17
2ej

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

DISERTACION SOBRE LA PROGRAMACION ORIENTADA A OBJETOS Y SU APLICACION EN LA CONSTRUCCION DE UN SISTEMA DE AUTORIZACIONES DE TARJETAS DE CREDITO FUERA DE LINEA

T E S I S P R O F E S I O N A L

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACION

P R E S E N T A

JUAN CARLOS CARDENAS FIGUEROA

ASESOR DE TESIS: ING. CRISTOBAL PEÑA OLIVO

MEXICO, D. F.

MARZO DE 1993

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Prólogo

La Orientación a Objetos (O-O) ya está en la industria comercial

Después de una larga espera en el campo de la investigación y el académico, la Orientación a Objetos (O-O) surge a la industria comercial del software, pasando de algunas pocas aplicaciones pioneras, a formar parte de los sistemas centrales de grandes corporaciones. Algunos ejemplos de esto (en E.U.A.) son [1]:

+ IBM ha adoptado un lenguaje O-O para el desarrollo de las aplicaciones de Presentation Manager.
+ General Motors tiene sistemas en producción hechos con O-O.
+ Texaco esta reconstruyendo la mayoría de sus sistemas utilizando la O-O.
+ Boeing, General Dynamics y otros contratistas del Gobierno de los E.U.A. se ayudan de la O-O para construir los aerotransportes del futuro.
+ Texas Instruments y el Departamento de la Defensa de los E.U.A. utilizan la O-O en sistemas CAD/CAE para el diseño y construcción de C.I.

Y está tomando cada vez mayor fuerza

Una parte, cada vez mayor, de las empresas de E.U.A. están considerando utilizar, o están utilizando, la O-O tanto para sus sistemas secundarios como centrales. Pero... ¿porqué está tomando tal importancia la O-O?. La respuesta es sencilla: ¡ porqué funciona ¡.

Existen dos grandes problemas para la adopción de la O-O

En México, la influencia de la O-O se deja notar en algunas (pocas) empresas de cómputo. Sin embargo, no

[1] Datos del libro "Object Oriented Information Systems", Taylor

ha tenido la penetración deseada, debido a varios factores entre los que destacan:

i) La falta de herramientas adecuadas, como lenguajes de programación, bases de datos, metodologías de análisis y diseño, ambientes integrados de desarrollo, etc... y,

ii) Falta de adecuada capacitación y asesoría en estos temas hacia los ingenieros, programadores y gerentes de departamentos de sistemas de cómputo.

Las herramientas de software evolucionan rápidamente

Las herramientas de software están evolucionando de una manera sorprendente. En la actualidad ya se cuenta con las herramientas mínimas necesarias para el adecuado desarrollo de sistemas O-O, para la mayoría de plataformas como son: DOS, Windows, OS/2, UNIX y sistemas POSIX. Inclusive se tienen proyectos para liberar versiones de FORTRAN y COBOL orientados a objetos. Entre los lenguajes de programación más populares se encuentran C++, SmallTalk, Pascal O-O y CLOS.

En México debemos impulsar la capacitación al personal de la industria y a los alumnos de las universidades

En México, el renglón de la capacitación y asesoría al personal de los departamentos de cómputo, y aún de las instituciones de enseñanza, tiene mucho por avanzar. Recientemente la carrera de Ingeniero en Computación, en la Facultad de Ingeniería de la UNAM, ofrece una introducción a la O-O en algunas de sus asignaturas. Sin embargo, considero de gran importancia ofrecer una mayor especialización en la O-O a los alumnos. Esta especialización debería abarcar aspectos de análisis, diseño, bases de datos y lenguajes, todos O-O.

Propósito del Texto

El propósito principal de este trabajo de tesis es exponer los conceptos básicos que forman la base de la Orientación a Objetos. Los objetivos alternos son: i)

aplicar estos conceptos en la construcción del sistema propuesto y ii) presentar conclusiones, tanto del sistema como de la O-O.

Acotamiento

Con fines de mantener el presente trabajo conciso, simple y ágil, se decidió sólo incluir los aspectos básicos de la O-O, de manera que no contempla aspectos más avanzados como son: Análisis, Diseño, Bases de Datos o algún lenguaje de programación en específico. De tal forma que, el presente es un modesto intento de explicar algo tan extenso y complejo como la O-O.

Público

Aunque la O-O tiene una base teórica formal muy firme, este libro tiene un enfoque intuitivo y práctico, que intenta emplear un lenguaje sencillo de manera que pueda servir como texto introductorio. Este texto está dirigido tanto a nivel gerencial como a nivel técnico.

Como leer este libro

La estructura general del texto presenta a la izquierda de la página una serie de oraciones que representan un resumen del párrafo al que se asocia. Si así lo desea, puede hacer una lectura rápida de los capítulos que no le sean de gran interés.

El capítulo 1 es fundamental

El capítulo 1, Resumen Ejecutivo de la Orientación a Objetos, está dirigido a todos los lectores, ya que representa la información mínima sobre la O-O y forma la base general de este libro.

El capítulo 2 presenta los orígenes de la O-O

El capítulo 2, Antecedentes, contiene la justificación de la evolución del concepto de la O-O. Si usted no está muy interesado en estos aspectos, puede dejar este capítulo para leerlo después.

El capítulo 3 ve al detalle los conceptos

El capítulo 3, Conceptos de la Programación Orientada a Objetos, trata en detalle lo visto en el capítulo 1, adicionando otros conceptos también. Si sólo le interesa una introducción básica, con el capítulo 1 será suficiente.

El capítulo 4, Implantación del Sistema Propuesto, introduce el esquema general de operación de los sistemas de autorización de tarjetas de crédito, los dispositivos de punto de venta y el análisis y diseño general del sistema de autorizaciones. Este capítulo es más técnico y pára leerlo debe tener un conocimiento general de la O-O.

El capítulo 4 es la implantación del sistema

El Capítulo 5, conclusiones, presenta algunas reflexiones en cuanto al sistema que se desarrolló y sus posibles aplicaciones alternativas. Este capítulo presenta también las ventajas, desventajas y futuro de la O-O.

El capítulo 5 tiene las conclusiones del sistema y de la O-O

Juan Carlos Cárdenas Figueroa.
febrero de 1993.

Los hombres aprenden mientras enseñan.

- Séneca.

Indice

Capítulo 5:
Conclusiones

Capítulo 1:

Resumen Ejecutivo de la Orientación a Objetos

Vive como si esperaras vivir hasta los cien años, pero estuvieras listo a morir mañana.

- Ann Lee.

Resumen Ejecutivo de la Orientación a Objetos

Aunque el término programación orientada a objetos es el más conocido, la orientación a objetos es mucho más que solamente un lenguaje de programación. La orientación a objetos es toda una metodología, una filosofía, un nuevo **paradigma**. El paradigma de la orientación a objetos (**POO**) consta de su propia **arquitectura** y de metodologías de **análisis y diseño**. Aunque no se tiene un estándar para el análisis y diseño, generalmente se aceptan, hasta hoy día, los trabajos de Yourdon & Coad [1] para el análisis y de Booch [2] para el diseño.

Paradigma Orientado a Objetos (POO)

Dado que el objetivo principal de este trabajo es presentar los componentes esenciales del POO, no se discute al detalle las metodologías de análisis y diseño; sin embargo, sí se utilizaron para el desarrollo del sistema propuesto. Se introduce el análisis y diseño general en el capítulo 4: Implantación del Sistema. La persona que desee más información sobre el tema puede consultar la bibliografía al final del libro.

El objetivo principal de este trabajo es presentar los conceptos del POO

La necesidad de evolucionar a la orientación a objetos se debió a la llamada "crisis del software" que estamos viviendo hoy en día: los sistemas se entregan tarde y arriba del presupuesto original. Trataremos este tema con mayor detalle en el siguiente capítulo.

En el POO se manejan varios conceptos como: clase, objeto, mensaje, abstracción, encapsulado, polimorfismo, herencia, etc... Sin embargo resulta confuso hablar de

Elementos del POO.

[1] Object Oriented Analysis Coad & Yourdon
[2] Object Oriented Design Booch Grady

ellos separadamente, por lo que decidimos relacionarlos entre sí a partir de tres partes medulares de la siguiente manera:

1) Objetos

   Abstracción
   Encapsulado

2) Mensajes

   Reutilización de nombres (name overloading)
   Polimorfismo

3) Clases

   Modularidad
   Objetos compuestos (composite objects)
   Herencia

Relación entre los conceptos del POO

El cuadro anterior representa la estructura que escogimos para presentar los conceptos del POO. Hemos visto que esta forma de relacionar los conceptos del POO resulta más sencilla de entender. Generalmente cuando se presentan estos términos sin una relación evidente entre ellos (como en el trabajo de Booch[3]), resulta más difícil encontrar la relación entre ellos, especialmente para los nuevos lectores del POO.

Esta estructura parece ser la más sencilla de entender

Recordemos que el presente capítulo es un resumen ejecutivo de la programación orientada a objetos, por lo que sólo veremos algunos de ellos (los más ilustrativos sin perdernos en el detalle). Así pues, principalmente veremos los tres niveles superiores: **objetos, mensajes y clases**. Para ver el detalle de cada uno de los subniveles refiérase al capítulo 3.

Este capítulo explica, principalmente, a los tres primeros elementos: objetos, mensajes y clases

[3] OOD Booch

Objetos. Se puede pensar en los sistemas orientados a objetos como una colección de dispositivos virtuales independientes que se comunican entre sí, cada uno de ellos con su propia estructura interna. A estos dispositivos virtuales se les llama ***objetos.***

Definición de objeto

Un **objeto** es la abstracción de una entidad tangible del mundo real o de un concepto teórico. Un objeto tiene estado, comportamiento e identidad.[4] En este sentido, un objeto puede ser: un avión, una computadora, una lista ligada, un automóvil, una puerta, etc...

Un objeto tiene estado y comportamiento

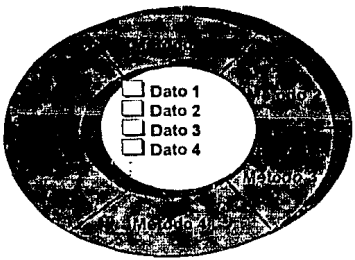
Un objeto se caracteriza por tener **estado** (peso, posición, altura) y **comportamiento** (aumentar velocidad, moverse de lugar, cargar combustible). El modificar el comportamiento provoca ya sea la ejecución de alguna acción, o el cambio en el estado del objeto. Un objeto tiene identidad porque es único y diferente a cualquier otro.

El estado se modela a través de sus datos internos y su comportamiento se modela con sus métodos

El estado de un objeto se modela a través de un conjunto de valores variables, que representan la abstracción del objeto y sus características en el tiempo. Aunque existen muchos términos para llamar esta representación -atributos, propiedades, variables miembro, etc ...-, en este trabajo les llamaremos **datos internos** o simplemente **datos**. Por otra parte, el comportamiento se modela a partir de un conjunto de funciones que llevan a cabo ciertas acciones o que modifican el estado del objeto. De nuevo, existen varias formas de llamarle a estas funciones -funciones miembro, operaciones, etc..-, aquí les llamaremos **métodos**. Los datos y métodos se manejan como un todo inseparable.

[4] OOD Booch pp. 76,77

Los datos internos de un objeto son inaccesibles a cualquier otro objeto o programa. Este es uno de los conceptos principales en el POO: el **encapsulado**.

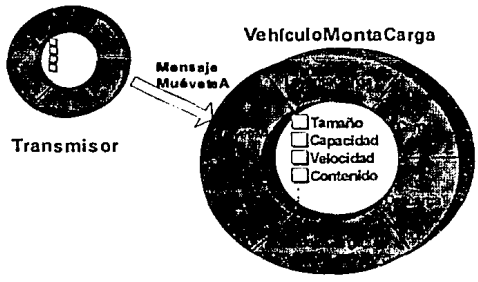


Los objetos se componen de datos y métodos como un todo inseparable

Como se puede apreciar en el diagrama anterior, los métodos del objeto protegen a sus datos de ser alterados por cualquier elemento externo. La única forma de modificar el estado del objeto es a través de un método, manteniendo así la integridad de los datos. El mantenimiento se simplifica al aislar los problemas en partes bien definidas y el uso del objeto se vuelve más sencillo, porque se ocultan los detalles internos de implantación.

Concepto de Mensaje. Los objetos se comunican entre sí a través de **mensajes**. Los mensajes modifican el comportamiento del objeto receptor, provocando que haga cierta acción o que cambie su estado. Por ejemplo, crear un nuevo objeto, modificar o preguntar por el valor de un dato, inicializar a un puerto serie, etc ...

Los objetos se comunican con mensajes



La única forma de modificar, dinámicamente, el comportamiento de un objeto es a través de un mensaje [5]

El concepto de **Polimorfismo** es muy importante en POO, se refiere simplemente a que cada objeto reacciona de manera muy particular a un mensaje que se le envía.

Por ejemplo: todos los animales comen, pero lo hacen de manera diferente; todas las figuras geométricas se pueden dibujar, pero se dibujan, también, de manera diferente. Mejor aún, si pongo otra figura geométrica u otro animal, él sabrá como comer o como dibujarse a sí mismo, yo sólo le envío el mismo mensaje: comer o dibujar. De esta manera el flujo normal del programa no cambia, volviéndose mucho más sencillo y de fácil mantenimiento.

[5] Es posible modificar su comportamiento de forma estática a través de la herencia.

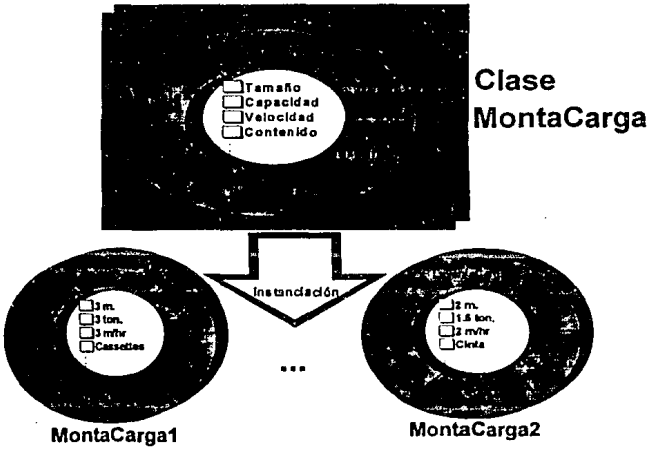
Concepto de Clase. Una clase es la definición abstracta de un grupo de objetos con datos y métodos comunes.

Definición de clase

Los objetos se crean mediante la instanciación de una **clase**. Mientras que un objeto es una entidad concreta que existe en el tiempo y el espacio, una clase representa solo una abstracción, la esencia del objeto.

Un objeto es la instanciación de una clase

Por ejemplo: dadas las variables ***i,j,k*** de tipo entero ***Int***, decimos que ***I,J*** y ***k*** son objetos de la clase ***Int***.



La actividad principal de cualquier lenguaje de programación orientado a objetos es definir nuevas clases (tipos) que actúen como "construidos de fábrica" (built in), por ejemplo: int, double, float y char del lenguaje C.

Una clase define las propiedades y métodos que comparten todas sus instancias, pero cada objeto tiene sus propios valores únicos; justo como se muestra en la gráfica anterior, cada montacargas tiene un tamaño, capacidad, velocidad máxima y contenido específico. Aunque cada objeto tiene sus propios datos, sí comparte los métodos de la clase con los demás objetos.

Cada objeto posee sus propios datos pero comparte los métodos

Un objeto siempre es la instanciación de una clase, pero una clase bien puede no producir objetos. A este tipo de clases, que solo sirven para expresar características comunes a un grupo de clases, se le llama **clase abstracta** y cobra sentido en una jerarquía de clases.

Clase Abstracta

Sin embargo, lo anterior no es suficiente para entender la importancia y trascendencia de las clases. Además de las funciones que vimos, las clases traen orden a los objetos organizándolos en **jerarquías de clase** -herencia- y en **objetos compuestos** (composite objects) -objetos que contienen otros objetos-. Estos conceptos se explican más adelante, pero veamos cual es su origen.

Las clases traen orden a los objetos

Métodos de Organización. Para entender el mundo que nos rodea, los seres humanos utilizamos varios métodos de organización.

El Organizar nos permite entender más fácilmente

De acuerdo con Yourdon & Coad[6], los métodos de organización que existen son tres:

[6] OOA Yourdon & Coad pp. 16,17

1) **Identidad.** Identificar los objetos por medio de sus atributos (estado) y comportamiento -p.e. un puma.

2) Organización PARTE DE (**objetos compuestos**) Descomponer al objeto en las partes que lo forman - p.e. un puma tiene ojos, patas y cabeza.

3) Organización TIPO DE (**jerarquía de clases**) Identificar la relación que tiene con otros objetos de características semejantes -p.e. el puma pertenece a los felinos.

Tipos de organización humana

A continuación presentamos la aplicación de estos métodos al POO.

El punto 1, Identidad, se explicó en el apartado de objetos. La actividad principal en este punto es la identificación y definición de los objetos.

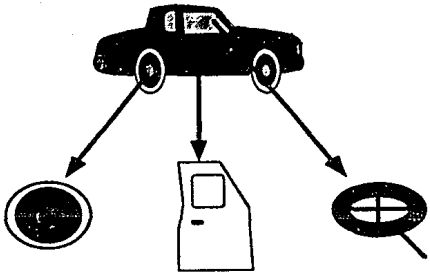
Los puntos 2 y 3 están íntimamente relacionados con las clases, recordemos que las clases son la abstracción de un grupo de objetos con características similares.

De nuevo, los dos métodos de organización restantes los utilizamos para simplificar nuestra apreciación del mundo.

Organización PARTE DE. Cuando le explicamos a un niño pequeño las partes que tiene un automóvil, le decimos:

Un automóvil tiene puertas, ruedas, volante y motor.

La organización PARTE DE explica al objeto en término de sus componentes. A este tipo de objetos que se componen a partir de otros se les llaman **objetos compuestos** (composite objects).



*La organización **parte de** describe a un objeto a través de las partes que lo componen (composite objects)*

Organización TIPO DE. Por otra parte, tomando el ejemplo anterior, cuando hacemos una descripción a un niño pequeño de lo que es un sofá, asumiendo que el niño sabe lo que es una silla, no le explicamos desde la nada, sino que le decimos:

*La organización **tipo de** describe al objeto en término de sus semejanzas con otros*

Un sofá es como una silla, solo que es más cómodo y más grande.

*El método, pues, que debemos adoptar para tratar de reconocer los grupos existentes en la naturaleza es seguir las indicaciones que el **Instinto** humano nos dicta, por ejemplo, para formar la clase de las aves y la clase de los peces.*

- Aristóteles.

Las clases actúan como una plantilla, agrupando objetos relacionados entre sí con propiedades y comportamiento similares.

En la organización **TIPO DE**, las clases se ordenan de una forma jerárquica en una **taxonomía** de superclases (padres) y subclases (hijos). Es precisamente esta taxonomía la que define la **jerarquía de clases.**

Jerarquía de clases

Con las subclases es posible definir objetos más especializados, conforme desciende la jerarquía. Las superclases expresan características comunes entre clases. Entre más cerca nos encontremos de la raíz, las clases son más generales; entre más cerca a las hojas, las clases son más particulares.

Entre más cerca de la raíz, más general; entre más cerca a las hojas, más especializado

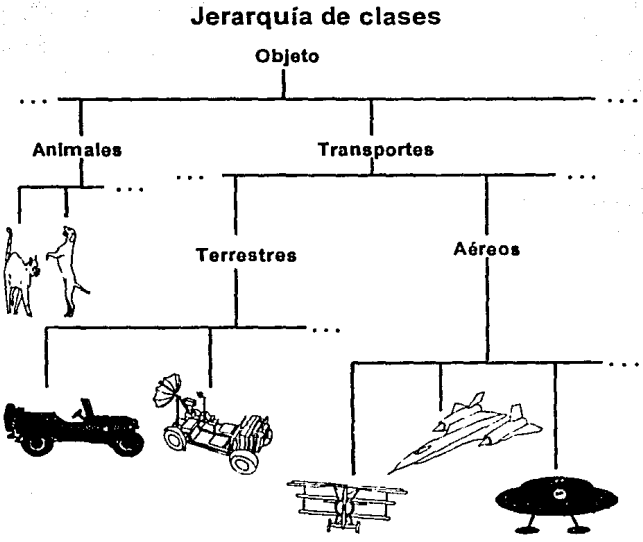
La jerarquización a la que se llega no es única, depende del enfoque y de la experiencia de la persona que la realiza.

Concepto de herencia. En términos generales, una subclase hereda todos los datos y métodos de la superclase a la que pertenece. La subclase puede adicionar datos y métodos a los que heredó de su padre, aunque también puede modificar los métodos heredados. En algunos lenguajes de programación, como el C++, es posible heredar de una manera selectiva (public, private y protected).

Concepto de herencia.

El hijo hereda las características del padre, adicionando otras.

En el diagrama que sigue se muestra el concepto de la herencia. Todos los transportes tienen un cierto tipo de combustible, una capacidad de carga y su objetivo principal es transportar personas y/o bienes de un lugar a otro. Todas las subclases de transporte heredan estas características. Además de la rama de transportes, todos los transportes terrestres comparten algunas características comunes, de la misma manera que lo hacen los transportes aéreos; por ejemplo los transportes terrestres se mueven en tierra firme, mientras que los aéreos por aire.



Un ejemplo de las ventajas de la herencia es que si se desea otra rama, por ejemplo transportes marítimos, su definición es mucho más fácil porque ya tenemos todas las características que comparten los transportes. Solo necesitamos definir la información específica sobre esta rama, como que los transportes marítimos se desplazan en el agua.

Al adicionar una nueva clase, podemos construirla sobre alguna jerarquía ya existente

La herencia constituye la principal forma de reutilizar el código en la programación orientada a objetos.

Capítulo 2:
Antecedentes

Ya sea que pienses que algo se puede o que no se puede hacer, lo más seguro es que se cumpla lo que pensaste.

- Anónimo.

Antecedentes

2.1. Crisis del Software

Las compañías modernas se enfrentan con un gran problema en el manejo de su información, cada vez se vuelven más dependientes del flujo continuo y adecuado de ésta, para el correcto funcionamiento de sus operaciones. Sin embargo, la habilidad de administrar sus propios datos es insuficiente; el volumen y la complejidad de la información rebasa su capacidad de manejarla eficientemente. Esto provoca que se "pierdan" entre sus propios datos.

Las compañías son incapaces de manejar su propia información eficientemente

La velocidad a la que avanza la tecnología en los componentes de **hardware** es sorprendente; año con año se construyen computadoras más veloces, poderosas y de menor precio. Aunque los costos del hardware han disminuido, los costos del **software** se han incrementado notablemente. El problema de construir buen software es mucho mayor que el de construir mejores computadoras.

El hardware es cada vez más rápido y barato

El problema no reside en el hardware. El problema se encuentra en los programas (software). Los cambios que requieren los sistemas, rebasan la capacidad de respuesta de los departamentos de cómputo.

El problema está en el software

Esta brecha entre el hardware y el software afecta a cualquier usuario de computadoras, aun a los más pequeños. Sin embargo, en las grandes industrias, estos efectos toman proporciones alarmantes, debido a la gran dependencia que tienen con sus sistemas de cómputo centrales, que generalmente son monolíticos y obsoletos. Es muy raro encontrar en la actualidad que un proyecto sea cumplido en el tiempo y con el presupuesto original.

La mayoría del software se entrega tarde y arriba del presupuesto

Peor aún, los sistemas que se liberan en estas condiciones están llenos de problemas y construidos de una manera muy rígida, volviéndose casi imposible hacer cambios importantes sin re-diseñar totalmente el sistema.

Al combinar estos problemas con la necesidad de las compañías por cambiar más rápidamente, se presagia un desastre. La mayoría del software corporativo es obsoleto, aún antes de ser liberado y generalmente es incapaz de evolucionar. Entre mayor sea el sistema, es mayor la probabilidad de fracaso. La capacidad de respuesta de los departamentos de cómputo está, pues, limitada por la complejidad, la carga de trabajo y por la velocidad de cambio del negocio. Esto indica que la forma en que se están realizando los programas en la actualidad simplemente no funciona.

Se necesita producir mejor software, más rápido

Para reducir estos efectos desde la década de los años 70´s, se han desarrollado varias metodologías y herramientas. Entre los trabajos más notables destacan el enfoque estructurado y el enfoque a datos. El enfoque estructurado responde a las preguntas: ¿Cuáles son las entradas y las salidas? y ¿Cómo se transforman los datos?. El enfoque orientado a objetos responde a las preguntas: ¿Cuáles son los elementos con los que estaré tratando? y ¿Cuál es su comportamiento?. En la siguiente sección, veremos brevemente los conceptos y la evolución de estos enfoques en los lenguajes de programación.

Se desarrollaron buenos esfuerzos para reducir estos efectos, aunque son eficaces ya no son eficientes

Aunque estas metodologías y herramientas han representado un gran avance y han sido efectivas durante largo tiempo, es evidente que la velocidad de cambio que requieren los negocios, han rebasado nuestra capacidad de respuesta. Es precisamente de estas metodologías y de la necesidad de afrontar estos problemas, que evolucionó el concepto de la orientación a objetos.

La orientación a objetos propone una solución a estos problemas

2.II. Evolución de los lenguajes de programación

La intención de esta sección no es dar una amplia introducción a la historia de las computadoras ni a los lenguajes de programación, por el contrario, esta exposición será breve y su propósito es solamente mostrar un marco de referencia que muestre de dónde evolucionó la programación orientada a objetos.

Esta sección sólo es un marco de referencia

En este momento se propone esta hipótesis:

Comparando el enfoque estructurado con el enfoque orientado a objetos, podemos decir que: la orientación a objetos representa una evolución del concepto, pero una revolución del método.

Hipótesis

Al final de esta sección trataremos de demostrarla.

Cuando surgieron las primeras máquinas automáticas -como la calculadora mecánica de Wilhelm Schickhard en la Universidad de Ttbingen (1623) y la de Blaise Pascal (1645)-, el concepto de instrucción programable no existía. La máquina era construida únicamente para servir a un propósito muy específico. Aún con las primeras computadoras programables de tubos al vacío -como la MARK I, construida por Howard Aiken en la Universidad de Harvard (1944)-, su programación era muy difícil. Se tenía que re-cablear el programa para que la computadora hiciera otra función.

Los lenguajes de programación no nacieron con las computadoras

No fue sino cerca de los años 40's que Konrad Zu, construyó la primera computadora de propósito general, con instrucciones lógicas alterables por interconexiones (lógica digital) . Es hasta este momento que cobra forma el concepto de instrucción programable.

Un programa es un conjunto de instrucciones. Un lenguaje de programación es la notación en la que se

Un lenguaje es la notación en la que se escribe un programa

escribe un programa. Existen tres tipos de lenguajes de programación:

1) Lenguaje de máquina

2) Ensambladores

3) Lenguajes de alto nivel.

Tipos de lenguajes

El lenguaje de máquina se escribe en código máquina con 1's y 0's. La computadora puede interpretar y ejecutar estos códigos directamente.

Dado que cada computadora tiene un conjunto único de códigos de operación, un programa escrito en una computadora no puede correr en otra. El lenguaje de máquina es tedioso y difícil de aprender.

El lenguaje de máquina es difícil

Los ensambladores son similares al lenguaje máquina, sólo que manejamos un mnemónico de hasta 4 caracteres. En forma general, los ensambladores son la forma más eficiente de programar. Sin embargo, el principal problema es que son aún dependientes de la computadora y difíciles de manejar.

La programación en ensamblador es la de más rápida ejecución

Los lenguajes de programación de alto nivel son, mayormente, independientes de la computadora. Generalmente una instrucción de un lenguaje de alto nivel, produce varias instrucciones de máquina, lo que simplifica la programación. Los primeros lenguajes de programación comerciales fueron: FORTRAN, COBOL, ALGOL, RPG, PL1, etc ..

Los lenguajes de alto nivel son más fáciles y menos dependientes de la máquina

Cada lenguaje de programación fue creado bajo una cierta metodología de diseño, bajo una cierta orientación. A continuación clasificamos a los lenguajes de programación de acuerdo a su orientación:

Orientación de los lenguajes de programación

1) Orientación a procedimientos.

2) Orientación a datos.

3) Orientación a objetos.

La clasificación anterior no es de ninguna manera la única ni la más completa -existe orientación lógica como prolog por ejemplo-, sin embargo, ésta parece ser la más útil para entender la evolución de los **lenguajes orientados a objetos** (LOO). A continuación explicamos cada orientación.

2.II.1. Orientación a Procedimientos

Bloque fundamental: el procedimiento

En los métodos de diseño orientados a procedimientos, los sistemas se construyen utilizando procedimientos (algoritmos) como el bloque fundamental de construcción. De aquí precisamente toma su nombre. La mayoría de los lenguajes de programación actuales se encuentran en esta clasificación -como C, pascal, cobol, basic,etc...

La mayoría de los lenguajes son de esta clasificación

Dado que la función es el foco de atención, los datos se ven como entradas que se transforman de alguna manera para producir ciertas salidas, gráficamente:

Entradas - transformación - salidas

En la orientación a procedimientos (programación estructurada), la actividad principal es dividir los procedimientos en tareas más pequeñas hasta llegar a un nivel atómico, que lo define las instrucciones básicas del lenguaje de programación. Los pasos que componen cada módulo o procedimiento y las particularidades de los datos sobre los que actúan, constituyen el soporte del programa. En esencia se busca cómo realizar una tarea.

Los procedimientos se subdividen en funciones más pequeñas

Estructurando el programa con base en módulos se hace más fácil su mantenimiento. Con la programación estructurada se incorporó un término muy importante: la **abstracción**. La abstracción se puede definir, en este contexto, como la capacidad para examinar algo sin preocuparse por los detalles internos. En la programación estructurada no es importante, desde el punto de vista del

Los módulos esconden la complejidad

usuario del módulo, como se realiza la tarea, siempre y cuando se haga bien.

La mayoría de los esfuerzos por mejorar el desarrollo del software se a enfocado en la modularización de los procedimientos. Pero existen otros elementos, aunque menos obvios, que son tan importante como los procedimientos: los **datos**. Los datos (conjunto de información) son operados por los procedimientos.

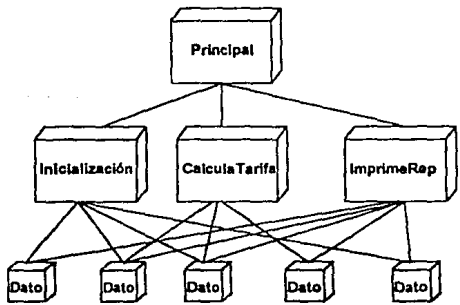
La modularización se ha enfocado a los procedimientos

Dado que las técnicas de la programación modular han evolucionado durante los años, se ha vuelto evidente que los datos también deben modularizarse.

Los datos se deben modularizar también

Si un programador requiere manejar solamente pocos datos, éstos se pueden compartir sin problemas en los módulos de todo el programa. El compartir los datos es muy conveniente para los programadores, pues se tiene una área común, donde las subrutinas pueden compartir los datos cuando necesiten comunicarse entre sí.

Las subrutinas pueden compartir pequeñas cantidades de datos



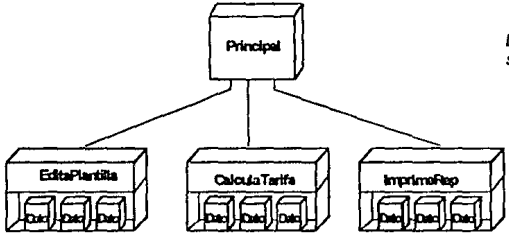
Compartiendo datos, en varias subrutinas

Pero cuando el número de datos crece en cientos o miles, esta solución tan simple lleva a errores misteriosos y a un comportamiento imprevisible. El problema radica en compartir los datos: esto representa la violación de la modularidad, la cuál requiere que los módulos sean tan independientes como sea posible. El permitir que todos los módulos interactuen libremente con los datos compartidos hace que cualquier módulo sea directamente dependiente del comportamiento de todos los demás. De echo, la información común se vuelve el desorden de la armonía en las subrutinas, que se había lograda con la programación estructurada.

Pero el compartir demasiados datos, conlleva problemas importantes

La solución a estos problemas es modularizar los datos junto con los procedimientos. Esto se logra, típicamente, al darle a las subrutinas sus propios datos, sin permitir que ningún otro módulo los accese. Esta estrategia del **ocultamiento de la información** (information hiding) previene de interacciones no deseadas entre subrutinas, permitiendo así que los módulos se diseñen y se mantengan de una manera más independiente.

La solución a estos problemas es "esconder" la información (data hiding)



Datos locales en una subrutina

Los programas pequeños, frecuentemente, no necesitan guardar datos, sólo toman los datos de entrada y producen la salida para ser utilizada inmediatamente. Por ejemplo, un programa que calcule tablas de amortización, tomando como datos de entrada un valor y el período de amortización; los resultados arrojados se envían a una impresora que entrega una hoja con los cálculos. Los programas de este tipo no requieren de mantener ningún tipo de información, porque la generan cada vez que se corren.

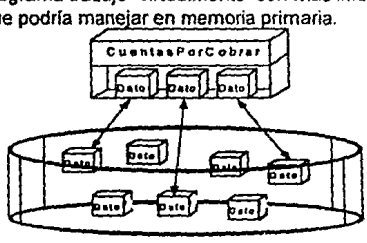
Algunos programas no tienen que guardar información

Sin embargo, los sistemas medianos y grandes (más sofisticados) generalmente trabajan con la misma información una y otra vez. Los programas de control del inventario, sistemas contables y herramientas de diseño en ingeniería simplemente no podrían funcionar si no existiera una forma de almacenar los datos entre corridas del programa.

Pero la mayoría necesitan reusar datos

La solución más simple a estos problemas es hacer que el programa guarde sus datos en un archivo externo. Cuando se termina de correr el programa, éste envía sus datos al archivo. Cuando empieza a correr el programa de nueva cuenta, carga sus datos del archivo en que los guardó. El utilizar archivos permite también que el programa trabaje -virtualmente- con más información de la que podría manejar en memoria primaria.

Los datos se almacenan fácilmente en archivos



Un programa puede guardar y cargar datos de un archivo

Los archivos externos proveen una solución adecuada al problema del almacenamiento de la información, siempre y cuando los datos sean accesados solamente por una sola persona y por sólo un programa. Cuando es necesario compartir los datos entre varios procesos y usuarios, se presentan problemas.

Pero esto no funciona cuando se necesitan compartir datos

Además del problema del acceso simultáneo, los sistemas que se diseñan con funciones como bloque fundamental, tienen otro problema. Dado que las funciones tienden a cambiar más que los datos, sería muy recomendable que los datos jueguen ahora el papel fundamental, o al menos igual de importante que las funciones en el diseño de los sistemas. Para ilustrar por que los datos son menos volátiles, veamos el siguiente ejemplo:

Los datos son más estables que los procedimientos

Digamos que tenemos un sistema que mide la velocidad de un avión en crucero. La forma (procedimiento) para medir la velocidad puede ser de muchas maneras: por revoluciones en un tacómetro, sensores ópticos, sensores electromagnéticos y sensores sónicos, por citar algunos. Si nuestro sensor de velocidad cambia, la interacción entre todo el sistema tendrá seguramente que cambiar. Sin embargo los datos que se deben mantener del sistema son los mismos: velocidad en crucero y estado del sensor. Como podemos ver los datos son menos volátiles que los procedimientos.

Para solventar estos problemas surge la orientación a datos.

2.II.2. Orientación a Datos

El analizar y diseñar con base en los datos lleva a la creación de sistemas más flexibles que permiten cambios más fácilmente, reduciendo así los altos costos en los sistemas rígidos tradicionales. Entre los principales objetivos de esta orientación se tiene:

La orientación a datos lleva a sistemas más flexibles

- Lograr la separación lógica entre las aplicaciones y el modelo de datos corporativo que las soporta.
- Establecer bases de datos, al mismo tiempo que sistemas, que controlen la integridad de los datos de las bases de datos.
- Provocar la participación activa de los usuarios en la definición de las estructuras de datos.

Objetivos de la orientación a datos

La orientación a datos toma los conceptos de la programación estructurada, pero refuerza la parte débil de los datos. Esta orientación se dio, principalmente, al devenir de las bases de datos. En las bases de datos la estructura e integridad de los datos son la parte fundamental. Un manejador de bases de datos mantiene la integridad de los datos y mantiene un control estricto en cuanto al acceso concurrente a los datos.

La orientación a datos retoma los conceptos de la orientación a procedimientos, pero refuerza la parte débil de los datos: integridad y concurrencia

Pero recordemos que no es nuestro objetivo analizar las bases de datos, regresemos al tema de la evolución de la orientación a objetos.

Podemos decir que el bloque de construcción principal, en la orientación a datos, son las estructuras de datos. Este énfasis a los datos ayuda enormemente a los sistemas de información dado que los datos son menos volátiles que los procedimientos.

Bloque de construcción: estructuras de datos

Para lograr el control en la integridad de los datos y el manejo de la concurrencia, se introduce el concepto de manejador de bases de datos, que es el frente de acceso a los datos. Este manejador de la base de datos puede ser tan sencillo o sofisticado como se desee. Generalmente también forma parte de la base de datos un "lenguaje de definición de los datos", en el que se define las estructuras de los datos a utilizarse.

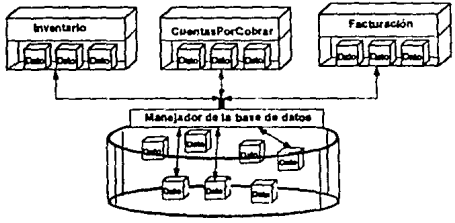
El manejador de la base de datos es el frente de los datos

Pero el manejar las estructuras de datos no es todo lo que realiza la base de datos; otra de sus funciones importantes es mantener las "relaciones" que existen entre los datos (con el lenguaje de definición). De esta manera es relativamente fácil relacionar la información de la manera que necesitemos para hacer referencias cruzadas, tan importante para la información estratégica del negocio.

La base de datos también contiene las relaciones entre los datos

Volviendo a los lenguajes de programación, el cambio que se dio no fié tan drástico como la transición de los ensambladores a los lenguajes de alto nivel. Por el contrario, el cambio fue sutil, manifestándose sólo como "extensiones" del lenguaje para el manejo de la base de datos. Sin embargo, el respaldo teórico que debe tener un programador para utilizar y definir una base de datos es mucho mayor.

Los lenguajes sólo son "extensiones" de los originales



La orientación a datos representa una gran ayuda para los sistemas de información, pues incluye toda una metodología bien fundamentada con bases teóricas y que ha sido objeto de constante investigación en el campo de la computación. Departamentos enteros en grandes compañías se encargan exclusivamente del diseño, operación y atención a las bases de datos corporativas.

Los sistemas de información han sido los más beneficiados con esta orientación

Las empresas como las casas de bolsa, bancos y en general cualquier institución financiera dependen en forma total de sus datos. Sin el empleo de éstas y sus características - como la reconstrucción de la base a partir de la bitácora (audit trail)-, sería casi imposible operar y crecer sanamente.

Hemos visto que la orientación a datos presenta grandes ventajas, en comparación con las que ofrece la programación estructurada por si sola. Lo interesante es que estamos construyendo otra arquitectura -la de datos- sobre la de orientación a procedimientos. Evoluciona el concepto de la orientación a procedimientos hacia la orientación a datos.

Estas ventajas se suman a las ventajas de la orientación a procedimientos

Sin embargo subsisten algunos otros problemas. El uso de la orientación a datos requiere de la aplicación de una arquitectura de datos y de gente mucho más preparada, especialmente en el manejo de las bases de datos. La arquitectura de la que hablamos depende básicamente de la experiencia y la calidad de los diseñadores de la base de datos, no es garantía el contar con una base de datos para resolver nuestros problemas; esto es, no existe una forma implícita en los lenguajes para promover el uso de la metodología. Peor aún, si no utilizamos correctamente la base de datos y no diseñamos bien nuestras relaciones entre los datos, resulta más problemático que no utilizar una base de datos.

Pero subsisten algunos problemas como la dependencia, ahora, de los datos

Otro gran problema es que la modificación de la estructura de los datos se vuelve, generalmente, prohibitivo. Dado que ahora nuestro foco de atención es la estructura de los datos, si hacemos cambios importantes se requiere de la reestructuración de la base de datos. Esto conduce a la re-compilación de la base y la adaptación de los datos anteriores a los actuales, lo que implica copias de un formato o de un lugar a otro. En pocos cientos de datos no parece haber problema, pero en varios miles o millones de ellos podría ser intolerable.

Y la posible reestructuración de la base de datos

Para evitar estos problemas, el concepto evoluciona a la **orientación a objetos**.

Por eso se evolucionó a la orientación a objetos

2.II.3. Orientación a Objetos

Sin importar todos los esfuerzos que se han hecho para construir mejores programas más fácilmente, la crisis del software sigue creciendo año con año[7]. Han pasado más de cuarenta años desde que se inventó la subrutina y seguimos construyendo los sistemas artesanalmente -"a mano", una instrucción a la vez. Se han desarrollado mejores métodos para este proceso de construcción que tratan de automatizarlo, pero no funcionan satisfactoriamente para sistemas grandes. Adicionalmente, estos métodos producen software que es difícil de mantener y modificar. [8]

Ninguno de estos esfuerzos han resuelto la crisis del software

Se necesita una nueva metodología para construir sistemas, uno que deje atrás los problemas tradicionales y que proponga una manera mejor y más rápida de hacer sistemas. Esta nueva metodología debe ser aplicable tanto a sistemas grandes como a sistemas pequeños, y debe crear sistemas flexibles, mantenibles y capaces de evolucionar al ritmo que lo hacen sus empresas. La orientación a objetos puede cumplir con estos retos y aun más.

La orientación a objetos puede vencer a la crisis del software

La orientación a procedimientos centra su atención en los procedimientos; la orientación a datos, en los datos; la orientación a objetos, en ambos como un todo inseparable.

La orientación a objetos toma a los datos y procedimientos como un todo inseparable

Al principio de esta sección (2.2 Evolución de los lenguajes de programación), propusimos la hipótesis:

La orientación a objetos representa una evolución del concepto, pero una revolución del método.

---

[7]Referirse a la sección 2.1 Crisis del Software
[8]Object Oriented Technology: A Manager's Guide Taylor

Se explica la evolución del método porque retoma los fundamentos de la orientación a procedimientos y de la orientación a los datos. Un objeto consta de datos y métodos (procedimientos) como un todo inseparable, ambos con la misma importancia. Por tanto no es un concepto que sale de la nada. Es efectivamente una evolución.

Evolución del concepto

Recordemos que en las otras orientaciones tenemos una arquitectura -modularización, abstracción, formas normales, etc... Esta arquitectura puede aplicarse o no, quedando a la voluntad de los diseñadores. Sin embargo en la orientación a objetos, y más especificamente en un lenguaje de programación orientado a objetos, la arquitectura -encapsulado, herencia, abstracción, etc...- está implícita y no se le separa de la programación[9]. Por eso decimos que: la orientación a objetos es una revolución del método porque nos obliga a seguir una cierta arquitectura, unas leyes básicas (como las de estática en la construcción de edificios) que nos garantizan un software más seguro, flexible, mantenible y reutilizable. Esto se hará evidente en el siguiente capítulo.

Revolución del método

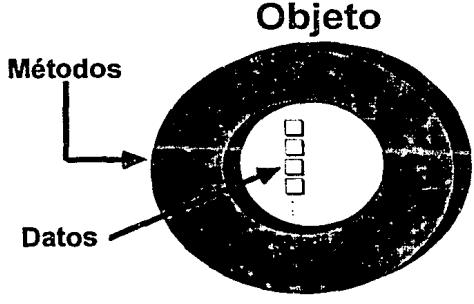
Tratar de explicar los fundamentos de la orientación a objetos solamente en unos cuantos párrafos sería demasiado aventurado. En el siguiente capítulo "Introducción a los Conceptos de la Orientación a Objetos", se tratan los elementos de la orientación a objetos.

Por ahora pensemos que la orientación a objetos centra su atención tanto en los procedimientos como en los datos, ninguno es más importante que el otro.

[9] Alan Kay, MIT Media Lab "El Padre de la Computadora Personal"

Pero usted se preguntará: ¿Entonces porqué se le llama orientación a objetos?. Se le llama orientación a objetos porque representan la abstracción de un objeto real, que consta de un estado -representado por los datos- y que exhibe cierto comportamiento -representado por los métodos. Así pues, el objeto representa más fielmente al elemento de la realidad en la computadora; la computadora se amolda al mundo real, en contraste con la programación tradicional donde amoldamos el mundo real a la computadora. En este sentido, la orientación a objetos es mucho más natural para la gente común que las técnicas tradicionales; aunque generalmente resulta más confusa para las personas que trabajamos con las otras orientaciones.

Un objeto es la abstracción de un elemento real que tiene estado y comportamiento



Aclaración. Esta evolución de los lenguajes de programación es discutible y representa únicamente el punto de vista del autor. Dado a que esta clasificación parece ilustrar mejor la evolución del concepto de la orientación a objetos, es que se presentó de esta manera. Sin embargo la orientación a objetos surgió

cronológicamente en paralelo con las otras orientaciones; aunque sí se enriqueció con ellas.

A continuación se recomienda leer el capítulo I: "Resumen Ejecutivo de la Orientación a Objetos", si no lo ha echo aún, y después continuar con el capítulo III.

Capítulo 3:

Introducción a los Conceptos de la Orientación a Objetos

Los tres fundamentos del aprendizaje son: mirar mucho, estudiar mucho y sufrir mucho.

- Catherall.

Entre más difícil una tarea, una vez realizada es más satisfactoria.

- J.C. Cárdenas.

Introducción a los Conceptos de la Orientación a Objetos

3.1. Introducción al capítulo

Esta es una introducción al contenido del capítulo

El objetivo de esta introducción es dar un panorama general de lo que veremos durante todo el capítulo.

Este capítulo presenta los conceptos de la orientación a objetos

Este capítulo introduce los conceptos que se manejan en la orientación a objetos. Estos conceptos se presentan con cierto detalle, por lo que ud. podría perder la esencia de la información. Por tanto, le recomiendo que lea antes - si no lo ha echo- el primer capítulo: "Resumen Ejecutivo de la Orientación a Objetos" para que tenga una idea general de estos conceptos.

La orientación a objetos se basa en la reutilización del software

La orientación a objetos es una nueva forma de construir sistemas de cómputo que promete resolver algunos de los problemas principales en la construcción tradicional del software. El punto central detrás de la orientación a objetos es que el software debe ser construido con componentes estándares y reutilizables, siempre que sea posible. Es precisamente por esta premisa que algunos expertos le han llamado a esta tecnología: la revolución industrial del software.

Existen muchos términos nuevos

Una de las barreras para entender la orientación a objetos es el vocabulario que se maneja. Mientras que algunas palabras de este vocabulario no son de gran relevancia, la mayoría de los términos valen la pena revisarse porque constituyen realmente nuevos conceptos, muy diferentes a los que manejamos en la programación tradicional.

Los diez términos mínimos, y generalmente suficientes, para entender a la orientación a objetos son: **objeto, método, mensaje, clase, subclase, instancia, herencia, encapsulado, abstracción y polimorfismo.**

Pero son diez los básicos

Aunque todos los términos son importantes, se les ordenó en tres grandes bloques a saber (se adicionaron también algunos otros):

1) Objetos

   Abstracción
   Encapsulado

2) Mensajes

   Reutilización de nombres (name overloading)
   Polimorfismo

3) Clases

   Modularidad
   Jerarquía
   Herencia
   Objetos compuestos (composite objects)

Estructura general de este capítulo

Inicialmente se presenta una pequeña exposición de la analogía entre objetos y células. Posteriormente este capítulo se divide en tres sub-partes, que explican cada uno de los apartados anteriores con un mayor detalle.

Antes se presenta una analogía con las células

Pero el lector se preguntará: ¿Porqué esta estructura?. En muchas lecturas del tema los conceptos: abstracción, encapsulado, modularidad, etc..., se muestran como partes sin mucha relación entre sí, lo que puede provocar confusión. La clasificación de arriba parece ser adecuada para disminuir la confusión, especialmente para los nuevos lectores de la orientación a objetos.

Esta estructura genera menos confusión

Como ya vimos, hay tres conceptos clave que forman la base de la orientación a objetos:

1) **Objetos**. Los objetos son paquetes de código que contienen los datos y procedimientos relacionados con ese objeto. En la mayoría de los casos, los objetos corresponden a objetos de la realidad (materiales o conceptos teóricos), tales como: productos, máquinas y personas.

2) **Mensajes**. Los mensajes son el medio de comunicación entre los objetos. En esencia, los objetos piden a otros que realicen cierta acción a través de un mensaje. Por ejemplo, los mensajes químicos entre las células.

3) **Clases**. Las clases son como una plantilla para definir tipos de objetos relacionados entre sí. Por ejemplo, todos los objetos que son una orden de pago se describirían como la clase: *orden de pago*. Las clases agrupan objetos con características similares de una manera natural -tal como lo hace la clasificación de los mamíferos.

Los tres conceptos claves de la orientación a objetos

Construyendo el software como si fuera hardware. Una de las ideas principales es construir el software de la manera en la que construimos el hardware. En la programación, haciendo un símil con el hardware, nos encontramos como en la época en que se construían las computadoras con transistores[10]. En lugar de construir los programas desde el principio cada vez, deberíamos construirlos con piezas de software estándares y reutilizables; en el software, debemos pasar de los transistores a los circuitos integrados -CHIPS.

Debemos pasar de los transistores al los chips, en la construcción del software

[10]Philippe Kahn, presidente y fundador de Borland International.

3.II. Los Objetos, Mensajes y Clases en la Naturaleza: Las Células.

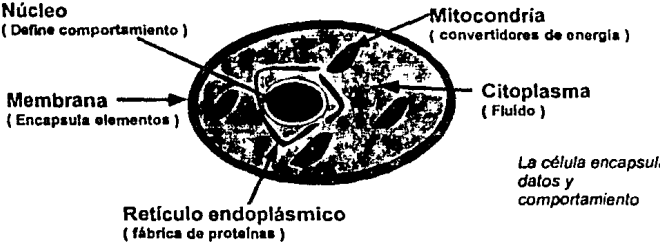
Se dice que la programación orientada a objetos es más natural que la programación tradicional, y es cierto en dos sentidos. En un sentido, la programación orientada a objetos es más natural porque nos permite organizar la información de una forma familiar para nosotros. En el otro sentido, es más natural porque refleja las técnicas de la naturaleza para manejar la complejidad. Para poder aclarar esta afirmación, veamos de forma muy general la manera en que la naturaleza le da poder al concepto de objeto.

La orientación a objetos es más natural

El bloque básico de construcción en la naturaleza, del cuál estamos hechos todos los seres vivientes, es la célula. Si nos detenemos a pensar un momento en sus implicaciones veremos que, toda la variedad de seres vivos que existimos sobre la tierra estamos constituidos de una misma cosa: células.

Toda la vida en la tierra se constituye de células

Célula



La célula encapsula datos y comportamiento

Si vemos la estructura interna de la célula, ignorando todas las posibles variaciones y complejidades, podemos hacer un par de observaciones interesantes:

Las células encapsulan datos y procedimientos

1) **La célula está protegida por una membrana** que controla el acceso a sus adentros. De echo, la membrana **encapsula** a la célula, protegiendo su forma interna de trabajar de la intromisión externa.

2) **La célula contiene formas de manejar información y comportamiento.** La mayoría de la información que define su comportamiento se mantiene en el núcleo, en el centro de la célula (ADN, ARN). El comportamiento, como son la síntesis de proteínas y la conversión de energía, se lleva al cabo en el cuerpo medio de la célula.

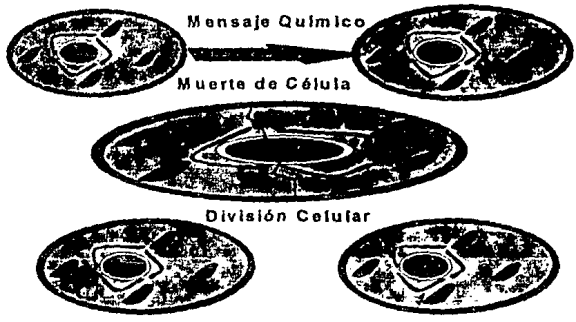
Las células encapsulan sus adentros y tienen un cierto comportamiento

Interacción entre células. Las células están rodeadas por una membrana, la cuál permite **intercambiar mensajes** químicos con otras células. Una célula puede permitir o no el paso del mensaje químico. La membrana representa su interface con el exterior.

Las células se comunican con mensajes químicos

Esta comunicación basada en mensajes simplifica grandemente la forma en que las células funcionan. No es necesario que una célula conozca el funcionamiento interno de otra, lo único que tiene que hacer es enviarle el mensaje apropiado para que la célula receptora ejecute esa acción. Por ejemplo, cuando una célula muere se liberan substancias con la que las otras células se dan cuenta; una de ellas se divide en dos para substituir a la que murió, regenerándose así los tejidos.

Simplificando la interacción entre células



Como podemos apreciar, la interacción entre células está basada en mensajes. Esto quiere decir que cuando una célula quiere afectar el comportamiento de otras células, aquella les envía un mensaje químico que provoca que éstas modifiquen su comportamiento al realizar cierta acción. Cuando una célula recibe un mensaje que tiene significado para ella, responde al estímulo de acuerdo con la información que tiene codificada en su núcleo.

Los mensajes modifican el comportamiento de la célula

En pocas palabras, las células no interactuan entre sí interfiriendo con el ADN de las otras. Las células se ocupan de sus propias actividades y hacen requerimientos de una manera ordenada, en lugar de tratar de controlar la operación interna de las otras células directamente. Esta situación ofrece dos formas diferentes de protección:

Los mensajes ofrecen ventajas muy importantes

1) Primero, **protege los adentros** de cada célula de ser modificados por otras. Cualquier célula viviente es mucho más compleja que la mayoría de los sistemas de cómputo jamás creados. Al proteger los

adentros de la célula del mundo externo, manteniendo su integridad.

2) Y más importante, **protege a otras células el tener que saber el manejo interno de esta célula.** Imagínese lo complicado que sería si cada tipo de célula tuviera que saber la forma de operar de cualquier otro tipo de célula.

La segunda es una forma de ocultamiento de la información (information hiding)

La segunda ventaja es un ejemplo vívido de lo que es el ocultamiento de la información (information hiding). De echo, las células funcionan tan bien como módulos en sistemas complejos como los seres vivos, gracias a que protegen al resto del sistema del tener que lidiar con la complejidad interna de sus adentros.

Si no protegemos la integridad de los adentros del módulo, podemos tener consecuencias desastrosas

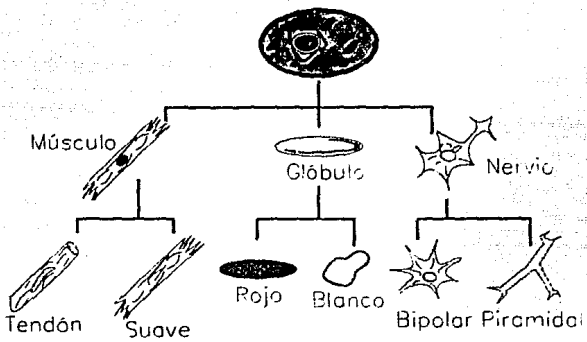
Cuando no se respeta esta independencia entre las células y la implantación interna de sus adentros, la célula enferma y muere. Veamos el problema con el virus del SIDA. Este virus se interna en la célula y la ataca desde adentro. Dado que el virus está arraigado en la célula, no puede ser destruida por los glóbulos blancos, porque se destruiría también la célula. ¿Es, a caso, algo parecido lo que hemos sufrido durante mucho tiempo en los sistemas de cómputo tradicionales al no proteger los adentros de nuestros programas ?

Las células operan bajo los mismos principios

Especialización de Células. La célula es realmente un bloque de construcción universal. Todas las células comparten una estructura similar y operan de acuerdo con los mismos principios básicos.

Sin embargo, diferentes tipos de células se especializan en una cierta actividad

Sin embargo, las células, en una gran variedad de formas, se especializan en ciertas funciones. Cualquier tipo de célula cae en una organización **jerárquica (en clases),** en donde cada forma especializada comparte características de todos los tipos arriba de ella.



Por ejemplo, las células del tallo de las plantas, tienen un recubrimiento rígido para mantenerla erguida; los glóbulos se mueven y transportan gases; las células de los tendones se pueden expandir o retraer en uno de sus ejes y comparten todas las características de las otras células de los músculos, pero también son capaces de transformar energía (mitocondrias), característica que comparten con todas las otras células.

3.III. Objetos

La llegada de los sistemas orientados a objetos ofrecen nuevas soluciones a los problemas de la construcción tradicional de los grandes sistemas de cómputo.

Para construir sistemas complejos y flexibles, la naturaleza no tiene igual

Cuando se trata de construir sistemas complejos (seres vivos), la naturaleza no tiene igual. De echo, la naturaleza construye sistemas que, no importando lo complejo que sean, son muy flexibles para adaptarse a los cambios del medio ambiente y que además se auto-reparan en la mayoría de los casos.

Los sistemas sin estas características mueren

Los sistemas que no cumplen con las características anteriores, simplemente son eliminadas por selección natural. Los dinosaurios reinaron la tierra durante millones de años. Recientemente surgió una teoría -la más aceptada- que dice que los dinosaurios perecieron debido a un cambio brusco en la temperatura de la tierra (por el choque de un asteroide, produciendo un polvo denso que cubrió los rayos solares enfriando la tierra). Como los dinosaurios no estaban preparados para sobrevivir a cambios de temperatura bruscos, perecieron; no fueron lo suficientemente flexibles al cambio. ¿A caso éste es el destino de los sistemas actuales, monolíticos e inflexibles?.

¿Es éste el destino de los sistemas de cómputo inflexibles y monolíticos?

3.III.1. Anatomía de un objeto

Aunque la mecánica real entre células y objetos difieren grandemente, en general sus funciones son muy parecidas. Ambos, objetos y células, encapsulan su comportamiento y datos asociados; ambos hacen uso de la comunicación basada en mensajes para esconder la complejidad; ambos forman jerarquías de tipos especializados; y ambos proveen la forma de construir infinidad de sistemas complejos.

Las células y los objetos son muy similares

De la misma forma que la naturaleza utiliza las células como bloque básico de construcción, en la orientación a objetos se utilizan objetos.

Los objetos son el bloque básico de construcción

En la orientación a objetos, los objetos representan una abstracción de algún elemento de la realidad. Los elementos de la realidad que representa el objeto puede provenir, principalmente, de dos grandes ramas:

Los objetos representan la abstracción de un elemento de la realidad

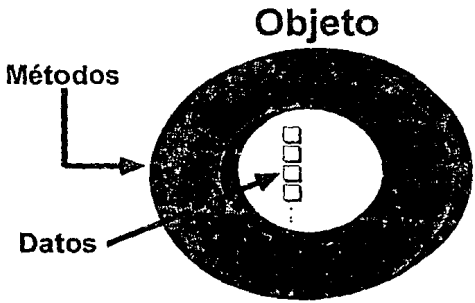
1) Una cosa tangible o visible (mesa, silla, río, mar, nube, martillo, avión, etc...), o
2) Un concepto del pensamiento, un concepto generalizado (lista ligada, compilador, cuenta de banco, autómata, base de datos, reacción química, etc ...)

Ya sea tangible o un concepto teórico

Como podemos ver, los objetos pueden provenir de muy variadas fuentes. Esto simplifica grandemente nuestro trabajo como programadores, ya que los artificios y conceptos que hemos creado con la programación tradicional, aún se pueden aplicar. Por ejemplo, podemos utilizar un objeto pila o cola circular en la construcción de un sistema operativo.

Aún podemos aplicar conceptos tradicionales de programación como pila, cola y autómata

Como ya lo hemos dicho repetidas veces, un objeto consta de datos internos que representan su estado, y de métodos (procedimientos), que representan su comportamiento; ambos, datos y métodos, forman parte medular del objeto y no se puede pensar en ellos en forma separada.



Como ya dijimos en el resumen ejecutivo, tenemos dos conceptos que asociamos al objeto: **abstracción** y **encapsulado**. Ambos conceptos son objeto de estudio en las sub-secciones siguientes. Por ahora, solamente pensemos en la abstracción como la representación de los objetos de la realidad, y en el encapsulado como la forma de proteger los datos del objeto de la intromisión externa.

3.III.2. Abstracción

El concepto de abstracción no es nuevo, en cambio, es una de las aportaciones más importantes de la programación estructurada (orientación a procedimientos).

La abstracción es una descripción simplificada que resalta solamente las características esenciales de un objeto de la realidad, despreciando las características no esenciales.

Concepto de abstracción

En la orientación a procedimientos se abstrae las funciones (los procedimientos) que hace el sistema, pero los datos quedan en un segundo plano.

En la orientación a procedimientos se abstraen las funciones

En la orientación a datos se dio un gran paso en el concepto de la abstracción: el *modelo entidad relación*. En el modelo entidad relación se modela los objetos del mundo real en términos de sus datos; aunque los procedimientos tienen una parte importante en el enfoque a datos (relaciones), se tratan datos y procedimientos como partes separadas.

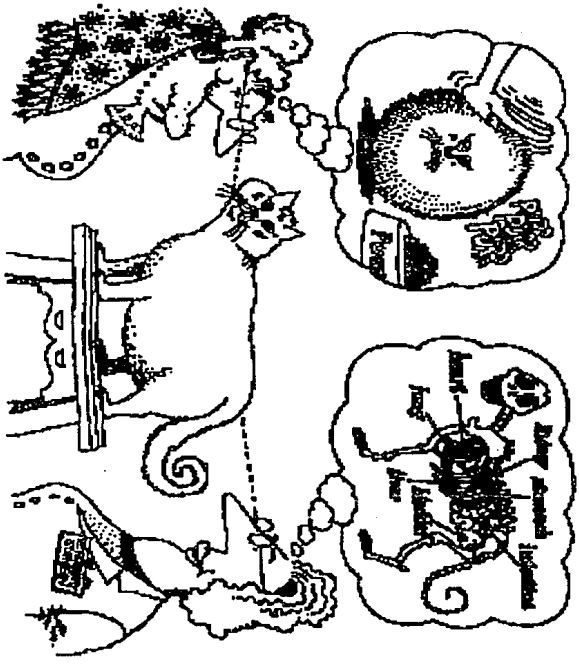
En la orientación a datos abstraemos objetos, en términos de sus datos

En la orientación a objetos se abstrae el estado del objeto (en término de sus datos) y de su comportamiento (métodos) como un todo inseparable, lo que lo hace más natural.

En la orientación a objetos la abstracción es sobre datos y procedimientos por igual

Como podemos apreciar, la abstracción es una descripción de las características esenciales del objeto. Pero, ¿Cuáles son esas características esenciales?. Bueno, eso depende del observador, de su experiencia y del punto de vista que utiliza para describir al objeto.

La abstracción depende del observador



El observador es importante en la abstracción[11]

[11] Ilustración del libro OOD, Booch pp.39

3.III.3 Encapsulado

Una de las grandes ventajas de la orientación a objetos es el **encapsulado**. Aunque este concepto no es exclusivo de la orientación a objetos, sí es la primera aplicación implícita (built in) en la arquitectura de los lenguajes de programación de esta orientación[12].

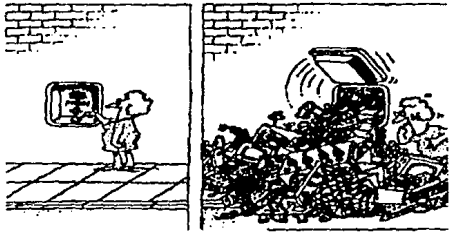
El encapsulado no es nuevo

El encapsulado protege a ciertos datos y ciertos métodos del objeto de la intromisión externa.

El encapsulado protege al objeto de la intromisión externa

Esto es, los usuarios del objeto NO pueden (y no necesitan) accesar esos datos y métodos si el diseñador del objeto no lo permite.

Al proteger la implantación interna simplificamos, también, la utilización de las cosas. Por ejemplo, un automovilista no necesita saber el funcionamiento de su auto para manejarlo; de la misma manera que un usuario de tarjeta de crédito no necesita saber todos los complicados procedimientos técnicos y administrativos que tienen que llevarse al cabo para una disposición de efectivo.



Con el encapsulado simplificamos el uso de un objeto [13]

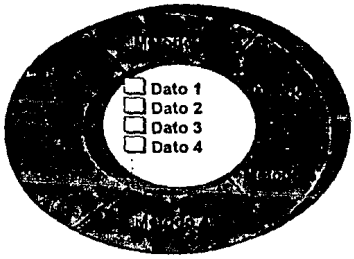
[12] La idea de encapsulado fue introducida por Parnas: Booch, OOD, pp.35
[13] Ilustración del libro OOB, Booch pp. 5

De nuevo, si tomamos la analogía con las células, el encapsulado lo representa la membrana de la célula. Para que se lleve al cabo las interacciones entre células, tiene que ser a través de mensajes químicos que la membrana permite o no pasar. Las células no interaccionan directamente con sus adentros (ADN). Desde hace mucho tiempo en programación sabemos lo que pasa cuando nos "metemos" con los adentros de los programas en una forma desordenada: el caos.

La célula esta encapsulada (protegida) por su membrana

Este concepto ya existía en la programación estructurada (en forma de mínimo acoplamiento entre módulos). Sin embargo, los lenguajes de programación con orientación a procedimientos no tienen un mecanismo implícito (built in) para promover el ocultamiento de información.

En la figura siguiente podemos ver que el objeto tiene datos y métodos. Los datos internos están encapsulados (protegidos) del exterior por sus métodos. La forma de modificar los datos internos del objeto es por medio de sus métodos. De esta forma podemos mantener un control estricto de la integridad de los datos, asegurando así la veracidad de la información.



Encapsulado: los datos internos de un objeto están protegidos por sus métodos

Las ventajas que obtenemos de la aplicación del encapsulado son principalmente:

1) Mejor manejo de la complejidad.

2) Los cambios a un objeto dado no afecta a los demás

3) Aislamiento de problemas de código

4) Flexibilidad

3.III.4. Relación entre abstracción y encapsulado

La abstracción y el encapsulado son conceptos complementarios. La abstracción permite a las personas modelar el problema con el que se enfrentan, mientras que el encapsulado permite la realización de cambios con relativamente poco esfuerzo.[14]

Relación entre abstracción y encapsulado

De echo para que funcione correctamente la abstracción, la implantación debe estar encapsulada. De otra forma la abstracción inicial se viola y se puede caer en inconsistencias y violar la integridad.

La implantación de una abstracción debe estar encapsulada

Esto nos conduce a dos conceptos importantes:

Cada objeto tiene dos partes importantes: interface e implantación.

La interface del objeto es la ***vista externa*** del objeto, la forma en que la ven los usuarios y está constituida por los métodos (públicos) del objeto. Es la que encapsula al objeto.

Interface de un objeto

La implantación del objeto son sus adentros, a través de la representación de la abstracción escogida. Esta parte interna contiene los artificios que utilizamos para modelar al objeto, por lo tanto debe estar protegida del exterior para evitar inconsistencias; el modificar un dato puede requerir modificar otros para mantener la integridad.

Implantación de un objeto

La implantación encapsula detalles sobre los cuales el usuario del objeto no debe hacer supuestos, los "secretos" de la abstracción.

[14] Booch OOD pp.45

3.IV. Mensajes, comunicación entre objetos

Como vimos en el resumen ejecutivo, los mensajes son la vía por la cuál se comunican los objetos. De acuerdo con el símil de las células, cuando una célula quiere que otra realice una acción (modificar su comportamiento), la primera le envía un mensaje químico a la segunda. En la naturaleza existen muchos tipos de mensajes, por ejemplo: **físicos** (acción-reacción), **químicos, psicológicos** ("Efecto Pigmalión" y pensamiento positivo), etc...

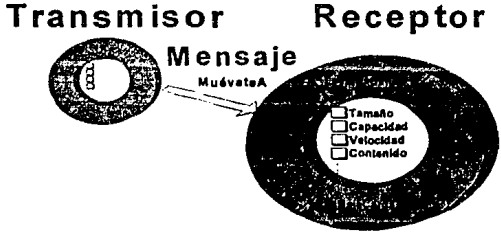
Mensajes en la naturaleza

Recordando que la interface de un objeto con el mundo exterior se compone de sus métodos (públicos):

Un mensaje es, simplemente, el requerimiento que hace un objeto a otro objeto, para que el segundo ejecute uno de sus métodos.[15]

Definición de Mensaje

Por definición, al objeto que envía el requerimiento se le llama objeto *transmisor* y al objeto que lo recibe se le llama el objeto receptor.



[15] OOIS, Taylor pp 49

El conjunto de todos los métodos del objeto determinan su comportamiento. Dado que el usuario del objeto sólo necesita conocer los métodos que le interesan para utilizarlo, la interface entre el objeto transmisor y el receptor se encuentra bien definida.

Los métodos definen el comportamiento del objeto

Como pudimos darnos cuenta, los objetos del mundo real pueden exhibir una gran variedad de interacciones de unos con otros. De esta gran variedad de comportamientos surge una pregunta interesante: ¿Cómo podemos representar toda esta variedad de interacciones en el software?.

Los objetos del mundo real,pueden interactuar de formas muy diferentes

Los autores del lenguaje Simula dieron una solución elegante: crear el concepto de mensaje. Simula es un lenguaje de programación especialmente diseñado para realizar simulaciones del mundo real, especialmente para simulación en el área de control. Simula tiene bloques fundamentales de construcción que podemos ver como objetos y la forma en que los objetos modifican su comportamiento es a través de mensajes. Un mensaje es solamente el nombre del método a ejecutar, seguido de los parámetros requeridos. Hablaremos de los parámetros en la siguiente sección.

Simula introduce el concepto de mensaje en los lenguajes de programación

3.IV.1. Anatomía de un mensaje

Estructuralmente, un mensaje consiste de tres partes:

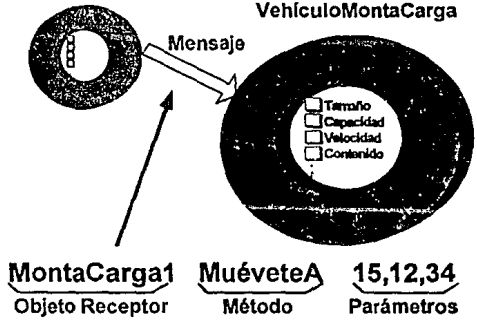
1) La identidad del objeto receptor,

2) El nombre del método que se desea ejecutar y

3) El conjunto de **parámetros** de ese método

Los parámetros pueden ser valores definidos o, inclusive, otros objetos (sus identificadores) que el método necesite para llevar al cabo su tarea.

Los parámetros son opcionales y generalmente varían, en número, de 0 a 4 ó 5.



Un mensaje consta de tres partes: objeto receptor, método a ejecutar y parámetros

El nombre de un método puede re-utilizarse sin problemas en otro tipo de objeto. Esto es, el método MuéveteA puede llamarse igual para el caso de que sea un montacargas, un autobús, un avión, una lancha, etc., siempre utilizando el mismo nombre. A lo anterior se le llama reutilización de nombres (**name overloading**).

Reutilización de nombres (name overloading)

Lo anterior es la esencia del **polimorfismo**, el cuál veremos en la siguiente sección.

Para un mensaje, la secuencia de los eventos es la siguiente:

Pasos en la ejecución de un mensaje

1) El transmisor envía el mensaje,

2) El receptor ejecuta el método, consumiendo los parámetros que le llegaron y finalmente

3) El receptor regresa al transmisor una respuesta (como un return de C), confirmando así que se recibió y que se ejecutó el mensaje.

3.IV.2. Polimorfismo

Dado que los objetos son independientes entre sí -característica del encapsulado-, es posible utilizar el mismo nombre de un método siempre y cuando se trate de objetos diferentes. Como vimos en la sección anterior, a este concepto se le llama reutilización de nombres (name overloading). Este concepto está estrechamente ligado con el polimorfismo.[16]

Podemos utilizar el mismo nombre en muchos métodos

Por ejemplo para imprimir, los instrumentos financieros: cheque, ordenDePago y chequeDeCaja. En programación tradicional debemos declarar funciones con nombres diferentes para cada tipo de instrumento:

```
imprimir_cheque( cheque )
imprimir_ordenDePago( ordenDePago )
imprimir_chequeDeCaja( chequeDeCaja )
```

En programación tradicional , todas las funciones deben tener nombres diferentes

Lo anterior genera una explosión en el número de funciones a manejar, haciendo que crezca la complejidad. En la POO podemos utilizar el mismo nombre: imprimir, sin tener que renombrar cada función. Reescribiendo para todos los instrumentos:

En la POO podemos utilizar el mismo nombre de un método para objetos diferentes (name overloading)

```
imprimir( cheque )
imprimir( ordenDePago )
imprimir( chequeDeCaja )
```

El compilador escoge la función que deseamos dependiendo del parámetro que estemos utilizando. El concepto de imprimir es el mismo, sólo que cada instrumento se imprime de manera diferente. Este es precisamente el concepto del **polimorfismo**.

[16] Algunos autores no hacen distinción entre ellos y lo consideran el mismo

El Polimorfismo se refiere a que cada objeto puede reaccionar de una manera diferente a un mismo mensaje.

Definición de Polimorfismo

Por ejemplo: considérese el mensaje "suma". Si se envía este mensaje a una orden de compra, puede significar: aumentar un nuevo tipo de artículo. Al enviárselo a un objeto departamento puede significar: dar de alta a nuevo empleado. Sin embargo la esencia es la misma: adicionar algo.

"suma" puede significar muchas cosas diferentes

Otro ejemplo de polimorfismo es la acción de comer en los animales. Todos los animales comen, pero cada animal come de manera diferente.

Todos los animales "comen", pero lo hacen de manera diferente

En los lenguajes de programación tradicionales ya se aplica el polimorfismo en los operadores de números pre-construidos (built in). El concepto de sumar es el mismo, ya sea que se trate de números enteros o de números fraccionarios. El compilador aplica el método de suma especifico, según el tipo de variables de que se trate. Este símil puede aplicarse también para "sumar" (concatenar) cadenas de caracteres, como lo hace basic y pascal; o para sumar dos objetos tipo número complejo, por ejemplo.

Los lenguajes tradicionales tienen polimorfismo en las operaciones built in (p.e. suma de enteros o fraccionarios)

Posteriormente retomaremos el polimorfismo y veremos la gran ventaja que tiene con las clases y las subclases.

3.V. Clase, clasificando y ordenando objetos

Los objetos comunicándose a través de mensajes forman la esencia de la orientación a objetos. Estos dos mecanismos solos, proveen la base para un lenguaje de programación poderoso. De echo, el lenguaje **Ada**, que utiliza el departamento de la defensa de USA, está construido con estos dos mecanismos.

Los objetos y mensajes son conceptos muy útiles

Sin embargo, es el concepto de **clase** el que incrementa dramáticamente lo útil de la orientación a objetos. Las clases mejoran el poder y la eficiencia de los objetos y los mensajes, en varios sentidos.

Pero las clases hacen la diferencia

Pero, ¿porqué el concepto de clase?. La razón principal es para expresar características comunes entre diferentes tipos de objetos, tratando así de lograr mayor flexibilidad, pero al mismo tiempo mayor facilidad en el mantenimiento al código. Estas características serán evidentes al terminar este capítulo.

Una clase agrupa objetos con características comunes, pero busca más flexibilidad y menor esfuerzo

3.V.1. Anatomía de una Clase

Como ya hemos dicho, una clase es una plantilla para un tipo específico de objetos. Si tenemos muchos objetos del mismo tipo, sólo es necesario definir una vez para toda la clase las características comunes de ese grupo de objetos. Haciendo una comparación con los lenguajes de programación tradicionales, una clase es como si definiéramos un nuevo tipo de variable preestablecida (bult-in), como son los enteros, reales o char. De esta manera, podemos tener variables tipo número complejo, elevador, avión, etc. Estos nuevos tipos de variables (clases), el compilador los tratará como cualquier otra estructura del lenguaje ya preestablecida. De echo la actividad principal de cualquier lenguaje de programación orientado a objetos es, precisamente, la definición de

Una clase es una plantilla de objetos similares

estas variables preestablecidas, mediante la definición de sus datos y de sus métodos.

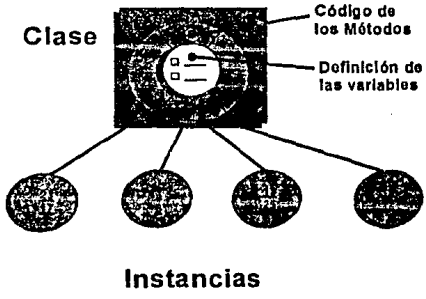
Por ejemplo, un sistema de compras tendrá no una si no miles de órdenes de compra. Sería tremendamente ineficiente duplicar el código en las definiciones de los datos y de los métodos para cada objeto orden de compra.

Por ejemplo: todos los objetos orden de compra pertenecen a una clase

Aquí es dónde las clases entran a escena. Las clases definen las características comunes de un grupo de objetos. Estos objetos son realizaciones o "**instancias**" de esa clase. Una analogía con los lenguajes de programación tradicionales decimos que la variable A es de tipo entero, entonces -en términos de la POO- el objeto A es una instancia de la clase entero.

Las clases eliminan la duplicidad

Otro ejemplo. Una clase llamada "OrdenDeCompra" define los métodos y datos asociados con todas las órdenes de compra. Las instancias de esta clase (objetos) contendrán solamente los valores de los datos para cada orden de compra, gráficamente:



Una clase es una plantilla de objetos comunes

En pocas palabras, podemos decir que: ***un objeto es la instancia de una clase***. Como tal, es una combinación de características que comparten varios objetos, expresadas dentro de la clase, y de un estado único representado por los valores que toman los datos de la instancia. Dada la definición anterior de objeto, es perfectamente correcto decir que un objeto tiene métodos asociados, aun cuando estos métodos estén en la clase a la que pertenece el objeto.

Un objeto es la instancia de una clase

Otra forma de ver la relación entre objetos y clases es la siguiente. Aquellos que están familiarizados con los sistemas de bases de datos, pueden pensar que las clases son como tablas y los objetos como registros. Una tabla define los nombres y los tipos de los datos de la información que contendrá, tal y como lo hace una clase. Como una instancia (objeto), un registro tiene valores específicos para un renglón de esa tabla. La principal diferencia en este esquema, a nivel conceptual, es que las clases, además de los datos, también contienen métodos.

Las instancias son como registros en una tabla

Servicio	Horas	Frecuencia
S1234	.5	6
S3321	1.5	3
S1422	3	7

Instancias como registros

Dado lo anterior, y aunque no es el objetivo principal de este texto, a continuación se presenta un pequeño ejemplo de la declaración de una clase en C++. El lector

Ejemplo de una clase en C++

que no esté interesado en los aspectos de programación puede continuar leyendo la siguiente sección.

En C++ la idea principal del diseñador del C++[17] fue aprovechar la infraestructura del lenguaje C. Por esta razón, no existen cambios drásticos en las estructuras del lenguaje. De echo, se conserva el lenguaje C casi íntegramente dentro del C++. Para los lectores familiarizados con la sintaxis del lenguaje C, una clase es virtualmente lo mismo que una estructura (o un record en PASCAL).

Una clase es como un struct de C, o un record de PASCAL

Hemos visto que una clase se compone de la definición de los datos (que representan su estado) y de métodos (que definen su comportamiento).

Pero además de datos, contiene métodos (funciones)

La "nueva" forma de llamar a la estructura (struct) en C++ es clase (class). Sin embargo podemos utilizar la palabra struct, con algunos cambios en la declaración de los datos y métodos. Dado que éste es un ejemplo, no explicaremos a ese detalle.

En C++, class es similar a un struct

Tomemos el siguiente ejemplo. Para representar en C++ el objeto del mundo real Elevador, una posible construcción de la clase sería de la siguiente manera:

```
class Elevador {
// Datos:
  int  funcionando;      // ¿está funcionando?
  int  piso;             // piso en que se encuentra

// Métodos:
  enQuePiso();           // método que dice el piso actual
  veAPiso( int pisoNo ); // ir a un piso determinado
  funciona();            // método: ¿está funcionando?
};
```

Clase Elevador en C++

[17] El diseñador de C++ es Bajme Stroustrup de la AT&T

Como podemos apreciar, la declaración de arriba se compone de **datos: funcionando y piso** que representan su **estado,** y de **métodos: enQuePiso, veAPiso y funciona** que representan su **comportamiento.**

Datos: funcionando y piso; métodos: enQuePiso, veAPiso y funciona

Los datos que contiene una clase pueden ser enteros, caracteres, así como también otras clases. A esto último se le llama objetos compuestos (composite objects). Este tipo de objetos de verán en la sección 3.IV.5: objetos compuestos.

Objetos compuestos (composite objects)

3.V.2. Modularidad

La **modularidad** se refiere a la acción de agrupar clases, cuando éstas se relacionen entre sí, en un mismo módulo o sub-programa. Esto se refiere, en programación tradicional, al uso de las unidades (units) de PASCAL o a los módulos de programas en C (programas con extensión OBJ).

Los módulos son como las unidades (units) de PASCAL

La modularización es la descomposición del problema en partes más pequeñas. De estas partes, tomamos las más relacionadas entre sí y las ponemos en módulos o subprogramas. Las clases que se encuentran en un mismo módulo deben estar altamente **cohesionadas** (agrupando abstracciones altamente relacionadas), pero el **acoplamiento** entre módulos debe ser mínimo (**minimizar las** dependencias entre módulos). Dado lo anterior podemos definir:

La modularidad es la propiedad que exhibe un sistema, tras su descomposición en módulos cohesionados y mínimamente acoplados.[18]

Definición de modularidad

El objetivo principal de la modularización es la de reducir los costos (tiempo y dinero) del software, mediante módulos que pueden ser diseñados y revisados de manera independiente.

El objetivo es reducir costos en el diseño y el mantenimiento

[18] OOD Booch, pp 52

3.V.3. Jerarquización

Los conceptos de abstracción, encapsulado y modularidad forman una sinergia. Los tres proporcionan los beneficios del último párrafo de la sección anterior. La abstracción es buena para manejar la complejidad pero, con excepción de los sistemas más triviales, generalmente encontramos más niveles de abstracción de lo que podemos entender a la vez. El encapsulamiento también ayuda al manejo de la complejidad, porque esconde la vista interna de esta complejidad. Pero con todo lo anterior no es suficiente, por lo que necesitamos una forma de dividir en niveles para reducir la complejidad.

Los conceptos de abstracción, encapsulado y modularidad están íntimamente relacionados

Todo lo anterior nos conduce a que debemos manejar a un sistema complejo mediante una **jerarquización**. La jerarquización es la consecuencia natural de manejar varias abstracciones (clases) comunes. La identificación de esta jerarquía simplifica grandemente el entendimiento de nuestro problema.

Pero para manejar mejor la complejidad los humanos jerarquizamos

Una vez que entendemos un nivel de abstracción, podemos bajar (o subir) de nivel. Por ejemplo, la mecánica de Newton se utiliza para describir una gran variedad de fenómenos mecánicos, pero no sirve para todos los casos. En cambio, la mecánica cuántica puede explicar cualquier tipo de fenómeno mecánico; sólo que es más complicado manejarla para la explicación de la mecánica de Newton. Son simplemente niveles diferentes de abstracción.

Manejamos una jerarquía -niveles de abstracción- para entender más fácilmente a los sistemas complejos

Hemos visto, pues, que para manejar la complejidad los humanos jerarquizamos las abstracciones. Los dos tipos de jerarquías más importante son:

Tipos principales de jerarquización

1) ***Tipo de***

2) ***Parte de***

Podemos decir, brevemente, que la jerarquización **tipo de** define nuevas abstracciones (clases), en términos de elementos conocidos. Por ejemplo, para definir a un objeto "sillón" -conociendo lo que es una silla-, podemos decir: *un sillón es como una silla, solo que es más grande y más cómodo.*

Jerarquización tipo de

La jerarquización **parte de** la hemos estado manejando durante la evolución de este texto como objetos compuestos (composite objects). Casi cualquier objeto de la realidad está compuesto de otros. Por ejemplo, un avión se compone, a grandes rasgos, de fuselaje, motor y sensores; a su vez, el fuselaje se compone de alas, alerones, etc...

Jerarquización parte de

A continuación cada tipo de jerarquización serán objeto de estudio en las secciones siguientes.

3.V.4. Herencia, jerarquización tipo de

La herencia es la base principal en la reutilización del software en la POO. Este mecanismo permite definir una clase añadiendo características (datos y métodos) a una clase ya existente. Pero veamos paso a paso la herencia, recordando siempre que es una jerarquización **tipo de**, con el fin de facilitar el manejo de la complejidad. De aquí en adelante cuando hablemos de **herencia** es lo mismo que la jerarquización **tipo de**.

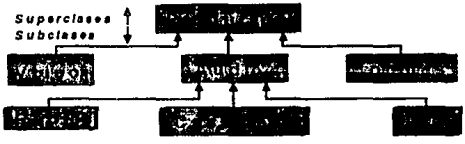
La herencia es la base principal en la reutilización de código

Especialización de Clases. En el mundo real existe una gran variedad de posibles clases, tantas como problemas se nos presenten. Esta gran variedad de clases no son aisladas. De echo, la mayoría de las clases que podemos definir son un caso especial de otra, formando así lo que llamamos una **jerarquía de clases.**

En la herencia, las clases se definen como casos especiales de otras clases

Por ejemplo, los productos que ofrece una compañía pueden ser definidos como una versión especializada de productos más generales, éstos a su vez pueden considerarse casos especiales de la clase más general: *Producto*. Formalmente, a estas especializaciones se les conocen como **subclases** (clases hijo), las clases de las que son casos especiales, se les conocen como **superclases** (clases padres).

Ejemplo: clases de productos



Especialización de Clases

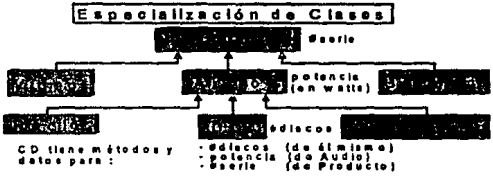
Posible jerarquía de clases en productos de audio y video

La ventaja de definir las clases en una jerarquía es que, a través de la herencia, todos los casos especiales comparten las características de la superclase. Del ejemplo de arriba, un objeto *CD* (compact disc player), heredaría todos los datos y métodos de un componente de *Audio*, el cuál a su vez heredaría todos los métodos y datos de la clase *Producto*.

La herencia permite que las clases compartan datos y métodos

Eliminación de redundancia con la herencia. La herencia es un mecanismo muy eficiente, porque cada método y cada variable se define una sola vez, en el nivel más general que aplique. Volviendo a nuestro ejemplo, todo producto comercial, de audio o video, debe manejar un número de serie, por lo que lo alojamos como dato en *Producto* (#serie) junto con los métodos apropiados para su manejo. De la misma manera, todos los productos de *Audio* manejan una cierta potencia -en watts-, y un *CD* maneja un cierto número de discos compactos.

Por ejemplo, un CD hereda varias propiedades de la jerarquía



Herencia de datos y métodos

Dado que CD hereda los datos y métodos de *Producto* y de *Audio*, sólo tiene que adicionar los datos y métodos para él mismo (#discos) sin preocuparse por los que hereda. De esta manera podemos aprovechar la plataforma existente, mejorando así la productividad en el desarrollo de los sistemas.

Cuando se agrega un nuevo objeto sólo es necesario adicionar sus datos y métodos

Se hace evidente ahora que un *CD* es un **tipo de** componente de *Audio* y que un componente de *Audio* es un **tipo de** *Producto*.

*Una subclase es un **tipo de** su superclase*

Del ejemplo anterior, podemos ver que la jerarquización de las clases van de lo más general a lo más particular. Dado que la subclase hereda los datos y métodos de su superclase, las jerarquizaciones siempre deben ser, como ya hemos dicho, de lo general a lo particular.

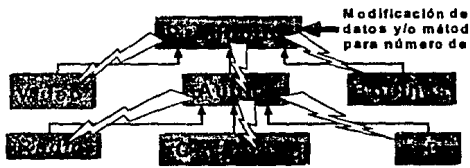
La herencia debe ser de lo general a lo particular

Propagación de los cambios. Al utilizar la herencia, el software se vuelve más fácil de modificar. Suponga que se vendieron más aparatos electrónicos de los que nunca se imaginó, y que comienzan a acabarse los números de serie. De alguna forma debemos extender la capacidad de la variable *#serie* y modificar los métodos que accesan este dato en *Producto*, para mantener la integridad de la clase.

La herencia hace que los cambios sean más fáciles

Dada la forma en que trabaja la herencia, sólo necesitamos realizar cambios en una clase. Como toda la variedad de productos que tenemos heredan de la clase *Producto*, las modificaciones -el *#serie* y/o los métodos del número de serie- "cambian" inmediatamente en todas las subclases que los heredan, sin necesidad de hacer programación adicional. Las subclases utilizan automáticamente estos nuevos métodos cada vez que los necesitan.

Ejemplo: cambio del número de serie



Los cambios en los datos y métodos se propagan automáticamente

Todas las subclases adoptan inmediatamente los cambios

Mediante la herencia, **mejoramos la productividad** en el desarrollo del software, minimizando los cambios en el código y reduciendo el tiempo de volver a probarlo - siempre que respetemos el encapsulado-.

Mejorando así la productividad

También es posible re-definir (over-ride) una característica heredada, de manera que lo que heredamos lo podemos modificar. Así mismo, estos cambios los heredarán mis subclases automáticamente. Para mantener la generalidad de este texto, no profundizamos en este tema.

También es posible re-definir (over-ride) datos y métodos heredados

Pero, ¿cómo sabemos que esta jerarquización es la correcta?. La jerarquización de clases (que implica la herencia) depende de la experiencia del diseñador y del contexto de la aplicación que estemos realizando. Quizás la jerarquización, de nuestro ejemplo anterior, no nos convenga en otro sistema, aunque se utilicen las mismas clases.

La experiencia del diseñador y el contexto del sistema son decisivos en la estructura de la herencia

3.V.5. Clases, mensajes y polimorfismo (el problema de la instrucción CASE)

Clases, mensajes y polimorfismo

A continuación discutiremos otro beneficio de utilizar mensajes y una jerarquía de clases. La mayoría de los lenguajes de programación modernos cuentan con una instrucción de ramificación, también llamada CASE o SWITCH, que significa: **"en caso de"**. Esta determina la acción que debe tomarse en función de una cierta condición. Esto se realiza, típicamente, mediante la comparación de una condición numérica de igualdad con un valor preestablecido.

La instrucción CASE controla el flujo de un programa

Por ejemplo, suponga que tiene un portafolio electrónico de instrumentos financieros, y que desea calcular su valor. Para hacer esto, podemos escribir un programa que recorra cada uno de los instrumentos del portafolio, calcule su valor y lo sume a un total global. Sin embargo la forma de calcular el valor de cada instrumento será, definitivamente, diferente. Para resolver este problema, el programa necesitará una instrucción CASE para asegurarse que la rutina correcta ha sido llamada.

Ejemplo: evaluación de un portafolio financiero

```
Si el instrumento es:
    Cheque          entonces   valor_Cheques(...
    ...
    PlazoFijo       entonces   valor_PlazoFijo(...
    ...
    TarjetaCrédito  entonces   valor_TarjetaCrédito(...
    ...
```

Ejemplo de una instrucción CASE

La instrucción CASE es aceptada y, muchas veces, utilizada con gran aceptación y respeto. De echo, es la mejor forma de atacar este problema con la programación estructurada. Sin embargo, la instrucción CASE genera algunos problemas:

La instrucción CASE crea varios problemas

- Una instrucción CASE es **tediosa de escribir** y constituye una buena parte del tamaño de los programas
- La instrucción CASE es **lenta al ejecutarse**, dado que la computadora debe ejecutar un cierto número de comparaciones para decidir la ramificación que tomará
- Será necesario la **repetición del CASE** en cada parte del programa en que se use. Del ejemplo anterior, necesitaremos insertar el mismo código -o al menos llamar una función- cada vez que necesitemos calcular el valor del portafolio.
- Con la instrucción CASE, **todas las opciones están en código (hard-coded)**. Si se desea adicionar o borrar una nueva opción al CASE, debemos modificar cada una de las partes del código en donde aparece.

El último problema es el más serio. Un sistema de gran magnitud puede tener cientos o miles de instrucciones CASE, de las cuáles depende fuertemente la estructura de muchos programas. Si cambiamos cualquiera de estas decisiones -por ejemplo, un nuevo instrumento financiero- puede requerir del mantenimiento de muchos programas, con problemas implícitos de re-desarrollo y de re-probar código, sin mencionar los problemas -bugs- que pudiéramos agregar.

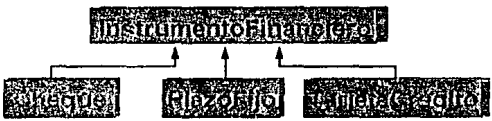
El problema más serio es el último: la rigidez

Flexibilidad a través de los mensajes. Al utilizar los mensajes en combinación con la jerarquía que proporciona la herencia, resolvemos este problema en una forma elegante. Cada instrumento financiero sabe como calcular su propio valor. El método para hacerlo se llama igual: "valor", para cada instrumento financiero.

Los mensajes ofrecen una solución simple a este problema

Gracias al polimorfismo, no importa con qué tipo de instrumento financiero estemos tratando, el método correcto se llamará automáticamente.

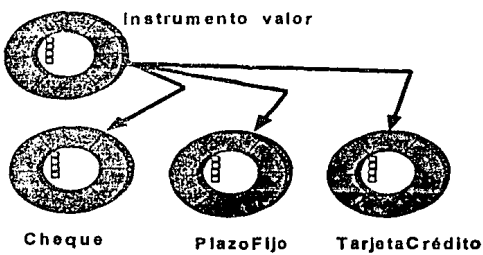
La siguiente figura ilustra una posible jerarquía de clases para los instrumentos financieros.



Jerarquía de instrumentos financieros

En una jerarquía de clases, una variable de una clase puede contener instancias (objetos) de ella misma o de cualquiera de sus subclases. Para nuestro ejemplo, digamos que tenemos una variable llamada *instrumento* que es de tipo *InstrumentoFinanciero*. De acuerdo a lo que hemos dicho, podemos instanciar a *instrumento* con un objeto de cualquiera de las clases: *Cheque, PlazoFijo o TarjetaCrédito*. Al pedir el valor de *instrumento*, el método correcto será llamado sin necesidad de utilizar una instrucción CASE, gráficamente:

Una variable de una clase puede contener instancias de ella misma y de sus subclases



Instrumento llama al método "valor" correcto, según el objeto de que se trate

En efecto, el polimorfismo elimina, en la mayoría de los casos, la necesidad de utilizar la instrucción CASE. Todo el código requerido para seleccionar la rutina correcta se reduce a una sola instrucción: *instrumento valor*. Esta forma de llamar directamente al método correcto, resuelve todos los problemas asociados con la instrucción CASE:

El CASE anterior se reemplaza por la instrucción: instrumento valor

+ Es **trivial al escribirla**: instrumento valor
+ **Se ejecuta rápidamente**, el programa ejecuta directamente el método correcto

Esto supera por mucho al CASE

+ Hace los **programas más pequeños** y más fáciles de mantener, porque elimina la duplicidad de las instrucciones CASE.
+ Hace los programas **más fácilmente modificables** dado que pueden agregarse opciones de ejecución, simplemente definiendo nuevas clases en la jerarquía.

Este último beneficio es el más poderoso porque resuelve el problema más difícil. En la orientación a objetos no es necesario codificar todas las opciones directamente en el programa. Si se desea agregar otras opciones, tales como nuevos instrumentos financieros, sólo necesitamos heredar de la clase *InstrumentoFinanciero*, implantar el método "valor" e instanciar a *instrumento* con un objeto de esa nueva clase.

La principal ventaja es la fácil modificación

3.V.6. Herencia Múltiple

Algunos lenguajes permiten a una clase heredar de varias superclases

Estrictamente hablando, a la herencia que hemos visto hasta este momento se le llama **herencia simple,** porque cada clase tiene a lo más una superclase. En muchos lenguajes de programación es posible que una clase herede de más de una superclase (como lo hace C++).

La herencia múltiple puede llevar a más complicaciones

Aunque la herencia múltiple puede simplificar ciertas situaciones, también puede provocar complicaciones. Por ejemplo, supongamos la situación en que heredo de dos superclases, si ambas tienen un método que se llama igual -imprime-, ¿cuál de los dos método heredaré?.

Y muchas veces es mal utilizada

Otro problema que trae la herencia múltiple es que muchas veces es mal utilizada. Para aplicarla correctamente, debemos estar seguros de que un objeto es realmente el resultado de dos o más clases.

La herencia múltiple, en general, es buena pero debe utilizarse con cuidado

Dadas las complicaciones anteriores, muchas personas se preguntan si los beneficios de la herencia múltiple son más que sus problemas -algunos la rechazan fuertemente. La tendencia en la orientación a objetos parece ser el soportarla, pero es definitivamente una característica que debe utilizarse juiciosamente.

Debemos evitar la "fiebre" de la herencia

Aunque la herencia es una herramienta muy valiosa, no es aplicable para todos los casos. De la misma manera que una persona que conoce por primera vez un martillo cree que todo es un clavo, los nuevos estudiantes de esta área piensan que todo es "heredable"; se debe tener cuidado con la "fiebre de la herencia".

*Preguntar si es un **tipo de***

Una regla sencilla para saber si una clase es o no heredable, es simplemente preguntar si la clase que queremos agregar es un **tipo de** la superclase de la que queremos heredar, si es así, proceda con la herencia. Por

ejemplo, una camioneta es un **tipo de** vehículo, pero una rueda no es un **tipo de** automóvil.

Sin embargo un automóvil tiene ruedas o, lo que es lo mismo, una rueda es una **parte de** un automóvil. Veremos precisamente esta jerarquización en la siguiente sección.

*Siguiente sección: jerarquización **parte de***

3.V.7. Objetos Compuestos, jerarquización parte de

Recordemos las jerarquizaciones básicas de la sección 3.IV.3. En la sección anterior vimos la jerarquización **tipo de**, representada por la herencia. Como ahora sabemos, aunque la herencia es de gran ayuda en la programación orientada a objetos, no siempre es aplicable. Otra gran rama, es la jerarquización **parte de**. Básicamente, esta jerarquización es la que hacemos al distinguir a un objeto y las partes que lo componen.

*La jerarquización **parte de**, construye a un objeto por medio de sus partes*

Por ejemplo, un avión -comercial, militar, de carga, etc.- consta de partes bien definidas como son: motores, fuselaje, alas, etc ... Estas a su vez se subdividen; por ejemplo, un motor consta de un compresor, combustor, turbina, etc y así sucesivamente. Dependiendo del nivel de abstracción que necesitemos, podemos llegar a hablar de tornillos, tuercas o inclusive átomos.

Ejemplo: un avión se compone de motores, fuselaje, alas, etc

Objetos que contienen objetos. Si los objetos sólo pudieran contener datos simples, como nombres y números, su utilidad sería muy limitada. Lo que hace gran parte del poder de los objetos es que pueden contener también otros objetos. A los objetos que contienen otros objetos se les conocen como: **objetos compuestos** (composite objects).

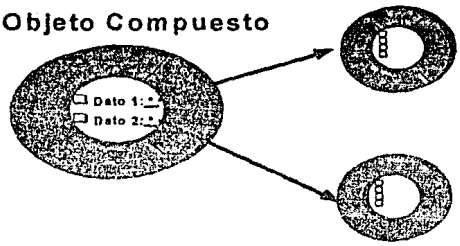
Objetos compuestos: objetos que pueden contener otros objetos

Así como la **herencia** representa a la jerarquización **tipo de**, los **objetos compuestos** representan a la jerarquización **parte de**.

*Los objetos compuestos representan a la jerarquización **parte de***

En la mayoría de los sistemas, los objetos compuestos no "contienen" literalmente otros objetos, si no que contienen datos que apuntan a una referencia a ese objeto (identificación del objeto: **object id**).

En realidad, se tiene un apuntador a los objetos que contiene



Las principales ventajas que proporcionan los objetos compuestos son dos:

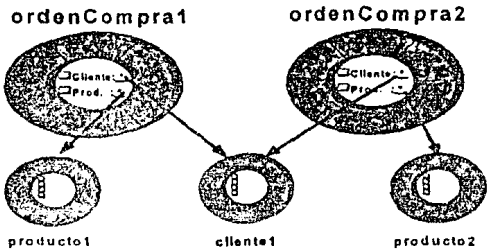
1) Los objetos **"contenidos" cambian más fácilmente**, tanto en tamaño como en composición, sin afectar la composición de los objetos que los contienen. Esto hace que el mantenimiento de los sistemas, que utilizan este anidamiento, sea más fácil y rápido.

Ventajas de los objetos compuestos

2) Los objetos **"contenidos" pueden ser parte de cualquier número de objetos compuestos**, en vez de estar ligados a un sólo objeto, evitando así la duplicidad.

Por ejemplo, la figura siguiente muestra dos objetos que representan a órdenes de compra. Constan de datos que contienen la información de los clientes y de los productos adquiridos. En lugar de que cada objeto OrdenCompra tenga los datos separadamente, sólo almacenamos referencias a los objetos que lo componen en forma de identificación del objeto (object id).

Ejemplo: órdenes de compra



Dado que las órdenes de compra tienen solamente las referencias al producto que contiene -y no al producto en sí-, el producto es libre de cambiar sin afectar al objeto OrdenCompra. Por ejemplo, podríamos dar al producto un campo para una descripción extendida, incrementando así el tamaño del objeto producto, pero sin afectar el tamaño del objeto OrdenCompra.

Cada producto -y cliente- es libre de cambiar

De la misma manera, un cliente puede participar en cualquier número de órdenes de pago, evitando así la duplicidad de información. Esta constituye otra ventaja de que los objetos compuestos contengan referencias, en lugar de los objetos físicamente.

Cada cliente -y producto- puede estar en muchas órdenes de compra

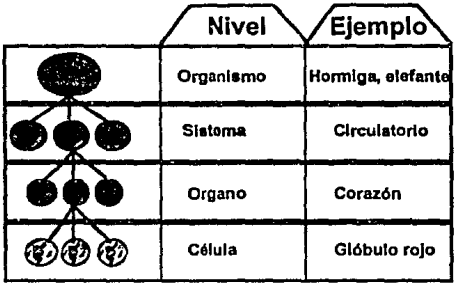
Múltiple anidamiento de objetos. Los objetos contenidos en objetos compuestos pueden a su vez ser objetos compuestos, y este anidamiento puede llegar a cualquier nivel. Esto quiere decir que podemos construir estructuras tan complejas como deseemos -recordemos que el objetivo de la jerarquía es manejar mejor a la complejidad. Esta apreciación es importante debido a que generalmente necesitamos más de un nivel de

Puede haber N niveles de anidamiento

modularización para evitar el caos, especialmente en sistemas de gran escala.

De nuevo podemos aprender mucho de la forma en que trabaja la naturaleza. Considere la forma en que la naturaleza maneja a los seres vivos (estructuras de alto nivel). Un ser vivo no es un montón de células revueltas, si no que las células se "organizan" en unidades funcionales llamadas órganos, tales como el corazón y el cerebro. Los órganos a su vez se agrupan en sistemas, tales como el circulatorio y el nervioso. Finalmente, estos sistemas constituyen a un organismo, que funciona como un todo.

La naturaleza utiliza ampliamente esta técnica



Cada nivel es entendible, casi independientemente de los demás

Los cuatro niveles de jerarquización de los organismos es muy conveniente para los biólogos, ya que trae un orden claro a lo que podría ser una colección caótica de interacciones entre células. De particular valor es que podemos entender un nivel sin tener que ver con los niveles alternos. Por ejemplo, podemos entender el funcionamiento de nuestro cuerpo como una interacción entre nuestros sistemas, sin tener que entrar a los detalles de los órganos ni de las células.

Aunque sólo podemos especular sobre la razón de esta modularidad, lo más posible es que la naturaleza actuó de esta manera porque facilita la evolución de los organismos. Entre más independientes sean los órganos entre sí, más fácilmente podrán evolucionar individualmente sin perturbar la operación de otros órganos y sistemas.

La división en niveles facilita la independencia de los cambios

Estas son las razones por las cuáles debemos utilizar niveles múltiples en la construcción de sistemas de cómputo complejos. De echo, podemos extender esta lógica a cualquier número de niveles, aunque 4 ó 5 son generalmente suficientes para las aplicaciones del mundo real.

Por eso debemos utilizar niveles múltiples en el software

Con esto terminamos el capítulo y la presentación de los conceptos de la orientación a objetos. En el siguiente capítulo presentamos un análisis y diseño general del sistema propuesto. Posteriormente concluimos presentando las ventajas y desventajas tanto del sistema propuesto como de la orientación a objetos.

Con esto concluimos el capítulo

Capítulo 4:

Implantación del Sistema Propuesto

Tanto que hay por hacer y tan poco hecho.

- Cecil Rhodes.

El mundo es del entusiasta que se mantiene sereno.

- William McFee.

Implantación del Sistema Propuesto

4.I. Introducción

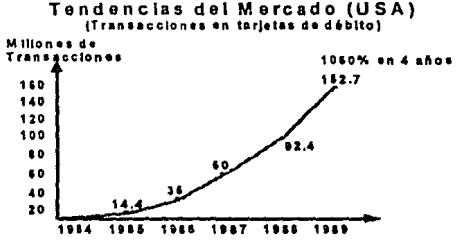
En la actualidad nos encontramos en un medio ambiente comercial sumamente competitivo. Existe todo tipo de establecimientos para la compra de bienes y servicios. Estos establecimientos nos ofrecen prácticamente todos los artículos que necesitamos para vivir, desde alimentos hasta equipos electrónicos.

Estamos en un medio comercial sumamente competitivo

Una gran parte de la población nos hemos acostumbrado al uso de las tarjetas bancarias -o simplemente tarjetas-, tanto de crédito como de débito, para el pago de los bienes y servicios de los que hablábamos en el párrafo anterior. No cabe duda de que esta forma de pago toma cada vez más importancia, tanto en México como en el resto del mundo, y que cada vez será más importante conforme pasen los años.

La utilización de las tarjetas -de crédito y débito- es cada vez mayor

Aunque no contamos con información específica del mercado Mexicano, éste sigue la mayoría de las tendencias que han tenido los E.U.A.. Veamos la siguiente gráfica:



Tendencias en el mercado de U.S.A., tarjetas de débito

En México vivimos un incremento importante en las transacciones por tarjeta cuando se introdujeron las tarjetas de débito: Invermático e inversión inmediata de Banamex y Bancomer respectivamente.

Cualquier comercio que pretenda ser mínimamente competitivo, debe aceptar esta forma de pago. La mayoría de las personas que manejamos tarjetas preferimos utilizar esta forma de pago, porque -en general- son más seguras y podemos llevar un mejor control de nuestros gastos. Claro que si nos sobre giramos, como cualquier otra forma de pago, nos buscamos problemas de solvencia económica.

Para competir, los comercios deben aceptar esta forma de pago

Hemos estado hablando de dos tipos de tarjeta hasta ahora, las de crédito y las de débito. Las tarjetas de crédito fueron creadas antes que las de débito. Las tarjetas de crédito tienen una línea de crédito asociada, mientras que las tarjetas de débito no. De manera que la disposición de dinero de una tarjeta de débito debe ser siempre verificada contra su saldo (disposición menor o igual al saldo), mientras que para la tarjeta de crédito se verifica contra su saldo disponible (esta cantidad y la acumulada no debe rebasar el **límite de crédito**).

Las tarjetas de crédito tienen un límite de crédito asociado, las de débito no

A continuación definiremos algunos conceptos. Llamamos **emisor** a la institución financiera que otorgó la tarjeta -de crédito o débito- al **cliente**. Este cliente tiene un contrato con el emisor, haciéndose responsable del buen uso de la tarjeta y de los pagos -o depósitos- correspondientes a su tarjeta. Decimos que el **comercio** es la institución a la cuál el cliente compra bienes o servicios. El comercio se encarga de realizar el cobro al emisor. El emisor se encarga de la transferencia de fondos entre el cliente y el comercio, cobrando por este servicio una comisión.

Definición de emisor, cliente y comercio

4.II. Proceso de Pago con Tarjeta

Sin importar el tipo de tarjeta que se trate (de crédito o débito), el proceso del pago es el mismo. A continuación describiremos el proceso del pago de un bien o servicio por medio de una tarjeta. Las personas que estén familiarizadas con este proceso, pueden continuar con la siguiente sección si así lo desean. Cabe aclarar que este proceso es bastante escueto y que se explica desde el punto de vista de un observador externo.

1) **Deseo de pagar con tarjeta.** El cliente desea pagar un bien o servicio que adquirió. El cliente entrega su tarjeta al cajero del comercio. El cajero toma la tarjeta y trata de conseguir una autorización con el emisor correspondiente.

2) **Proceso de autorización.** El emisor de la tarjeta puede aceptar o rechazar esa transacción. En caso de aceptarla, se proporciona un número de autorización.[19]

3) **Comprobante de la operación.** Se elabora el comprobante de la operación (voucher), ya sea manual o automáticamente, para que lo firme el cliente. Una vez firmado, el cajero verifica la firma y le entrega una copia al cliente.

4) **Afectación Contable.** El emisor realiza la transferencia de fondos de la cuenta del cliente a la cuenta del comercio, cobrando por esto una comisión que generalmente paga el comercio. La afectación contable puede realizarse al momento de pedir la autorización, al final del día o al depositar los vouchers en el banco.

Proceso de pago con tarjeta

[19] Se detalla este punto en la sección: proceso de autorización

4.III. Dispositivos POS

Dispositivos POS

Para la atención a las autorizaciones que necesitan los comercios, existe una gran gama de marcas de dispositivos a los que se les llaman dispositivos TPV (terminal punto de venta) o, de sus siglas en inglés, dispositivos POS (Point Of Service).

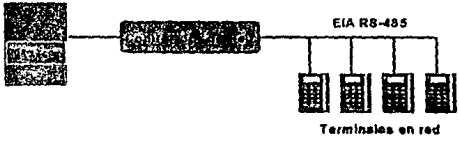
Clasificación de los dispositivos POS

A pesar de la diversidad de los dispositivos POS, podemos clasificarlos, en general, en tres grandes grupos:

1) **Terminales individuales de llamado automático.** También llamadas en inglés terminales dial-up. Este tipo de terminales marcan en la red pública telefónica el número correspondiente para cada emisor. Una vez conectados con el equipo central, le transmite los datos de la tarjeta a autorizar (p.e. no_cta, fecha_vencimiento, cantidad). El equipo central le contesta con una aprobación de la transacción, proporcionando un número de autorización. La principal ventaja de este tipo de terminales es el poder afectar el saldo del cliente de inmediato. El problema principal es que en horas pico, y aun en poco tráfico de llamadas, el tiempo de respuesta es demasiado elevado para un ambiente transaccional.

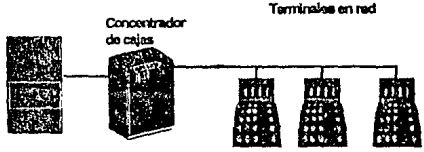
La comunicación es, generalmente, asíncrona a bajas velocidades (300 bps en México y 1200 en USA).

2) **Redes alternas de terminales.** Actualmente en México existen muchas cajas registradoras. Sin embargo, en la mayoría de los casos, estas cajas registradoras no cuentan con la infraestructura necesaria para procesar transacciones de tarjetas, por lo que se hace necesario tener una red alterna de terminales POS. Esta red alterna es, típicamente, una configuración de bus común con la interface RS-485, a la que se conectan todas las terminales de la red. Las terminales son controladas por un dispositivo llamado controlador de red con un protocolo del tipo poll-select. Las funciones principales de un controlador de red son: i) "hablar" con la red y ii) "hablar" con el host (equipo central de autorizaciones). El controlador de la red se encuentra conectado al host mediante una línea de conexión permanente (radio, línea privada, etc...).



Las ventajas principales es la velocidad de respuesta y que se tiene una línea compartida hasta el host. Los proveedores más conocidos de esta tecnología, al igual que las terminales dial-up, son: Verifone y Hypercom, entre otros.

3) **Cajas registradoras.** La siguiente gran rama de clasificación son las cajas registradoras con lectores, impresoras e infraestructura de software necesaria para manejar directamente las transacciones de tarjeta. Estas cajas empiezan a funcionar en todo México, principalmente en las grandes tiendas departamentales. La línea de cajas se conecta a un concentrador de las mismas, éste a su vez se conecta a un switch de autorización o directamente a los emisores para conseguir las autorizaciones. Los proveedores líderes de cajas registradoras son NCR, IBM e ICL, entre otras.



4.IV. Proceso de Autorización

Tenemos dos formas de autorizar una transacción

Sin importar el tipo de dispositivo punto de venta que estemos utilizando, cuando pagamos con tarjeta, la autorización -desde el punto de vista del comercio- puede realizarse de dos maneras:

En línea (online)

1) **Autorización en línea (online).** El dispositivo POS se conecta directamente al equipo de autorizaciones del emisor. El equipo central consulta el saldo real del cliente para autorizar la transacción. Definitivamente esta es la mejor forma de hacer una autorización de tarjeta, sin embargo, puede ser demasiado lenta (dial-up), demasiado cara (línea privada) o simplemente imposible (no existe infraestructura), por lo que se hace necesario la:

Fuera de línea (offline)

2) **Autorización fuera de línea (offline).** La autorización offline se refiere a que no se realiza una conexión directa con el emisor de la tarjeta. Generalmente este tipo de autorización se hace por dos vías: i) llamar verbalmente al emisor y ii) consultar el boletín de tarjetas prohibitivas (archivo negativo), que contienen -típicamente- las tarjetas robadas, perdidas y sobregiradas.

En la mayoría de los países Latinoamericanos es necesaria la autorización offline

En muchas ocasiones es imposible manejar una comunicación en línea con el emisor, por lo que la opción de autorización fuera de línea se vuelve indispensable, principalmente en países como México y los de toda Latinoamérica.

Llamar verbalmente al emisor es trivial, pero lento

Para la primera forma de autorización fuera de línea -llamar verbalmente al emisor-, el proceso es trivial. Sólo es necesario llamar al emisor correspondiente dando el número de la tarjeta, la fecha de vencimiento y la

cantidad. En caso de aprobarse, el emisor contesta verbalmente el número de autorización de esa transacción. Sin embargo, el tiempo requerido es demasiado, para algunas situaciones, por lo que este método es solamente efectivo cuando hay un volumen y tráfico de transacciones reducido.

La segunda opción de autorizar una transacción fuera de línea -consulta del boletín- es un poco más elaborada. Los pasos a seguir son:

1) Verificar que la tarjeta no esté vencida

2) Que la cantidad a autorizar no pase del límite de piso. Este límite de piso (cantidad máxima para autorizar) es asignado por el emisor dependiendo del tipo de tarjeta que sea. Si sobre pasa este límite de piso, es necesario llamar verbalmente al emisor

3) Asegurar que el número de la tarjeta no se encuentre entre las boletinadas

4) En caso de cumplir con las condiciones anteriores, se toma la identificación del boletín correspondiente como el número de autorización

5) El cajero imprime el comprobante de la operación para que el cliente lo firme. Después de que el cliente firma el comprobante, el cajero verifica la firma, terminando así la transacción

Pasos en una autorización offline por boletín

4.V. Introducción al Sistema Propuesto

Problemas de los dispositivos POS

Problemas con los dispositivos POS. Aunque los dispositivos POS incrementan, generalmente, la velocidad de respuesta para obtener una autorización, se presentan algunos problemas en su uso, a saber:

1) Las terminales dial-up dependen de la red telefónica para su buen funcionamiento (en México este servicio es poco eficiente). En México, el tiempo de marcado, conexión y transmisión de los datos se encuentra en el rango de 1 a 5 min. (por reintentos en el marcaje). Además en horas picos es poco menos que imposible comunicarse al centro de autorizaciones. Todo lo anterior hace poco conveniente su utilización en un intenso tráfico transaccional.

Las terminales dial-up son lentas y dependientes del servicio telefónico

2) Las redes de terminales necesitan líneas privadas dedicadas, o ancho de banda en radiofrequencia, para cada uno de los emisores que maneje. Esto implica una mayor "inteligencia" de los equipos, ya que estos deben decidir el tipo de tarjeta de que se trata para enviarla al emisor correspondiente, lo que resulta en un mayor costo. Estos enlaces dedicados son muy caros y, en algunos casos, imposibles de conseguir o son incosteables.

Las redes de terminales dependen de enlaces dedicados caros y a veces incosteables

3) Para el caso de las cajas registradoras, el concentrador de éstas se debe conectar, también, con un conmutador (switch) de transacciones y éste a su vez con cada uno de los emisores de las tarjetas que acepta. De nuevo, esta línea debe ser dedicada, para conseguir las autorizaciones en línea.

Para las cajas registradoras también se necesitan enlaces dedicados

Las opciones anteriores, además de ser muy caras, comparten una debilidad:

Los tres tipos de dispositivos comparten...

Cuando existe un problema con las comunicaciones, los sistemas simplemente no funcionan.

En la conexión online mantener el enlace es vital. Se vuelve un talón de Aquiles cuando tenemos fallas en la comunicación. En México, como en toda Latinoamérica, las comunicaciones vía telefónica no son buenas, tenemos otras alternativas como de radiofrecuencia o microondas. Estas últimas no siempre son posibles técnicamente o son incosteables (relación costo-beneficio). No importa lo sofisticado del equipo POS, si no podemos enlazarnos al equipo de autorizaciones del emisor, todo ese equipo queda inoperante.

Los dispositivos POS quedan inoperantes si no hay conexión al emisor

Cuando tenemos estos problemas de comunicación entre los dispositivos POS del comercio y el equipo central de autorización de cada emisor, sólo nos queda un camino: la autorización fuera de línea.

Cuando tenemos problemas en la comunicación, debemos autorizar fuera de línea

Problemas de la autorización offline. La autorización fuera de línea nos resuelve una gran número de situaciones. Sin embargo, implica otro tipo de problemas, el principal de ellos es el tiempo de respuesta de atención al cliente. Para las tiendas departamentales y de servicio rápido, se vuelve inaceptable que una transacción de tarjeta tarde más de 1 minuto. Además de lo anterior, tenemos problemas inherentes en este esquema, como son: costo de producción y manejo del boletín de cada emisor, riesgos de autorizar tarjetas recién boletinadas, control en las disposiciones de la tarjeta, etc...

Pero la autorización fuera de línea implica otro tipo de problemas

Pero ...

¿Existe algún sistema que integre las funciones de un controlador de red y proporcione autorizaciones fuera de línea por boletín automáticamente, como un sólo dispositivo ?

¿Podemos, en este sistema, manejar cualquier número de emisores de tarjetas?

¿Podemos atender a cualquier dispositivo POS?

¿Podría adaptarse a las necesidades propias de cada comercio y de cada emisor?

¿Es posible contar con toda esta funcionalidad a un precio razonable e implantarse en plataformas tipo PC ?

El propósito del sistema propuesto es responder con un SI a todas las preguntas anteriores.

En pocas palabras, el sistema propuesto se encargará de la autorización fuera de línea de transacciones por tarjeta -de crédito y débito-, mediante un sistema de cómputo que automatice, lo más posible, el proceso manual de autorización offline.

Propósito general del sistema de cómputo propuesto

4.VI. Análisis y Diseño General del Sistema

La metodología que puede utilizarse para resolver este problema es muy variada. De echo, podemos resolverlo con cualquiera de las orientaciones que vimos en el capítulo de antecedentes; esto es, orientación a procedimientos con la programación estructurada, con la orientación a datos por medio de una base de datos (digamos relacional), o con la orientación a objetos. Dado que el tema central de la tesis es la orientación a objetos y que ésta promete ser más flexible y de más fácil mantenimiento, decidimos resolver el problema con esta orientación.

El sistema se hará en el marco de la orientación a objetos

Una vez identificado lo anterior, debemos escoger un lenguaje de programación adecuado. Los lenguajes de programación orientados a objetos que pueden utilizarse son muy variados, entre los más utilizados en la actualidad se encuentran SmallTalk y C++. Tomando en cuenta que C++ es más portable, eficiente y flexible que la mayoría de los demás lenguajes, escogimos a C++ para implantar este sistema.

El lenguaje de programación escogido es el C++

A continuación, presentamos un análisis y diseño general del sistema propuesto. En aras de mantener la claridad y simplicidad del texto, no utilizaremos ninguna notación especial para el análisis y el diseño que se presentan aquí. Sin embargo, debemos aclarar que para hacerlo, seguimos la guía de los textos: de Yourdon & Coad[20] y de Booch[21].

La notación utilizada es general pero basada en autores ampliamente reconocidos

Objetivos del Sistema. En las siguientes líneas presentamos los objetivos generales que perseguimos en la concepción del sistema.

[20] Object Oriented Analysis, Coad & Yourdon
[21] Object Oriented Design , Booch

Objetivos:

1) **Automatizar el proceso de autorización fuera de línea** -por boletín- de transacciones de tarjetas, mediante la utilización de un sistema de cómputo confiable y flexible.

2) **Incrementar la velocidad de respuesta.** Reducir este tiempo -típicamente- de 2 min. a 20 seg., mediante la automatización del proceso de autorización fuera de línea por boletín.[22]

3) **Mejorar el control de disposiciones de la tarjeta.** En las tiendas donde existen muchas de cajas y se autoriza manualmente por boletín, no es posible mantener un control adecuado de las compras que ha echo el cliente. Con este sistema podremos registrar el importe acumulado y el número de veces que se ha autorizado una tarjeta, manteniendo siempre sus respectivos límites.

4) **Mantener una arquitectura abierta.** El sistema contemplará la posibilidad de atender a cualquiera de los tres tipos de dispositivos POS. Sin embargo, para efectos prácticos de esta tesis, se simulará la comunicación con una sola terminal, pero la arquitectura del sistema estará preparada para atender a cualquier dispositivo POS.

5) **Posibilidad de extender funciones.** El sistema estará abierto para albergar módulos adicionales que puedan aportar un valor agregado tanto para el comercio, como para el cliente y el emisor. Por ejemplo, módulo de comprador frecuente.

Objetivos del sistema

[22] Este proceso se describió en la sección 4.4 Proceso de Autorización

Diagrama de contexto. A continuación, presentamos el diagrama equivalente al diagrama de contexto, que se utiliza en el análisis estructurado. A éste se le llama subject layer en el análisis orientado a objetos (AOO)[23], y top-level class category en el diseño orientado a objetos (DOO)[24]. Sin importar el nombre que deseemos darle, su propósito es mostrar la arquitectura y el funcionamiento general del sistema. Cada rectángulo representa una agrupación lógica de clases, estas agrupaciones de clases representan las funciones de más alto nivel que debe realizar el sistema para cumplir con sus objetivos.

El diagrama de contexto muestra la arquitectura general del sistema

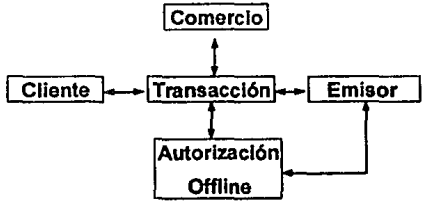


Diagrama de contexto

Del diagrama anterior, podemos ver que la transacción es la entidad principal del sistema. Una transacción se construye a partir de los datos del cliente (datos de su tarjeta), al que va a realizarse el cargo, y de los datos del comercio, al que se le va a abonar el dinero. Lo anterior explica la conexión existente entre el comercio y el cliente con la transacción.

La entidad transacción es la principal y se construye a partir de datos del cliente y del comercio

Una vez que se construye una transacción con los datos que mencionamos, se debe someter al proceso de autorización. Esto explica la conexión que existe entre una transacción y el proceso de autorización.

Una vez construida la transacción, se trata de autorizar

[23] De acuerdo con Coad & Yourdon en su libro Object Oriented Analysis
[24] De acuerdo con Booch en su libro Object Oriented Design

La interacción entre la autorización offline y el emisor se hace necesaria. En la transacción tenemos los datos de la tarjeta, pero ¿cómo decidir de qué emisor es esa tarjeta?. Lo anterior es necesario para saber qué archivo negativo (boletín) vamos a consultar y para saber el límite de piso asignado a esta tarjeta. Para mantener el respeto de la independencia entre clases -encapsulado-, lo que hacemos es "preguntarle" estos datos al emisor -por medio de mensajes-, por lo que necesitamos que la autorización offline y el emisor se comuniquen entre sí.

La relación entre autorización offline y emisor es debido al límite de piso y al archivo negativo

Clases principales. Con base en el diagrama de contexto y por medio de un análisis de dominio (domain analysis)[25], identificamos las clases principales del sistema. Estas son:

1) Transacción

2) Fecha y hora

3) Tarjeta

4) Comercio

5) Autorización offline

6) Lista de emisores y emisor

7) Lista de archivos negativos y archivo negativo

8) Control de disposiciones

Clases principales

Ahora haremos algunas observaciones sobre las clases de arriba.

Continuamos con observaciones de lo anterior

[25] Este se describe en el libro de Booch, OOD pp. 142 y 143

Como podemos ver en el diagrama de contexto, la entidad transacción continúa siendo la más importante y pasa a ser una clase directamente. Un dato muy importante para una transacción es la fecha y la hora en que se realizó, por lo que se necesita que una clase represente estos datos (punto 2 del cuadro anterior).

Transacción se convierte en una clase directamente

El cliente es representado por la clase tarjeta, no es necesario mantener otro tipo de información sobre él. La información que contiene la tarjeta es suficiente para nuestro sistema, y en general para cualquier sistema de tarjetas.

La tarjeta es suficiente para representar al cliente

Tanto comercio como autorización offline se convierten en clases del sistema. Mientras que comercio es simple, la autorización offline se vuelve un poco más complicada e interesante. Recordemos que la autorización offline involucra una serie de pasos, que vimos en la sección 4.4. de este capítulo. Los puntos 6, 7 y 8 del cuadro anterior forman parte de la autorización offline, pero esto lo veremos con más detalle un poco más adelante.

Comercio y autorización offline también son clases

Diagramas de las clases. Los diagramas que estamos a punto de presentar, no forman parte de alguna notación formal. Sin embargo, éstos se escogieron con la finalidad de clarificar y de presentar una aplicación práctica de los conceptos teóricos del capítulo tres.

Los siguientes diagramas aplican los conceptos de la orientación a objetos

En estos diagramas ilustramos las relaciones que guardan las clases entre sí. Por lo tanto, se definen solamente los datos -donde tenemos algunos objetos compuestos-, los métodos se ilustrarán en el proceso de autorización offline.

Los siguientes diagramas ilustran las relaciones entre las clases

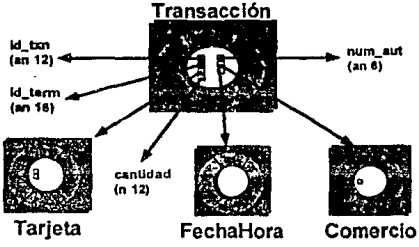


Diagrama para la clase Transacción

Del diagrama anterior podemos hacer algunas observaciones. Este es un ejemplo práctico de una jerarquía **parte de** o de clases compuestas (objetos compuestos). La clase Transacción se forma de otras clases: Tarjeta, FechaHora y Comercio. Pero también tiene datos simples como: identificación de la transacción (id_txn) -alfanumérico de 12 posiciones-, identificación de la terminal (id_term) -alfanumérico de 16-, cantidad -numérico de 12- y número de autorización (num_aut) -alfanumérico de 6-.

Este es un ejemplo de una clase compuesta. Además de otras clases, también tiene datos simples

A partir del diagrama anterior, detallemos un poco más cada clase.

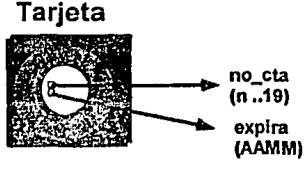


Diagrama de la clase Tarjeta

Aunque la cinta magnética que tienen las tarjetas contienen mucha más información (se graba información en tres tracks o pistas diferentes), los dos datos del dibujo

Aunque las tarjetas tienen muchos datos grabados, sólo escogimos estos dos

anterior son suficientes para nuestro sistema inicial. Pero si en algún momento necesitáramos datos adicionales, esta clase puede cambiar sin interferir con otras clases, gracias a las ventajas del encapsulado. El número de cuenta es numérico de hasta 19 caracteres y la fecha de expiración es de 4 posiciones, que indica el año y el mes de vencimiento.

La siguiente clase es FechaHora. Este ejemplo es interesante. Supongamos que ya tenemos una biblioteca que escribimos hace algún tiempo y que nos sirve para este diseño. Esta biblioteca es FechaHora. Dado que ya está lista para funcionar, nosotros solamente la utilizamos como una biblioteca, de manera que no necesitamos hacer más especificaciones ni codificar nada. Conforme trabajamos más con la orientación a objetos, vamos acumulando un conjunto de bibliotecas de clases que nos van a ayudar en el desarrollo de nuevos programas. También podríamos comprar bibliotecas en la tienda (off the shelf), tal como ahora compramos una ALU en una tienda de electrónica. Esto es, precisamente, la productividad de los sistemas orientados a objetos.

Como FechaHora es una biblioteca ya existente, sólo la utilizamos

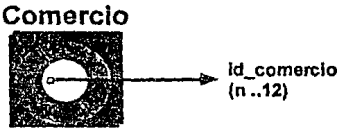
FechaHora

Esta es la base de la productividad en los sistemas orientados a objetos

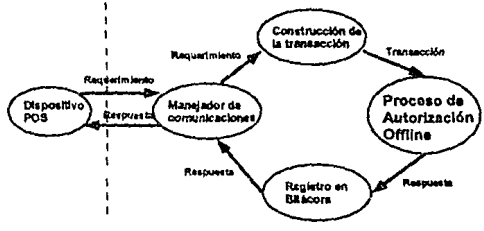
La siguiente es la clase Comercio. Esta clase es sumamente sencilla, sólo tiene un dato, que es la identificación del comercio (id_comercio) -numérico de 12-. Esta identificación es un número que asignan los emisores de las tarjetas. Es la forma en que los emisores identifican la cuenta a la que se va a realizar el abono. De nuevo, la dejamos como clase debido a que es muy

Comercio es una clase sencilla, pero lista para evolucionar

posible que cambie con el tiempo, hay muchos más datos que se pudieran agregar. Por ejemplo, dirección, teléfono de voz, teléfono de modem, responsable de la tienda, etc...



Flujo del sistema. A continuación presentamos el flujo normal que seguirá un requerimiento de un dispositivo POS, para pedir una autorización. Este diagrama es básicamente indicativo, de manera que busca ubicarnos en el proceso que deberá realizar internamente el programa.



El siguiente diagrama ilustra el flujo del sistema

Cuando llega un requerimiento de algún dispositivo POS (el cuál es un elemento externo), el manejador de comunicaciones se encarga de todos los detalles técnicos del protocolo de linea. De manera que le llegan los datos correctos al módulo que construye la transacción, sin importar si se tuvo una liga directa, un modem, etc..., o si el protocolo es del tipo poll-select o X.25.

El manejador de comunicaciones encapsula toda la variedad de los dispositivos POS

Una vez que llega el requerimiento al módulo de construcción de la transacción, se substraen los datos correctos para rellenar a la transacción: id_term, Tarjeta y cantidad. El sistema le adiciona la FechaHora de llegada y la identificación del comercio. Una vez hecho esto, se envía la transacción al proceso de autorización. Este proceso puede autorizar o rechazar la transacción. De cualquier manera, la respuesta debe guardarse en la bitácora de las transacciones. En caso de ser aceptada, el mismo proceso de autorización llena el dato del número de autorización (num_aut). En la siguiente sub-sección explicaremos mejor este proceso.

Conforme la transacción pasa por el sistema, sus datos se van llenando

Diagramas de objetos. Hasta ahora hemos visto algunos diagramas de clase y el diagrama general de flujo del sistema. Ahora veremos rápidamente un diagrama de objetos. La diferencia principal entre un diagrama de clase y un diagrama de objetos es que, en este último, podemos ver cómo los objetos se envían los mensajes. Un diagrama de objetos nos acerca más a la forma en que se codificará nuestro código.

El diagrama de objetos muestra el envío de los mensajes entre objetos

Un diagrama de objetos ilustra un mecanismo[26] del sistema. De la misma manera que la definición de las clases forman el "vocabulario" del sistema, los mecanismos son el "alma" del diseño. Los mecanismos son las partes del diseño que le darán su funcionalidad al sistema. En el diagrama anterior, ilustramos el mecanismo general del sistema. A continuación presentamos más al detalle el mecanismo que realiza la autorización de las transacciones.

Un diagrama de objetos ilustra un mecanismo

[26] Mecanismo es un término utilizado por Booch (OOD pp. 148) para describir el comportamiento de un grupo de objetos.

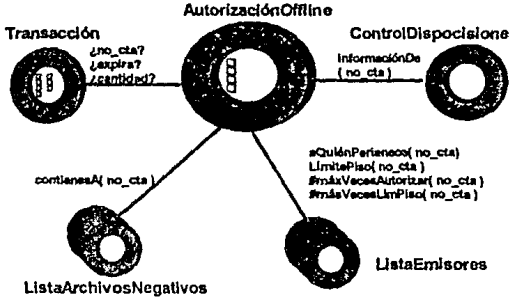


Diagrama de objetos para autorización offline

En este diagrama de objetos tenemos un nuevo elemento del que no habíamos hablado: ControlDisposiciones. Recordando los objetivos de nuestro sistema -al principio del capítulo-, el sistema dará un mejor control a las disposiciones que ha echo el cliente, en número de veces y en cantidad dispuesta. Por emisor se establece el número máximo de veces que se puede autorizar y el número de veces que la cantidad acumulada puede sobre pasar el límitePiso asignado, ambos para un mismo cliente (no_cta). Recuérdese que, aunque no lo definimos aquí explícitamente, debe haber una definición de clase para ControlDisposiciones, ListaEmisores y ListaArchivosNegativos.

En este diagrama de objetos presentamos a ControlDisposiciones

El diagrama de arriba nos presenta una idea más general de la forma en que se utilizan los mensajes. Mediante éstos, los objetos piden que se lleve al cabo una acción, o solicitan información uno del otro.

Un mensaje pide la realización de una acción o pide información

Sin embargo, un diagrama de objetos no nos permite saber la secuencia en la que se ejecutan los mensajes. Por lo tanto, se hace necesaria una forma más expresiva para indicar esta secuencia. Una forma sencilla y aún aceptable es una descripción textual breve.

Pero necesitamos documentar la secuencia de los mensajes

Para la autorización offline, nuestra descripción es la siguiente:

1) Extraer: no_cta, expira y cantidad de Transacción

2) Verificar que la tarjeta pertenezca a algún emisor: listaEmisores aQuiénPertenece(no_cta)

3) Ver vigencia: expira <= fechaHoy

4) Asegurar que: cantidad>=0 y cantidad<=límitePiso(no_cta)

5) Número de cuenta no exista en el negativo: !(Negativo contienesA(no_cta))

6) Cargar información de disposiciones: controlDisposciones informaciónDe(no_cta)

Hacemos esto mediante una descripción textual breve

7) Verificar que:

   #vecesHaAutorizado(no_cta) <= #máxVecesAutorizar(no_cta)

   cantidadAcum(no_cta) + cantidad <= límPiso(no_cta) * #máxVecesLímPiso(no_cta)

8) Si pasa todas las condiciones anteriores

   Actualizar información de controlDisposiciones

   Generar num_aut

De esta manera terminamos la presentación de la implantación del sistema propuesto, no sin antes recordar

que el objetivo de este capítulo (como el de todo el presente trabajo) es entender más claramente los conceptos de la orientación a objetos, mediante su aplicación práctica.

Capítulo 5:
Conclusiones

El éxito de la vida consiste en seguir siempre adelante.

- Samuel Johnson.

La imaginación es más importante que los conocimientos.

- A. Einstein.

Conclusiones

5.1. Introducción

Este capítulo presenta las conclusiones del sistema y de la orientación a objetos

Este capítulo es el más breve de todos y presenta tanto las conclusiones correspondientes al sistema de autorizaciones de tarjeta, como de la orientación a objetos en general.

La orientación a objetos tiene aplicación en todos los campos de la computación

Los conceptos de la orientación a objetos se pueden aplicar a una gran variedad de sistemas y tienen interés potencial para cualquier empresa que esté desarrollando, o piense desarrollar, cualquier sistema de cómputo. Debido a la inmadurez del mercado, las aplicaciones se han encasillado generalmente en los institutos de investigación y las instituciones de enseñanza. Además de la inmadurez de los lenguajes de programación, hay otro problema aún más grave, no existen -ni teóricamente- estándares en cuanto a bases de datos orientadas a objetos. Es común ver que para poder dar persistencia a los objetos (guardar objetos a disco), se utilice una base de datos relacional.

La base de la productividad en la orientación a objetos se encuentra en la reutilización (de objetos, librerías e incluso aplicaciones enteras)

La base de la productividad en los sistemas orientados a objetos se encuentra en la creación de estructuras reutilizables (clases) con estado y comportamientos específicos. Es más, no sólo es posible reutilizar a los objetos, la idea de la reutilización llega hasta el nivel de aplicaciones completas -como contabilidad, hojas de cálculo, etc.. Estas aplicaciones se "conectan" como lo harían una tarjeta de expansión en una PC-compatible, como tarjetas de video o de CD-ROM, pero para ver esto comercialmente tendrán que pasar todavía algunos años.

A continuación presentamos las conclusiones referentes al producto (sistema de cómputo propuesto).

5.II.Conclusiones del Producto

5.II.1. Aplicación a las Instituciones Financieras.

Este sistema fue concebido para trabajar en el ambiente que impera en México, donde las comunicaciones (principalmente telefónicas) no son demasiado confiables. Dado que Latinoamérica es, muchas veces, un espejo de México, este sistema puede aplicarse a casi cualquiera de estos paises.

Este sistema encuentra una amplia aplicación en México y en toda Latinoamérica

El sistema es adecuado para las instalaciones donde no existe infraestructura de comunicaciones, o donde la inversión del enlace al equipo central de autorizaciones no se justifica.

Es aplicable en donde no se puede o no es costeable un enlace directo al equipo central

El sistema puede manejar cualquier número de emisores y es especialmente valioso en configuraciones de redes alternas de terminales. En este sentido, el sistema puede aplicarse en un gran número de centros comerciales y tiendas departamentales, tanto del interior de la república -donde encuentran el campo más fértil- como en el D.F.. Cuando tenemos una configuración de redes de terminales, un sólo sistema de este tipo puede atender a todas las terminales conectadas a la red. Mejor aún, puede utilizarse la misma terminal para todos los emisores, sólo es necesario que cada emisor provea su propio archivo negativo (boletín). Esto es muy conveniente para los comercios, ¿cuántas veces hemos visto establecimientos con -típicamente- 3 terminales POS (para procesar Banamex, Bancomer y Carnet) que sólo provocan más problemas al comercio?.

Puede manejar cualquier número de emisores y es especialmente útil en redes de terminales

Elimina la utilización del boletín, lo que genera un ahorro substancial de tiempo y dinero. Al no tener que distribuir los boletines, el comercio gana tiempo y

Elimina el costo de producción y manejo del boletín

eficiencia de sus cajeros y el banco ahorra gastos de producción y distribución de los boletines.

La velocidad de respuesta se incrementa dramáticamente, -típicamente de 2 min a 20 seg-. Con esto estimulamos el pago con tarjeta, lo que conviene a los tres involucrados (cliente, comercio y emisor). Muchas veces el pago con tarjeta (cuando se encuentra totalmente automatizado) resulta más rápido que el pago con efectivo.

Incrementamos la velocidad de respuesta

La finalidad del sistema es la autorización de transacciones por tarjeta en donde se desea automatizar el proceso de autorización fuera de línea -por boletín-.

La utilidad principal: la automatización de las autorizaciones fuera de línea

Sin embargo, la tendencia mundial es realizar todas las autorizaciones por tarjeta en línea, siempre que sea posible. Gracias a que desde su concepción el sistema tiene una arquitectura abierta, su evolución se hace más fácil. Además del diseño abierto, la orientación a objetos juega un papel determinante para que esto sea posible, gracias al fácil mantenimiento, reutilización de código, independencia entre entidades y en general todas las ventajas que conlleva utilizar esta tecnología.

Este sistema puede evolucionar fácilmente a trabajar en línea y fuera de línea, dando así un servicio non-stop

Otra característica que proporciona este sistema al esquema tradicional de autorización offline, es el control de las disposiciones. Las personas deshonestas que roban y hacen fraude con las tarjetas, tienen bien localizados los lugares donde pueden comprar mercancía con tarjetas recién robadas. Como estas tarjetas acaban de ser robadas no están en el boletín, y si las compras fraudulentas que realicen no rebasan en límite de crédito, en una misma tienda -o centro comercial- pueden hacer un fraude cuantioso, con muchas disposiciones menores al límite de piso. Claro que esto sólo funciona dentro de una misma tienda o centro comercial.

Valor agregado: control de disposiciones

Proporciona una gran flexibilidad al ser capaz de atender a cualquier tipo de dispositivo POS. Esto es, podemos conectar terminales dial-up, en red o cajas registradoras al sistema, a todas podrá darles una autorización. Dada la arquitectura que guarda el sistema, puede integrarse cualquier dispositivo pos con un mínimo de cambios.

Puede atender a cualquier dispositivo POS, con un mínimo de cambios

5.II.2. Aplicaciones Alternativas.

Dada la arquitectura que tiene, el sistema es fácilmente modificable y puede adaptarse a otras aplicaciones similares con relativa facilidad. Por ejemplo:

1) Compañías aseguradoras

2) Instituciones médicas

3) Gasolinerías

4) Bonos y ayudas gubernamentales (comida, tortillas, etc...)

Se puede adaptar fácilmente a aplicaciones similares

Lo anterior implicaría un cambio al corazón del sistema, pero las partes de comunicaciones quedarían casi intactas, ganando así tiempo en el desarrollo, lo que desemboca en una mayor productividad y el abaratar los costos.

Lo que conduce a la productividad en el desarrollo de sistemas

5.III.Conclusiones de la Orientación a Objetos

Debemos tomar en cuenta costos y beneficios

Antes de que podamos tomar una decisión de si los sistemas orientados a objetos nos sirven o no, debemos tomar en cuenta tanto sus beneficios como sus problemas. Esta sección ofrece un resumen general de los beneficios y costos asociados a la orientación a objetos[27].

5.III.1. Beneficios Potenciales

1) **Desarrollo más rápido.** Una ventaja obvia de lo que pueden hacer los sistemas orientados a objetos es la rapidez en que se desarrollan. Esta rapidez está constituida básicamente en tres aspectos: i) construcción del sistema a partir de objetos estándares, ii) reutilización de subprogramas y aplicaciones completas y iii) la utilización de prototipos. Nótese que el rápido desarrollo no es mágico, sino que viene de la reutilización de objetos y programas existentes. De echo, si no tenemos una infraestructura en bibliotecas de clase, programar un sistema desde lo más básico, consume más recursos en la orientación a objetos que con la metodología tradicional. Afortunadamente, existen comercialmente muchas bibliotecas disponibles.

2) **Mejor calidad.** Dado que los objetos a partir de los que producimos nuevo software ya están probados, no tenemos que probar de nuevo toda la jerarquía. Solamente utilizamos el objeto como tal o heredamos de él algunas características, para adicionar otras nuevas. De manera que lo que existía sigue funcionando igual -gracias al

[27] Esta sección se basa principalmente en el libro de Taylor, Object Oriented Technology.

encapsulado- y sólo debemos probar las adiciones que hicimos.

3) **Mantenimiento más fácil.** Mediante la utilización de la herencia dejamos intacto el código original, lo que nos ayuda a evitar la adición de nuevos problemas (bugs). Mediante el encapsulado, aislamos problemas de manera que si algo está fallando, es más fácil encontrarlo. Este es un punto de gran importancia, especialmente si tomamos en cuenta que el 80% del personal de los departamentos de cómputo actualmente se dedican al mantenimiento de programas.

4) **Reduce costos.** Esto se lleva a cabo básicamente por tres medios: i) desarrollo más rápido (punto 1), ii) comprar bibliotecas generalmente es más barato que hacerlas y iii) se desvían menos recursos al mantenimiento (punto 3).

5) **Mayor flexibilidad.** Este punto representa el grado de adaptabilidad que puede tener un sistema. ¿Cuántas veces cuando por fin se termina un sistema es ya obsoleto?, desgraciadamente muy frecuentemente. Este punto se lleva al cabo por tres puntos principalmente: i) escalabilidad, que es la habilidad de construir sobre lo existente sin perturbarlo (herencia) de manera que la funcionalidad del sistema va evolucionando poco a poco; ii) estructuras más completas, estructuras que además de representar datos hacen operaciones sobre ellos (como una base de datos orientada a objetos); y iii) adaptabilidad, que es la capacidad que tienen los sistemas de cumplir con requerimientos ad-hoc, de manera que cubren las necesidades de los usuarios y no las condiciones súper puestas por el diseñador original.

5.III.2. Problemas Potenciales

1) **Madurez de la tecnología.** Este punto es muy importante. Si no tenemos las herramientas adecuadas (p.e. compiladores o lenguajes deficientes), nos creamos un problema mayúsculo. Debemos escoger con mucho cuidado los compiladores, herramientas y sistemas operativos sobre los que vamos a trabajar.

2) **Necesidad de estándares.** Dado que muchas compañías quieren imponer su estandard, no existen más que tendencias del mercado. Pero si escogemos irnos por una tendencia que, al final de cuentas no fue la correcta, nos buscamos también muchos problemas.

3) **Necesidad de mejores herramientas.** Hoy por hoy, todavía es necesario que hagamos mucho trabajo manual, desde el análisis y diseño (como sucedió con la metodología CASE) hasta las bases de datos orientadas a objetos.

4) **Velocidad de ejecución.** Aunque C++ generalmente no crea tanta sobrecarga al sistema (si está bien diseñado y se justifica su uso), la mayoría de los otros lenguajes orientados a objetos imponen una carga bastante pesada a la computadora.

5) **Disponibilidad de personal capacitado.** El personal capacitado en esta área es limitado, por lo que es necesario educar a nuestros desarrolladores en esta tecnología. Desafortunadamente la curva de aprendizaje es prolongada -típicamente de 8 meses o más.

6) **Costos de conversión.** El entrar a un esquema de orientación a objetos puede requerir de una fuerte inversión en herramientas, educación y consultoría en la nueva tecnología. Por lo anterior, es necesario planear bien la introducción de esta tecnología por secciones, según convenga a cada empresa.

5.III.3. En Balance

A pesar de las múltiples desventajas, considero que vale la pena intentar la introducción de la orientación a objetos. Los primeros que tomen correctamente la orientación a objetos, tendrán una ventaja competitiva clave sobre sus competidores.

Las ventajas parecen superar las desventajas

No cabe duda que para entrar a un esquema de orientación a objetos, se requiere de una planeación muy minuciosa y de la ayuda de consultores reconocidos. Si no construimos sólidamente nuestras bases en la orientación a objetos, lo más seguro es que fracasemos. Por el contrario, si avanzamos correctamente en este campo, la competitividad y productividad que podemos alcanzar pueden asegurar nuestra supervivencia como empresas en este medio ambiente tan competitivo.

Pero al tomar la orientación a objetos debemos planear muy bien nuestros pasos y apoyarnos de consultores reconocidos

Quizás lo mejor es instalar un proyecto piloto en donde podamos evaluar su efectividad. Es importante recalcar que la atención que requiere este proyecto piloto es vital para su éxito, si no le damos el suficiente apoyo seguro fracasará.

Se puede empezar con un proyecto piloto

Si no comenzamos nosotros, alguien más lo hará.

5.III.4. Futuro.

Conforme pasen los años seguramente tendremos el nacimiento de nuevos estándares, la tecnología se encontrará más madura, nuevas y mejores herramientas surgirán y tendremos más y mejor personal capacitado.

Con el pasar de los años muchos de los problemas mencionados desaparecerán

No cabe ninguna duda que el mercado se dirige vertiginosamente hacia la orientación a objetos. Quizás esto se deba principalmente a que los sistemas orientados a objetos y los sistema abiertos van de la mano. Grandes compañías como IBM, AT&T, Microsoft, etc... están apoyando fuertemente la orientación a objetos. A menos que surja otra tendencia radical, la orientación a objetos ocupará los lugares privilegiados que tienen ahora los métodos estructurados y las bases de datos relacionales.

Sin ninguna duda, el mercado se dirige vertiginosamente hacia la orientación a objetos

También veremos, más que bibliotecas de clase, aplicaciones completas de aparador (off the shelf). Este tipo de aplicaciones serán capaces de acoplarse a nuestro sistema de la misma forma que las tarjetas electrónicas de expansión se adaptan a nuestra PC. De esta manera, el software alcanzará -a largo plazo- la confiabilidad, eficiencia y el bajo costo que tienen los componentes electrónicos hoy en día.

También veremos la proliferación de bibliotecas y aplicaciones completas de aparador (off the shelf)

Todo lo anterior nos conduce a una cosa (desde mi punto de vista):

La orientación a objetos es algo que DEBEMOS adoptar tarde o temprano, si queremos competir eficazmente en la industria del software....

... hablemos de temprano.

Bibliografía

Con excepción de las criaturas, no hay cosa más maravillosa que un libro. Es un mensaje para nosotros de personas que nunca habíamos conocido, pero a pesar de todo nos levantan, nos asustan, nos enseñan, nos confortan y abren sus corazones como si fuéramos hermanos.

- Kingsley.

Bibliografía

+ Booch, Grady "Object Oriented Design with Applications" (OOD), Benjamin-Cummings, 1991.

+ Coad Peter & Yourdon Ed. "Object Oriented Analysis" (OOA), Yourdon Press, 1990.

+ Taylor, David "Object Oriented Technology: A Manager's Guide" (OOT), Addison Wesley, 1990.

+ Taylor, David "Object Oriented Information Systems", John Wiley & Sons, 1992.

Glosario

Abstracción de Tipos de Datos.
El proceso de definición de nuevos tipos de datos de alto nivel, para la representación -en un sistema computacional- de las entidades del mundo real.

C++.
Una lenguaje de programación orientado a objetos desarrollado en los laboratorios Bell AT&T, por Bjarne Stroustrup, durante los principios de la década de los 80's. C++ es un lenguaje de programación híbrido que combina al lenguaje "C" y a la orientación a objetos.

Clase.
Es una plantilla que define los métodos y los datos que componen a un tipo de objeto. Todos los objetos de una misma clase son idénticos en forma y comportamiento, pero cada uno contiene sus datos individuales que representan su estado.

Clase Abstracta.
Una clase que no tiene instancias, solamente se crea con el propósito de crear una clase base para la definición común que afectará a las clases que deriven de ella en la jerarquía (de clases). El término "clase virtual" es el mismo concepto.

Clase Virtual.
Ver Clase Abstracta.

Crisis del Software.
El término utilizado para describir la continua y creciente dificultad de producir rápidamente buen software, que responda a las necesidades cambiantes de los negocios. La crisis del software se caracteriza por la entrega tardía, costos elevados, defectos constantes y sistemas que son difíciles de mantener y modificar.

Dato.
Información que sirve para representar el estado interno de un objeto. Un dato puede ser un tipo de dato simple -como entero, caracter, decimal, etc.- o puede ser una referencia a otro objeto (composite objects).

Encapsulado.
Es la técnica mediante la cual se protege tanto la representación interna de un objeto (datos), como su comportamiento (métodos) de la intromisión externa.

Herencia.
Es un mecanismo mediante el cual una clase puede hacer uso de los datos y de los métodos definidos en todas las clases que se encuentran arriba de ella, en una jerarquía de clases.

Herencia Múltiple.
Cuando en una jerarquía de clases, una clase puede tener dos o más superclases.

Instancia.
Término que se utiliza para referirse a un objeto que pertenece a una clase dada. Por ejemplo, Morelos es una instancia de la clase Estado.

Jerarquía de Clases.
Una estructura de árbol que representan las relaciones existentes entre varias clases. Las jerarquías representan una especialización de clases, de más general -representada por su raíz- a más particular - representada por sus hojas. Puede haber cualquier número de niveles, y cualquier número de clases en cada nivel.

Mensaje.
Es una señal de exitación de un objeto hacia otro, que le indica al objeto receptor que ejecute uno de sus métodos.

Método.
Es un procedimiento dentro de un objeto. Los métodos de un objeto forman la interface (o protocolo) de éste con otros objetos. Los datos internos del objeto sólo pueden ser cambiados mediante sus métodos.

Objeto.
Es una estructura que representa a un elemento de la realidad o un concepto teórico generalizado. Un Objeto exhibe cierto estado (representado por sus datos internos) y cierto comportamiento (representado por sus métodos).

Objeto Compuesto (composite object).
Se le llama así a un objeto que contiene uno o más objetos, típicamente mediante el almacenamiento de referencias -en sus datos internos- a esos objetos.

Ocultamiento de la Información (information hiding).
La técnica de hacer que ciertos detalles internos de un módulo sean inaccesibles a otros módulos. Esto permite tanto proteger a los módulos de la intromisión externa, como evitar que los módulos dependan de la implantación interna de otros.

Overloading (sobrecarga).
Asignación de diferentes significados a un mismo nombre de función, permitiendo que un mismo mensaje ejecute funciones diferentes dependiendo del objeto que lo recibe y de los parámetros que lo acompañan.

Paradigma.
Una forma arraigada de pensar sobre algo, que da forma al pensamiento y a la acción, tanto consciente como inconsciente. La importancia de los paradigmas radica en que proveen un modelo cultural compartido que rige la forma de pensar y de actuar de una comunidad. Sin embargo, pueden ser una barrera importante para las innovaciones.

Parámetro.
Un dato que se incluye en un mensaje que provee al método indicado la información que necesita para ejecutar su trabajo. Un método puede tener cualquier número de parámetros, incluyendo cero. Otra forma de llamar a los parámetros es argumento.

Polimorfismo.
Término griego que indica "muchas formas". En la orientación a objetos, la habilidad de ocultar diferentes implantaciones detrás de una interface común, lo que simplifica la comunicación entre objetos y el flujo del programa.

Subclase.
Una clase que es la especialización de otra. Una clase que tiene una clase definida arriba de ella en una jerarquía de clases. Por ejemplo, un León es una especialización (subclse) de Felino.

Superclase.
Es una clase que se encuentra más arriba en la jerarquía, con respecto a otra clase. Una superclase es más general que sus subclases. Por ejemplo, transporte es la superclase de automóvil.

Tipo de Dato.
Definición genérica de una unidad elemental de información para un sistema de cómputo dado. Ejemplos de tipos de datos comunes son: números enteros y decimales, fechas, strings, etc... Es posible definir estructuras de datos de mayor nivel, si se soportan los tipos de datos abstractos.

Tipo de Dato Abstracto.
Es un tipo de dato que es definido por el programador y no forma parte de las estructuras pre-construidas del lenguaje (built-in). Los tipos de datos abstractos se utilizan, principalmente, para crear estructuras de datos de alto nivel, que representan objetos del mundo real en un programa. Por ejemplo, número complejo, automóvil, parser, lista ligada, etc...