TECNOLOGÍA DE ORIENTACIÓN A OBJETOS

Compilado y elaborado por Luciano RIPANI

SEMINARIO DE SISTEMAS

UTN - FRR

1994

TECNOLOGÍA DE ORIENTACIÓN A OBJETOS

1. INTRODUCCIÓN 1

1. I MOTIVACIÓN 1

1. II EVOLUCIÓN DE LOS MÉTODOS DE ANÁLISIS 3

DESCOMPOSICIÓN FUNCIONAL 3
DESCOMPOSICIÓN DE FLUJOS DE DATOS 4
INFORMATION MODELING 5
ORIENTACIÓN A OBJETOS 5

2. EL MODELO DE OBJETOS 8

2. I OBJETO 8
2. II PROGRAMA 9
2. III MENSAJE 9
2. IV ABSTRACCIÓN 9
2. V ENCAPSULAMIENTO 10
2. VI CLASE 12
2. VII RELACIONES ENTRE CLASES 13
2. VIII JERARQUÍA 13
2. IX HERENCIA 14
2. X ENSAMBLE 19
2. XI POLIMORFISMO 20

3. CICLO DE DESARROLLO EN EL PARADIGMA DE OBJETOS 23

4. OBJETIVOS DE LA ORIENTACIÓN A OBJETOS 26

5. VENTAJAS DE LA ORIENTACIÓN A OBJETOS 27

6. PROBLEMAS CON LA ORIENTACIÓN A OBJETOS 28

7. BIBLIOGRAFÍA UTILIZADA 29

8. EJEMPLOS 30

8. I **EJEMPLO 1-1 :** *OBJETOS* 30
REFERENCIAS 30
POLÍGONO 30
PILA (Tipo abstracto de datos) 30
VENTANA 30
CRÉDITO 31

8. II **EJEMPLO 2-1 :** *MENSAJES* 31

8. III **EJEMPLO 3-1 :** *INTERFACE E IMPLEMENTACIÓN* 31

8. IV **EJEMPLO 3-2 :** *ENCAPSULAMIENTO Y CAMBIOS EN LA IMPLEMENTACIÓN* 33

8. V **EJEMPLO 5-1 :** *HERENCIA* 34

8. VI **EJEMPLO 5-2 :** *HERENCIA Y REDEFINICIÓN* 35

8. VII **EJEMPLO 5-3 :** *CLASES ABSTRACTAS* (RUMBAUGH) 36

8. VIII **EJEMPLO 5-4 :** *JERARQUÍA DE CLASES* 37

8. IX **EJEMPLO 5-5 :** *RED DE CLASES CON HERENCIA MÚLTIPLE* (MEYER) 38

9. APÉNDICE 39

PROYECTO PHOTO CD de KODAK 39

1. INTRODUCCIÓN

En los últimos años ha tomado consenso la idea de que el paradigma de objetos será un elemento primordial en la ingeniería de software de los años '90. Se han publicado numerosos libros y revistas especializadas dedicados a la divulgación de este paradigma que vió su nacimiento a fines de la década del '70. La madurez actual de esta tecnología permite disponer además de metodologías específicas de análisis y diseño, de poderosas herramientas de desarrollo y lenguajes de programación, tales como Smalltalk y C++, y aún de bases de datos orientadas a objetos. También, es posible acceder a capacitación especializada y diversos congresos internacionales y conferencias de divulgación. Es por todo esto, y porque principalmente se han comprendido las ventajas de la aplicación de la orientación a objetos, que varias empresas (especialmente en EE. UU.) comenzaron a utilizarla en sus desarrollos de sistemas.

1.1 MOTIVACIÓN

El desafío principal de la Ingeniería de Software consiste en **construir software eficiente y económico.**

Pero la situación actual está muy distante de la ideal, por causa de lo que se podría llamar el **caos en informática.** Este se evidencia en la gran desorganización que podemos encontrar en cualquier PC, como ser : redundancia de procesos y datos, escasa documentación y desorden en el disco rígido (demasiados programas y datos sin estar muy bien definidas su función y utilidad). En el ambiente PC los usuarios se desvinculan de las áreas de sistemas de sus empresas y comienzan a crear ellos mismos sus propias aplicaciones.

Estos problemas se producen en parte porque el software es inherentemente complejo. Sin embargo, es la tarea de cualquier profesional en informática presentarle al usuario un producto simple y fácil de usar.

LA CRISIS DEL SOFTWARE [1] : En los últimos años la creciente complejidad del software ha vuelto a poner al desnudo la "Crisis del Software". Esta se manifiesta básicamente en los costos crecientes del desarrollo del software, la imposibilidad de las organizaciones de cumplir con sus compromisos, etc.

[1] Nota : extractado de [PRIETO]

Hay tres aspectos esenciales para combatir la crisis del Software:

I. La necesidad de bajar los costos de mantenimiento, extensión y modificación del software. Según Cox el principal problema es la actitud frente al cambio; y el cambio es el principal enemigo del software. ¿ A qué se llama cambio ? A la necesidad de introducir modificaciones en el software, fundamentalmente por cambios en el dominio de la aplicación.

Hay dos estrategias para resolver el problema :

- **Estrategia convencional** : negar el cambio.

- **Estrategia moderna** : tomar el cambio como una parte integral del proceso de desarrollo de software.

II. Desarrollar software reusable. La naturaleza rutinaria del desarrollo de software es sorprendente. Sin embargo, muy pocas veces se utilizan componentes de software construidas previamente para los sistemas nuevos.

III. La necesidad de reducir el "gap" semántico. La construcción de programas representa un mapeamiento de entidades pertenecientes al dominio de los problemas a las representaciones abstractas del dominio de las soluciones. Es evidente que cuanto más próximos se encuentran estos dominios, más fácil será la construcción de programas. La distancia entre ellos se denomina "gap" semántico.

La orientación a objetos pretende producir software robusto que pueda ser fácilmente reusado, refinado, probado, mantenido y extendido, en consecuencia asegurándole una madurez productiva.

El software es **reusado** cuando se lo usa como parte de un software distinto de aquél por el que fue creado. El software es **refinado** cuando se lo usa como base para la definición de otro software. Es **probado** cuando se determina si su comportamiento está de acuerdo con la especificación o no. Es **mantenido** cuando se le encuentran y corrigen errores. Y el software es **extendido** cuando se incorpora nueva funcionalidad a un sistema ya existente.

En conclusión, si se quiere lograr software ***eficiente y económico*** se hace indispensable que en su construcción se pueda utilizar (reutilizar) componentes de código (diseño o incluso análisis) que ya estén creadas y probadas, que el software sea fácilmente mantenible y extensible y que el proceso de desarrollo pueda ser iterativo y flexible (para que una mala decisión inicial no obligue a tirar todo y comenzar de cero).

Las técnicas orientadas a objetos pretenden solucionar estos problemas así como las técnicas estructuradas lo intentaron en su momento. Es importante aclarar que el paradigma de objetos no descarta las técnicas estructuradas, sino que pretende ser un paso más en la evolución del desarrollo de software y como tal aprovecha las técnicas anteriores.

1. II EVOLUCIÓN DE LOS MÉTODOS DE ANÁLISIS

Análisis : es el estudio del dominio del problema, con el objeto de lograr una especificación de la conducta externamente observable del sistema. Es una definición completa, consistente y viable de lo que se necesita. Análisis implica el proceso de determinar o extraer los requerimientos de un sistema, es decir, lo **que** el sistema debe hacer para el cliente y **no como** será implementado el sistema.

Los requerimientos incluyen operaciones funcionales y características operacionales (cuantificadas) como ser facilidad de uso, confiabilidad, disponibilidad, mantenibilidad y performance. También incluyen, las interfaces con las que el software debe interactuar, ambientes a los que el software se debe adaptar y cualquier otra restricción de diseño aplicable (visiones de contexto).

Los tres primeros métodos (que se describen a continuación) han sido aplicados en proyectos informáticos, algunas veces con resultados exitosos y otras con grandes fracasos. Los aspectos más importantes y útiles de estos métodos tienen su lugar específico en el enfoque de orientación a objetos. Por lo tanto, es importante destacar que no se está tratando de abolir los viejos y más establecidos métodos, sino que se intenta incorporar las mejores ideas de los tres primeros métodos en uno más amplio e integrador.

Lo primero fue la programación solamente sin definiciones ni análisis previo. Cuando se fue más allá de las primeras aplicaciones en las grandes máquinas y a medida que éstas se fueron haciendo más complejas, se hizo evidente la necesidad realizar Análisis y Diseño.

DESCOMPOSICIÓN FUNCIONAL

Se considera que el sistema cumple funciones del mundo real y abstraemos las funciones de ese mundo que el sistema desarrolla. Entonces se definen las funciones y subfunciones que el nuevo sistema debe ejecutar. El énfasis está puesto en **qué procesamiento** es requerido por el sistema.

DESCOMPOSICIÓN FUNCIONAL = FUNCIONES + SUBFUNCIONES + INTERFACES FUNCIONALES

Los problemas que trae aparejado este enfoque son :

- Dificultad para identificar las funciones.
- Es muy probable que dos analistas distintos identifiquen funciones diferentes para un mismo sistema.

◈ No se hace abstracción sobre los datos que se utilizarán, lo cual provoca inconsistencia y redundancia de datos.

◈ La especificación resultante modela en forma indirecta el dominio del problema, se hace difícil para el analista y para el usuario evaluar si los requerimientos constituyen una especificación correcta de lo que el sistema debería hacer.

◈ Otro problema es elegir las funciones y subfunciones de manera que la interface interfuncional sea mínima.

DESCOMPOSICIÓN DE FLUJOS DE DATOS

De acuerdo a este método, también conocido como técnicas estructuradas, el mundo está formado por flujos de datos y procesos que transforman los datos. Por lo tanto el analista describe el mundo real en base a flujos de datos y burbujas. Se torna necesario para el analista , y más significativamente para el usuario, el seguimiento del flujo de los datos en el mundo real y volcar este flujo en análisis y especificación. Pero el seguimiento del flujo de datos no es una forma natural de manejar la complejidad cuando se está analizando un problema, por lo tanto aparecen diversas dificultades.

Es importante realizar una distinción entre las **técnicas estructuradas clásicas y las modernas**. En las primeras se define un DFD inicial con el usuario y luego se hace descomposición funcional mediante DFD's de niveles inferiores. El problema principal es que no hay ninguna técnica ni criterio para hacer está descomposición. En las técnicas modernas se soluciona este problema considerando a los sistemas como de respuestas preplaneadas y por lo tanto el criterio para la organización de los DFD's es el **particionamiento por eventos**. La ventaja resultante es que los DFD's son mas fáciles de definir ya que es más natural identificar eventos que funciones, además, es más sencillo descomponer las burbujas ya que en general agrupan una cantidad relativamente manejable de funciones.

Las dificultades que origina este método son :

◈ El gran desafío consiste en la elección de las burbujas, el particionamiento por eventos ayuda solo cuando la cantidad de eventos no está más allá de lo manejable.

◈ El modelo de procesos y el modelo de datos no están integrados y se crean problemas para conciliar ambas visiones.

- Se generan grandes cantidades de documentación (especialmente en cuanto al diccionario de datos).

- No es verdad que los usuarios entiendan los DFD's y que por consiguiente se los puede utilizar en una entrevista para validar el conocimiento que el analista tenga del sistema.

- La descomposición por flujo de datos sigue teniendo un énfasis muy fuerte en la abstracción procedural y ,por lo tanto, se está apoyando en los aspectos menos estables de un sistema.

- El pasaje del análisis al diseño implica un cambio de modelo. En análisis se utilizan DFD's básicamente y el diseño se documenta en base a diagramas de estructura (modelos de procedimientos). Es decir, que al pasar al diseño hace falta además de agregarle al modelo los detalles de implementación, un cambio en la forma de representación.

INFORMATION MODELING

De acuerdo a este enfoque se modela al mundo real **en base a los datos.** Este método ha evolucionado a través de los años desde su herramienta de modelización primaria, el diagrama de entidad relación, hasta los modelos semánticos de datos.

INFORMATION MODELING = OBJETOS + ATRIBUTOS + RELACIONES + SUPERTIPOS/SUBTIPOS + OBJETOS ASOCIATIVOS.

La vieja estrategia consiste en : desarrollar una lista tentativa de atributos, asociar los atributos a objetos, agregar relaciones, refinar con subtipos/supertipos y objetos asociativos, y finalmente normalizar.

La nueva estrategia es muy similar, excepto en que la etapa inicial se identifican objetos del mundo real y se los describe con atributos.

La ventaja de este método es que achica el gap semántico, ya que el modelo tiene una relación mas directa con el mundo real y con la forma en que las personas lo entienden.

El problema fundamental que Information Modeling presenta es que genera un modelo parcial al no tomar en cuenta los procesos.

ORIENTACIÓN A OBJETOS

Se ha creado una gran confusión en base a la orientación a objetos, especialmente en cuanto a qué es un objeto. Esto es porque el termino objeto proviene de dos áreas diferentes de estudio como son Information Modeling y la Programación Orientada a Objetos. Además, hoy en día todo producto comercial se dice orientado a objetos más porque parece ser la moda del momento que por haber sido desarrollado para soportar el Paradigma de la Orientación a Objetos.

En el momento de diseñar la arquitectura de un sistema, el diseñador de software se enfrenta con una decisión fundamental : **¿ Se basará la estructura sobre las acciones o sobre los datos ?**. En la respuesta a esta pregunta radica la diferencia entre los métodos de diseño y análisis tradicionales y el enfoque orientado a objetos.

Enfoques híbridos deben ser utilizados, ya que al final los datos y las acciones cumplen un rol dentro de la estructura. Sin embargo, la cuestión de mayor importancia es qué usar como criterio primario para describir la descomposición de mayor nivel de las estructuras.

El paradigma de la **orientación a objetos** construye **estructuras de software basadas en los objetos** (datos) que cada sistema o subsistema manipula, más que en las funciones que debe cumplir. Qué debe hacer un sistema es por supuesto una cuestión importante para obtener una implementación que funcione; esta cuestión debe resolverse tarde o temprano. Desde el punto de vista de la orientación a objetos es mejor tarde que temprano. Usando diseño orientado a objetos, se difieren las decisiones concernientes a la implementación hacia el final, para concentrarse primero en la porción más estable de todo sistema que son los datos.

El motivo para utilizar los datos como criterio para efectuar la modularización del sistema está basado en algunos aspectos de la calidad del software como ser : compatibilidad , reusabilidad (utilizar componentes previamente creados en distintos sistemas), extensibilidad (facilidad para adaptar a cambios de especificación y requerimientos).

√ **Compatibilidad** : es un significativo incentivo para utilizar los datos como criterio para descomposición. Es difícil combinar acciones si las estructuras de datos a las que acceden no son tomadas en cuenta. ¿ Por qué no combinar, en cambio, estructuras enteras de datos ?.

√ **Reusabilidad** : también sugiere algunos argumentos. Para cualquier aplicación que incluya estructuras de datos no triviales, es difícil construir componentes reusables si éstas engloban acciones solamente e ignoran la parte de los datos.

√ Finalmente se observa que **extensibilidad** provee el argumento más convincente : a través del tiempo, las estructuras de datos (y no las funciones), al menos vistas al suficiente nivel de abstracción, constituyen el verdadero aspecto estable del sistema. Por lo tanto, una arquitectura basada en los datos está menos expuesta a ser modificada como consecuencia de cambios en los requerimientos del sistema.

2. EL MODELO DE OBJETOS

2. I OBJETO

El elemento primordial del enfoque orientado a objetos es el objeto.

Un objeto es una entidad que tiene un conjunto de responsabilidades y que encapsula un estado interno.

Las responsabilidades (o servicios) que realiza el objeto se implementan mediante métodos, los cuales describen el comportamiento del objeto. El estado interno del objeto se implementa mediante un conjunto de propiedades o atributos **encapsulados.**

Por lo tanto, los objetos combinan procedimientos y datos en una misma entidad.

Un objeto tiene estado, comportamiento y una identidad.

El **estado** del objeto comprende las propiedades estáticas del objeto más los valores actuales de cada una de esas propiedades.

Por ej. : el objeto CAJA DE AHORRO tiene los atributos número, titular y saldo que constituyen su estado interno. Estas son propiedades estáticas. Por otro lado, el contenido actual de cada atributo (por ej. 2345/13 'PEDRO' y 1200.56) representan el valor dinámico de sus propiedades, y son afectados por el comportamiento del objeto.

El hecho de que cada objeto tiene estado implica que todo objeto ocupa una cierta cantidad de espacio, ya sea en el mundo físico o en la memoria de una computadora.

El **comportamiento** de un objeto es cómo el objeto actúa y reacciona, en términos de sus cambios de estados y de los mensajes que pasa. Es decir que, el comportamiento de un objeto está completamente definido por sus acciones.

Por ej. : el comportamiento del objeto CAJA DE AHORRO está compuesto por las acciones SALDO, EXTRAER , DEPOSITAR , etc.

La **identidad** es la propiedad de un objeto que lo distingue de todos los otros objetos.

Por ej. : un objeto CAJA DE AHORRO es distinto de otro objeto CAJA DE AHORRO. Pueden o no tener el mismo número, titular o saldo. El primero puede tener como titular a 'PEPE" y el segundo a 'JUAN'.

Ver ejemplo 1-1 (pag. 30).

2. II PROGRAMA

Un programa construido con el modelo de objetos es una colección de objetos cooperantes. La cooperación se lleva a cabo mediante mensajes.

2. III MENSAJE

Un mensaje consiste en el nombre de una operación y de los argumentos por ella requeridos.

Los mensajes constituyen el medio por el cual los objetos se comunican con el fin de lograr una acción cooperativa. Cuando un objeto envía un mensaje a otro objeto, el emisor le está requiriendo al receptor que realice la operación nombrada y (posiblemente) devolverle alguna información. Cuando el receptor recibe el mensaje, realiza la operación requerida en la forma en que sepa hacerla. La petición que hace el emisor no especifica cómo debe ser realizada la operación. Tal información está siempre oculta para el emisor.

Por ej. : imagine una red de computadoras corriendo un sistema operativo en el cual una de las responsabilidades de un objeto archivo es saber como imprimirse el mismo. Un mensaje que se puede imaginar bajo estas circunstancias sería : **imprimir**. Cuando los usuarios le piden a un archivo en particular que se imprima, raramente les importará cuales son específicamente los pasos necesarios para producir la salida impresa. Solamente les preocupará tener la posibilidad de enviar el mensaje **imprimir** al archivo que deseen imprimir y obtener la salida en un lapso de tiempo razonable.

Ver ejemplo 2-1 (pag. 31).

2. IV ABSTRACCIÓN

La abstracción es una de las formas fundamentales con que los seres humanos se enfrentan a la complejidad, se la define como :

"Una abstracción denota las características esenciales de un objeto que lo distinguen de todas las otras clases de objetos y que por lo tanto proporcionan límites conceptuales bien definidos, con relación a la perspectiva del observador". [Booch]

La abstracción se concentra en la visión exterior de un objeto y , por lo tanto, sirve para separar el comportamiento esencial del objeto de su implementación. La abstracción , en otras palabras, captura el comportamiento completo de un objeto, nada más y nada menos.

Todas las abstracciones tienen propiedades dinámicas y estáticas. Por ejemplo : un objeto archivo ocupa cierto espacio en un dispositivo de memoria determinado, tiene un nombre, y tiene contenido. Estas son todas propiedades estáticas. El valor de cada una de estas propiedades es dinámico, relativo al tiempo de vida del objeto: un objeto archivo puede aumentar o disminuir su tamaño, su nombre puede cambiar y su contenido también. En un estilo de programación procedural, la actividad que cambia los valores dinámicos de los objetos es la parte central de todos los programas ; las cosas ocurren cuando los subprogramas son llamados y las sentencias se ejecutan. En un estilo de programación orientado a objeto, las cosas ocurren cuando le enviamos un mensaje a un objeto. Al invocar una operación sobre un objeto se produce una reacción de éste. Qué operaciones se pueden realizar sobre un objeto y cómo ese objeto reacciona constituyen el comportamiento entero del objeto.

2. V ENCAPSULAMIENTO

Significado del encapsulamiento. La abstracción de un objeto debería preceder las decisiones sobre su implementación. Una vez que una implementación ha sido seleccionada, debería ser tratada como un secreto de la abstracción y permanecer oculta al resto de los objetos. Ninguna parte de un sistema complejo debería depender de los detalles internos de alguna otra parte. Mientras la abstracción ayuda a la gente a pensar qué es lo que está haciendo, el encapsulamiento permite que se hagan cambios en los programas en forma confiable y con un esfuerzo limitado.

La abstracción y el encapsulamiento son conceptos complementarios, ya que, la abstracción se concentra en la visión exterior de un objeto y el encapsulamiento evita que los objetos clientes accedan a su visión interna, donde el comportamiento de la abstracción es implementado. De esta forma, el encapsulamiento provee barreras explícitas entre diferentes abstracciones. Por ejemplo : en el diseño de una aplicación de base de datos, es una práctica estandar escribir los programas de manera que no se preocupen por la representación interna de los datos, sino que, dependen solamente de un esquema que denota la visión lógica de los datos.

Incluso podría decirse que, para que la abstracción funcione, las implementaciones deberían ser encapsuladas. En la práctica esto significa que cada clase (objeto) debe tener dos partes : una interface y una implementación. La **interface** de un objeto captura solamente su visión externa. La **implementación** de un objeto comprende la representación de la abstracción y los mecanismos necesarios para lograr el comportamiento deseado. La división explícita interface/implementación representa una clara separación de conceptos; la interface es el lugar donde se expresan todos los supuestos que un cliente puede hacer sobre el objeto, y la implementación encapsula detalles sobre los cuales ningún cliente puede hacer suposiciones.

Para resumir se define encapsulamiento como sigue :

"Encapsulamiento es el proceso de esconder todos los detalles de un objeto que no contribuyen a sus características esenciales". [Booch]

Ver ejemplo 3-1 (pag. 31).

Encapsulamiento también se conoce por el nombre de Information-hiding u ocultamiento de la información. Aunque algunos autores establecen una distinción entre estos términos, en el presente trabajo serán tratados como sinónimos.

El ocultamiento de la información permite sacar de la vista una porción de aquellas cosas que forman parte de un objeto. Esto es útil para incrementar los beneficios obtenidos de la abstracción y para diseñar código que pueda ser modificado, mantenido y extendido más fácilmente.

El encapsulamiento distingue entre la habilidad para realizar cierta acción y los pasos específicos necesarios para llevarla a cabo. Públicamente, un objeto revela sus habilidades : "Puedo hacer estas cosas" y declara "Puedo contar estas cosas". Pero no cuenta cómo las hace o sabe, ni necesita que otros objetos estén al tanto de ello. En cambio, algún otro objeto que le pide una operación o alguna información actúa como un buen gerente. Especifica el trabajo o pide la información y luego se va. No se queda dando vueltas viendo como se hace el trabajo o se calcula la información.

Es decir que, un objeto mediante su interface hace públicas las acciones que puede realizar y la información que puede devolver, pero guarda internamente ,como un secreto, los detalles de qué métodos o procedures ejecuta para llevar a cabo dichas acciones , ni tampoco dice si la información que devuelve es directamente el valor de un atributo o el resultado de una función. Por lo tanto, el objeto cliente que le está requiriendo un servicio no puede hacer ninguna suposición sobre la forma en que el objeto servidor implementa su comportamiento y ,como consecuencia, el modelo de objetos asegura que la implementación en el objeto cliente es independiente de la implementación del objeto servidor. Está característica es sumamente beneficiosa, ya que cualquier cambio en la implementación del objeto servidor (ya sea por mantenimiento, modificación o extensión) no provocará ningún cambio en el objeto cliente, siempre y cuando no sea necesario modificar la interfaz del servidor.

Los objetos solo conocen las operaciones que pueden requerir a otros objetos. Esto ayuda a tomar una visión abstracta del objeto mientras se lo analiza o diseña. Algunos detalles que todavía pueden no ser importantes o desconocidos, pueden ser diferidos para más adelante en el ciclo de desarrollo. Y entonces es posible concentrarse en la esencia del análisis primero y del diseño después.

Otra ventaja aparece más tarde en la vida del sistema. Se puede cambiar la representación interna de un objeto o implementar un algoritmo superior para una operación específica sin cambiar la interface pública y abstracta del objeto. Los objetos que dependen de los resultados de las operaciones o de ciertos valores no se ven afectados por el cambio.

Ver ejemplo 3-2 (pag. 33).

El encapsulamiento inteligente localiza las decisiones de diseño que son propensas al cambio. A medida que el sistema evoluciona, sus desarrolladores pueden descubrir que en el uso real, ciertas operaciones consumen más tiempo del aceptable o que ciertos objetos ocupan mayor espacio que el disponible. En tales casos, la representación de un objeto es frecuentemente cambiada de manera que se puedan aplicar algoritmos más eficientes o que se ahorre espacio al calcular en vez de almacenar ciertos datos. Esta habilidad para cambiar la representación de una abstracción sin perturbar ninguno de sus clientes es el beneficio esencial del ocultamiento de la información.

El encapsulamiento aisla una parte del sistema de otras partes, permitiendo que el código sea modificado y extendido, y que los errores sean corregidos, sin el riesgo de introducir efectos secundarios innecesarios y no intencionales. Los objetos constituyen la forma de poner este principio en la práctica dentro de un programa.

1 - Primero se abstrae la funcionalidad e información relacionada y se la encapsula en un objeto.

2 - Luego se decide qué funcionalidad e información otros objetos requerirán de este objeto, y el resto se esconde. Se diseña una interface pública que permita a otros objetos acceder a lo que necesiten, y la representación privada es protegida por defecto del acceso de otros objetos.

2. VI CLASE

Se dice que los objetos que comparten el mismo comportamiento pertenecen a la misma clase.

Una clase es una especificación genérica para un número arbitrario de objetos similares.

Se puede pensar a una clase como un molde para un tipo específico de objetos; así como, la definición de una tabla en un RDBMS es un modelo para las tuplas que la componen (aunque las tuplas solo tienen datos). Lo que diferencia a objetos de una misma clase es su identidad y su contenido (datos).

Una clase permite construir una taxonomía de objetos a un nivel conceptual y abstracto. Las clases nos permiten describir en un lugar el comportamiento genérico de un conjunto de objetos, y entonces crear objetos que se comporten en esa forma cuando los necesitemos.

Los objetos que se comportan en la forma especificada por una clase se llaman **instancias** de esa clase. Todos los objetos son instancia de alguna clase. Una vez que se crea la instancia de una clase, se comporta como todas las otras instancias de su clase, capaz de realizar cualquier operación, para la cual tenga métodos, inmediatamente después de haber recibido un mensaje. Puede también requerir a otras instancias, ya sea de la misma o de otras clases, que realicen otras operaciones en su nombre. Un programa puede tener tantas instancias de una clase en particular como sea necesario.

Mientras que un objeto es una entidad concreta que existe en el tiempo y el espacio, y que lleva a cabo un determinado rol dentro del sistema en general; una clase representa solo una abstracción, la "esencia" de un objeto, que captura la estructura y el comportamiento de todos los objetos que a ella pertenecen. Por lo tanto, podemos hablar de la clase Mamífero, que representa las características comunes a todos los mamíferos. Y para identificar a un mamífero en particular en dicha clase, debemos hablar de "este mamífero" o "aquél mamífero".

Las clases y los objetos son conceptos separados aunque están íntimamente relacionados. Específicamente todo objeto es instancia de alguna clase, y toda clase tiene cero o más instancias. Para prácticamente todas las aplicaciones, las clases son estáticas; por lo tanto, su existencia, semántica y relaciones se fijan en forma previa a la ejecución de un programa. Similarmente, la clase de la mayoría de los objetos es estática, es decir que una vez que un objeto es creado su clase es fija. Por el contrario, los objetos son típicamente creados y destruidos a una tasa (velocidad) importante durante la ejecución de una aplicación.

2. VII RELACIONES ENTRE CLASES

Existen tres tipos de relaciones entre clases :

- Jerárquicas de herencia
- Jerárquicas de ensamble
- Contractuales, de colaboración o de asociación : estas relaciones se dan cuando un objeto necesita para poder llevar a cabo su comportamiento un servicio provisto por otro objeto.

2. VIII JERARQUÍA

La abstracción es algo muy bueno, pero en casi todas las aplicaciones encontramos muchas más abstracciones (objetos) de las que podemos comprender a la vez. El encapsulamiento ayuda a manejar esta complejidad al esconder la visión interna de las abstracciones. Pero esto no es suficiente. Un conjunto de abstracciones frecuentemente forman una jerarquía, e identificando estas jerarquías en el análisis y en el diseño, se simplifica enormemente el entendimiento del problema.

Jerarquía es un ranking u ordenamiento de abstracciones (objetos).

Las dos jerarquías más importantes en un sistema complejo son la **herencia** y el **ensamble** (todo-parte).

La jerarquía de ensamble describe las relaciones de tipo "es parte de" que se dan entre objetos.

Básicamente, la herencia define una relación entre clases, donde una clase comparte estructura interna y comportamiento definido en una o más clases (herencia simple y múltiple respectivamente). Por lo tanto, la herencia representa una jerarquía de abstracciones, en la cual una subclase hereda de una o más superclases. Típicamente, una subclase aumenta o redefine la estructura y comportamiento existentes de sus superclases.

2. IX HERENCIA

El paradigma de objetos incluye otro mecanismo de abstracción : la herencia.

Herencia es un mecanismo que acepta que la definición de una clase incluya el comportamiento y estructura interna definidos en otra clase más general.

En otras palabras, se puede decir que una clase es justo como otra clase excepto que la nueva clase incluye algo extra. La herencia provee un mecanismo para la clasificación, con ayuda de él se pueden crear taxonomías de clases.

La herencia permite concebir una nueva clase de objetos como un refinamiento de otra, con el objeto de abstraerse de las similitudes entre clases, y diseñar y especificar solamente las diferencias para la clase nueva. De esta manera se pueden crear clases rápidamente.

También hace posible la reutilización de código. Las clases que heredan de otras clases pueden heredar comportamiento, frecuentemente una gran cantidad de comportamiento, si la clase o clases de donde heredan fueron diseñadas para poder ser refinadas. Si una clase puede usar algo o todo el código que hereda, no necesita re-implementar esos métodos. Este mecanismo de abstracción puede proveer un modo poderoso de producir código que pueda ser reusado una y otra vez.

Subclase es una clase que hereda el comportamiento y estructura interna de otra clase. Una subclase generalmente agrega su propio comportamiento y estructura interna para definir su propio y único tipo de objeto.

Por ejemplo : suponiendo un sistema que incluye una variedad estandar de clases, incluyendo una clase llamada ARRAY que permite especificar un número de elementos, indexarlos, y manipularlos por medio de un índice. Una aplicación puede necesitar una clase para manipular un string de caracteres. Esta clase (que llamaremos STRING) se comporta como un array de caracteres, y entonces heredará de la clase ARRAY de manera que sus elementos puedan ser manipulados a través de un índice. Sin embargo, los strings tiene otra característica, que es la de poder ser comparados y dispuestos en orden alfabético. Por lo tanto, la clase STRING agregará los métodos necesarios para llevar a cabo dicho comportamiento.

Se llama **superclase** a una clase de la cual otras clases heredan su comportamiento y estructura interna. Una clase puede tener solo una superclase (**herencia simple**) o puede tener varias (**herencia múltiple**) combinando comportamiento de varias fuentes y agregando solo un pequeña porción por sí sola para producir su propio y único tipo de objeto.

En el ejemplo anterior, se puede asumir que la habilidad para comparar dos instancias de una clase, para determinar que una es menor que la otra y ordenar el resultado, está definida por una clase llamada MAGNITUD. Entonces se podría definir a STRING de manera que también fuera subclase de MAGNITUD y definiendo su orden de modo que fuera alfabético. En ese caso, la clase STRING tendría dos superclases: MAGNITUD y ARRAY. El comportamiento que hereda de ellas tiene que ver con el conocimiento de como comparar y ordenar sus elementos, y como manipularlos a través de índices. El comportamiento que agrega tiene que ver con el conocimiento de que el orden es alfabético.

Ver ejemplo 5-1.

Redefinición : Una subclase puede redefinir un método de la superclase simplemente definiendo un método con el mismo nombre. El método en la subclase redefine y reemplaza al método de la superclase. Hay varias razones por las que puede ser útil redefinir un método : para especificar comportamiento que depende de la subclase, para lograr mayor performance a través de un algoritmo más eficiente para la subclase o guardando un atributo de cálculo dentro de la estructura interna del objeto, etc.

Por ejemplo : en una aplicación en la que sea necesario manipular figuras geométricas, podríamos tener la clase POLÍGONO y la subclase RECTÁNGULO, entre otros mensajes tanto un POLÍGONO como un RECTÁNGULO deberían informar cuál es su perímetro a través del mensaje PERÍMETRO. En el caso del polígono se podría implementar como la suma de la longitud de sus lados, mientras que en el rectángulo se podría redefinir el mensaje perímetro para se lo calcule como el doble de la suma de la longitud de dos lados consecutivos.

Se pueden redefinir métodos y valores por defecto para los atributos. Toda redefinición debe preservar la cantidad o tipo de parámetros utilizados en un mensaje y el tipo del valor devuelto. Nunca debe redefinirse un método de manera que sea semánticamente inconsistente con el originalmente heredado. Una subclase es un caso especial de su superclase y debería ser compatible con ella en todo respecto.

Ver ejemplo 5-2.

Las clases que no han sido creadas para producir instancias de sí mismas se llaman **clases abstractas**. Existen para que el comportamiento y estructura interna comunes a una determinada variedad de clases pueda ser ubicado en un lugar común, donde pueda ser definido una vez (y más tarde, mejorado o arreglado de una sola vez, si es necesario), y reusado una y otra vez. Las clases abstractas especifican en forma completa su comportamiento pero no necesitan estar completamente implementadas. Pueden también especificar métodos que deban ser redefinidos por cualquier subclase. Un ejemplo de esto puede ser la implementación de cierto comportamiento por defecto para prevenir un error del sistema, pero dicha implementación debe ser redefinida o aumentada por los métodos implementados en las subclases.

Las subclases concretas heredan el comportamiento y estructura interna de sus superclases abstractas, y agregan otras habilidades específicas para sus propósitos. Puede ser que necesiten redefinir la implementación por defecto de sus superclases abstractas, para poder comportarse de alguna forma que tenga sentido para la aplicación de la cual son parte. Estas clases completamente implementadas crean instancias de sí mismas para hacer el trabajo útil en el sistema.

Ver ejemplo 5-3

En un ejemplo anterior se nombraba una clase llamada MAGNITUD que contenía comportamiento para comparar dos objetos, para decir si uno es más grande que otro y para ordenar el resultado. Imaginando que esto es virtualmente todo lo que la clase MAGNITUD puede hacer, claramente, una instancia de esta clase no será útil para una aplicación real. La clase MAGNITUD existe para capturar en un solo lugar el comportamiento que implementa. Muchas clases requieren la habilidad de comparar y ordenar elementos, entonces es útil implementar dichas habilidades en un solo lugar, permitiendo que otras clases las hereden. Si un algoritmo superior o más eficiente es descubierto, se lo podría implementar en la clase MAGNITUD, y toda clase en el sistema que herede su código es automáticamente mejorada. Mas aún, si se encuentra un error, hace falta corregirlo en un solo lugar y no en varios.

Debido a las relaciones de herencia el **mecanismo de respuesta a un mensaje** se comporta del siguiente modo :

- El objeto recibe un mensaje.
- Se busca el mensaje en la clase del objeto receptor.
- Si el mensaje es encontrado, se ejecuta el método correspondiente.
- Si por el contrario, no se encuentra el mensaje en la clase del objeto, se lo busca en la superclase. Si no se lo encuentra en la superclase se lo busca en la superclase de la superclase, y así sucesivamente.
- Si finalmente el mensaje no es encontrado, se detiene la ejecución y se da un mensaje de error porque el objeto no entiende el mensaje.

La jerarquía de **herencia** es también conocida como jerarquía de **generalización / especialización**. Estos tres términos se refieren a aspectos de la misma idea y frecuentemente son utilizados indistintamente. Se utiliza generalización para referirse a la relación entre clases, mientras que herencia se refiere al mecanismo de compartir atributos y métodos utilizando la relación de generalización. Generalización y especialización corresponden a dos puntos de vista diferentes de la misma relación, desde la perspectiva de las superclases o de las subclases. La palabra generalización deriva del hecho de que la superclase generaliza las subclases. Especialización se refiere al hecho de que las subclases refinan o especializan la superclase. En la práctica más allá de esta distinción teórica estos términos serán usados indistintamente.

A medida que la jerarquía de herencia evoluciona, la estructura interna y el comportamiento que son iguales para clases diferentes tenderá a migrar hacia superclases comunes. Este es el motivo por el cual se dice que la herencia es una jerarquía de generalización/especialización. Las superclases representan abstracciones generalizadas, y las subclases representan especializaciones en las cuales campos y métodos de la superclase son agregados, modificados o aún ocultados. De este modo la herencia permite establecer las abstracciones con una economía de expresión. Hace posible definir software nuevo de la misma forma en que le introducimos conceptos a los novatos, comparándolos con algo que ya les sea familiar.

Herencia se ha transformado en sinónimo de reusabilidad de código dentro de la comunidad de orientación a objetos. Luego de modelizar un sistema, el desarrollador analiza las clases resultantes y trata de agrupar clases similares y reusar código común. Frecuentemente hay código disponible de trabajo anterior (como una librería de clases); el desarrollador puede reusarlo y modificarlo, donde sea necesario, para lograr el comportamiento preciso y deseado. Pero otro uso igual de importante (y menos reconocido) de la herencia es la simplificación conceptual que proviene de reducir el número de características independientes en un sistema.

Analizando el proceso de construcción del software desde la perspectiva de los módulos que lo componen y considerando a cada clase como un módulo, la herencia es una técnica clave de reusabilidad.

Un módulo (clase) es un conjunto de servicios que se ofrecen al mundo exterior. Sin herencia, cada módulo debe definir por sí mismo todos los servicios que ofrece. Por supuesto que las implementaciones de esos servicios pueden basarse en los servicios que suministran otros módulos, esté es precisamente el propósito de la relación cliente-servidor. Pero no hay ninguna forma de definir un módulo nuevo simplemente añadiendo servicios nuevos a módulos previamente definidos.

La herencia provee precisamente esta posibilidad. Todos los servicios de una superclase están automáticamente disponibles para la subclase, sin necesidad de definirlos nuevamente. La subclase es libre de agregar nuevas características de acuerdo a sus propósitos específicos. Un grado extra de flexibilidad se logra con la redefinición, la cual permite que la subclase pueda utilizar si lo desea implementaciones diferentes a las ofrecidas por la superclase.

Esto favorece un estilo de desarrollo de software completamente diferente de los enfoques tradicionales. En vez de tratar de resolver cada problema nuevo de la nada, la idea es construir sobre logros previos y extender sus resultados. Con esto se logra economía, porque no se rehace lo que ya está hecho, y robustez, porque se construye a partir de componentes de software convenientemente probados y libres de error.

El beneficio completo de este enfoque se entiende mejor en términos del principio abierto-cerrado (open-closed) enunciado por Meyer. El principio establece que una buena estructura de módulo debería ser abierta y cerrada :

- **Cerrada**, porque los clientes necesitan de los servicios del módulo para poder proseguir con su propio desarrollo, y una vez que se ha establecido una versión del módulo no deberían verse afectados por la introducción de nuevos servicios que no necesiten.

- **Abierto**, porque no hay ninguna garantía de que sean incluidos desde el principio todos los servicios potencialmente útiles para todos los clientes.

Este requerimiento doble es un dilema, y las estructuras de módulos clásicas no ofrecen una respuesta satisfactoria. Pero la herencia lo resuelve. Una clase es cerrada, ya que puede ser compilada, almacenada en una librería, y usada por otras clases clientes. Pero también es abierta, porque cualquier clase nueva la puede usar como superclase, agregando nuevos servicios. Cuando se define una subclase no hay necesidad de cambiar la original o perturbar a sus clientes. Esta propiedad es fundamental en la aplicación de la herencia a la construcción de software reusable.

Una de las cuestiones más difíciles en el diseño de estructuras de módulos reusables es la necesidad de tener en cuenta aspectos comunes que pueden existir entre grupos relacionados de abstracciones de datos (ej. : tabla secuencial, tabla hash, lista encadenada, array). El paradigma de objetos provee una forma de lograrlo, usando redes de clases conectadas por herencia (jerarquía de clases), entonces es posible tomar ventaja de las relaciones lógicas que existen entre estas implementaciones.

Ver ejemplos 5-4 y 5-5.

Existe una tensión saludable entre los principios de abstracción y encapsulamiento con el de herencia. La abstracción de datos intenta proveer una barrera detrás de la cual métodos y atributos están ocultos; la herencia requiere abrir esa interface hasta cierto punto y puede permitir que tanto métodos como atributos sean accedidos directamente. Para una clase determinada, usualmente hay dos tipos de clientes: los objetos que invocan operaciones a instancias de la clase, y subclases que heredan de la clase. Con ayuda de la herencia el encapsulamiento puede ser violado de tres formas : la subclase puede acceder a un atributo de la superclase, llamar a un método privado de la superclase, o referirse directamente a la superclase de su superclase.

En la práctica esta tensión se hace visible cuando por ejemplo por determinada razón se debe modificar un atributo de una clase y como consecuencia, deben modificarse también algunas partes de los métodos donde dicho atributo es utilizado. Debido al encapsulamiento se puede estar seguro de que dichos cambios no afectarán a las clases que sean clientes de ésta (siempre que la interface no sea modificada), pero si es posible que las subclases deban ser modificadas en la medida en que realicen alguna operación con dicho atributo. Finalmente, se puede decir que todo cambio realizado en una clase que no afecte su interfaz externa, podrá tener impacto en el código de la misma clase y de sus subclases.

2. X ENSAMBLE

Ensamble es la relación "todo-parte" o "es parte de" en la cual los objetos que representan los componentes de algo son asociados a un objeto que representa el todo". [Rumbaugh]

La jerarquía de ensamble describe las relaciones de tipo "es parte de" que se dan entre objetos, es decir que identifica cada una de las partes que componen un objeto. Entonces un objeto compuesto está formado por objetos componentes que son sus partes. Cada objeto parte tiene su propio comportamiento y el objeto compuesto es tratado como una unidad cuya responsabilidad es realizar la coordinación de las partes.

Por ejemplo : un documento está formado por párrafos que a su vez están formados por oraciones. Una factura está formada por líneas que a su vez consisten en un artículo, una cantidad y un precio.

Las partes pueden o no existir fuera del objeto compuesto o pueden aparecer en varios objetos compuestos. Por lo tanto, la existencia del objeto componente puede depender de la existencia del objeto compuesto del cual forma parte, ó en otro caso, el objeto componente puede tener una existencia independiente (esto último ocurre cuando algún objeto que no sea el objeto compuesto pueda enviarle mensajes directamente al objeto parte).

Es importante diferenciar las relaciones de ensamble y de colaboración. Si dos objetos están fuertemente acoplados por una relación de todo-parte es un ensamble. Si por el contrario los dos objetos son usualmente considerados como independientes, aunque se envíen mensajes frecuentemente, la relación es contractual o de colaboración. Por ejemplo : una empresa es un ensamble de divisiones, que a su vez están compuestas de departamentos. Pero una empresa no es un ensamble de empleados, puesto que empresa y personas son objetos independientes de la misma estatura.

Ensamble no es lo mismo que generalización (herencia). La relación de ensamble relaciona dos instancias. Dos objetos distintos están involucrados, uno es parte del otro. La herencia relaciona clases y es una manera de estructurar la descripción de un solo objeto. Tanto la superclase como la subclase referencian las propiedades de un único objeto. Con generalización, un objeto es simultáneamente una instancia de la superclase y de la subclase. La confusión puede aparecer porque tanto el ensamble como la generalización forman jerarquías. Un árbol de ensamble está compuesto de instancias de objetos que son parte de un objeto compuesto; mientras que, un árbol de herencia está compuesto de clases que describen un objeto.

2. XI POLIMORFISMO

Polimorfismo es la habilidad de dos o más objetos de responder a mensajes con igual nombre, cada uno de su propia forma.

Esto significa que un objeto no necesita saber a quién le está enviando un mensaje. Solo debe saber que se han definido varios tipos diferentes de objetos para que respondan a ese mensaje en particular.

Ejemplos : En el caso del POLÍGONO y el RECTÁNGULO ambas clases responden al mensaje PERÍMETRO devolviendo un número que representa el perímetro de la figura. Ambos, objetos responden al mismo mensaje y cada uno lo hace de la manera que es más apropiada. Por ello, al escribir una porción de código en la que se necesita saber el perímetro de una figura solo hace falta enviar el mensaje PERÍMETRO sin preocuparnos si el objeto es una instancia de la clase POLÍGONO o de la clase RECTÁNGULO.

Otro caso de polimorfismo podría darse en un sistema bancario, donde tanto la clase CUENTA_CORRIENTE como la clase CAJA_DE_AHORRO responden a los mensajes EXTRAER(monto) o SALDO().

Objetos de una variedad de diferentes, pero similares, clases pueden reconocer algunos mismos mensajes y responder en forma similar y apropiada. Una respuesta apropiada para una clase de objeto podría ser completamente inapropiada para otra clase. El emisor no necesita preocuparse por el método que se ejecuta como resultado del envío del mensaje.

El polimorfismo permite reconocer y explotar las similitudes entre diferentes clases de objetos. Cuando reconocemos que varios tipos diferentes de objetos pueden responder al mismo mensaje, estamos reconociendo la distinción entre el nombre del mensaje y un método. Un objeto envía un mensaje: si el receptor implementa un método que corresponda al mensaje, responderá. Diferentes respuestas son posibles, por lo tanto métodos diferentes tienen sentido para clases diferentes, pero el emisor puede simplemente enviar el mensaje sin preocuparse de la clase del receptor.

Un concepto relacionado al de polimorfismo es el de ***Binding dinámico : significa que el enlace entre el receptor del mensaje, y el mensaje, se realiza en forma dinámica (en tiempo de ejecución).***

Ejemplo :

En una aplicación de gestión comercial es necesario realizar la impresión del libro de IVA que está formado por un resumen de las facturas, notas de crédito y notas de débito emitidas en el mes. Suponiendo que hay un archivo donde están almacenados los documentos del mes, el problema se reduce a listar este archivo. Para calcular el importe de cada documento a fin de poder imprimirlo en el listado solo hace falta enviarle el mensaje IMPORTE() a cada objeto, sin preocuparse si es un objeto FACTURA, NOTA_CRED o NOTA_DEB, ya que en tiempo ejecución el binding dinámico permite decidir el método de qué clase debe ejecutarse como respuesta a la recepción del mensaje.

Pero si estuviéramos trabajando con un lenguaje procedural sería necesario por un lado definir tres funciones con nombres distintos que realizarán los cálculos para cada tipo de documento, y por otro lado para poder imprimir el listado sería necesaria una sentencia CASE que en función del tipo de documento realizara la llamada adecuada.

Con un lenguaje orientado a objetos :

```
WHILE  haya documentos por imprimir HACER
       .
       .
       .
       PRINT DOC.IMPORTE()
       .
       .
       .
END
```

Con un lenguaje procedural :

```
WHILE  haya documentos por imprimir HACER
       .
       .
       DO CASE
            CASE TIPO_DOC="FA"
                 IMPORTE:=IMP_FAC()
```

```
                CASE TIPO_DOC="NC"
                     IMPORTE:=IMP_NC()

                CASE TIPO_DOC="ND"
                     IMPORTE:=IMP_ND()
           ENDCASE
           PRINT IMPORTE
           .
           .
END
```

La utilización de polimorfismo combinada con binding dinámico permite :

- que grandes sentencias CASE sean innecesarias, ya que cada objeto implícitamente conoce su propio tipo.

- Escribir código más genérico.

- Reducir el esfuerzo de extensión de aplicaciones (por ej. cuando se agregan nuevos tipos de datos). (Si en el ejemplo anterior agregáramos un nuevo tipo de documento, solo es necesario que implemente el mensaje IMPORTE() para que se pueda realizar el listado. Pero si trabajáramos con un lenguaje procedural haría falta agregar una cláusula adicional a la estructura CASE).

3. CICLO DE DESARROLLO EN EL PARADIGMA DE OBJETOS

El enfoque tradicional para el desarrollo de software es un proceso secuencial llamado frecuentemente "modelo en cascada". Este ciclo de vida está compuesto por cuatro etapas : análisis, diseño, codificación y prueba. El proceso se lleva a cabo en secuencia y cada etapa comienza solo cuando la anterior ha sido finalizada. Solo cuando la última etapa haya terminado se podrá instalar el sistema.

Los problemas de aplicar rígidamente este enfoque son bien conocidos :

I. *El modelo de cascada asume una progresión relativamente uniforme de las etapas de elaboración*.

II. *Se asume que el análisis de requerimientos inicial no tiene error*, y una vez que ha sido hecho el resto de las etapas fluirán en forma natural. Pero en el mundo real, entender los requerimientos de un sistema no es tarea fácil y generalmente el análisis inicial tiene que ser revisado y modificado.

III. *La participación del usuario final* es pobre. El sistema se transforma en inteligible para el usuario en la etapa de diseño cuando sus requerimientos se traducen en especificaciones técnicas. El resultado es que el cliente no puede validar el diseño hasta que el producto está completo, entonces se pueden llegar a producir programas que no cumplan completamente con las necesidades del usuario final.

IV. El diseño y la codificación se derivan del análisis inicial, de manera que son específicos para los requerimientos de determinada aplicación. Por lo tanto, *el enfoque tradicional tiende a no reusar código existente*. Esto provoca duplicación de esfuerzo y reducción de productividad.

V. *Los sistemas se construyen de manera que los diferentes componentes (subsistemas, módulos, programas) son muy interdependientes*, como lo son programas y datos. Esto hace difícil las modificaciones y el mantenimiento. Por ejemplo, para agregar una nueva función o cambiar el formato de algún dato hace falta modificar varios programas en diferentes lugares y después, pasar por largas compilaciones y pruebas. Por lo tanto, los cambios en los requerimientos del negocio o las nuevas oportunidades de hardware o software no pueden ser fácilmente explotadas.

VI. *El modelo de cascada no se adapta al desarrollo de tipo evolucionario que se hace posible con la prototipación rápida y con los lenguajes de cuarta generación*.

Si bien, cuando se trabaja con orientación a objetos se puede utilizar el modelo de cascada, es recomendable y hasta más natural recurrir a otros ciclos de desarrollo de tipo iterativo (como el espiral de Boehm) o a la prototipación rápida.

El ciclo de desarrollo orientado a objetos, en contraposición con el modelo secuencial de cascada, enfatiza la idea de que el proceso de desarrollo real consiste de una serie de iteraciones en cada una de las etapas del proceso. En vez de ser un proceso secuencial es un enfoque "circular" en el cual se completará el sistema luego de un cierto número de iteraciones.

El análisis comienza con una porción del sistema que pasa luego por diseño, implementación y prueba, avanzando luego a otra fase de análisis para comenzar una nueva iteración. Al pasar por cada iteración el sistema va evolucionando. Este proceso se detiene cuando el sistema esté lo suficientemente maduro como para cumplir con los requerimientos eficientemente.

Una porción más grande del tiempo es utilizada en el diseño. Esto es porque el software se está diseñando para ser fácilmente reusado, mantenido y modificado. Las herramientas de programación orientadas a objetos no aseguran, por sí mismas, la reusabilidad, mantenibilidad y extensibilidad del software. Ni tampoco son, de hecho, absolutamente necesarias. Sin embargo, las herramientas de programación orientadas a objetos pueden ayudar en un proyecto en el cual los miembros del equipo dediquen su tiempo a explorar el problema y hacer un diseño cuidadoso. Un gran esfuerzo es necesario para diseñar código para reusar, pero es un esfuerzo que paga generosamente al final. El tiempo utilizado en realizar un diseño cuidadoso permite entender el problema más profundamente. La implementación se acelera porque se ha aprendido bastante sobre el problema. Por lo tanto, el tiempo total requerido para un ciclo puede permanecer sin cambios o aún disminuir. Y como el software fue diseñado desde el principio teniendo en mente el mantenimiento, la extensión y la reusabilidad, el tiempo requerido para cualquier esfuerzo subsiguiente decrece radicalmente.

Rumbaugh nos dice que los impactos de un enfoque orientado a objetos son :

I. *Cambio del esfuerzo de desarrollo hacia las etapas de análisis y diseño*. Gran parte del esfuerzo de desarrollo se traslada a las etapas de análisis y diseño. Alguna veces es desconcertante el hecho de gastar más tiempo durante el análisis y el diseño, pero el esfuerzo extra está más que compensado por una implementación más rápida y simple. Como el diseño resultante es más claro y adaptable, los cambios futuros son más sencillos.

II. *Se enfatiza en la estructura de datos por sobre los procesos*. Este cambio de énfasis le da al proceso de desarrollo una base más estable y permite el uso de un solo y unificado concepto de software a través del proceso : el concepto de objeto. Todos los otros conceptos, como funciones, relaciones y eventos, están organizados alrededor de los objetos de manera que la información recolectada durante el análisis no se pierde o transforma cuando el diseño y la implementación se llevan a cabo.

Las estructuras de datos de una aplicación y las relaciones entre ellas son mucho menos vulnerables a los requerimientos cambiantes de lo que lo son la operaciones ejecutadas sobre los datos. El hecho de organizar un sistema alrededor de objetos en vez de procesos le confiere al proceso de desarrollo una estabilidad que está ausente en los enfoques orientados a los procesos.

III. *Proceso de desarrollo suave (sin sobresaltos)*. Como el enfoque de orientación a objetos define un conjunto de objetos orientados al problema muy temprano en el proyecto y continua usando y extendiendo esos objetos a lo largo del ciclo de desarrollo, la transición entre las distintas etapas del ciclo de vida es mucho más suave. El modelo de objetos desarrollado durante el análisis es utilizado para el diseño y la implementación, y el trabajo se centra en refinar el modelo a niveles progresivamente más detallados, en lugar de convertir de una representación a otra (como se hace en las técnicas estructuradas). El proceso es suave porque no hay discontinuidades en las cuales la notación de una etapa es reemplazada por otra notación en una etapa diferente.

IV. *Iterativo en vez de secuencial*. La suavidad del proceso de desarrollo hace más fácil repetir las etapas en niveles de detalle progresivamente más finos. Cada iteración adiciona o clarifica características más que modificar trabajo que haya sido realizado previamente, por lo tanto hay menos oportunidad de introducir inconsistencias y errores.

También, cabe destacar que, el usuario final puede involucrarse en todas las etapas del ciclo de desarrollo. Los objetos en el sistema tienen relación directa con los objetos del problema, lo que permite un mejor entendimiento de los objetos y sus interacciones. Además, la posibilidad de utilizar prototipos desde las primeras iteraciones del proceso de desarrollo permite ir validando el trabajo realizado con la ayuda de lo usuarios. Por lo tanto, los sistemas desarrollados con orientación a objetos tienen mayor probabilidad de cumplir con los requerimientos del cliente.

4. OBJETIVOS DE LA ORIENTACIÓN A OBJETOS

La introducción de la tecnología de objetos en el desarrollo de software responde a los siguientes objetivos:

- ***Favorecer la productividad*** promoviendo la reusabilidad como una meta primordial, proveyendo herramientas más adecuadas para modelizar el mundo real y permitiendo la utilización de ciclos de desarrollo más flexibles.

- ***Incrementar la calidad*** : la utilización de componentes reusables produce sistemas más robustos, debido a que cada componente al haber sido usada varias veces ha sido sometida a muchas pruebas y por lo tanto, tiene una confiabilidad mayor. Por otra parte, al ser posible corregir errores cometidos en etapas iniciales con relativa facilidad, se logra que los sistemas orientados a objetos sean más resistentes al cambio, un aspecto de la calidad que será preponderante en la computación del futuro.

- ***Mejorar el mantenimiento*** : debido a que los requerimientos de un sistema se encuentran en estado de cambio permanente, la clave para lograr un mejor mantenimiento se encuentra en desarrollarlo de manera que se le puedan incorporar suavemente los cambios que no puedan ser anticipados. La manera de lograrlo es separando las partes del sistema que son intrínsecamente volátiles de las que son más estables. Las técnicas estructuradas no logran este objetivo y por lo tanto, necesitan imponer un congelamiento de los requerimientos del sistema. El paradigma de objetos considera en cambio al sistema en su evolución en el tiempo y provee herramientas para soportar un ciclo de desarrollo evolutivo.

5. VENTAJAS DE LA ORIENTACIÓN A OBJETOS

- Mayor modularidad al encapsularse funcionalidad en objetos.

- Mayor extensibilidad debido a la modularidad y a la utilización de la herencia combinada con polimorfismo.

- Desarrollo de software más reusable.

- Simplicidad de diseño.

- Software modificable con mayor facilidad.

- Mejor comprensión del mundo de las aplicaciones.

- Utilización de ciclos de desarrollo más flexibles que el tradicional de cascada.

- Incrementa la consistencia entre análisis, diseño y programación, debido a que se eliminan los "saltos" entre estas etapas.

- Continuidad en la representación desde el análisis, pasando por el diseño hasta la programación : el modelo que resulta del análisis se expande en el diseño y luego en la programación, pero en ningún momento hace falta cambiar el tipo de modelo (como la conversión del DFD a los diagramas de estructura, en las técnicas estructuradas).

6. PROBLEMAS CON LA ORIENTACIÓN A OBJETOS

- Falta de consenso respecto de metodologías de desarrollo.

- Necesidad de capacitar y producir un cambio cultural en el equipo de sistemas : nuevas y diferentes metodologías de desarrollo, herramientas de programación, roles que cumplir, etc.

- Necesidad de establecer una metodología para administrar la reusabilidad.

7. BIBLIOGRAFÍA UTILIZADA

- [AMANDI] Amandi, Analía, Alvarez, Luis, y Prieto, Máximo (LIFIA, UNLP) : **REUSO EN EL PARADIGMA DE OBJETOS**, (material de apoyo del tutorial dictado en el marco del Foro II de la Informática) Rosario, Octubre 1992.

- [BOOCH] Booch, G. : **Object Oriented Design with Aplications**, Benjamin-Cummings, 1991.

- [COAD] Coad, P. y E. Yourdun : **Object-Oriented Analysis**, Prentice Hall, 1990.

- [COAD] Coad, P. y E. Yourdun : **Object-Oriented Design**, Prentice Hall, 1991.

- [MEYER] Meyer, B. : **Object-Oriented software construction**, Prentice Hall, 1991.

- [PRIETO] Prieto, Máximo (LIFIA, UNLP) : **Técnicas estructuradas vs. Técnicas orientadas a objetos**, (apunte de apoyo de curso dictado en la Semana de la Productividad, organizado por la revista Compumagazine) Buenos Aires, Agosto 1993.

- [PRIETO 92] Prieto, Máximo, Amandi, Analía, Alvarez, Luis y Leonardi, M. Carmen (LIFIA, UNLP) : **ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS**, (material de apoyo del tutorial dictado en el marco del Foro II de la Informática) Rosario, Octubre 1992.

- [ROSSI] Rossi, G. y Amandi, A. : **Tecnología de orientación a objeto**. En : Compumagazine, Magazine Publishing srl., Buenos Aires, nros. 58,59 y 61, 1993.

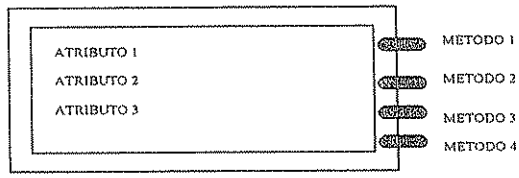
- [RUMBAUGH] Rumbaugh, J. y otros : **Object Oriented Modeling and Design**, Prentice Hall, 1991.

- [WIRFS] Wirfs-Brock, R. y otros : **Designing Object-Oriented software**, Prentice Hall, 1990.

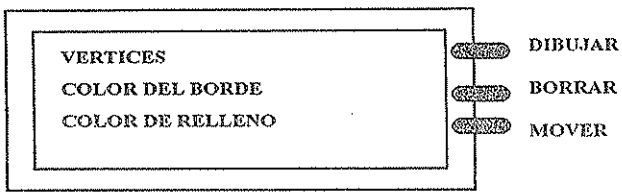
8. EJEMPLOS

8. I EJEMPLO 1-1 : *OBJETOS*

REFERENCIAS

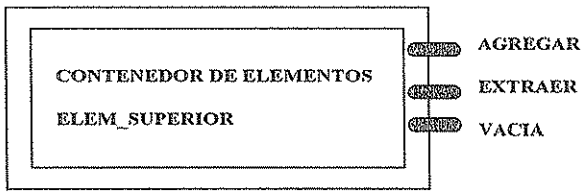


POLÍGONO



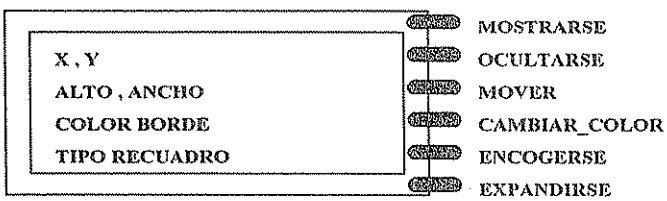
El objeto polígono tiene como responsabilidades : **Dibujar, Mover y Borrar.** Para ello necesita como estado interno, conocer : sus **Vértices, Color del Borde y Color de Relleno.**

PILA (Tipo abstracto de datos)

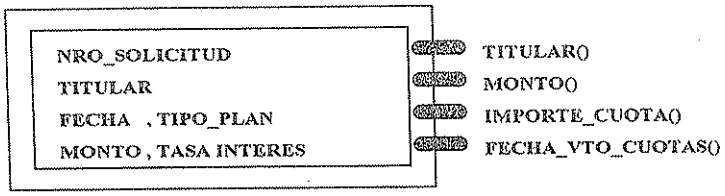


Un objeto podría ser una estructura de datos común (tipo abstracto de datos), como lo es una Pila. El comportamiento que tendrá consta de **Agregar** o **Extraer** un elemento, e informar si la pila está **Vacía.**

VENTANA

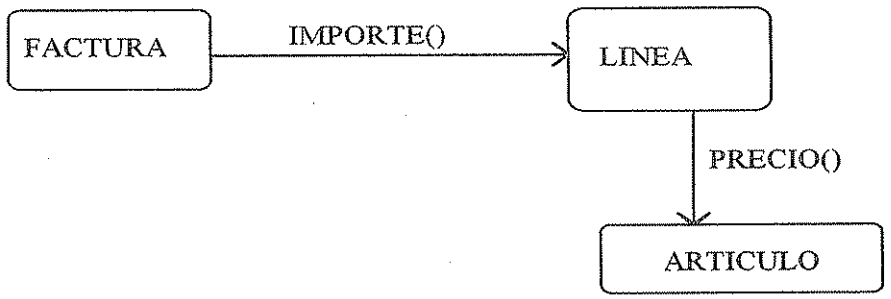


CRÉDITO



8. II EJEMPLO 2-1 : *MENSAJES*

El objeto **Menú** le envía el mensaje **Mostrarse()** a **Ventana**, cuando el usuario ha elegido una opción que hace necesario mostrar la ventana.



Un objeto **Factura** está formado por **Líneas** , cada una de las cuales contiene un **Artículo** y una cantidad.

Para que la **Factura** pueda calcular su importe, le envía el mensaje **Importe()** a cada una de sus **Líneas**. Cada **Línea** al recibir el mensaje **Importe()** ejecuta el *método* correspondiente, enviando el mensaje **Precio()** a su **Artículo** para multiplicarlo por la cantidad y así devolverle a **Factura** el importe correcto.

8. III EJEMPLO 3-1 : *INTERFACE E IMPLEMENTACIÓN*

En el ejemplo 1-1 se presenta el objeto VENTANA que representa una ventana de la pantalla. La interface externa de este objeto está constituida por los mensajes que el objeto puede recibir del exterior :

◆ MOSTRARSE : produce que la ventana se haga visible en la pantalla.

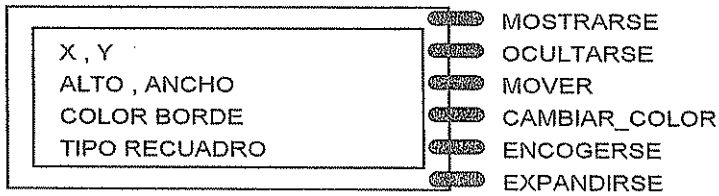
◆ OCULTARSE : la ventana se hace invisible.

◆ MOVER : la ventana se cambia de ubicación dentro de la pantalla.

◆ CAMBIAR COLOR : cambia el color de la ventana.

◆ ENCOGERSE : reduce a la mitad las dimensiones de la ventana.

◆EXPANDIRSE : aumenta al doble las dimensiones de la ventana.



Para que el objeto pueda responder a estos mensajes necesita tener una implementación constituida por un estado interno y de algunos métodos. Habrá una porción de código que conformará el método para cada mensaje, y el estado interno podría ser :

◆ X , Y : dos enteros que representan las coordenadas del extremo superior izquierdo de la ventana.

◆ ALTO , ANCHO : enteros que representan el alto y el ancho de la ventana.

◆ COLOR BORDE : especificación del color del borde.

◆ TIPO RECUADRO : especificación del tipo de recuadro.

Debido al encapsulamiento el estado interno del objeto VENTANA es inaccesible para cualquier otro objeto, esto implica que la única forma de cambiar el tamaño de la ventana sea enviándole el mensaje EXPANDIRSE o el mensaje ENCOGERSE. Suponiendo que el objeto VENTANA estuviera direccionado por la variable unaVentana, el código correspondiente sería :

```
unaVentana.EXPANDIRSE()

unaVentana.ENCOGERSE()
```

Pero no sería posible asignar un valor directamente a ANCHO o a ALTO **desde afuera del objeto unaVentana.** Es decir que un código como el siguiente **generaría un error** :

```
ALTO:=ALTO / 2

unaVentana.ALTO:= 4
```

8. IV EJEMPLO 3-2 : *ENCAPSULAMIENTO Y CAMBIOS EN LA IMPLEMENTACIÓN*

Supongamos que, para aprovechar más velocidad en la graficación, decidiéramos hacer un cambio en el estado interno del objeto VENTANA, de manera que guardará las coordenadas del extremo inferior izquierdo en vez del largo y el ancho. Entonces, el estado interno sería : X , Y , X2 , Y2 , COLOR BORDE , TIPO RECUADRO.

Este cambio tendría claramente un impacto sobre varios métodos que constituyen la implementación del objeto, como por ejemplo los métodos que responden a los mensajes MOSTRARSE y ENCOGERSE. Probablemente, la implementación de mostrarse se simplificaría porque no haría falta calcular la otra coordenada, y la implementación de encogerse seguramente se complicaría por el hecho de tener que calcular el largo y el ancho de la ventana.

Una vez realizadas las modificaciones pertinentes en la implementación del objeto, el resto del programa quedaría inalterado, ya que por ejemplo la forma de enviar los mensajes MOSTRARSE y ENCOGERSE no sufriría ningún cambio :

```
unaVentana.MOSTRARSE()

unaVentana.ENCOGERSE()
```

Esta propiedad del software orientado a objetos, que permite a través del encapsulamiento construir programas más resistentes al cambio constituye una de las ventajas fundamentales de está tecnología.

Cabe aclarar que en este caso se pudo circunscribir las modificaciones al ámbito de la implementación del objeto, debido a que los cambios introducidos no afectaban la interface pública del objeto. Si hubiéramos reemplazado por ejemplo los mensajes EXPANDIRSE y ENCOGERSE por un único mensaje REDIMENSIONAR(factor) (si factor > 1 corresponde expansión, si es menor que 1 se encoge la ventana), habría sido necesario realizar modificaciones en todos los objetos que enviarán los mensajes EXPANDIRSE o ENCOGERSE, cambiando :

```
unaVentana.EXPANDIRSE()
```

por

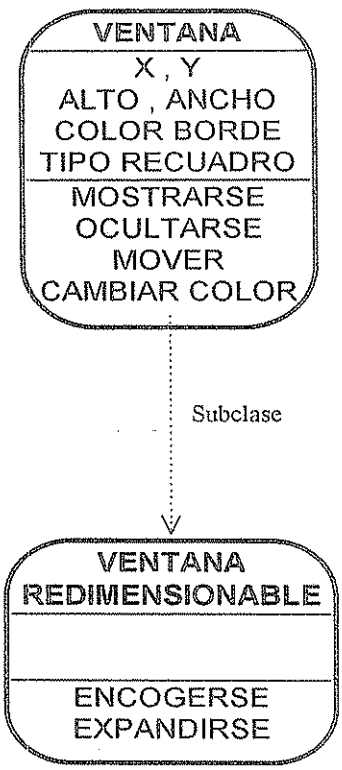
```
unaVentana.REDIMENSIONAR(2)

unaVentana.ENCOGERSE()
```

por

```
unaVentana.REDIMENSIONAR(0.5)
```

8. V EJEMPLO 5-1 : *HERENCIA*

Retornando al ejemplo de la **ventana**, se puede suponer que en un sistema será necesario contar tanto con ventanas de dimensiones fijas como de dimensiones variables. Por lo tanto, se deben definir dos clases distinta: ***Ventana*** y ***Ventana Redimensionable***.



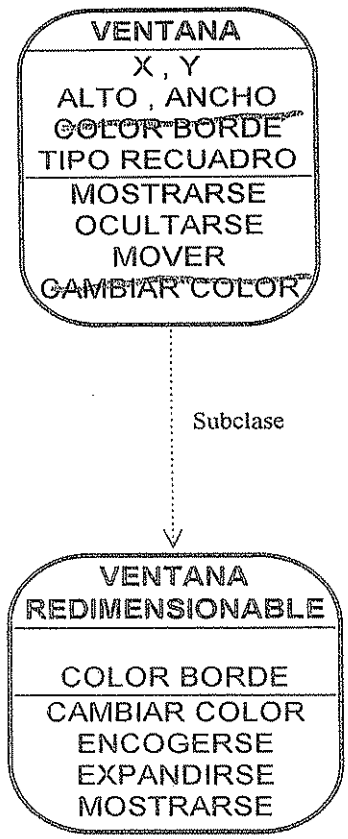
Todo objeto de la clase ***Ventana*** tendrá dimensiones determinadas en el momento de su creación y podrá **Mostrarse, Ocultarse, Moverse** y **Cambiar de Color.**

Todo objeto de la clase ***Ventana Redimensionable*** tendrá como estado interno heredado de ***Ventana*** : **X , Y , Alto , Ancho , Color Borde** y **Tipo Recuadro;** además tendrá como comportamiento heredado : **Mostrarse, Ocultarse, Moverse** y **Cambiar de Color.** Agregará como comportamiento propio : **Encogerse** y **Expandirse.** Es decir que, ***Ventana redimensionable*** además de tener los mismo atributos y poder hacer las mismas cosas que ***Ventana,*** puede también **Encogerse** y **Expandirse.**

Ventana es superclase de [illegible], y esta última es subclase de la primera.

8. VI EJEMPLO 5-2 : *HERENCIA Y REDEFINICIÓN*

El ejemplo anterior se podría modificar de manera que la ***Ventana*** de dimensiones fijas tenga siempre el color estandar como color de borde, y que la ***Ventana Redimensionable*** se diferencie de su superclase por poder cambiar su color de borde, además de sus dimensiones.

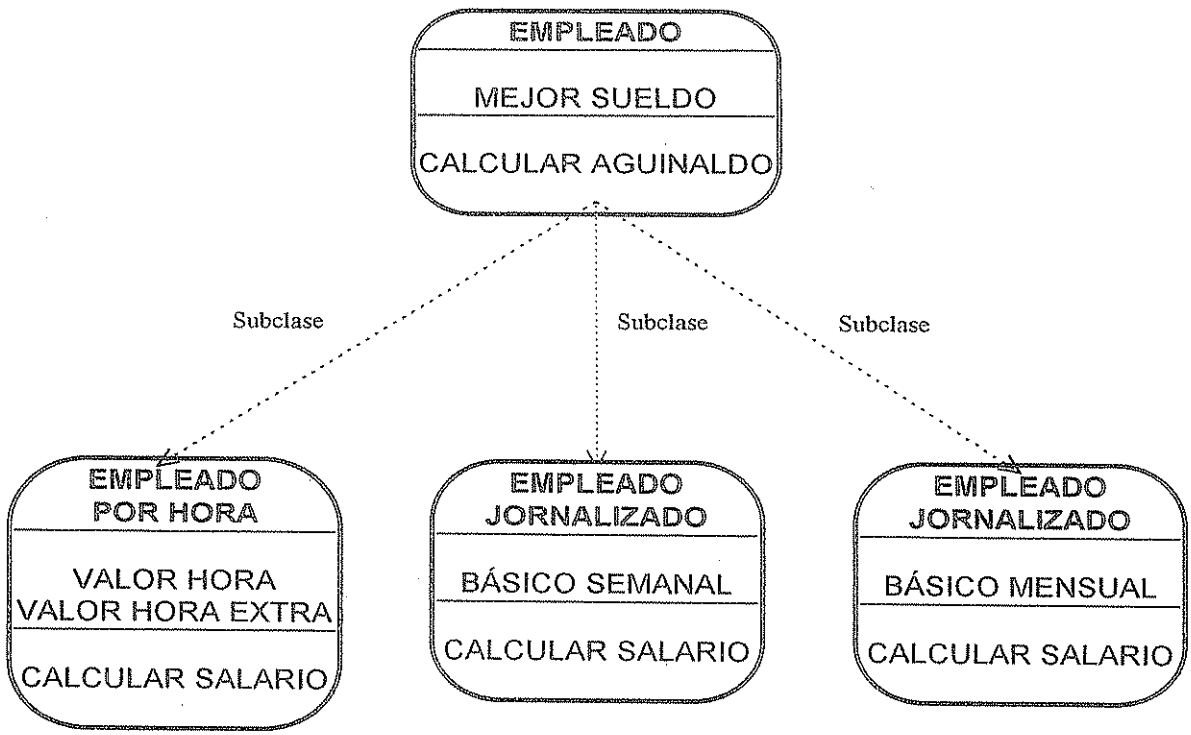


Todo objeto de la clase ***Ventana*** tendrá dimensiones determinadas en el momento de su creación y podrá **Mostrarse, Ocultarse, Moverse.** Además, el color del borde será el estandar, y no podrá modificarse.

Todo objeto de la clase ***Ventana Redimensionable*** agregará al estado interno heredado : **Color Borde** ; y agregará al comportamiento heredado : **Cambiar Color, Encogerse** y **Expandirse.** Y por último deberá **REDEFINIR** el mensaje **MOSTRARSE,** debido a que el método heredado de su superclase no utiliza el atributo color borde para graficar la ventana.

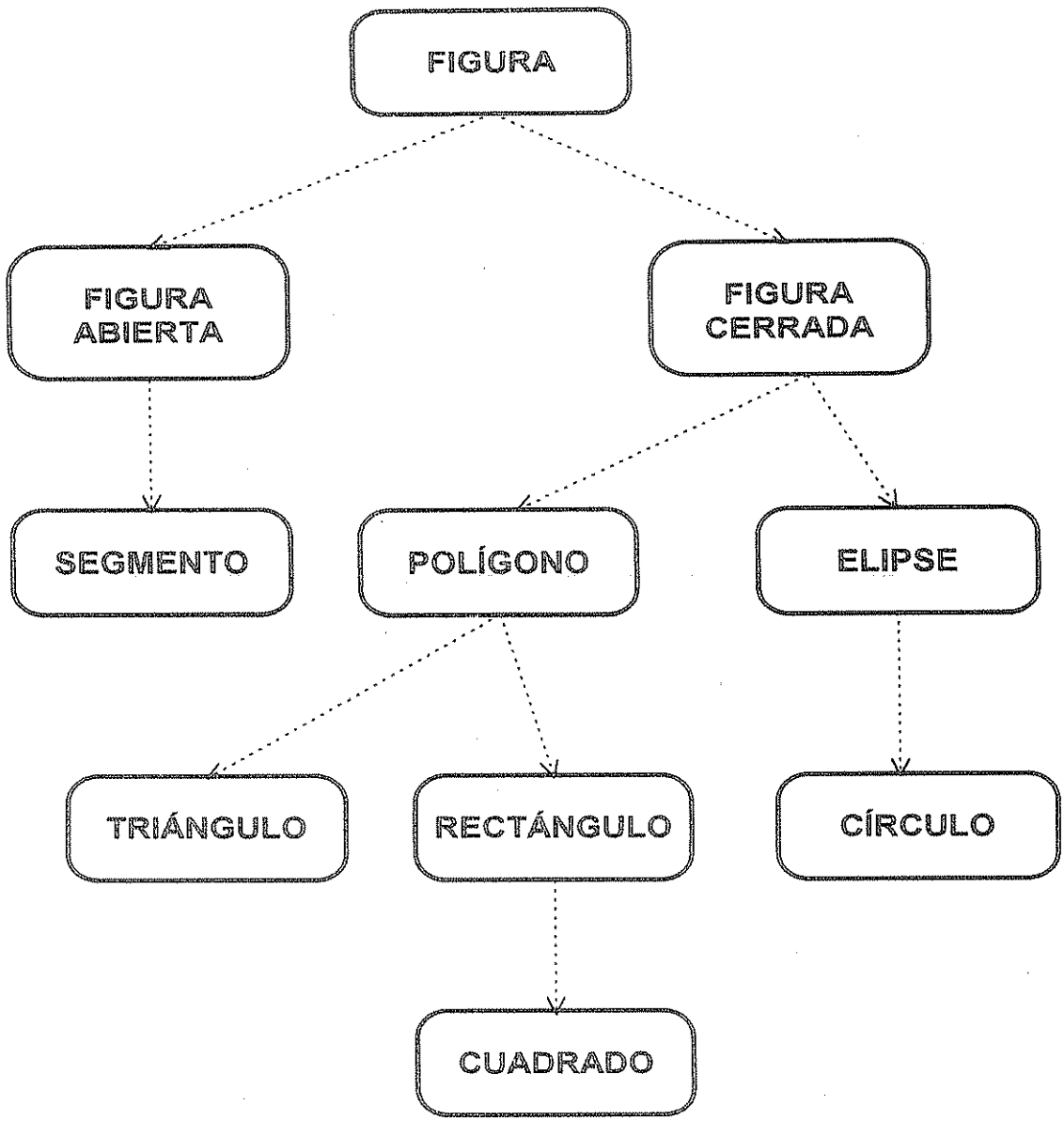
8. VII EJEMPLO 5-3 : *CLASES ABSTRACTAS* (RUMBAUGH)

En un sistema de liquidación de sueldos se podría tener una jerarquía como la siguiente :

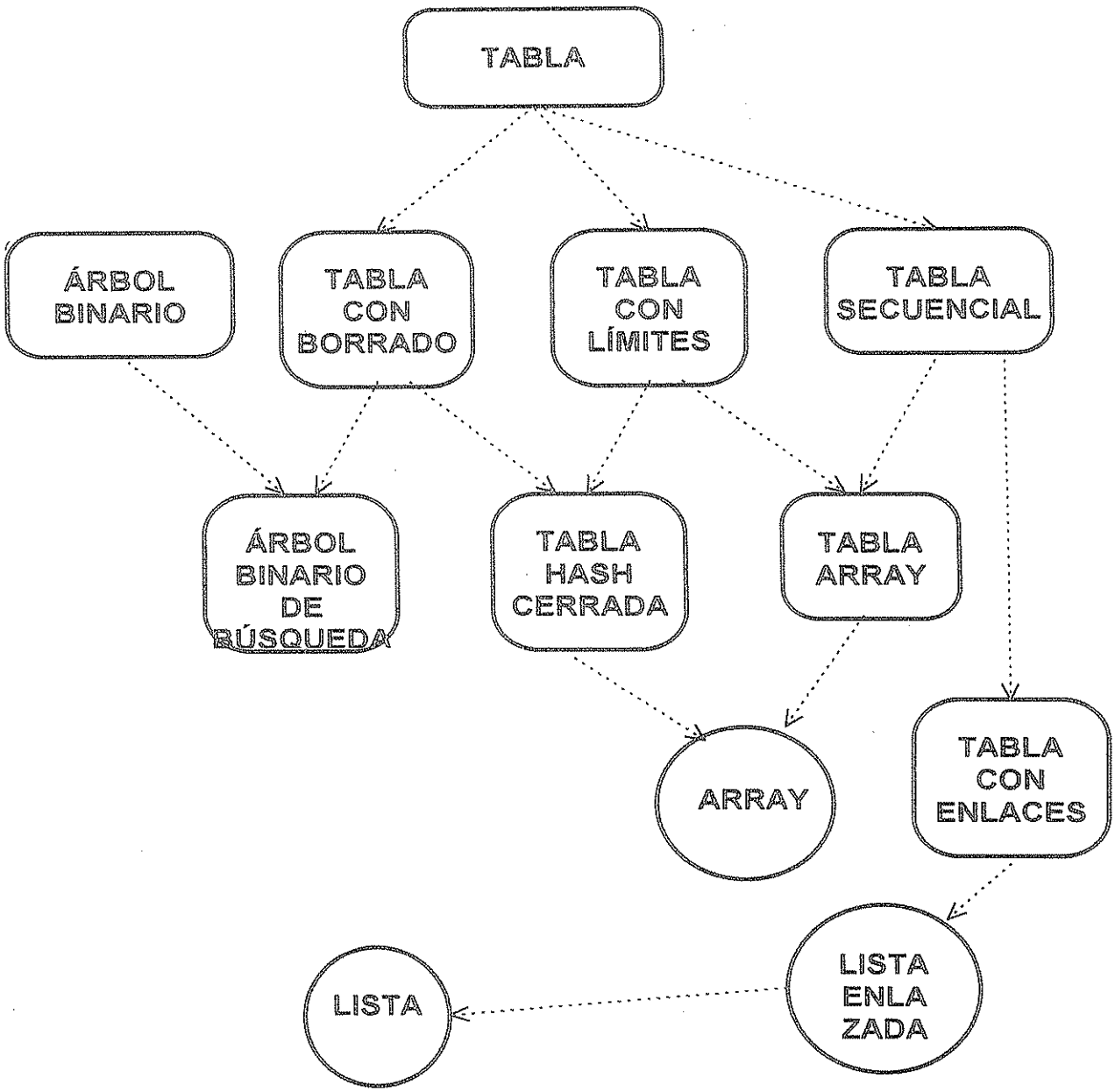


La clase ***Empleado*** es una clase abstracta, ya que todo empleado será un ***Empleado por hora, Jornalizado o Mensualizado.***

8. VIII EJEMPLO 5-4 : JERARQUÍA DE CLASES



8. IX EJEMPLO 5-5 : *RED DE CLASES CON HERENCIA MÚLTIPLE* (MEYER)



9. APÉNDICE

En el artículo siguiente se comenta un caso real donde la aplicación de la Tecnología de Orientación a Objetos a un proyecto concreto permitió a la empresa KODAK obtener una ventaja competitiva.

PROYECTO PHOTO CD de KODAK

La tecnología de objetos acelera el tiempo de lanzamiento al mercado.[2]

Joe De Luca representa al no especialista que se transforma en el sponsor de la tecnología de objetos para un proyecto comercial crítico. En 1989, De Luca fue nombrado gerente de producto de software para el estratégico proyecto de desarrollo PhotoCD de Kodak, responsable del desarrollo de software, control de calidad, integración de sistemas y planificación. Sorprendentemente, éste fue su primer cargo en software en toda su carrera.

"Tengo una formación técnica," explica De Luca, " pero en ingeniería mecánica, no en software o computadoras. Me he especializada en planificación de negocios, y trabajé con el tipo de concepto de manejo de equipo que la compañía concibió para PhotoCD"

El sistema PhotoCD permite a los aficionados almacenar imágenes en CD-ROM además de hacer impresiones convencionales y diapositivas. El sistemas de software maneja una variedad de hardware: scanners de alta velocidad, grabadores de CD, impresoras térmicas y de otro tipo, workstations, y tableros de compresión que "escanean" más de 400 imágenes por hora. Las workstations para PhotoCD corren aplicaciones para "escaneo", escritura y copiado de imágenes. El grupo de desarrollo está también construyendo aplicaciones de multimedia y para procesamiento de imágenes en puestos de escritorio.

"Sabíamos que enfrentábamos mucho riesgo con PhotoCD", recuerda De Luca. "Teníamos una planificación de tiempos agresiva y un montón de nuevas tecnologías : los scanners, los grabadores de CD, el formato de archivos, y los estándares de performance eran todos nuevos. Teníamos que decidir cómo manejar aquel riesgo". Como ésta era una oferta de producto importante y nueva, y a su vez una fuente significativa de ganancias, el tiempo para lanzar al mercado era particularmente crítico.

La dirección de Kodak enfatizó que el proyecto debería usar cualquier herramienta y tecnología que fueran necesarias para lograr los objetivos. La tecnología de objetos era uno de los posibles enfoques de desarrollo. De Luca escuchó al grupo principal de ingenieros recomendando un enfoque orientado a objetos, y decidió que ellos podían realizarlo.

---

2 Traducido de OBJECT MAGAZINE, SIGS Publications INC, New York, nro. 3 pag. 72 , 1993

"Una clave para la decisión de utilizar objetos fue nuestra necesidad de una familia de productos comprendiendo varias aplicaciones con funcionalidad solapada. Los objetos nos permitían reusar funcionalidad común para diferentes miembros de la familia. Con objetos, uno puede 'definir una caja' alrededor de un trozo de software o funcionalidad y manejarla separadamente definiendo claramente sus interfaces. El proceso de administración se hace mas fácil cuando uno puede invertir en un sistema particular o en un elemento de software y manejarlo como una entidad independiente."

De Luca remarca que Kodak trató de realizar las inversiones correctas en capacitación, "Pero no había muchos cursos u otro tipo de fuentes de información, particularmente para el análisis y el diseño, que sabíamos era la etapa crítica del proyecto". Afortunadamente, el ingeniero senior Tim Nichols había estado trabajando en análisis y diseño y su trabajo fue tomado como un curso oficial.

De Luca intencionalmente no uso sus expertos originales en objetos para los pocos primeros proyectos. El quería estimar cuán bien sus programadores de objetos recientemente capacitados habían hecho el cambio de paradigma. De esta manera, le era posible medir la curva de aprendizaje realista para un desarrollador típico.

"Vimos en la tecnología de objetos una oportunidad única para barrer con nuestros viejos y malos hábitos de desarrollo. Nuestro análisis nos ayudaba a identificar funciones similares y ver cómo los sistemas se relacionarían con el flujo de trabajo. Pudimos definir interfaces claras y repartir nuestro trabajo entre aplicaciones de software diferentes. Era fácil desarrollar elementos de software específicos en paralelo y luego integrarlos suavemente. Nuestros desarrolladores rápidamente se sintieron más cómodos con el reuso de elementos de programas."

El equipo de PhotoCD ha construido un número de módulos independientes basados en C++ y los ha incluido dentro de diferentes productos. De Luca destaca que reuso significa mayor calidad, ya que estos módulos ya están probados antes de ser usados. "Vemos a estos elementos de programa probados como nuestros kits de herramientas de software, y tenemos confianza en nuestra habilidad para integrarlos exitosamente dentro de nuevas aplicaciones."

"El enfoque de objetos ha sido definitivamente justificado para nuestros proyectos," observa De Luca. "Es más barato eliminar defectos en el análisis o en el diseño que después de haber pasado por la codificación." Otro beneficio que descubrió De Luca fue que pequeños elementos de software de objetos permiten una prueba más temprana, reduciendo el riesgo de fracaso. Antigüamente, cuando los grandes sistemas eran integrados recién al final del proyecto, el factor de riesgo era mucho mayor".

"Usted sabe, unos pocos años atrás no podíamos obtener mucho reuso de nuestros tableros de cables a medida (custom hard-wired boards). En estos días, con circuitos integrados, hay más reuso, más uso de elementos estandar en el hardware que pueden ser intercambiados. Ahora los objetos están haciendo lo mismo para el software. Sí, la tecnología de objetos es evolucionaria, pero es un paso mayor hacia adelante. Ahora tenemos una cultura completa en el grupo de PhotoCD que piensa de la misma manera : los comerciantes y los desarrolladores."

Diagrama de Interacción para la operación de extracción

Este diagrama muestra el orden en que se van ejecutando los métodos y a qué clases pertenecen:

1. Arranca el sistema y *CAJERO* recibe el mensaje *MENU_PCIAL*
   El usuario elige opción Extraer. El menú le envía el mensaje *EXTRAER* al *CAJERO*.
2. Ingresar el número de cuenta.
3. Buscar la cuenta.
   Cuenta_De_Cajero busca secuencialmente pidiendo a cada cuenta su nro.
4. Obtener puntero a la cuenta.
5. Ingresar monto.
6. Registrar la extracción.