ***"Cuentas del departamento"*** como un todo y podría tener un método llamado ***"Produce_Ofertas"***. Las Cuentas están compuestas de 3 objetos o capas : ***"CompradorLibroMayor"***, ***"OfertasLibroMayor"*** y ***"NominaLibroMayor"***. El método ***"ProduceOfertas"*** es implantado por el método ***"ProduceInvolucra"*** del objeto ***"Ofertas"***, denotando ***"Ofertas.ProduceInvolucra"***.

En el contexto de un tradicional Análisis Top-down, podríamos comenzar con un diagrama estilo "Diagrama de Flujo de Datos", mostrando objetos externos y diagramas de flujo atravesado el límite del sistema.

El sistema es entonces el objeto de alto nivel (capa) con métodos definidos por ese flujo de datos. Cada método es únicamente implantado por ligas y, es más obvio qué debería suceder cuando un componente requiere los servicios del objeto fuera de la capa contenida. Considerando la Fig. A.47, podemos observar que hay dos casos.

Caso 1, ocurre cuando una clase B que no ha sido implantado por ligas, necesita los servicios de E. En este caso B debe ser removido de la composición de estructura desde que éste

no implanta ningún servicio para él. Esto puede invocar un menor rediseño, pero es una buena práctica en la mayoría de los casos.

Caso 2, es el caso donde una subclase "A" necesita los servicios de *otra* "Y" o "E". Aquí "A" está ejecutando un servicio para "X", en el cual se requiere enviar un mensaje a "E" que debe ser retomado a través de "X" recursivamente. Si después parece completamente permisible entrenar una correspondencia directa o colaboración, pero las semánticas del diseño deben ser examinadas cuidadosamente para ver si la anomalía puede ser removida, para los diagramas pueden obtener un pequeño desorden. En otras palabras, las capas pueden contener referencias de objeto en sus componentes.

La notación para capas necesita ser relajada algunas veces como capas, realmente tiene un aspecto dual. Pueden ser convenidas extemamente, como entidades coherentes o, internamente como colecciones de objetos que pueden recibir comunicaciones desde afuera del mundo en una base individual. Graham encontró que ambos puntos de vista se originan de manera natural durante el análisis. Así que cuando el punto de vista interno es adoptado, las flechas pueden penetrar la capa y buscar fuera objetos individuales. Desde su punto de vista, las capas deben, si es posible, convenir la misma forma como objetos envueltos, en otras palabras, como objetos de alto nivel por sí mismos. Cada objeto recibe y envía mensajes y, recibe y delega responsabilidades a objetos, los cuales encapsulan.

Por omisión, los métodos de objetos envueltos son la unión de todos éstos componentes de los métodos del objeto. Si es la responsabilidad del objeto envuelto por sí mismo resolver referencias polimórficas. Por ejemplo, si dos objetos sin la capa tienen el mismo método y el enviador no tiene especificado cuál aplicar, una regla es requerida en la capa objeto para determinar cómo resolver el conflicto.

A.10.2) Encontrando objetos

Los objetos son identificados como usuales usando los métodos de modelamiento de datos, sugerido por Coad y Yourdon o Shlaer y Mellor, la técnica textual de Análisis como la usada con Booch, HOOD y CRC, o en otra forma. SOMA añade las técnicas de identificación de objetos tomadas del conocimiento de elicitación y la práctica HCI. Específicamente incluye: entrevistas estructuradas y enfocadas, análisis de ideas y tareas de análisis. Una nueva técnica de refinamiento llamada ***"análisis de dictamenes"*** es añadida. Esas técnicas son muy útiles cuando un escritor de requerimientos declara que no existen especificaciones y que las entrevistas son necesarias.

Cuando no existen especificaciones otra técnica útil usada con SOMA está basada en el uso de cases Objectory. Un diagrama mostrando entidades externas y actores es construida y usa-guiones preparados para cada actor, las excepciones son definidas como subguiones y el texto resultante usa las tarjetas extendidas preparadas CRC. Esas pueden entonces ser usadas para conducir caminos con los usuarios y analistas.

En la práctica, Graham emplea algunas de las técnicas de CRC para identificar objetos y refinar su definición y organización cuando una especificación escrita de requerimientos está disponible. Esta es una buena forma de comenzar pero requiere las técnicas adicionales de SOMA cuando el sistema es complejo o la semántica es importante. Encontrando a las tarjetas CRC como un excelente vehículo para la enseñanza de los conceptos de Análisis Orientado a Objetos para principiantes.

Los modelos de estado de sistemas pueden ser usados para identificar y refinar objetos como en la entidad de cambio de estado, transición de estado o diagramas de historia de vida con correspondencia de efecto diagramas tomados de SSADM Versión 4. El uso de técnicas múltiples es recomendada para aumentar la confianza en sus productos, los cuales deben ser los mismos, o al menos equivalentes a cualquier técnica usada.

La notación SOMA es meramente la reposición final para toda la información descubierta y unifica el modelo de objeto estático con el modelo del comportamiento dinámico empleando aserciones y reglas.

A.10.3) Estructuras y datos semánticos

Las estructuras a ser identificadas son de 3 tipos principales : uso, clasificación y composición de estructuras. Las estructuras de uso muestran las rutas de mensajes permitidos a través del sistema, la clasificación de estructuras muestra herencia de características y la composición de estructuras muestra la formación de objetos agregados y capas.

El uso de estructuras registra la topología del paso de mensajes del sistema o equivalentemente, la visibilidad o relaciones cliente/servidor. Esta estructura generaliza la jerarquía de precedencia HOOD. El objetivo es minimizar la *"homología"* o complejidad de la estructura y ésta homología puede ser una medida útil de colectar. La homología es encontrada por el conteo del

número de fallas en la estructura, con respecto al doble encabezado de puntas como dos puntas de una sola cabeza. En la **Fig. A.48** la homología es la 2.

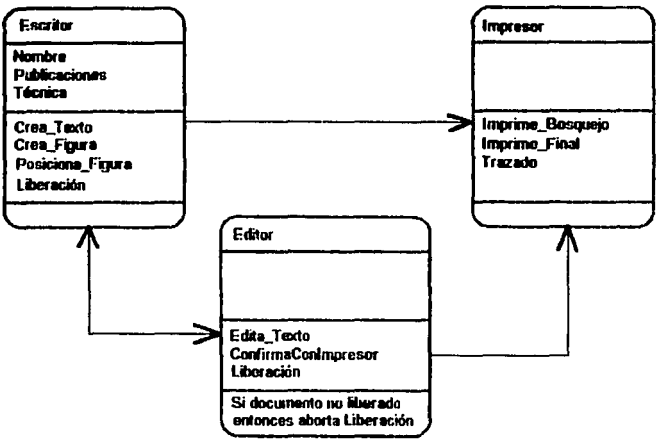


Fig. A. 48 Un fragmento del uso de una estructura. Los escritores pueden preguntar a un impresor que imprima un bosquejo y los editores pueden preguntarles a ellos acerca de imprimir una copia final. Los escritores pueden liberar el texto a un editor y los editores verificar los cambios con el autor, quien puede mejorarlo. No todos los atributos, métodos y reglas son mostradas.

Aquí es totalmente permisible construir modelos de comportamiento separado para objetos y sus interacciones. SOMA es neutral en la notación a ser utilizada, puesto que el modelo de comportamiento global es descartado después y el resultado es implantado en los objetos de SOMA. El estado local cambia los diagramas que pueden ser implantados en los objetos. Las notaciones posibles y los métodos del comportamiento de modelamiento incluyen a : Booch ' 91 , TMO, OSA, SSADM ELH's y, la notación del Modelo de Comportamiento de Objeto de Martin y Odell.

Esto significa la existencia de CASE y las herramientas de diagramación pueden ser utilizados para esta actividad. Los resultados están consolidados usando la notación SOMA.

Los mensajes pueden ser mostrados explícitamente con capas en los muchos estados tempranos de análisis. Los diagramas formales, sin embargo, son compilados bajo la regla de que una flecha uniendo dos iconos representa la posibilidad de que algún mensaje esté siendo transmitido, los mensajes actuales están siendo determinados de los métodos ventana del receptor. Esas flechas son equivalentes al uso de estructuras o relaciones cliente/servidor. Los mensajes polimórficos ilegales deben ser explícitamente declarados en la descripción del método o como un comentario en las reglas ventana. Por ejemplo, si es necesario permitir utilizar objetos para enviar mensajes a objetos de la clase "empleado", pero no prohíbe al usuario descubrir una edad de empleado. Se propone que las herramientas que soporten SOMA deban incluír facilidades para simulación dinámica de eventos/mensajes.

A.10.3.1) Clasificación de Estructuras

Las notaciones de algunos métodos son algunas veces ambiguas. Un ejemplo es la notación de Coad/Yourdon para la clasificación y composición dada en las Fig's. A.5 y A.6. Una alternativa sugerida para ambos se muestra en las **Fig.'s A.49** y **A.50.**

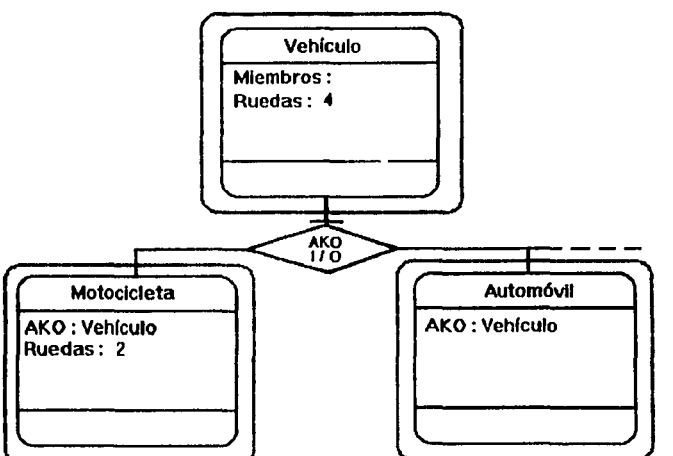
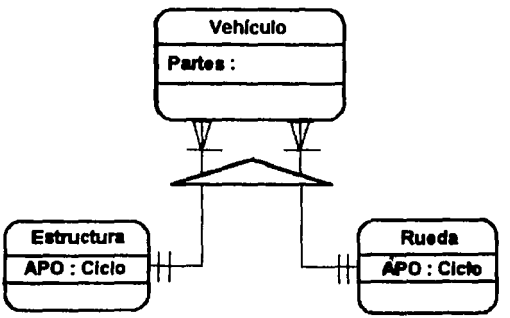


Fig. A. 40 (a) Una mejorada notación - clasificación. Nota los atributos especiales "Miembros:" y "AKO: (Un tipo de)";
(b) Diagrama de árbol informal

La notación dentro del diamante es interpretada exactamente como. Eso es, E/O permanece para "exclusivo/opcional" e I/O significa "inclusive/opcional". El témino "exclusivo" indica que cada intersección de la subclase con otras subclases está vacío y, que "inclusivo" indica que las subclases pueden superponerse. La "opcional" indica que la lista no es exhaustiva : podría haber más que todavía no han sido identificadas. Un "M", para "mandatario" indica que un miembro de la superclase debe estar en al menos una de las subclases, ésto es, las subclases son una lista exhaustiva o partición de clases.



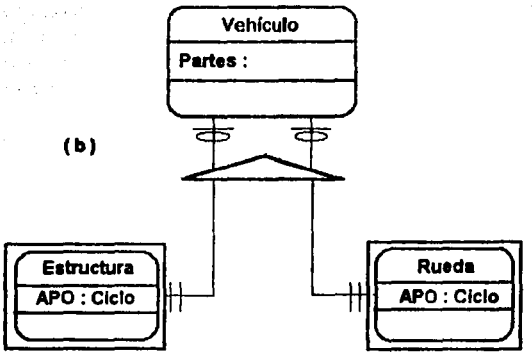


Fig. A. 60 Una notación compuesta mejorada. Atributo especial APO : (Una parte de). (a) Nivel clase (b) Nivel instancia

La clasficación tampoco puede ser mostrada en esta manera, en TMO o en la notación de Coad/Yourdon o más simplemente, como árboles simples como en la Fig. A.53. De conformidad con Coad/Yourdon, una "X" contra un atributo o método (en la izquierda) registra el hecho de que puede no ser heredado de una superclase. Las ventanas de atributo de clases que participan en la clasificación deben contener los atributos especiales APO y Miembros: De preferencia deben tener el atributo especial Isa: si es deseado, una distinción puede ser hecha entre los atributos especiales "Miembros" y "Subclases" aunque el contexto siempre lo hace claro. Simplemente, si el atributo "Miembro" es convenido como parte de la especificación, este compromiso reusa, porque todo el tiempo un nuevo miembro es añadido y debe ser cambiado. Sin embargo, si nosotros imaginamos el soporte automatizado para éste método, el problema se evapora desde que es un procedimiento enteramente mecánico para actualizar los Miembros: atributo automáticamente todo el tiempo un miembro es adicionado o borrado de la clase. Estrictamente, los miembros no son parte de la especificación pero meramente un auxilio navegacional conveniente.

Este podría ser argumentado por aquellos entregados a un método basado en objetos para análisis, en los que la herencia es una idea de implantación la cual no debe entrar dentro del proceso de análisis y, que los usuarios pueden ser confundidos por él y distraídos de los metas del sistema. Este es un argumento genuino pero dañado de dos debilidades : *la primera*, confunde la

elicitación de requerimientos en el proceso conceptual de análisis con la búsqueda de abstraer los conceptos de la aplicación independientemente de las percepciones de los usuarios. Posteriormente, las características estructurales del campo son revelados, incluyendo nociones naturales de especialización y generalización. *La segunda*, la herencia y otras estructuras son una parte importante del campo semántico: la forma en la que los objetos son clasificados define el campo y, a menudo el propósito, de la aplicación. Por ejemplo, los objetos de sociología e historia son los mismos, la gente y las organizaciones, pero en la historia no podríamos clasificar por grupo socio-económico, por clase social o más aún, notacionalmente. **Las estructuras representan el propósito de los análisis.**

La clasificación, herencia o estructuras de generalización/especialización, las estructuras permiten al analista registrar las semánticas de clasificación. La herencia puede ser solamente usada en este sentido en SOMA y algún tipo de polimorfismo ad hoc o herencia meramente para los propósitos de código compartiendo éstas amonestaciones en la etapa de análisis. La notación para esas estructuras sigue el estilo Chen de modelación ER. Así pues, un conector en forma de diamante señalado AKO (Un Tipo De) indica que los objetos adjuntos participan en una clasificación de estructura. Una barra en un conector indica una liga a una superclase y está ausente una liga a una subclase. Las barras pueden ser nombradas si se desea. En el caso de herencia múltiple habrá 2 ó más barras. Normalmente, la herencia múltiple es conjuntiva; esto es, la subclase es un tipo de todas éstas superclases. En casos raros la herencia disjuntiva es útil. En este caso, donde la subclase es un tipo de una de las superclases, una barra doble es empleada. Esta también puede ser nombrada y anotada. Implícitamente todas las relaciones AKO son asociaciones en la forma de "muchos-a-uno" de un tipo especial de clasificación denotada. Sin embargo, incluyendo esta multiplicidad de información redundante no solamente no añade nada a nuestro entendimiento del problema sino también resulta usual y altamante confundible.

Ciertamente el estilo Chen para señalamiento con herencia múltiple trabaja mejor que el estilo Bachman adoptado, tales CRC, ORACLE*CASE y Ptech, donde las subclases son mostradas por inclusión. Esto es aún cierto para herencia simple donde la jerarquía es profunda y tenemos cajas, sin cajas, etc. Martin y Odell (1992) dejan de cumplir con ésto es para permitir a los diagramas helecho (fern) producidos por Kappa, ser empleados también como la inclusión de cajas.

Otras notaciones, tales como Coad/Yourdon, interpretan la composición de ligas como conexiones "nivel-instancia". Se encuentra útil permitir una interpretación nivel-clase tan bién. En la Fig. A.50 un mecanismo particular (una instancia) puede pertenecer a solamente un ciclo a la vez. Sin embargo, el tipo de mecanismo (esta clase) puede ser una parte de muchísimos modelos o tipos de biciclo. En el nivel instancia, para todo mecanismo hay en la mayoría un biciclo que es "una

parte de". Sin embargo, en el nivel clase, para todo tipo de mecanismo puede haber muchos tipos de bicycle que incorporar.

Las ventanas de atributo de clases o instancias que participan en la composición de estructuras deben contener los atributos especiales : "Partes y APO". Exactamente la misma aplicación remarca al APO: atributo de partes como aplicar a los Miembros: atributo de superclases. Son ayudas navegacionales las cuales deben ser actualizadas automáticamente donde las "Partes:atributo" es añadido o sustraído de, de cualquier forma todo el tiempo de un objeto compuesto fue restructurada la especificación de esas partes, lo cual podría haber sido alterado. Las notaciones E/O, E/M, I/O y I/M pueden ser usadas pero las partes no son usualmente superpuestos.

SOMA ha sido criticado por el uso del mismo símbolo de diamante para la clasificación y composición, el cual podría ser confundido pero se encuentra la distinción suficientemente clara en la práctica.

A.10.3.2) Asociaciones

Otros tipos de asociaciones y su semántica/estática deben ser catalogados. Los datos semánticos en Coad/Yourdon son solamente anotados en el nivel instancia. En SOMA ha diso encontrado útil permitir las conexiones nivel-clase de éste tipo como con la composición de estructuras. Esto sirve para enriquecer la semántica. Las conexiones pueden ser a través de **asociaciones;** esto es, otras relaciones estructurales que usan, clasificación o composición. En algunos casos esas asociaciones tienen propiedades y deben ser expandidas dentro de objetos en su propio derecho. En otras la asociación puede ser tan importante para la aplicación (por ejemplo, el trono o reinado en antropología) que unas 4 ó 5 estructuras pueden ser adicionadas al modelo. Las ligas de datos semánticos indican la multiplicidad y modalidad que pueden ser empleados en ligas de composición de estructuras.

Los elementos de la rica notación de TMO son usadas para la compleja grabación de composición de estructuras y asociaciones. En particular la notación para la composición recursiva mostrada en la Fig. A.16 es recomendada.

En este punto es posible levantar la idea, de cualquier forma un concepto particular será incluído como un tipo de objeto o meramente como un atributo de otro objeto. Las consideraciones del modelamiento usual de datos se aplican y, generalmente los objetos con 1 ó muy pocos atributos y/o métodos son mejor tratados como atributos de otros objetos. Se remarca que en la

práctica, el análisis de datos, la posesión de objeto y la validación deben ser especificadas tan claramente como sea posible.

La normalización dentro de la tercera forma normal proveé pautas valuables en la creación de la decisión acerca de cualquier intento de algo como un objeto o como un atributo. 3NF es la forma normal la cual elimina la mayoría de las anomalías actuales. Sin embargo, la normalización definitivamente no forma parte del modelo objeto y debe ser usado solamente como una guía.

La regla es: el modelo del mundo real y piensa acerca de las redundancias y anomalías potenciales.

Es notable que cuando una relación es representada explícitamente con sus propios atributos y/o métodos, el icono normal del objeto 4-ventana es usado de preferencia que la notación de diamante de los modelos Entidad-Relación. Siguiendo con Desfray, no son normalmente permitidas las relaciones que tengan operaciones y que a través de ellas puedan, raramente, tener reglas.

A.10.4) Atributos y operaciones

Los atributos son descubiertos usando técnicas de modelamiento de datos estándar. Dos especiales listas-valuadas de atributos son usadas : "AKO y Partes". La "AKO: atributo" es una lista-valuada y contiene una lista de objetos que compone el objeto. Donde el soporte automatizado está disponible a los atributos duales, "Miembros : y APO:", pueden ser empleados para ayudar a la navegación a través del modelo, pero eso debe ser actualizado automáticamente, de otra forma el reuso es comprometido.

En SOMA todo atributo tiene un tipo, el cual debe ser un usuario-definido valido o la clase primitiva en el sistema. Los típicos primitivos incluirán reales, enteros, cadenas, listas, datos y etc. Los atributos almacenan asociaciones. Por ejemplo, si tenemos la relación semántica la cual dice que los empleados deben trabajar para exactamente un departamento mientras los departamentos pueden tener 0 ó más empleados, entonces podríamos registrar

Worksin : (DEPT, 1 , 1)

como un atributo de EMPLEADO y

Employs : (EMPLEADO, 0 , n)

como un atributo de DEPTO o produce el diagrama equivalente "el alimento de los cuervos". En efecto, convenimos los atributos como divididos dentro de 2 clases : atributos puros y atributos representando asociaciones.

La extensión según la cual cada asociación compromete el reuso ha sido ampliamente debatido, pero los requerimientos de los sistemas comerciales desarrollados parecen tener traer el conceso redondo a su aceptación y, las asociaciones son aún parte del Modelo de Objeto Abstracto OMG. Esas ligaduras son heredadas a lo largo de la clasificación de ligas en la siguiente manera por omisión.

Superclase	Subclase
1 - a - muchos	0 - a - muchos
0 - a - muchos	0 - a - muchos
1 - a - 1	0 - a - 1
0 - a - 1	0 - a - 1

Esas asunciones por omisión pueden ser sobrepuestas. En el caso de sobreposición los diagramas deben mostrar la liga compuesta remplazada como en la Fig. A.51.

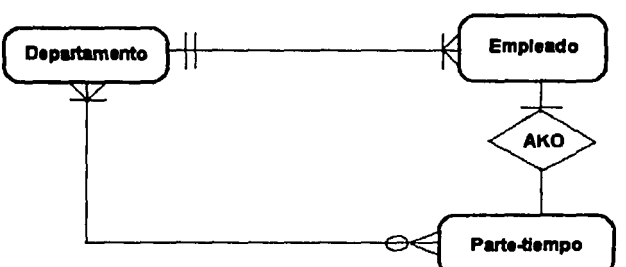


Fig. A. 51 Imponiendo la semántica por omisión dada una liga compuesta de clasificación

Los atributos almacenán las omisiones usuales, seguridad, propiedades y códigos de acceso y el rango de limitación los cuales no están definidos por su tipo.

Los métodos u operaciones. Aquí es donde SOMA toma prestado casi todo, desde otros métodos orientados a objetos y métodos convencionales. Los STD's pueden ser usados como en TMO y en OSA. Aún SSADM ELH's pueden ser empleados y los DFD's pueden ser usados como sigue : toma un almacén de datos y encuentra los objetos a encapsular. Repete para cada almacén de datos y refina los resultados. El método HOIST (una extensión de HOOD) y el método CGI Yourdon a menudo refine versiones de ésta técnica. SOMA es deliberadamente neutral en la técnica a ser usada porque en ésta forma los profesionales pueden construír su experiencia previa. Como con los métodos ternarios a analizar la pregunta sería dónde referir STD's en el sistema o en entidades individuales. TMO envía antiguos aspectos y OSA o posteriores. La filosofía de SOMA, comparte con Coad/Yourdon y OSA o una extensión menor, es que las técnicas estructuradas convencionales son usadas internamente en un básico "objeto-a-objeto" y no para la descripción global del sistema. Lo último es la función del uso de diagramas en el AOO. Sin embargo, a través del chequeo de los productos de un Diagrama de Flujo de Datos global con los métodos identificados sin SOMA pueden actuar como un chequeo adicional por completo y no existe discusión, ni dudas.

Cuando el comportamiento del objeto es muy complejo el estado de diagramas de transición ha provisto una herramienta efectiva para la identificación de operaciones. Todas las técnicas para la especificación de sistemas de computadora se encuentran a la mano para el especificador de cada método : diagramas de flujo de datos, Petri neto para métodos concurrentes, diagramas de transición de estado, árboles de decisión y aún buena descomposición funcional del viejo Top-down.

Las operaciones tienen una lista de parámetros y regresa un tipo. Las condiciones de pre-, post e invariancia pueden ser atacadas para cada operación y son heredadas.

Es esencial hacer una distinción entre operaciones que apliquen instancias, tales como el calculo de la edad de las personas desde su fecha de nacimiento y, operaciones las cuales se apliquen a clases enteras, tales como el cálculo de la edad promedio de todos los empleados. *Si la distinción de **arises** entonces esos tipos de método son llamados **instancia de métodos y clases de métodos** , respectivamente. Las clases de métodos son precedidas con un signo de "$", siguiendo la convención de la Técnica del Modelamiento de Objetos.*

A.10.5) Reglas

En la ejecución del AOO y la construcción de un modelo, un gran número de lecciones pueden ser aprendidas, desde los sistemas AI que se construyen usando semántica neta y todo el modelamiento de datos semánticos. Las especificaciones que exhiben reusabilidad y extensibilidad

son todos muy buenos, pero no necesariamente contienen el significado intencionado por el analista o el usuario. Al reusar una especificación de un objeto deberiamos ser capaces de leer qué es (estructura de datos), qué hacen (operaciones), por qué lo hacen
y cómo se relacionan con otros objetos. Enfocándonos al contenido semántico, está parcialmente contenido en las estructuras de clasificación, composición, uso, y las reglas, las cuales describen su comportamiento.

La notación Coad/Yourdon para la composición o partes jerárquicas por ejemplo, como se ha remarcado, no toma cuenta de la investigación de estrategias. Esta no es meramente una idea de implantación pero si una pregunta de la aplicación de la semántica. La herencia jerárquica usa el conocimiento implicito en ellos en una forma análoga a investigar y, busca lo que puede ocurrir en un regreso o adelanto de la forma cambiante, justo como un experto en sistemas.

El hecho es que toda la semántica compromete el reuso. En la especificación de un sistema ambos aspectos son igualmente importantes y el trueque debe ser bién entendido y manejado con cuidado, dependiendo de las metas de los analistas y sus clientes.

Definiendo los datos semánticos es mejor mantener separados la definición de ambos objetos y atributos. Como ya se ha señalado, los Sistemas de Bases de Datos Relacionales a menudo requieren mucho de ésta información, incorporándola en los programas de bases de datos orientadas a objetos, o aún bases de datos relacionales o extendidas o semi-orientadas a objetos, es que esas semánticas pueden ser capturadas en el ambiente de desarrollo destino en buena medida, como en el análisis en forma completamente explícita. La explicitud delibera flexibilidad cuando los datos semánticos o las reglas de negocios cambian la aplicación. Si vamos a construir ejecutables, especifcaciones reversibles, es crucial que la información no se pierda en el código. Esto incluye todos los tipos de información semántica. Los datos semánticos pueden ser capturados en relaciones y almacenados explícitamente en BDOO o deductivas. La semántica funcional puede ser capturada explicitamente como reglas. La idea de reglas y su representación se tratarán próximamente.

En la opinión de Ian Graham, el defecto más serio de muchos métodos de AOO es su escasez de soporte para la semántica funcional, la descripción del control global o reglas de negocios. Ahora entraremos en lo que en lo sucesivo se denominará **semántica declarativa**, porque el procedimiento semántico es encodigado como métodos.

Es necesariamente en ésta etapa de un análisis, que se registran reglas de negocios, el mantenimiento excepcional y el comportamiento del paso de parámetros del objeto. El método

Coad/Yourdon tiene el defecto de una notación para la excepción de mantenimiento. Esto no es siempre importante pero, para esas aplicaciones donde está, una notación similar a OOSD puede ser usada en anotaciones para cada objeto sin hacer violencia a la simplicidad de la notación que recomendamos.

Los Métodos Orientados a Objetos, tales como Coad/Yourdon pueden obviamente ser extendidos con herencia múltiple aunque esto no está incluído en su descripción, excepto en el nivel de notación. La extensión puede incluir provisión para la anotación del mantenimiento de conflicto cuando el mismo atributo o método es heredado diferentemente desde 2 objetos padres. Debe casi permitir semánticas diferentes para subclases las cuales particionarán sus super-clases dentro de un exhaustivo conjunto de subclases y de quienes son incompletos, en el sentido que presuntamente no especifican subclases pueden ser adicionadas.

La sugerencia obvia cubre esa semántica declarativa, donde se adicionan reglas a objetos SOMA los cuales no pueden solamente desambiguar herencia múltiple pero casi definen reglas prioritarias por omisión y demonios. Un demonio es un método que se levanta cuando se necesita : cuando un valor cambia, o es adicionado o borrado. Esto es, esas reglas pueden determinar cómo resolver el conflicto que se origina cuando un atributo hereda 2 valores diferentes o, además especifica donde el valor por omisión debe ser aplicado antes o después de un demonio. Pueden casi especificar la relativa prioridad de herencia y demonios. Esas reglas por sí mismas pueden tener omisiones. Las reglas de negocios especifican el segundo-orden de información, tales como dependencias entre atributos, por ejemplo una dependencia entre la edad de un empleado y su entretenimiento vacacional. Las condiciones : "global, pre y post" que se aplican a todos los métodos pueden necesitar ser especificadas como reglas. Una regla de negocios típica en una aplicación personal puede incluir "cambio del entretenimiento vacacional a 6 semanas cuandos los servicios exceden 5 años" como una regla en las ventana de regla del objeto Empleado. Con esta extensión la notación puede arreglárselas con problemas de análisis donde una base de datos relacional o deductiva con características orientadas a objetos es concebida como el ambiente destino.

Esas reglas son encapsuladas dentro de objetos, de preferencia siendo globalmente declarados como es el caso con todas las implantaciones actuales en los LOO's. Ellos pueden ser heredados y sobrepuestos. El beneficio de ello, radica en el impacto de la estrategia de control. Además, el analista puede inspeccionar el impacto de la estructura de control en todo objeto - usando, de preferencia, su curiosidad - y no tiene que anotar diagramas complicados para describir los efectos locales del control global. Si un aspecto global es deseado entonces los Diagramas de Estructuras de Datos o Diagrmas de Evento Ptech pueden opcionalmente ser usados. Las reglas

genuinamente globales son contenidas en un objeto de alto nivel, llamado "objeto", y serán heredadas por todos los objetos que no sean eliminados. Sólo un diagrama de transición de estado o cambio de estado puede ser usado para describir la semántica procedimental de los métodos, así los árboles de decisión pueden ser útiles en la colocación de descripciones complejas de reglas.

Uno debe asegurarse de la necesidad de decidir, cuando las reglas pertenecen a métodos individuales que no pueden ser expresados en un lenguaje basado en reglas. Sin embargo, la distinción hecha aquí no es entre la forma de las expresiones y el contenido de las reglas. Las reglas que relatan muchísimos métodos no pertenecen a esos métodos y, las reglas las cuales definen la dependencia entre los atributos también se refieren al objeto al que pertenecen sin uno de éstos métodos. El más importante tipo de reglas del "objeto completo" son reglas de control las cuales describen el comportamiento del objeto, dado que éste participa en las estructuras a las que pertenece, las reglas a controlar el mantenimiento de omisiones, herencia múltiple, excepciones y relaciones generales con otros objetos.

La actividad de las semánticas declarativas (reglas) está en el más novel aspecto de SOMA. Aumenta el modelo usual de objeto añadiendo un conjunto de reglas-conjunto para cada objeto. Así pues, mientras un objeto normalmente consiste de identificador, atributos y métodos, un objeto de SOMA consiste de identificador, atributos, métodos y reglas. Las reglas pueden estar subdivididas dentro de tipos. Lo cual presenta un número de consecuencias interesantes, la más remarcable de las cuales es que esos objetos que son entidades locales pueden encapsular las reglas para el comportamiento global del sistema.

Otra consecuencia es que el conjunto-de-reglas p.ieden ser convenidas como objetos para desarrolladores de sistemas expertos. En ese caso un método natural es la regla-conjunto de objetos teniendo un pequeño número de atributos y métodos.

Como con los atributos y métodos, la interface del objeto solamente despliega el nombre de una regla-conjunto o regla. Este uso especializado de reglas puede ser de muy diferentes tipos. De preferencia podríamos tener : reglas de negocios y, reglas de control. Las reglas de negocios típicamente relatan dos ó más atributos y sentencias que relacionan atributos a métodos.

Por ejemplo:

Regla de negocios:

IF Servicio_longitud > 5 THEN

vacaciones = 25

A través de la sentencia :

Cuando Salario + Incremento_Salario > 20,000

corre por medio de la CIA AwardCoCia.

Por ejemplo, la clase "InsuranceSalesman" podría contener las reglas para dar la mejor sugerencia a un cliente en la forma :

If cliente es retirado y Client.RiskAverse : es falso
then MejorProducto : es "Anulado"
If cliente es jove y Client.RiskAverse : es falso then
MejorProducto : es "Surtido"
If Client.RiskAverse : es verdadero then
MejorProducto : es "Garantizado"
If Client.Children : > 0 then
Client.RiskAverse es verdadero

Las reglas son questionadas cuando un valor para "MejorProducto" es necesitado. Todas las reglas no comprometen la encapsulación del cliente colocando el valor de Risk.Averse en ese objeto. El "Salesman" se asume meramente como en el caso de pérdida de datos.

La integridad de las reglas es convenida como parte de los datos semánticos y es almacenada como parte del tipo de información.

Vamos a examinar un simple pero interesante ejemplo en el que se involucra la herencia múltiple y los valores por omisión de los atributos. Supongamos que deseamos construir un juego de aventuras basado en texto. Vamos a agregar la llamada "Quest". En un aventura del tipo representado por "Colossal Cave" usualmente tenemos el siguiente orden de entidades: localidades, actores y cosas. El juego tiene localidades que pueden contener objetos movibles listados en el contenido de los atributos o tenemos misterios propios descritos por los métodos. Los atributos de localidades describen varias entradas y salidas. Existen jugadores, y la computadora pretende contener una guía antropomórfica quien, si tu preguntas como comer el aperitivo, dice, lo reemplazaremos con algo como "Acabo de perder el apetito". Esos actores pueden convencionalmente ser considerados como personas y, tenemos nuestra primer estructura de herencia como se muestra en la Fig. A.52. Nótese que la guía no está disponible para llevar los objetos (indicados por la X) pero hereda la habilidad de mover desde un lugar a otro. Los jugadores pueden utilizar comandos tales como "come aperitivo", "toma antorcha", o "ve al Norte".

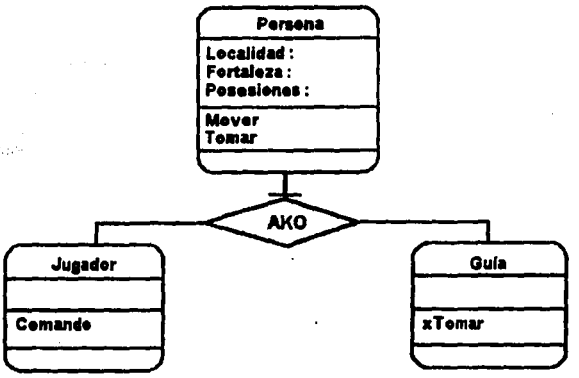


Fig. A. 62 El actor de "la búsqueda"

Cuando examinamos la "Cosa" para la estructura, ésta puede ser clasificada dentro del árbol, el cual tiene valores en puntos, armas y utiliza cada una como comida, balas y etc. Una estructura provisional se muestra en la **Fig. A.53**, donde puede verse que la herencia múltiple ha introducido: la "espada enjoyada" como arma y tesoro. El tesoro tiene un valor positivo en puntos, y las armas no tienen ninguno. Así que ¿ cuál es el valor que la espada enjoyada podría heredar de la arma y el tesoro ?.

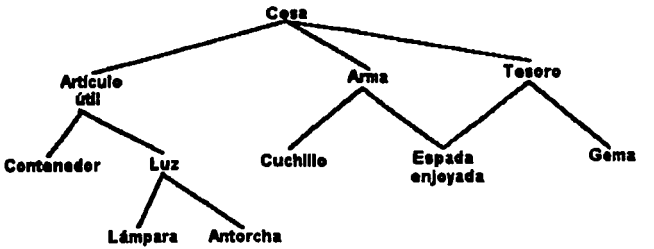


Fig. A . 53 La estructura de las cosas

Existen muchísimas estrategias para el mantenimiento de los conflictos de la herencia de éste tipo. El sistema podría reportar el conflicto del usuario y preguntar por un valor y, ésta es la estrategia más común.

Desafortunadamente, la corrida en un ambiente por lotes es algo más que una simple molestia, es costoso. Todos los otros métodos requieren que al último cuando las longitudes de la ruta son iguales.

La Fig. A.54 muestra un ejemplo de una regla de conflicto rota.

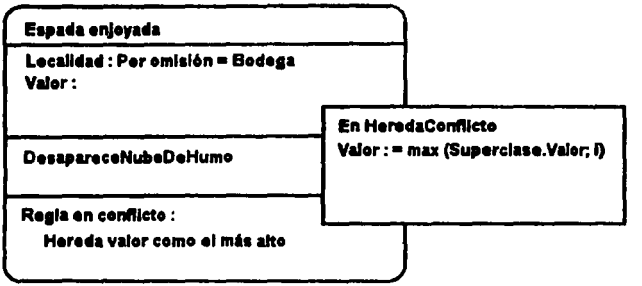


Fig. A . 54 Notación de objeto SOMA incluyendo un regla de conflicto abierta

Considerando un Diseño Formal : las "Condiciones"contra las "Reglas"

Todo lo que pueda ser expresado como las reglas lo hacen, en principio, será expresado como una combinación de "pre-", "post-" e "invariancia". Sin embargo, hacer ésto puede resultar complejo e ilegible. La ventaja de las reglas es la claridad y consistencia. El analista debe ser capaz de identificar cuál es mejor en un caso individual. Como un ejemplo, la regla acerca de *"Vacaciones"* podría ser expresada como una postcondición del atributo *"servicio"*, así pues, cualquier valor cambia en alguna forma al llamar a un método, el cual actualiza *"Vacaciones:"*. Esta es realmente una post-condición en el método para actualización de la extensión del servicio, no en el atributo. Desde que éste método estándar no está usualmente exhibido, se añade la dificultad del mantenimiento de éstas condiciones y las reglas del método parecen ser mucho más claras en el balance. En la implantación ésta decisión puede ser fácilmente invertida. Donde la corrección formal es un problema, las reglas deben ser evadidas debido a la posibilidad de efectos secundarios y a la escasez de una teoría formal de prueba. Graham considera difícil que ésto ocurran en sistemas comerciales, donde las reglas son usualmente la mejor forma de limpiar la expresión para propósitos de análisis.

A menos que la implantación sea un Lenguaje-basado-en-reglas, los diseñadores de SOMA reemplazarán las reglas de los analistas de SOMA con aserciones. Esto es más preciso pero menos comunicable a los usuarios, así que es importante trazar la conversión o aún automatizarla.

BIBLIOGRAFIA

* **Object Oriented Desing with Applications,**
Grady Booch,
Benjamin Cummings Publishing Company, 1991, 580p.

* **Designing Object-Oriented Software,**
Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener,
Prentice Hall, Englewood Cliffs, New Jersey, 341p.

* **Análisis y Diseño de Sistemas de Información,**
James A. Sean,
McGraw Hill, 1991, 941p.

* **Methods Oriented Object,**
Ian Graham,
Addison-Wesley Publishing Company, 1993. 473p.

* **Introducción a la Programación Orientada a Objetos,**
Timothy Budd,
Addsion-Wesley Iberoamericana, 1994. 490p.

* **Fundamentos de Programación,**
Ernesto Peñaloza Romero,
U.N.A.M.,1994, 352p.

* **Object-Oriented Programming in C ++,**
Nabajyoti Barkakati,
SAMS, 1991, 666p.

* **Manual de Referencia de C ++ con Anotaciones,**
Bjarne Stroustrup y Margaret A. Ellis,
Addison-Wesley / Diaz de Santos, 1994, 541p.

* **Aplique C ++,**
Bruce Eckel,
McGraw Hill, 1991, 521p.

* **Programación Orientada a Objetos,**
Miguel Katrib,
Cuba, 1995.