Universidad Autónoma de Querétaro
Facultad de Informática

"Programación Orientada a Objetos"

Guía de Maestro

Que para obtener el título de

LIC. EN INFORMATICA

Presenta:

Marco Antonio Celedón Rosiles

Querétaro, Qro. Mayo de 1997

Agradecimientos:

A Dios por darme la vida y la capacidad de razonar ...

A mis padres Antonio y Cecilia por su apoyo e invaluable motivación ...

Este trabajo lo dedico especialmente a mi esposa Claudia, a mis hijas Paola y Andrea, quienes son mi principal motivación e inyección de energía para hacer las cosas cada día mejor y lograr así mis metas ...

No puedo dejar de agradecer su cooperación a mis hermanos Roberto, Leticia, Gabriela, Verónica, Alejandra, Juan Manuel y Francisco, quienes en los momentos difíciles de la carrera me orientaron para lograr mi objetivo ...

Gracias a mi tío Isauro y a mi abuelita María Luisa por su incondicional y valioso apoyo ...

Agradezco a mis amigos, compañeros de generación por los momentos que disfrutamos.

Quiero agradecer de manera muy especial los maestros de la facultad quienes contribuyeron en mi formación.

Universidad Autónoma de Querétaro
Facultad de Informática

C.U. a 30 de Abril de 1997.

MARCO ANTONIO CELEDÓN ROSILES
PRESENTE:

En base al oficio que presento al H. Consejo Académico, en el cual solicita la aprobación de la guía del maestro de la materia de **Programación III** sobre el tema **"Programación Orientada a Objetos"**, este H.Consejo tuvo a bien autorizarle dicha guía según lo establecido en el artículo 65 del Reglamento de Titulaciòn Vigente.

A T E N T A M E N T E:

"LO RACIONAL Y AUTOMÁTICO PARA INFORMAR"

ING. TERESA GUZMÁN FLORES
SECRETARIA ACADÉMICA

C.c.p. Acta 86
Exp. del Alumno

TGF/all*

Centro Universitario Av. Universidad Esq. Gabino Barrera s/n Querétaro, Qro. A.P. 284
C.P. 76010 Tels. (91-42) 16 32 42 Ext. 163/16 84 30

INTRODUCCION

La frase "Orientada a Objetos" se ha transformado en sinónimo de modernidad, calidad y eficiencia en los círculos de la tecnología de información. Las ventajas que se destacan de la orientación a objetos son la reutilizabilidad y la extensibilidad. Es decir, los sistemas orientados a objetos se tienen que ensamblar a partir de componentes previamente escritos con un esfuerzo mínimo, y el sistema ensamblado será fácil de ampliar sin necesidad de retocar los componentes reutilizados.

La "orientación a objetos" se refiere a algo más que a la programación orientada a objetos. Comprende una filosofía completa de desarrollo de sistemas que abarca la programación, el análisis de requisitos, el diseño de sistemas, el diseño de bases de datos y muchos otros asuntos relacionados.

Con todo lo anterior, los sistemas orientados a objetos sí tienen un papel determinante, y lo importante no es investigar si un sistema está orientado a objetos o no, sino en que forma está orientado a objetos y de qué manera proporciona las ventajas asociadas.

OBJETIVOS GENERALES

- Proporcionar al alumno las herramientas necesarias tanto en la teoría como en la practica, para que comprenda la metodología de la programación orientada a objetos.
- Comprender y aplicar los conceptos de la programación orientada a objetos en el desarrollo de sus aplicaciones.
- El alumno conocerá e identificara plenamente los elementos básicos y conceptos del paradigma orientado a objetos.

TEMARIO

MODULO I: LA PROGRAMACIÓN ORIENTADA A OBJETOS

OBJETIVOS PARTICULARES

- El alumno conocerá el ambiente de la programación orientada a objetos.
- Identificar plenamente los elementos de la POO y sus diversas aplicaciones en la realidad.
- El alumno conocerá y comprenderá el concepto de objeto.

- Conocer la implementaron de objetos y la comunicación entre ellos.
- Identificar los diferentes tipos de objetos, clases así como sus representación.
- Aplicar los conceptos de herencia y encapsulamiento entre objetos.
- Comprender el concepto de herencia así como su implementaron en la POO.
- Conocer el concepto de polimorfismo y su uso en el paradigma de la POO.
- Comprender el encapsulamiento de los datos en los objetos y la manera de accesar al información de los mismos.
- Identificar plenamente las diferencias entre clases.

MODULO II: EL PARADIGMA DE LA POO

OBJETIVOS PARTICULARES

- El alumno comprenderá el paradigma de la programación orientada a objetos.
- Conocer el proceso de desarrollo de un sistema desarrollado en un ambiente orientado a objetos.
- Conocer la diferencias entre la programación estructurada y la programación orientada a objetos.

MODULO III: EL LENGUAJE SMALLTALK

OBJETIVOS PARTICULARES

- El alumno conocerá el lenguaje de programación orientado a objetos denominado smalltalk.
- Conocer la forma de crear y manipular objetos en smalltalk.

MODULO IV: EJEMPLOS EN EL LENGUAJE DE PROGRAMACION SMALLTALK/V

ANEXO I: PROGRAMACIÓN ESTRUCTURADA VERSUS EL PARADIGMA DE LA POO

MODULO I

1.1 ANTECEDENTES HISTORICOS.

La aparición de la orientación a objetos refleja y resume la historia de la computación como un todo. Los primeros trabajos en computación, remontándose a los últimos años de la década de los 40's, tenían que ver exclusivamente con lo que ahora conocemos como programación. Sólo más tarde surgió un interés consciente sobre el diseño y el análisis como cuestiones independientes. Del mismo modo, fue la programación orientada a objetos la que primero llamó la atención, y, solo posteriormente, el diseño orientado a objetos e, incluso, mas recientemente, el análisis orientado a objetos, se transformaron en áreas importantes de trabajo.

Aunque Ten Dyke y Kunz en 1989, afirman que en al año de 1957 ya se habían utilizado técnicas rudimentarias de orientación a objetos, la historia de la programación orientada a objetos comienza realmente con el desarrollo del lenguaje de simulación de sucesos discretos, Simula, en el año de 1967, y continúa con el desarrollo de un lenguaje que casi toma como fetiche el concepto de objeto, Smalltalk, en los años 70's. Los lenguajes de influencia intermedia incluyen el Alphard y el CLU. Desde entonces ha habido muchos lenguajes que se han inspirado en esos desarrollos y reclaman el título de "orientación a objetos".

La fase siguiente, más o menos en los años 80's, puso de manifiesto una explosión de interés por la interfaces de usuario (UI, user interface). Los pioneros comerciales más conocidos, Xerox y, después, Apple, introdujeron el mundo de las interfaces universales WIMP (windows, icons, mouse and pointers). Muchas de las ideas de Smalltalk están estrechamente vinculadas con esos desarrollos. Por un lado, la programación orientada a objetos dio soporte al desarrollo de esas interfaces, mientras que, por otro, el estilo de los lenguajes orientados a objetos estuvo fuertemente influenciado por el paradigma WIMP.

A medida que la ingeniería del software comenzó a madurar, el interés se desplazó hacia los métodos de diseño orientados a objetos y las especificaciones o el análisis orientado a objetos. Las ventajas de la reutilizabilidad y de la extensibilidad se pueden aplicar tanto a los diseños y a las especificaciones como al código. Tanto Biggerstaff y Ritcher en 1989, como Prieto-Díaz y Freeman en 1987, y Sommerville en 1989 han sostenido, en un contexto de ingeniería del software más general, que mientras más alto sea el nivel de reutilización, mayores son las ventajas.

Más recientemente, el interés se ha volcado en los estándares. La tecnología de objetos sólo puede imponerse ante la inercia de las prácticas existentes si los usuarios pueden lograr la confianza que se requiere para cambiar a sistemas abiertos. Si las aplicaciones orientadas a objetos son todas incompatibles mutuamente, si las bases de datos orientadas a objetos no pueden trabajar entre

sí y con las bases de datos relaciónales, y si no hay una notación estándar y en términos para el análisis orientado a objetos, hay pocas esperanzas para ello.

Por lo anterior, el futuro de los métodos orientados a objetos parece muy prometedor. Dado el especial énfasis en los sistemas distribuidos y abiertos, la metáfora de los objetos se presenta como la más natural que se debe adoptar, dada la importancia que se tiene el encapsulamiento y el paso de mensajes. El aumento de la preocupación por los costos de mantenimiento bien puede conducir al reconocimiento de la reutilizabilidad como el problema clave de la programación, el análisis y el diseño.

1.2 ASPECTOS GENERALES DE LA PROGRAMACION ORIENTADA A OBJETOS.

1.2.1 Una revolución industrial en el software.

Una de las preocupaciones actuales más urgentes de la industria de la computadora es la de crear software y sistemas corporativos en tiempos más cortos y de más bajo costo. Para hacer un buen uso del poder cada vez mayor de las computadoras, necesitamos un software de mayor complejidad. Aparte de más complejo, también es necesario que dicho software sea más confiable. La alta calidad es esencial en el desarrollo del software, ya que una calidad pobre trae como consecuencia perdida de dinero.

La necesidad de crear un mejor software se aplica tanto en el interior de la propia industria del software como dentro de las empresas de todo tipo que crean sus propias aplicaciones computacionales. Las organizaciones relacionadas con la tecnología de la información necesitan crear y modificar aplicaciones en forma mucho más rápida. Si el desarrollo de aplicaciones tarda de dos a tres años, con un retraso de creación de varios años, las empresas no pueden crear más aplicaciones ni reaccionar ante la competencia de manera rápida. Se pierde la capacidad vital para el cambio dinámico. Para poder crear software con mayor rapidez y calidad requerimos cambiar en los aspectos de:

- Complejidad
- Capacidad de diseño
- Flexibilidad
- Rapidez en el desarrollo
- Facilidad de modificación
- Confiabilidad

Las técnicas orientadas a objetos permiten las aplicaciones se construyan a partir de objetos de comportamiento especifico. Los propios objetos se pueden construir a partir de otros y, que a su vez estos pueden estar formados por otros objetos. Esto se puede comparar con una maquinaria que esta construida por partes, subpartes, sub-subpartes, etcétera.

El desarrollo de aplicaciones en el mundo orientado a objetos comienza al estudiar los objetos en su ambiente, así como los eventos que interactúan con dichos objetos. Además de debe volver a utilizar clases de objetos ya existentes y, en caso necesario construir nuevas clases. Al modelar una empresa, los analistas deben identificar los tipos de objetos y las operaciones que hagan que estos se comporten en determinada forma.

A través de la historia de la ingeniería, ha permanecido un principio: *la gran ingeniería es la ingeniería sencilla*. Las ideas que se tornan muy rebuscadas, inflexibles y problemáticas tienden a ser reemplazadas por otras nuevas y claras desde el punto de vista conceptual y con sencillez estética. Cuando el programa de un programador parece telaraña, es hora de rediseñar todo el programa. El software corre el riesgo constante de convertirse en rebuscado, inflexible y problemático. Los enlaces entre las diversas partes tienden a multiplicarse si se añaden nuevas características y si cambian las necesidades de los usuarios. De no tener claros los conceptos, los diseñadores solo producirán intrincadas telarañas.

El término revolución industrial en el software se ha utilizado para describir el paso hacia una era en la que el software será compilado a partir de componentes de objetos reutilizables, con lo que se crearía una enorme biblioteca de componentes. Debemos pasar de una era de paquetes monolíticos de software, donde un vendedor construye todo un paquete, hasta una era en la que el software sea ensamblado a partir de componentes y paquetes de muchos proveedores (de la misma forma en que las computadoras y los automóviles son ensamblados a partir de componentes de muchos proveedores). Los componentes serán cada vez más complejos desde el punto de vista interno, pero será más sencillo interactúar con ellos. Serán ***cajas negras*** donde no podremos mirar al interior.

Tan solo las técnicas orientadas a objetos no nos pueden proporcionar la magnitud del cambio necesario. Se deben conjuntar con otras tecnologías de software. Las técnicas orientadas a objetos han existido durante dos décadas, pero las primeras de ellas se preocupaban más por la codificación en lenguajes como Smalltalk y C++. Entre tanto, con base en las herramientas CASE, los generadores de código se han desarrollado con base en el depósito, el motor de

inferencias y las técnicas de inteligencia artificial. Las técnicas orientadas a objetos son muy poderosas cuando se combinan con otras tecnologías. La combinación sinérgica de estas tecnologías traerá una revolución en el software, pero para que esto se lleve a cabo se requiere de un equipo de trabajo, educación y una buena administración, junto con la cooperación de las compañías de software respetando los estándares de desarrollo.

Las técnicas orientadas a objetos se piensan a veces en términos de lo que puede hacerse con los lenguajes de programación orientados a objetos como Smalltalk. Esto es un punto de vista muy restringido. Los métodos se pueden

crear con cualquier técnica adecuada, a veces se utilizan lenguajes por procedimientos, tales como C, COBOL Y FORTRAN; en otras ocasiones son más adecuados los lenguajes no procedurales como el SQL y los generadores de informes. Los lenguajes funcionales, como LISP, resuelven otros problemas. También se pueden implantar métodos con un motor de inferencias y el procesamiento basado en reglas y las técnicas de la inteligencia artificial y PROLOG. Los métodos pueden diagramarse con diagramas de acción o con técnicas alternativas.

El enfoque orientado a objetos puede incluir las demás técnicas de programación. En general, hay que utilizar las técnicas más poderosas, lo que por lo general quiere decir utilizar una herramienta CASE para el análisis y el diseño con un generador integrado de códigos y tal vez un motor de inferencias.

La programación orientada a objetos ha mejorado de manera esencial la creación del software, pero por si misma no implica el mejoramiento masivo necesario para la industria de la computación. Las técnicas orientadas a objetos deben combinarse con todos los aspectos disponibles de la automatización del software.

Las técnicas OO modificarán:

- Toda la industria del software
- La forma en que se venden los programas de aplicación
- La forma de uso de las computadoras
- La forma de uso de las redes
- La forma de analizar los sistemas
- La forma de diseñar sistemas
- La manera de utilizar las herramientas CASE
- La forma de planear las empresas
- El trabajo de las personas que desarrollan sistemas de información

1.3 CONCEPTOS BASICOS DE LA PROGRAMACION ORIENTADA A OBJETOS

Algunas de las ideas fundamentales que subyacen en la tecnología orientada a objetos son las siguientes:

- Objetos
- Métodos
- Mensajes
- Herencia
- Encapsulado
- Polimorfismo
- Clases

Aunque estos conceptos son la base del software orientado a objetos, son similares a los que forman la base de todos los seres vivientes.

1.3.1 Definición de objeto

El concepto de objeto nos proporciona un mecanismo para modelar los fenómenos físicos del mundo real. Las entidades en el dominio problema pueden ser modeladas como objetos. Las operaciones que puede desarrollar una entidad en el mundo real deberá ser modelada como funciones de interface para los objetos, es decir, el comportamiento de la entidad en el mundo real deberá ser el mismo para el objeto creado.

Un objeto puede contener una entidad de información (modularidad), donde los datos son protegidos (ocultamiento de información) y el manejo de la información almacenada será solo a través de operaciones legales permitidas por el objeto.

A continuación mencionaremos algunos conceptos de Objeto en la metodología de programación orientada a objetos:

Un objeto es una entidad capaz de almacenar un estado (información) y que ofrece varias operaciones (comportamiento) que nos permite examinar o afectar su estado.

Un objeto es el elemento básico de la programación orientada a objetos. Un objeto es aquel que es capaz de retener información y sabe como llevar a cabo sus operaciones.

Un objeto es un conjunto de variables de software y métodos relacionados

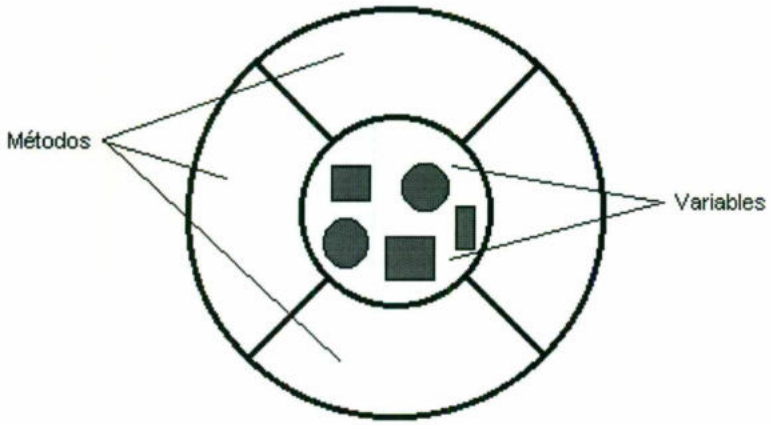
1.3.1.1 ¿Qué es un Objeto?

Como implica el nombre objeto-orientado, los objetos son llaves para comprender la tecnología orientada a objetos. Si observáramos a nuestro alrededor ahora pudiéramos ver muchos ejemplos de objetos de la vida real:

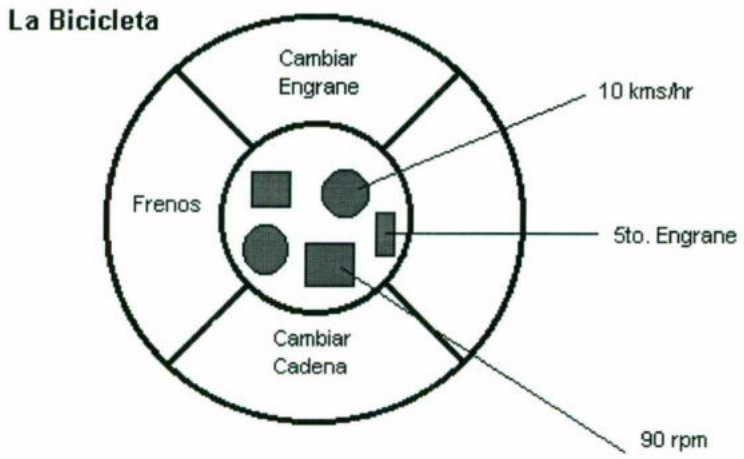
- un perro
- un escritorio
- una televisión
- una bicicleta
- un carro
- una factura
- una empresa
- un avión
- una computadora
- un árbol
- una lista

Estos objetos tiene dos características básicas: **el estado y el comportamiento.** Por ejemplo una bicicleta tiene estado (su actual engrane, pedales, cadena, llantas, numero de engranes) y tiene comportamiento (frenar, mover pedales, velocidad más lenta, cambio de engrane). Dichos objetos mantiene su estado mediante variables y cambian su comportamiento por medio de los métodos.

La siguiente ilustración es una representación visual común de un objeto:



Cada cosa que el objeto conoce (estado) y lo que puede hacer (comportamiento) es expresada por las variables y métodos propias del objeto. Un objeto que modela una bicicleta del mundo real tiene variables que indican su estado actual de la bicicleta, como lo es su velocidad en este momento es de 10 kms/hr, la cadena del pedal es de 90 rpm y el actual engrane es el número cinco. Estas variables y métodos son normalmente conocidas como variables de instancia y métodos de instancia para distinguirlos de las variables y métodos de clase.



El objeto bicicleta deberá tener métodos para frenar, cambiar pedal, cadena del pedal y cambiar de engrane.

Cualquier cosa que el objeto no reconozca o bien no pueda realizar, es excluida. Por ejemplo la bicicleta probablemente no tenga nombre, no puede correr o traer cosas. Por consiguiente no existen variables o métodos para esos estados y comportamientos en la clase bicicleta.

Como podemos observar en el diagrama anterior, las variables del objeto son el núcleo del mismo. Los métodos rodean y ocultan el núcleo de otros objetos en el programa. Al empacamiento de las variables del objetos y la manipulación mediante métodos es llamado ENCAPSULAMIENTO. Cuando se requiere cambiar de engrane de la bicicleta, realmente no interesa como trabaja el mecanismo, solamente se necesita conocer el nivel en el cual se dejará. De manera similar funcionan los programas de software, no se requiere conocer como una clase es implementada, solo se necesita saber cuales son los métodos correctos para invocarla. Así los detalles de la Implementación pueden cambiar mientras otras partes del programa son afectadas.

1.3.1.2 ¿Qué es un objeto en la POO?

Las personas nos formamos conceptos desde temprana edad. Cada concepto es una idea particular o una comprensión de nuestro mundo. Los conceptos adquiridos nos permiten sentir y razonar acerca de las cosas en el mundo. A estas cosas a las que se aplican nuestros conceptos se llaman objetos.

En el análisis y diseño orientados a objeto, nos interesa el comportamiento del objeto. Si construimos software, los módulos de software orientados a objetos (OO) se basan en los tipos de objetos. El software que implanta el objeto contiene estructuras de datos y operaciones que expresan dicho comportamiento. Las operaciones se codifican como métodos. La representación en software OO del objeto es entonces una colección de tipos de datos y métodos. ***En software OO un objeto es cualquier cosa, real o abstracta, acerca de la cual almacenamos datos y los métodos que controlan dichos datos.***

Un objeto puede estar compuesto por otros objetos. Estos últimos, a su vez, pueden estar compuestos de objetos, del mismo modo que una máquina está formada por partes y éstas, también, están formadas por otras partes. Esta estructura intrincada de los objetos permite definir objetos muy complejos.

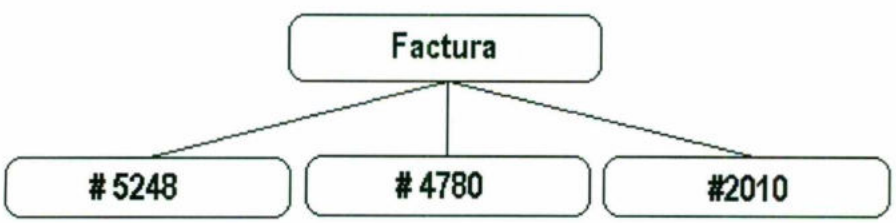
1.3.2 Definición de Tipo de Objeto

Los conceptos que poseemos se aplican a tipos determinados de objetos. Por ejemplo, *empleado* se aplica a los objetos que son personas empleadas por alguna organización. Algunas instancias de empleado podrían ser Juan Pérez, María López, etcétera. En el análisis orientado a objetos, estos conceptos se llaman tipos de objetos; las instancias se llaman objetos.

Un *tipo de objeto* es una categoría de objeto Un *objeto* es una instancia de un tipo de objeto

El tipo de objeto Empleado y sus instancias Juan Pérez, María Sanchez, etc.

Otro ejemplo, un tipo de objeto podría ser Factura y los objetos serían la factura # 5248, #4780, #2010, etc.



1.3.3 Métodos

Los métodos son una colección de instrucciones de programa que serán ejecutadas por el objeto que permite recibir un mensaje con el mismo nombre del método invocado.

Los métodos son implantaciones de las operaciones relevantes para una clase de objetos. Los métodos son invocados en respuesta a los mensajes.

Un método es sinónimo de operación. Es invocado cuando un objeto recibe un mensaje.

Cuando un mensaje es enviado a un objeto, un método con un mensaje modelo verifica que el mensaje sea buscado en la clase del objeto receptor. Si el método es encontrado es evaluado, en caso contrario, la búsqueda continua en la superclase del objeto. Si el método no se llegara a encontrar sucede un error.

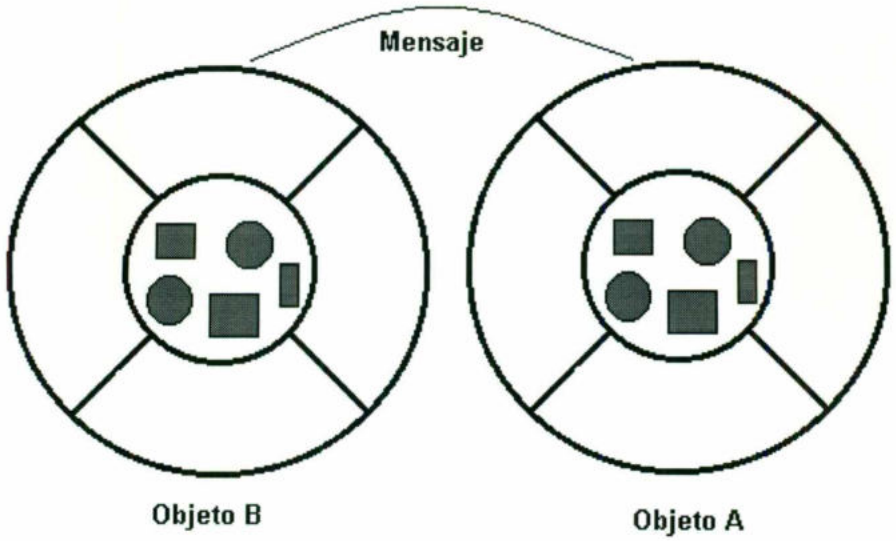
El mensaje modelo consiste de el selector de mensajes, junto con los nombres de los argumentos requeridos.

Cabe resaltar que la sintaxis de los lenguajes de programación para el envío de mensajes son diferentes, por convención primero se envía el nombre del método seguido de los parámetros.

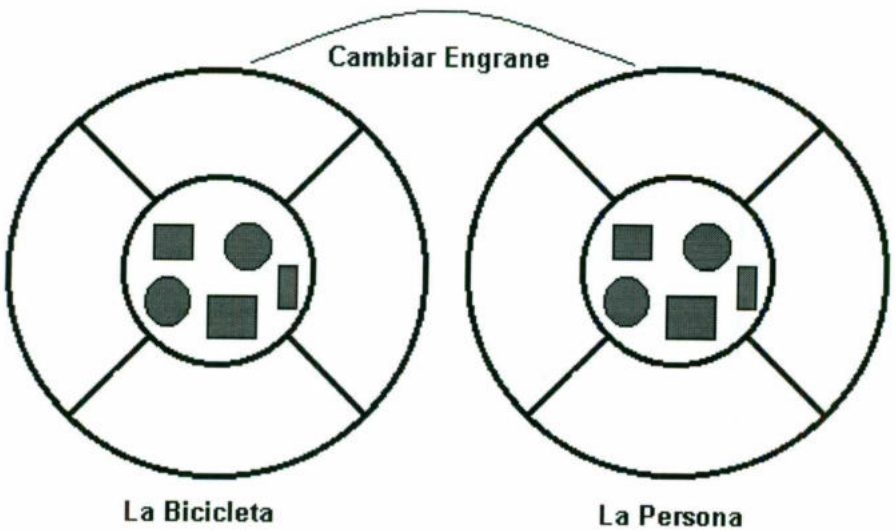
1.3.4 Mensajes

Un solo objeto por si solo generalmente no es muy útil y comúnmente aparece o forma parte como un componente de un gran programa o aplicación que contiene otros objetos. A través de la interacción de estos objetos, los programadores pueden alcanzar un alto grado de funcionalidad y una mayor conducta. Retomando el ejemplo de la bicicleta mencionado anteriormente, si nosotros la colgáramos de un gancho incrustado en la pared, no tiene razón de ser el objeto, puesto que por si solo es incapaz de realizar cualquier actividad. La bicicleta solamente es útil cuando otro objeto (es éste caso una persona) interactúa con ella (haciendo girar sus pedales).

Los objetos en el software orientado a objetos interactúan y se comunican con otros a través del envío de mensajes. Cuando un objeto A requiere que el objeto B realice un método propio de él, el objeto A envía un mensaje al objeto B.



En algunas ocasiones el objeto que recibe el mensaje requiere de información adicional para comprender exactamente que es lo que va ha realizar, por ejemplo, cundo cambiamos el engrane de la bicicleta y ésta se encuentra en movimiento, necesitamos indicarle en que engrane queremos posicionarnos. A ésta información adicional al mensaje se le conoce como **parámetro**.



Los componentes básicos de un mensaje son:

1. El objeto a quien el mensaje es enviado (la bicicleta).

2. El nombre del método que se ejecutará (Cambiar Engrane).

3. Los parámetros que el método necesita para ser ejecutado (Engrane pequeño)

Estos tres componentes son suficientes para que el objeto receptor lleve a cabo la operación requerida.

1.3.4.1 Beneficios de los mensajes:

- El comportamiento de un objeto es expresado a través de sus métodos, así como los mensajes que soporta para la interacción con otros objetos.

- Los objetos no necesitan estar en el mismo proceso o aún en la misma máquina para enviar o recibir mensajes de otro objeto.

Los objetos pueden ser muy complejos puesto que pueden contener muchos subobjetos, éstos a su vez pueden contener subobjetos, etc. La persona que utilice el objeto no requiere conocer su complejidad interna, sino la forma de comunicarse con él y la forma en que responde.

Tomando otro ejemplo, podemos comunicarnos con una televisión enviando mensajes por medio del control remoto. La TV responde mediante determinada acción y presenta las respuestas en la pantalla. Todas las televisiones de una marca, ejemplo SONY, se controlan mediante el mismo tipo de interfaz. Si tratáramos de comunicarnos con una televisión marca JVC no respondería puesto que requiere una interfaz distinta.

1.3.5 HERENCIA

1.3.5.1 Análisis del concepto de Herencia.

Un tipo de objeto de alto nivel puede especializarse en tipos de objetos de bajo nivel. Un tipo de objeto puede tener subtipos. Por ejemplo, el tipo de objeto persona puede tener subtipos militar civil y militar. Militar puede tener subtipos oficial y enrolado. Oficial puede tener subtipos, teniente, capitán y mayor, y también subtipos como marino o zapador, oficial en servicio activo u oficial retirado. Existe una jerarquía de tipos, subtipos, sub-subtipos, etcétera.

Una clase implanta el tipo de objeto. Una subclase hereda propiedades de su clase padre; una sub-subclase hereda propiedades de las subclases; etc. Una subclase puede heredar la estructura de datos y los métodos, o algunos de los métodos, de su superclase. También tiene sus métodos e incluso tipos de datos propios.

Cuando las clases de las entidades en una aplicación han sido identificadas, se debe investigar la semejanza entre ellas. Para la identificación de su comportamiento y características comunes, se debe establecer una jerarquía de clases. El mecanismo de la herencia se puede utilizar para expresar las similitudes entre clases.

La herencia puede también ser utilizada como un mecanismo para compartir y reutilizar código. Las nuevas clases pueden ser creadas utilizando una clase existente como un modelo, las cuales sirven como fuente para la herencia y son llamadas supercalases o clases base, y las clases de ésta son llamadas clases derivadas. Una clase que ya existe puede servir como una clase base para más de una clase derivada, o bien, lo que significa que una superclase puede tener varias subclases.

Cuando la herencia es empleada para crear una nueva clase de una ya existente, dicha subclase o clase derivada tiene acceso a la información encapsulada de su superclase. De esta manera, los sistemas que manejan la herencia permiten la construcción de estructuras encapsuladas, las cuales pueden ser aplicadas de diferentes maneras sin necesidad de modificar el código.

Generalmente, la clase base es una definición de clases genéricas de objetos, mientras que las clases derivadas de la clase son definidas de manera más especifica o de casos especializados de objetos.

Bien, ahora analizáremos el concepto de herencia en términos de los lenguajes basados en clases.

> Todos los objetos son una misma instancia de la misma clase, las clases forman una jerarquía, cada subclase en la jerarquía puede adicionar o modificar el comportamiento del objeto y agregar un estado. Una ventaja de la herencia es que ésta soporta la programación diferencial, la cual consiste en tener un objeto similar a otro, pero con ciertas modificaciones.

Una explicación del mecanismo de la herencia:

> Quizá el concepto más poderosos en los sistemas de programación orientados a objetos es la herencia. Los objetos pueden ser creados heredando las propiedades de otros objetos, eliminando así la necesidad de escribir código redundante, por ejemplo, un programa para procesar números complejos consiste de partes reales e imaginarias. En un número complejo, las partes reales e imaginarias se comportan como los números reales, de tal manera que todas las operaciones como +, -, /, *, Seno, Coseno, etc. pueden ser heredadas de la clase de objetos llamada REAL, en vez de tener que ser escritos en código. Lo anterior tiene un mayor impacto en la productividad del programador.

La idea de la jerarquía de la herencia es una extensión de la propia herencia. A continuación se explica la idea de la jerarquía de la herencia:

> Una ambulancia es una clase de camión, un camión es una clase de vehículo motorizado y un vehículo motorizado se puede definir como una clase de vehículo. Existen también otra clase de vehículos; tales como aviones, trenes, barcos, etc., y tienen características en común de todos los vehículos y algunas de los vehículos motorizados. Si escribiéramos las relaciones entre las diferentes clases de vehículos, obtendríamos una estructura de árbol. Dicha estructura es normalmente conocida como la jerarquía de la herencia.

Una ventaja significativa de la herencia, es que soporta la programación diferencial, la cual significa crear una nueva clase de objetos realizando pequeñas modificaciones a una clase ya existente. En resumen, para proveer un mecanismo para la reutilización y compartición de código, la herencia también facilita el mantenimiento de software para todos los bugs fijos y modificaciones a la clase base o superclase. los cuales son propagados de manera automática a las clases derivadas.

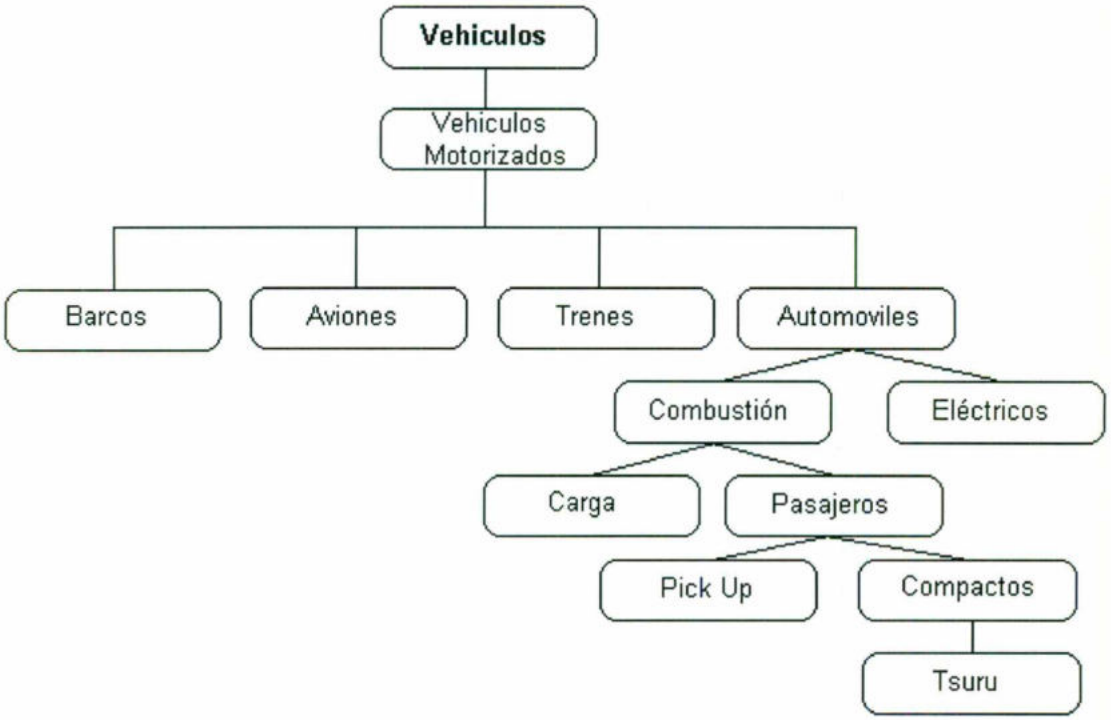
La programación diferencial o programación por herencia reduce dramáticamente la cantidad de código en los grandes sistemas; es una poderosa herramienta para la organización, facilita las modificaciones a los programas, su extensión y el mantenimiento.

La herencia en Smalltalk es basada en la idea de las subclases, por ejemplo cuando definimos una clase, es una subclase de otra. Las clases son adoptadas en una simple jerarquía de herencia con la clase de más arriba. Una clase puede tener cualquier número de subclases, pero cada clase tiene solo una superclase.

Una clase que es una subclase de otra, automáticamente hereda o comparte la representación y el protocolo de la clase. En resumen las subclase tienen también:

- La capacidad de soporte adicional agregando nuevos métodos.
- Aumentar la representación agregando clases y variables de instancia.

Tomando el ejemplo de los vehículos trataremos de explicar la jerarquía de las clases:



Jerarquía de Clases

Cada subclase hereda el estado de la superclase en la forma de la declaración de las variables. Los barcos, aviones, trenes y automóviles comparten algunos estados tales como: la velocidad, distancia recorrida, etc.. Además las subclases heredan métodos de la superclase, tales como: cambiar velocidad, cambiar de dirección, etc., los cuales determinan su comportamiento.

Sin embargo, las subclases no son limitadas por el estado y comportamiento de la superclase, puesto que estas pueden adicionar variables y métodos propios de ellas.

1.3.5.2 Implementación de la herencia.

La implementación de la herencia es un tipo de herencia relacionada con una clase derivada de la clase base o superclase, y sucede cuando algunas de las funciones de la clase derivada son delegadas de las que han sido implementadas en la clase base. De esta manera la clase derivada hereda las funciones de su superclase, dando así la reutilización de código. La implementación de la herencia as apropiada para situaciones donde la clase derivada es un subconjunto o una especialización de la clase base o superclase.

1.3.5.3 Herencia y Encapsulamiento.

Encapsulamiento es el resultado o acto de ocultar los detalles de implantación de un objeto respecto de su usuario. Es el empaque conjunto de datos y métodos. El objeto esconde sus datos de los demás objetos y permite el acceso a los datos mediante sus propios métodos. Esto recibe el nombre de ocultamiento de información. El encapsulado evita la corrupción de los datos de un objeto. El encapsulado protege los datos del uso arbitrario y no pretendido. Además oculta los detalles de su implantación interna de los usuarios de un objeto. Los usuarios se dan cuenta de las operaciones que pueden solicitar del objeto pero desconocen los detalles de como se lleva a cabo la operación. Todos los detalles específicos de los datos del objeto y la codificación de sus operaciones están fuera del alcance del usuario.

Como mencionamos con anterioridad, una nueva clase puede ser creada a partir de una superclase. Las variables y métodos pueden ser agregados a una nueva clase para obtener la funcionalidad requerida. La herencia es un mecanismo a través del cual una subclase tiene acceso a las variables de su superclase. Por lo mencionado anteriormente, los clientes de la subclase pueden también ser considerados como clientes de la superclase. El encapsulamiento ofrecido por la clase puede ser limitado por el acceso permitido a sus variables de las subclases por medio de la herencia. En una subclase, las propiedades generales especificadas en la superclase, pueden ser reemplazadas por propiedades de mayor especialización. De esta manera, es posible modificar las propiedades definidas en la superclase redefiniendo las propiedades en la subclase.

1.3.5.4 Interface de la Herencia y Herencia de Datos.

Algunas veces una clase es creada solo para especificar el conjunto de operaciones que son permitidas en una clase de objetos, el objetivo es especificar la clase genérica para la clase de objetos fuera de cualquier intención de definir un solo objeto. Tales clases sirven para definir la funcionalidad de la interface, además son también llamadas como interface de la herencia por algunos investigadores, porque la razón para utilizar este tipo de herencia es que permiten la misma funcionalidad para ser presentadas por todos los objetos que son miembros de clase y los derivados de dichas clases. Las clases son utilizadas solamente por la interface de la herencia pudiendo no tener datos o cualquier implementación de funciones. Por lo anterior, solamente sirven como una relación entre sus clases derivadas y el resto del universo.

La herencia de datos se presenta cuando una clase base es creada solo para definir sus datos y las clases derivadas tienen sus propias funciones e implementadas de manera diferente. Este concepto es lo contrario de la interface de la herencia, desde entonces el énfasis es en la generación de clases derivadas que comparten solamente los datos con su clase base. Dichas clases nos indican como serán utilizados los datos en las clases derivadas, y que operaciones son posibles en dichos datos.

1.3.5.5 Herencia y Generalidad.

La Generalidad es una alternativa de la herencia, la cual refuerza los beneficios de la extensión, reutilización y compatibilidad de los componentes. La generalidad es una técnica para la definición de los componentes de software que tienen más de una interpretación por la representación de los tipos de parámetros. la herencia, en cambio, permite la definición de nuevos componentes de software en términos de los ya existentes, estos es, como una extensión o restricción de los que ya existen.

La generalidad es utilizada en lenguajes tales como ADA y CLU. En el primero, existe una forma de generalidad llamada TIPO DE GENERALIDAD, la cual proporciona la capacidad de parametrizar componentes de software por uno o más tipo. Dichos componentes de software pueden ser desarrollados en términos de tipos genéricos.

La generalidad puede ser clasificada como "Cronstrained" y "Unconstrained". Cuando los componentes de software no solo requieren tipos parametrizados sino también funciones u otras operaciones aplicables en esos tipos se dice que

existe generalidad "constrained", en cambio, cuando existen requerimientos específicos en los tipos que pueden ser utilizados para instanciar los parámetros genéricos se da la generalidad "unconstrained".

Esta última puede ser vista como una técnica para desviar los requerimientos necesarios impuestos por la verificación de tipos estáticos. No existen requerimientos específicos en los tipos que pueden ser utilizados como parámetros genéricos. En otros casos, sin embargo, una definición genérica puede ser solo significativa si los parámetros genéricos satisfacen algunas condiciones. Esta forma de generalidad es definida como "constrained".

1.3.5.6 Herencia y Delegación.

En los sistemas orientados a objetos, la definición incremental y compartición pueden ser implementados por dos métodos alternos: **la Herencia y la Delegación**.

La delegación es considerada por muchos investigadores más poderosa que la herencia. En ésta, cada objeto es una instancia fuera de una clase. Este concepto es considerado de una manera ortogonal a el concepto de clases. Por lo tanto, en los lenguajes que son menos basados en clases son más probables para utilizar la delegación. Algunos investigadores han promovido a la delegación como una técnica de compartición de código que puede servir como una alternativa para la herencia clásica.

La delegación permite la definición incremental en los objetos, pero no en las clases. Así, los objetos pueden ser definidos en términos de otras instancias. El nuevo objeto (instancia), creado por los atributos de la delegación desde uno o más objetos base (instancias) es conocido como un PROTOTIPO. Aunque no existen las clases en la delegación, los métodos y las variables de instancia pueden ser compartidos. Los objetos en la jerarquía de la delegación no son independientes. Cualquier cambio a un objeto modifica los atributos delegados y los valores en el prototipo que dependen de él. Los investigadores afirman que la delegación pueden capturar el comportamiento de la herencia.

La herencia permite la definición incremental de las clases cuando un tipo de alguna instancia de un clase puede ser definida en términos de los tipos de las instancias de otras clases. Sin embargo un objeto individual o instancia no puede ser definido en términos de otras instancias u objetos. Nuevamente, utilizando la herencia, las instancias pueden compartir solamente los atributos y su comportamiento, pero no los valores. Las instancias son de este modo

independientes, y sus valores no son compartidos. Si los valores de una variable de una instancia son modificados, estos no afectan las otras instancias de la misma clase.

Como conclusión afirmaremos que la diferencia básica entre la herencia y la delegación, es que la primera es permitida solo en las clases y en tanto la segunda en todos los objetos.

1.3.5.7 Herencia Múltiple.

La herencia múltiple es una generalización de la herencia que permite múltiples superclases. Esto es muy útil cuando creamos nuevos objetos que comparten un comportamiento común el algunas clases ya existentes de objetos.

En algunos casos es necesario diseñar una nueva clase que herede atributos y comportamiento de dos superclases diferentes. Por ejemplo, los anfibios pueden heredar atributos y comportamiento de los animales vivientes de la tierra y de los animales que viven en el agua. Esto es llamado herencia múltiple. En esta, la jerarquía de la superclase y de las subclases pueden ser presentadas mediante una gráfica.

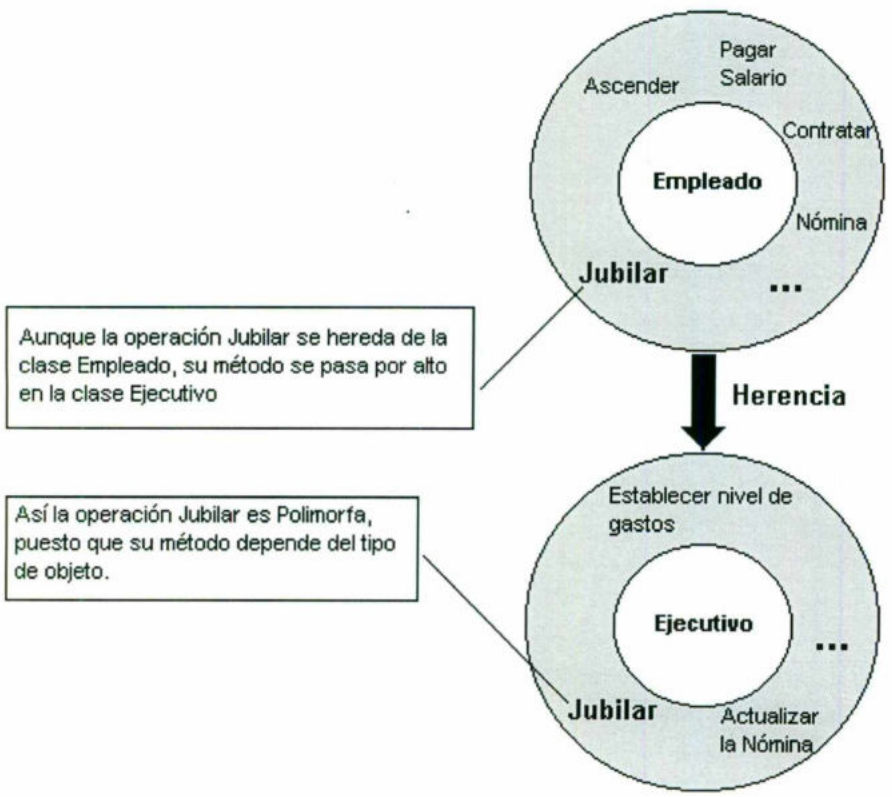
La herencia múltiple, es el hecho que facilita la combinación de diferentes tipos de comportamiento en un nuevo objeto. Es fácil comprender este concepto cuando las características de un objeto no son relacionadas y relativamente independientes. Por ejemplo, los objetos de la clase Ballenas pueden ser creados combinando comportamientos de la clase mamíferos con la clase peces. La herencia múltiple es correcta cuando las estructuras y las operaciones no son duplicadas y las variables no son definidas en cada superclase de la subclase.

Existen dos maneras de analizar la herencia múltiple:

- Como un mecanismo para compartir código donde la subclase se acerca a la superclase y es enfátizada como un mecanismo para la reutilización de código y,
- Como un mecanismo para la generalización de la herencia donde un concepto es clasificado como una especialización de conceptos e implica el orden de los conceptos generales.

1.3.6 Concepto de Polimorfismo.

Uno de los objetivos principales de las técnicas orientadas a objetos es utilizar otra vez el código, sin embargo, algunas de las operaciones requieren adaptación para resolver necesidades particulares. Como es mostrado en el siguiente ejemplo, la clase empleado define una operación de retiro(jubilación). En las implantaciones orientadas a objetos todas las subclases de empleado heredan ésta operación en forma automática. Sin embargo, una organización puede tener distintos métodos para retirar a un ejecutivo y a un empleado. En este caso, el método para el retiro de un ejecutivo está por encima del método para el retiro de los empleados en general. Aun así, aunque los métodos son distintos, llevan a cabo el mismo propósito operativo. Este fenómeno se conoce como polimorfismo. La palabra polimorfismo se aplica a una operación que adopta varias formas de implantación, tanto si el objeto es un empleado o un ejecutivo.



El método se puede implantar de manera diferente en una clase y su subclase. Las subclases pueden implantar la misma operación con métodos diferentes. Esto se llama polimorfismo.

La palabra Polimorfismo, según las etimologías griegas significa "Muchas Formas" o "Muchos Tipos". Esto significa que el ítem referenciado puede ser de tipos diferentes.

Polimorfismo significa que el transmisor de un estímulo no necesita conocer la clase de la instancia receptora. El transmisor da únicamente un requerimiento del evento especificado, mientras que el receptor sabe como ejecutar este evento.

El enlace de los estímulos recibidos con la operación apropiada a ejecutar es realizada asociando los estímulos con esta operación. Si este enlace ocurre durante la compilación, hablamos de un **enlace estático**. La característica polimorfica hace algunas veces incierta la determinación de la clase a la que una instancia pertenece al tiempo de compilación. Esta incertidumbre no será aclarada hasta el momento de la ejecución. Si la asociación ocurre cuando se envía el estímulo, esto a tiempo de ejecución, se habla de **asociación dinámica**. Otros nombres para este concepto son asociación tardía, retardada o virtual. La asociación dinámica es flexible, pero reduce la eficiencia; en principio, la operación del algoritmo de búsqueda as ahora llevada a cabo durante la ejecución.

La ventaja con la asociación dinámica es que se obtiene un sistema flexible. Esta flexibilidad es de mayor uso en sistemas que son modificados regularmente. Mediante la no asociación previa a la ejecución, muchas de las modificaciones hechas no afectaran al objeto que transmite.

La asociación estática, sin embargo, es más segura y eficiente. Es segura, si tenemos un lenguaje con tipo, debido a que los errores son notificados al momento de compilación y no se permite que causen fallas durante la ejecución. Es más eficiente debido a que el algoritmo de búsqueda se efectúa únicamente en la fase de compilación.

El polimorfismo muchas veces entiende que el receptor de un estimulo, no puede asociar este a una operación antes de que dicho estímulo sea enviado al momento de la ejecución, por la razón de que la clase se desconoce hasta entonces. Si no conocemos de antemano (durante la compilación) que operación se debe ejecutar en el receptor de cierto estimulo, entonces debemos utilizar algún tipo de asociación dinámica. Esta es por lo tanto, una manera de implantar el polimorfismo.

El polimorfismo puede, de hecho, ser usado sin tener asociación dinámica. Lo anterior lo podemos ejemplificar si declaramos que una variable A es una instancia de la clase B, y se declara una operación X a ser ejecutada en esta

clase sin que sea sobrescrita en ninguno de sus descendientes. Si asignamos a A cualquier instancia cuya clase descienda de B, la operación X será conocida durante el momento de compilación. Podemos entonces asociar estáticamente la operación al estimulo. Aquí hemos usado polimorfismo, ya que no se sabe exactamente la clase a la que esta asociada la instancia almacenada en A.

1.3.6.1 Tipos de Polimorfismo.

En la Programación orientada a objetos existen tres grandes variedades de polimorfismo:

a).- Polimorfismo de Inclusión.

Un objeto puede pertenecer a diferentes tipos que no necesitan encontrarse separados. El objeto puede incluir uno o mas tipos relacionados. En la jerarquía de clases, los objetos que pertenecen a una clase se manipulan no solamente como pertenecientes a su tipo, sino que también como a su supertipo. De este modo, ciertas operaciones sobre objetos pueden trabajar no solamente sobre objetos de las clases, sino que también sobre objetos de las superclases.

b).- Polimorfismo Parámetrico.

Tipos de parámetros implícitos o explícitos son utilizados para determinar el actual tipo de argumento requerido para cada una de las aplicaciones polimorficas. De este modo la misma operación puede ser aplicada a los argumentos de diferentes tipos.

c).- Polomorfismo Ad Hoc.

Cuando un procedimiento trabajo o aparenta trabajar en varios tipos, es llamado polimorfismo ad hoc. Esto es similar al sobrecargamiento, y no es considerado como un verdadero polimorfismo.

El polimorfismo de inclusión, es el más común en la programación orientada a objetos. Desde el punto de vista de programación, polimorfismo significa múltiples clases con nombres de métodos comunes y la habilidad para enviar mensajes (para activar métodos) a las instancias de cualquier clase sin conocer a cual clase de objetos nos referimos.

1.3.7 CLASES.

1.3.7.1 Análisis del concepto de una clase.

El paradigma orientado a objetos hace posible modelar los sistemas en términos humanos, en términos del pensamiento humano simulando un lenguaje humano. Al definir sistemas en términos de objetos y acciones desarrolladas entre objetos, el paradigma orientado a objetos depende de la tendencia humana. Los humanos tendemos a clasificar nuestro conocimiento y a organizar la información en base a características. De manera similar el concepto de una clase en el paradigma orientado a objetos, debe tener sus orígenes en la tendencia humana para clasificar las entidades en los problemas del mundo real de acuerdo a sus similitudes y diferencias. Las clases facilitan la clasificación de objetos con propiedades similares, y pueden llegar a formar subclases por medio de la especialización de las propiedades generales.

El concepto de clase es un complemento de la idea de los tipos definidos por el usuario, además ayuda para la implementación del ocultamiento de la información. Un programador que utiliza una clase no necesita conocer el interior de la misma. Como resultado del ocultamiento de información, el código en el cual la clase es utilizada podría no depender de los detalles de implementación de la clase.

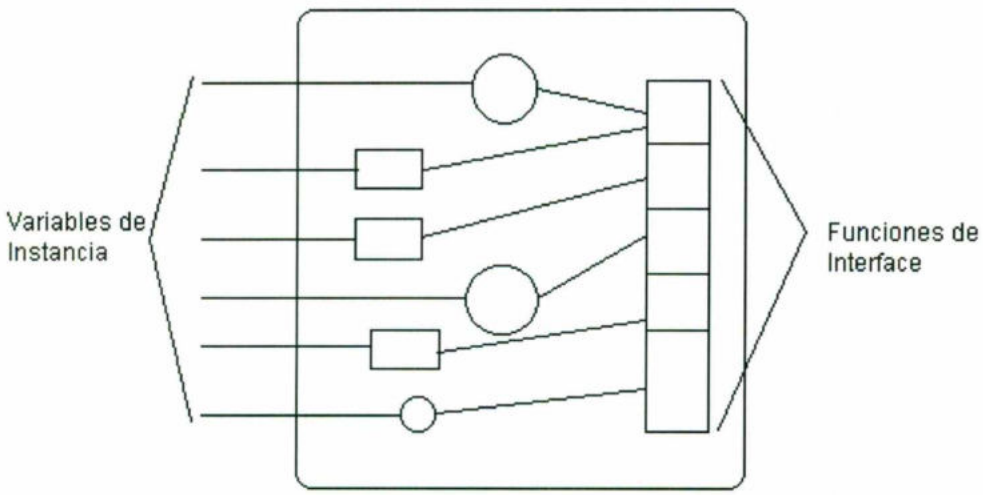
Existen dos maneras de ver una clase:

- Vista Exterior (La especificación)
- Vista Interior (La implantación)

Cuando una clase es utilizada en un programa, solo se necesita conocer su especificación, la cual es descrita por las funciones de interface y éstas a su vez definen las operaciones que pueden pueden ser desarrolladas sobre las instancias de la clase.

El código escrito por un programador para utilizar las interfaces de una clase no necesita ser recompilado siempre y cuando la implementación de una función de interface es modificada. En otras palabras, el ocultamiento de información para una clase ayuda a minimizar la recompilación. La razón de lo mencionado anteriormente, es porque al generar la clase el código escrito en el interior de la misma ya fue compilado.

La implementación de una clase puede ser por consiguiente considerada como la vista interior, ésta se encuentra compuesta por los detalles de la representación de los datos, la implementación de las funciones de interface y otras operaciones similares como es mostrado en la siguiente figura:



Objeto: Datos Encapsulados y Funciones de Interface

En algunos lenguajes orientado a objetos tales como Smalltalk, las funciones de interface son conocidas como "Métodos", mientras que en otros lenguajes como C++, son llamadas funciones de miembro (member functions). En Smalltalk se realizan sistemas orientados a objetos para expresar los requerimientos computacionales. Dichos mensajes son enviados a través de los métodos, los cuales pueden cambiar el estado de los objetos.

1.3.7.2 Clases.

Una clase es el mecanismo de abstracción fundamental. Esta agrupa objetos con características similares. Las clases permiten la abstracción de atributos comunes y conocer el comportamiento de un conjunto de objetos. Una clase describe el protocolo común para cada objeto en una colección y los objetos individuales seguidos de una descripción son llamados instancias. Una descripción de clase sirve para describir todas las instancias de la clase.

Bien, por lo mencionado anteriormente podemos describir a una clase y a una instancia de la siguiente manera:

Clase: Es la descripción de un conjunto de objetos con características y atributos similares.

Instancia: Es un objeto individual descrito por una clase particular.

Algunos ejemplos de clase comunes en los lenguajes de programación son: Integers, Characters, Strings, Booleans, Arrays, etc., y ejemplos de instancias son: un número 3 es una instancia de la clase Integers, la palabra "HOLA" es instancia de la clase Strings, la letra "A" es instancia de la clase Characters, etc.

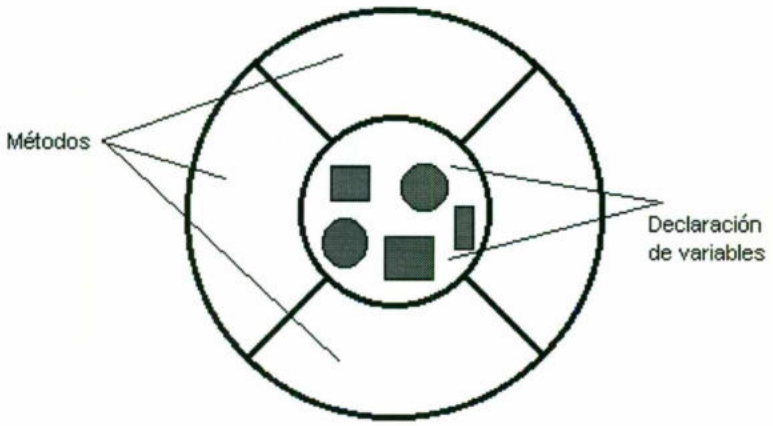
1.3.7.3 ¿Qué son las Clases?

En el muno real tenemos muchos objetos de mismo tipo. Por ejemplo una bicicleta no es mas que una de muchas bicicletas en el mundo. Utilizando la terminología orientada a objetos podemos afirmar que un objeto bicicleta es una instancia de la clase de objetos conocida como bicicletas. Las bicicletas tienen estado y comportamiento en común. Por consiguiente cada estado de la bicicleta es independiente y puede ser diferente del de las otras bicicletas.

Cuando las bicicletas son construidas, los productores toman ventaja del hecho de que tienen características similares por modelo, lo cual podría ser ineficiente para producir un nuevo modelo para las que son producidas.

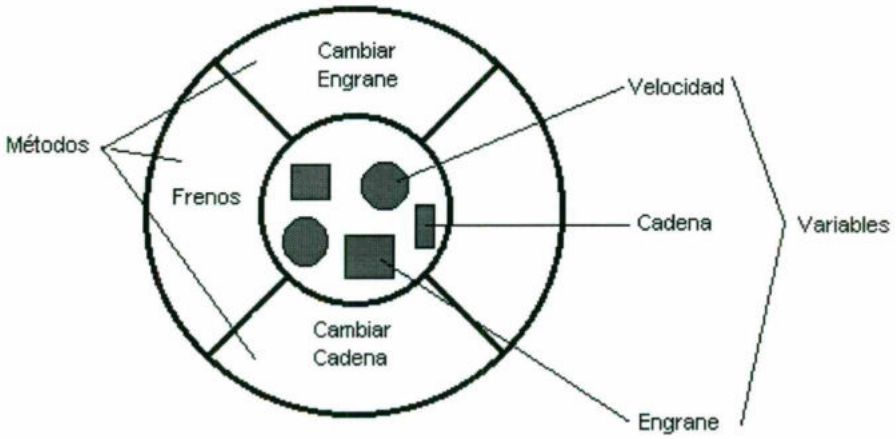
En software orientado a objetos, también es posible tener muchos objetos del mismo tipo que compartan características: los rectángulos, los registros, vídeo clips, etc.. De manera similar que en la construcción de las bicicletas, los programadores pueden tomar ventaja del hecho de que los objetos de la misma clase son similares y se puede crear un modelo para dichos objetos. En éste software los "Modelos" para los objetos son llamados clases. De esto podríamos obtener una nueva definición para una clase:

Una clase es un modelo o prototipo que define las variables y métodos comunes para todos los objetos de cierto tipo.



Por ejemplo, podríamos crear la clase bicicleta que contiene algunas variables para conocer el engrane actual, la cadena, y así sucesivamente para cada objeto bicicleta. La clase debe ser declarada y proporcionar implementaciones para los métodos que permiten cambiar el engrane, los frenos y cambiar la cadena del pedal.

La clase bicicleta.



Los valores para las variables son proporcionados por cada instancia de la clase. De ésta manera, después de ser creada la clase bicicleta, se deberá instanciar (creando una instancia) antes de que pueda ser utilizada. Cuando se crea una instancia de una clase, se crea un objeto del tipo y el sistema operativo asignará memoria para las variables declaradas en la clase.

Hasta éste momento se pueden invocar los métodos del objeto para realizar con ellos operaciones. Las instancias declaradas de la misma clase comparten las implementaciones de los métodos, los cuales residen en la misma clase.

El beneficio de las clases:

Los objetos proveen el beneficio de la modularidad y ocultamiento de información. Las clases proporcionan la reutilización de código. Los programadores al utilizar la misma clase, utilizan el mismo código y por consiguiente pueden crear muchos objetos.

1.3.7.4 Tipos de datos definidos por el Usuario y Clases.

Muchos lenguajes de programación han incorporado el concepto de tipos de datos definidos por el usuario, tales como el ADA y CLU permiten al usuario definir sus propios tipos de datos. Este rasgo a sido referenciado tradicionalmente como "Tipos de Datos Abstractos". El creados de un tipo de dato abstracto es responsable del manejo de los datos y tiene que proporcionar mecanismos para la creación y eliminación de variables de los tipos de datos definidos por el usuario.

Los Types son motivados por el tipo de verificación. Estos pueden ser definidos por un predicado para el reconocimiento de las expresiones del Type. El concepto de una clase es una extensión de la idea de los tipos de datos definidos por el usuario. Las clases son utilizadas como abstracción para los objetos, y son definidas por las plantillas para la creación de objetos. Mientras que los Types tienen una semántica para su chequeo, las clases la tienen para la creación de instancias.

1.3.7.5 Clases e Instancias.

Una clase es una plantilla para los objetos y desde la cual los objetos de instancias pueden ser creados. Cuando la clase es definida, el comportamiento de los objetos de instancia es determinado por el momento de la definición de la misma.

Cuando la clase es diseñada e implementada, es correcto asumir que el implementador es el único usuario para la clase. Tal afirmación o suposición generalmente conduce a la simplificación del diseño. En lugar de lo anterior, la clase deberá ser diseñada para un amplio uso, para evitar las posiciones acerca de los usuarios.

Cuando se crea una instancia de la clase, la memoria para las variables de la instancia es asignada. Cuando los objetos de la instancia se encuentran fuera del alcance, la memoria que ha sido designada para el objeto será reclamada.

Lenguajes de programación OO tales como Smalltalk incorporan mecanismos automáticos de "colección de basura", mientras que otros como C++ no.

En cualquier sistema que modelamos, deben existir objetos que se comuniquen entre sí. Algunos de estos objetos tendrán características comunes y podemos agruparlos de acuerdo a estas.

Con el objeto de describir a todos los objetos que tienen comportamiento y estructura de información similares, podemos identificar y describir una clase que represente estos objetos.

> Una clase es una definición, una plantilla o un molde que habilita la creación de objetos nuevos y es, por lo tanto, una descripción de las características comunes de varios objetos. Los objetos que comprenden una clase tiene esta plantilla en común.

Una clase representa una plantilla para varios objetos y describe la forma como estos se estructuran internamente. Los objetos de la misma clase tienen la misma definición tanto para sus operaciones como para su estructura de información.

Las clase son llamadas algunas veces tipos de los objeto, sin embargo, un tipo y una clase no son la misma cosa. Como hemos mencionado un tipo abstracto de datos es definido por un conjunto de operaciones. Un tipo es definido por las manipulaciones que se pueden hacer con el tipo. Una clase es más que estos, una clase la podríamos ver como una implantación de un tipo.

En los sistemas orientados a objetos, cada objeto pertenece a una clase. La clase describe la estructura (su comportamiento e información) de la instancia, mientras que el estado actual de la instancia esta definido por las operaciones ejecutadas sobre la instancia.

Podemos así definir una clase computadora, y cada objeto que represente a una computadora se convierte en una instancia de esta clase. Como ejemplo, podemos describir la clase computadora, donde IBM, MAC, HP son instancias de esta clase.

1.3.7.6 Clases y Metaclases.

Una clase es un depósito para la información de las instancias. Por ejemplo los métodos de las instancias son almacenados en la clase. Para propósitos de la ejecución, cuando enviamos un mensaje a un objeto se ejecuta un proceso de búsqueda que inicia por:

- Extraer la clase del objeto receptor.
- Buscar en la clase un método con el mismo nombre.
- Si alguno es encontrado, un contexto apropiado para la ejecución es establecido y por consiguiente el método es ejecutado.
- Si ninguno es encontrado, la superclase es invocada y se repite el proceso de búsqueda hasta que el método es encontrado o no existen más superclases, en caso de no encontrar se dispara un mensaje de error.

Ahora ¿Qué sucede si un mensaje es enviado a la clase en vez de la instancia?. Exactamente lo mismo, pero al no realizar el mecanismo de búsqueda al menos, ¿Debe tener un caso especial para probar lo siguiente?:

IF un método es encontrado para una instancia
ENTONCES
    busca en el lugar reservado para los métodos de la instancia
SI NO
    busca en el lugar reservado para los métodos de la clase

La respuesta pudiera haber sido de esta manera, pero se invento una mejor forma. Exactamente el mismo mecanismo puede ser usado para ambos, excluyendo las pruebas de los casos especiales si los métodos para las clases son almacenados en otros objetos en lugar de la misma clase. Dichos objetos son llamados **Metaclases**.

Bien. ahora ya conocemos que las clases son objetos como las instancias, por lo tanto, también tienen una clase.

Una metaclase es un deposito para la información de las clases, esto es una clase para las clases. Cada clase tiene una metaclase.

En algunos lenguajes de programación orientados a objetos tales como Smalltalk, las clases son consideradas por si mismas para ser instancias de una clase llamada metaclase. Por consiguiente la clase también es un objeto, y puede ser manipulada como un objeto. Los mensajes pueden ser enviados a una clase de igual manera como son enviados a sus instancias.

Las clases son instancias de una instancia de la clase metaclase. Esta es una clase de todas las metaclases. De este modo, las clases heredan las propiedades de las metaclases. Desde entonces una metaclase es una clase, y llega a ser en cierto momento una instancia de la metaclase. Cada una de estas tiene exactamente una instancia, es decir, al momento de ser declarada hace referencia a la clase que pertenece.

1.3.7.7 Clases Abstractas.

Mientras que el comportamiento de clase nos ayuda a comprender el comportamiento abstracto de los objetos, no todas las clases necesitan ser asociadas con objetos. Las clases abstractas son definidas únicamente con el propósito de derivar otras clases (subclases) a partir de estas. Las clases abstractas son un concepto importante y proveen un mecanismo para la reutilización de código. Las clases abstractas son comúnmente creadas para servir como la raíz de la jerarquía de clases. De hecho, durante la fase del diseño en la programación orientada a objetos, intenta interactuar con el modelo de comportamiento de los objetos, destacando de una manera considerable la alta necesidad de las clases abstractas.

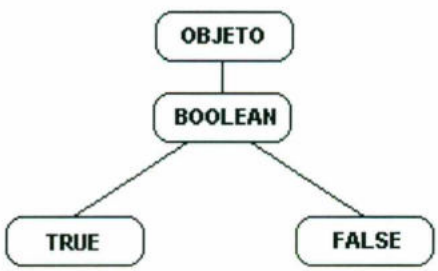
Una clase abstracta actúa como una plantilla para otras clases o mejor dicho como una plantilla para objetos.

Las clases abstractas no proporcionan una implementación compuesta, puesto que tienen algunas funciones virtuales indefinidas llamadas "funciones abstractas". Estas clases definirán otras funciones en términos de las funciones abstractas. Dichas clases derivadas definirán las funciones abstractas, y entonces estarán disponibles para utilizar las funciones heredadas definidas en la primer clase abstracta.

Las implementaciones de las funciones abstractas pueden ser implementaciones figuradas. La idea de las clases abstractas es muy similar con el concepto de "esqueletos de programa" que son empleados para reutilizar la estructura del programa. Sin embargo, cambiando las clases abstractas es mas fácil las modificaciones por el concepto de la herencia que modificar los esqueletos de programas. En el caso de estos últimos, los cambios tienen que ser realizados en cada instancia.

1.3.7.8 Clases Booleanas.

Las clases booleanas se basan en el protocolo general de TRUE o FALSE. Por razones de eficiencia, el protocolo general es especializado vía subclase true o false como es mostrado en la siguiente figura. Los Objetos True o False son las únicas instancias de las clases True o False respectivamente.



La jerarquía Booleana

El protocolo proporcionado por BOOLEAN, genera un pequeño número de métodos genéricos que aplican tanto a True como False. El protocolo para True, en cambio, aplica solo para la instancia true. Lo mismo sucede en el protocolo para False. Por consiguiente si aplicáramos el método de negación (NOT) funcionaría de manera distinta en cada subclase.

MODULO II: EL PARADIGMA DE LA POO.

2.1 EL PROCESO DE DESARROLLO DEL SOFTWARE.

El desarrollo del software es una tarea compleja que abarca múltiples fases de esfuerzo y tiempo.

1. En la primera fase, el problema es analizado para conocer que es lo que queremos hacer.
2. En la segunda fase, todas las posibles soluciones son analizadas e identificada la más efectiva.
3. Por lo que respecta a la tercera fase y con frecuencia la más crucial, es generar una solución para la propuesta a partir de un diseño de alto nivel.
4. Utilizando dicho diseño como una guía, realizaremos un diseño detallado para la implementación de la solución y ésta se genera en la cuarta fase.
5. La programación inicia en la quinta fase después de que todos los detalles han sido eliminados.
6. Mientras los programas son desarrollados, la sexta fase, conocida como la fase de pruebas, comienza. En esta etapa se involucra la unidad de ejecución, integración y pruebas del sistema.
7. Finalmente, cuando el sistema supera las pruebas, es sujeto de una aprobación por parte del usuario.

Algunas de las fases antes mencionadas pueden ser aprobadas en el primer intento, en otras ocasiones esto no sucede así y se tiene que regresar a fases anteriores en un método interactivo.

La mejor manera de desarrollar software es "NO DESARROLLARLO". Reutilizar el software es más rápido que crear nuevo código. Existen varias ventajas para la reutilización de software, la programación es un proceso de "BUG-RIDDEN". Hasta los mejores programas desarrollados por un programador experto pueden tener BUGS. Esto es porque durante el desarrollo del software nunca se realizan de manera extensiva las fases de pruebas. De esta manera, la mayoría del nuevo código desarrollado, es el objetivo por el cual se harán los procedimientos de prueba. Como un corolario de éste principio, la mayoría del código reutilizable, la menor parte de los bugs introducidos y sobre todo el software desarrollado es menor en costos.

La reutilización de código es posible solo si existe una manera eficiente para guardar el código probado de cierta manera en una librería y si existe alguna forma de accesar a dicha librería fácil y eficientemente. La reutilización no es un nuevo concepto. Las librerías de funciones matemáticas y estadísticas deben ser disponibles para dos décadas. Sin embargo, una nueva función implementada para la misma, deberá tener diferentes nombres para los tipos de argumentos, y el desarrollador del software será el responsable de invocar la función apropiada con los argumentos correctos. Por ejemplo, en la mayoría de los sistemas operativos UNIX, las funciones alog(rl), dlog(dpl) y clog(cxl) son usados para calcular el logaritmo natural para números reales, dobles y parámetros complejos, respectivamente. La reutilización de código será mejor promovida generando librerías de funciones de soluciones genéricas para los problemas. De ésta manera existe la necesidad de cambiar la solución para una instancia especifica de un problema, dentro de la solución genérica que puede ser manejada de otras formas del mismo problema.

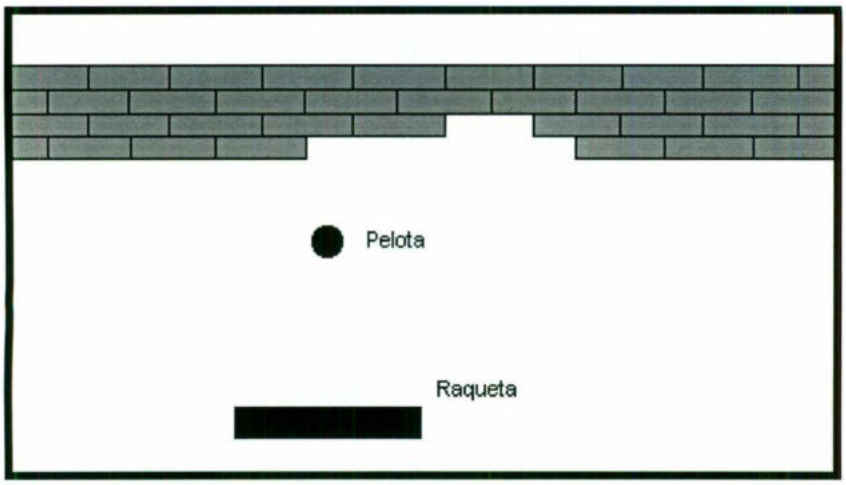
2.2 La POO puede verse como una programación por simulación

La POO es fácilmente descrita como programación por simulación. La metáfora de la programación es basada en la personificación de los objetos físicos o conceptuales del mundo real; por ejemplo, los objetos son los clientes en los negocios, la comida en una tienda de productos o las partes en una fábrica. Trataremos de reencarnar objetos del dominio del problema dentro de los modelos computacionales dando a los objetos del programa las misma características y capacidades como las propias en el mundo real. Este proceso es con frecuencia denominado como la programación antropomórfica o la programación por personificación.

El poder de la simulación como una metáfora de programación puede ser vista como el suceso de interfaces de usuarios basados en ventanas, ahora comúnmente visto en estaciones de trabajo personales.

La computadora Apple de Macintosh por ejemplo, utiliza la metáfora DESKTOP, en la cual los iconos representan objetos comunes de oficina como documentos, carpetas y hasta el icono de basura puede aparecer en el escritorio. De manera interactiva, el usuario puede abrir documentos, copiar, almacenar un documento con otros documentos en una carpeta o enviar un documento a la basura. Las operaciones con los objetos en el DESKTOP, limitan la manera como son manipulados en la vida real. Cuando la implementación tiene relación directa con el problema del objeto, el software resultante es difícil de entender y utilizar.

Considerando la siguiente especificación de problema para un simple vídeo juego. La pantalla del juego es mostrada en la siguiente figura, el objetivo es desaparecer todos los ladrillos de la pared. Cuando la pelota toca un ladrillo éste desaparece, la pelota puede cambiarse de dirección utilizando la barra, la cual el jugador puede mover hacia la izquierda o derecha. A cada jugador se le asignan tres pelotas, una pelota se pierde si no es tocada por la barra y pasa por debajo de ella. Si el jugador desaparece todos los ladrillos es el ganador.



Una solución orientada a objetos para este problema, es simular los objetos como es en la realidad el vídeo juego. El software de los objetos debe ser construido para representar la barra, los lados, la pelota, los ladrillos en la pared, etc. Además, las operaciones en estos objetos deben representar las tareas del dominio-problema tales como el movimiento de la pelota y la barra, determinando si la barra hace contacto con la pelota o bien si la pelota pasa por debajo de ella y se pierde, desaparecer los ladrillos, y así sucesivamente.

2.3 EL PARADIGMA ORIENTADO A OBJETOS

El paradigma de la programación orientada a objetos es un diseño de software y tecnología de desarrollo que incorpora sofisticados y eficientes mecanismos, los cuales contienen una estructura organizacional para el desarrollo de proyectos de software grandes y complejos. En comparación con las técnicas de programación estructurada, la tecnología orientada a objetos se enfoca al desarrollo de sistemas de software con la finalidad de facilitar la funcionalidad de la producción y relación de los datos. Para algunos investigadores, el paradigma

orientado a objetos es mucho más que el mejor método para el desarrollo de software; éste es el objetivo hacia el cual desarrolladores y diseñadores de sistemas de software desearían llegar.

La programación orientada a objetos es un método "orientado a objetos" para el diseño y desarrollo de software, donde los datos son encapsulados en objetos y son manejados por mensajes. Un objeto son datos encapsulados que pueden ser accesados o manipulados por medio de funciones de interface o manejo. Un mensaje es el mecanismo por medio del cual una operación es ejecutada con los datos encapsulados dentro de un objeto. De este modo un objeto es definido en términos de datos encapsulados y las operaciones que dichos datos permiten por medio de las funciones de interface.

El encapsulamiento de los datos permite el ocultamiento de los mismos. El actual método de almacenamiento de datos encapsulados es una implementación detallada, la cual es independiente de la manera como son usados los datos. Las operaciones que pueden ser ejecutadas por medio del encapsulamiento de los datos son especificadas como parte de la interface del objeto. Estas operaciones son también llamadas como "funciones de interface". La implementación de dichas funciones es de manera interna para el objeto, los detalles de la implementación de las operaciones que manipulan los datos almacenados pueden ser cambiados por fuera afectando la interface. De esta manera, el concepto de un objeto contempla el ocultamiento de información y la abstracción de los datos.

La manera en que un objeto puede ser manipulado es por medio de la asignación de las funciones de interface. Las operaciones que son definidas, son operaciones genéricas y son aplicables a otros objetos similares, todos los objetos que tienen características similares pertenecen a la misma clase. El concepto de una clase es importante para el paradigma orientado a objetos, una clase es una definición abstracta de las características de los objetos que tienen apariencias similares y que permiten operaciones semejantes para ejecutar los datos encapsulados.

De esta manera, una clase es una definición abstracta de los datos encapsulados existentes junto con la definición del conjunto de operaciones que pueden ser ejecutadas en las instancias de la clase. Un objeto es una instancia de una clase particular y es completamente distinta de otras instancias de la misma clase.

El concepto de herencia es un punto central del paradigma de la programación orientada a objetos. La herencia permite la creación de nuevas clases utilizando

2.3.2 Características de las Técnicas Orientadas a Objetos.

El análisis y diseño orientado a objetos tiene algunas características importantes:

- Cambian nuestra forma de pensar sobre los sistemas. Para la mayoría de las personas, la forma de pensamiento orientado a objetos es más natural que el estructurado. Después de todo, el universo esta orientado por objetos. Al principio de nuestra vida comenzamos a aprender sobre ellos y descubrimos que tienen diferente comportamiento y los categorizamos.

- Los sistemas suelen construirse a partir de objetos ya existentes, llevando un alto grado de reutilización, ahorro de dinero, menor tiempo en desarrollo y mayor confiabilidad del sistema.

- La complejidad de los objetos que podemos utilizar sigue en aumento, puesto que los nuevos objetos se construyen a partir de otros objetos, y estos a su vez están construidos a partir de otros objetos, etc.

- El depósito CASE debe contener una creciente biblioteca de tipos de objetos, algunos comprados y otros construidos. Existe mucha probabilidad de que estos tipos de objetos sean más poderosos conforme crezca su complejidad. La mayoría de este tipo de objetos será diseñados de forma que se adapten a las necesidades de los diferentes sistemas.

- La creación de sistemas con un funcionamiento correcto es más fácil con las técnicas OO. Esto se debe, en parte, a que las clases OO están diseñadas para reutilizarse; y en parte, a que las clases están autocontenidas y divididas en métodos. Cada método se puede construir, depurar y modificar con relativa facilidad.

- Las técnicas OO se ajustan de manera natural a la tecnología CASE. Existen ciertas herramientas elegantes y poderosas para la implantación OO. Muchas otras herramientas CASE necesitan ciertas mejoras para controlar el análisis y diseño orientado a objetos.

2.3.3 Beneficios de la Tecnología Orientada a Objetos

Muchos de los beneficios sólo se alcanzan cuando el análisis y diseño orientado a objetos se utiliza con las herramientas OO de CASE, generadoras de códigos basadas en depósitos.

- **Reutilización.** Las clases están diseñadas para que se utilicen en muchos sistemas. Un objetivo fundamental de las técnicas OO es lograr la reutilización masiva al construir un software.

- **Mejor comunicación entre los desarrolladores de sistemas de información y los empresarios.** Los empresarios comprenden mas fácilmente el paradigma OO, piensan en términos de objetos, eventos y políticas empresariales que describen el comportamiento de los objetos.

- **Interacción.** El software de varios proveedores puede funcionar como un conjunto. Un proveedor puede utilizar clases de otros, pues ya que existe una forma estándar de localizar clases e interactuar con ellas.

- **Migración.** Las aplicaciones ya existentes, sean OO o no, pueden preservarse si se ajustan a un contenedor OO, de modo que la comunicación con ellas sea a través de mensajes estándar OO.

- **Bibliotecas de clases para las empresas.** Las compañias de software venden bibliotecas para las diversas áreas de aplicación. Las bibliotecas de clases independientes de la aplicación también son importantes y se proporcionan mejor como una capacidad de las herramientas CASE.

MODULO III: EL LENGUAJE SMALLTALK.

3.1 INTRODUCCION

Convertirse en un programador productivo en Smalltalk requiere mucho más que conocer el lenguaje. Uno debe familiarizarse con el uso de las herramientas que brinda *el* entorno de programación Smalltalk y, quizá lo más importante de todo es manejar la extensa biblioteca de clases existentes en el sistema Smalltalk. Es fundamental la experimentación interactiva y en línea. El beneficio que puede obtenerse con todo este esfuerzo, es la capacidad que se tiene para desarrollar aplicaciones orientadas a objetos de gran calidad como son aplicaciones gráficas con todas las características de las interfaces modernas de usuario (por ejemplo, ventanas, menús, interacción con el ratón, etc.), todo esto a un bajo costo. Las aplicaciones en Smalltalk pueden ser desarrolladas con una alta productividad debido a la filosofía de Smalltalk basada en la reutilización de código existente en lugar de rehacer código.

La programación en Smalltalk exige un conocimiento sólido de los siguientes aspectos:

- Los elementos fundamentales del lenguaje; como objetos, mensajes, clases y herencia.
- La sintaxis y semántica de Smalltalk.
- Cómo interactuar con el entorno de programación de Smalltalk para construir nuevas aplicaciones (Smalltalk es un lenguaje interactivo donde se aprende haciendo o explorando la programación) y
- El sistema fundamental de clases, tal como las clases numéricas, de manejo de colecciones, gráficas y de interfaces de usuario. El diseño de nuevas aplicaciones en Smalltalk requiere conocimiento de las capacidades existentes del sistema Smalltalk. La programación en Smalltalk es con frecuencia llamada **programación por extensión.** Las aplicaciones nuevas se construyen extendiendo la biblioteca de clases existente en Smalltalk.

En esta parte, consideramos los primeros dos procedimientos así como una introducción al entorno de programación. Durante las prácticas del curso hablaremos sobre el entorno de programación y se explicarán algunas bibliotecas de funciones.

Suponemos que el participante es un programador con alguna experiencia en un lenguaje estructurado tradicional. Siempre que esto sea posible, haremos un contraste entre el código de otro lenguaje equivalente a los programas Smalltalk que presentemos, hacemos esto para acelerar el proceso de explicación,

3.2 OBJETOS EN SMALLTALK

Como mencionamos antes, los elementos en Smalltalk son los objetos. Los componentes del sistema (como compilador y depurador), los elementos primitivos de datos (tales como enteros, variables lógicas y caracteres), así como los elementos gráficos (tales como áreas rectangulares, plumas para dibujar y mapas de bits) son todos ellos objetos. Como veremos después en este módulo, las estructuras de control son implantadas por medio del paso de mensajes a objetos.

¿Qué es un objeto en Smalltalk ?

Método Es un sinónimo de operación. Es invocado cuando un objeto recibe un mensaje.

Objeto Es un componente del sistema Smalltalk representado por algún dato privado y un conjunto de métodos u operaciones.

Conceptualmente, un objeto puede ser entendido como una computadora virtual con una memoria y un conjunto de instrucciones propio. Un objeto es también capaz de realizar cómputo. Puede responder a algunos mensajes dentro de un conjunto predefinido. Este conjunto de mensajes es llamado **protocolo de mensajes** que soporta el objeto. Cuando un objeto recibe un mensaje, él debe decidir primero si "entiende" el mensaje y en caso afirmativo debe generar una respuesta. Si un objeto puede responder a un mensaje de manera directa, entonces se selecciona un método o función correspondiente al mensaje y se evalúa. El resultado de la evaluación del método se regresa al emisor de mensaje como resultado.

Protocolo de mensajes. Es el conjunto de mensajes que un objeto puede responder.

Como un ejemplo concreto de objeto Smalltalk, supongamos que tenemos un objeto, digamos un punto, que representa una posición en la pantalla de despliegue de Smalltalk. El estado de un punto podrá contener dos componentes: primero, un objeto que representa la coordenadas de la posición del objeto en la pantalla y segundo, la coordenadas. En términos de otro lenguaje, podemos pensar que un punto es un registro con dos campos: coordenadas X y coordenadas Y.

¿Qué protocolo de mensajes debe ser soportado por el objeto unpunto? Supongamos que unpunto permite al emisor preguntar sus coordenadas **X** e **Y** soportando el protocolo **X** e **Y**. Por ejemplo, el envío del mensaje **X** a unpunto se hace con la siguiente expresión en Smalltalk:

unpunto **X**

que regresará un objeto que representa su coordenada X. Para que nuestra discusión tenga un mayor interés, supongamos que un punto soporta también el protocolo **distanciaDe**: otropunto. El efecto de enviar el mensaje **distanciade:** a unpunto mediante una expresión en Smalltalk de la forma

unpunto **distanciaDe:** otropunto

es regresar la distancia entre un punto y otropunto.

Ocultamiento de información: Vistas interna y externa de un objeto.

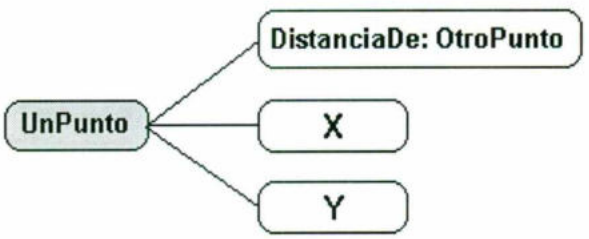
Los objetos en Smalltalk encapsulan tanto datos como procedimientos. Ellos soportan el concepto de **ocultamiento de información.** Para controlar la complejidad en programas muy grandes, debemos particionarnos en módulos. Más aún, debemos ocultar tanta información como se pueda dentro de un módulo y minimizar la interfaz presentada a los usuarios.

Es útil pensar que los objetos en Smalltalk brindan dos vistas diferentes: una para los usuarios del objeto y otra para quienes implantan el objeto. Llamaremos a esas vistas externas e internas respectivamente.

La **vista interna** describe la representación del objeto y los algoritmos para implementar los métodos (u operaciones).

La **vista externa** es la vista del objeto como lo perciben los otros objetos.

La vista externa, o lo que se puede hacer con el objeto, se describe por su protocolo de mensajes. Para un usuario, la vista interna de un objeto es privada. Esta es propiedad del objeto y no puede ser manipulada por otros objetos a menos que se les brinde un protocolo específico que permita hacerlo. Por ejemplo, en la siguiente Figura, se muestra la vista externa de **un punto.** Si **un punto** no soporta un protocolo para tener acceso a sus coordenadas X e Y, será imposible que otro objeto tenga acceso a esta información. La única manera de preguntar si un objeto realiza cierto proceso es enviando un mensaje.



Esta idea contrasta fuertemente con la forma de programar en otro lenguaje, que prácticamente no brinda soporte para el ocultamiento de detalles de implantación. Si queremos tener acceso *o* modificar el campo **coordenadas** de **unpunto,** podemos hacerlo fácilmente con una expresión de la forma

unPuntocoordenadaX

Más aun en otro lenguaje, no tenemos manera de prevenir que un programador tenga acceso directo a la representación de **unpunto,** mientras que en Smalltalk si, porque a menos que el programador brinde métodos específicos de acceso como parte del protocolo de mensajes del objeto, es imposible usar la estructura interna del objeto.

Objetos Literales

Hay ciertos tipos de objetos en Smalltalk que se pueden describir literalmente. Por ejemplo, se usan literales para describir **números, símbolos, caracteres, cadenas** y **arreglos.** Este tipo de objetos serán discutidos con mayor detalle un poco más adelante. Por el momento solo veremos algunos ejemplos.

SMALLTALK	COMENTARIO
34	El entero 34.
-17.62	El número en punto flotante -17.62.
1.56e-3	El número en punto flotante 0.00156 escrito en forma exponencial.
'una cadena'	Una cadena de caracteres.
#soluciones	Un símbolo con el nombre soluciones. Cada símbolo es único dos símbolos con el mismo nombre no pueden coexistir.
$c	La letra minúscula c.
#(-15 'una cadena' $c)	Un arreglo de tres objetos. A diferencia de otro lenguaje, los objetos dentro de un arreglo no tienen necesariamente que ser homogéneos. Los objetos individuales dentro de un arreglo pueden ser reverenciados usando índices enteros que van de 1 al tamaño del arreglo.

3.3 ENVIO DE MENSAJES

Los mensajes en Smalltalk describen quién recibe el mensaje, qué operación debe ser seleccionada, y qué argumentos se necesitan para llevar a cabo la operación requerida. Los componentes del mensaje se llaman el **receptor,** el **selector,** y los **argumentos** respectivamente. Por ejemplo, en la expresión en Smalltalk

1+5

el entero 1 es el receptor del mensaje, + es el selector que identifica de manera única la operación a ser seleccionada, y el 5 el argumento necesario para llevar a cabo la operación. Lo más importante en la nueva terminología involucrada aquí es la manera como se evalúa esta expresión. Como se ilustra en la siguiente Figura, las expresiones en Smalltalk y otro lenguaje son evaluadas de maneras muy diferentes.

Operadores unarios

Los mensajes de los operadores unarios no tienen argumentos, sólo un receptor y un selector. Ellos son equivalentes a las funciones en otro lenguaje con un solo argumento. Se dan a continuación algunos ejemplos.

Smalltalk	Comentario
5 factorial	Se envía el mensaje **factorial** al objeto entero 5. El resultado es el objeto entero **120.**
16.79 rounded	Se envía el mensaje **rounded** al número en punto flotante **16.79.** El resultado es el objeto entero **17.**
$a **asInteger**	Se envía el mensaje **asInteger** al objeto carácter **a.** El resultado es el objeto entero que representa el valor ordinal del carácter.
'abcdef' **size**	Se envía el mensaje **size** a **la cadeda'abcdef'.** El resultado es el objeto entero **6** que representa a la longitud de la cadena.
3 **negated**	Se envía el mensaje **negado al** entero. **3.** El no equivalente +3 resultado es el objeto entero **-3.**

Mensajes Binarios

Además del receptor, los mensajes binarios tienen un argumento. Ellos son equivalentes a los operadores binarios de otro lenguaje estructurado. Los selectores de mensajes binarios tienen uno o dos caracteres. Los caracteres selectores simples incluyen las operaciones aritméticas y de comparación comunes tales como +, -, *, /, <, >, =. Los caracteres selectores dobles incluyen operadores como ~= (no igual), <= (menor o igual que), >= (mayor o igual que),y // (división de números enteros. Los siguientes son ejemplos de mensajes binarios.

Smalltalk	Comentario
55 + 100	Se envía el mensaje +100 al entero 55. El selector es + y el argumento es 100. El resultado es el entero 155.
'abc'~='def'	Se envía el mensaje **~='def** ' al objeto cadena **'abc'.** El selector es ~= y el argumento es la cadena 'def'. El resultado es el objeto lógico **true.**

Mensajes con palabras clave

Los mensajes con palabras clave contienen una o más palabras, cada una de ellas tiene un argumento simple. Los nombres de las palabras clave terminan siempre con dos puntos (:). Los dos puntos son parte del nombre (no son un terminador especial). Los mensajes con palabras clave son equivalentes a las funciones un lenguaje de programación estructurada, con uno o más argumentos. Veamos algunos ejemplos:

Smalltalk	Comentario
28 **gcd:12**	Se envía el mensaje **gcd: 12** al entero 28.El selector es **gcd:** y el argumento es el entero **12.** El resultado es el máximo común divisor del receptor **28** y el argumento **12,** esto es, el objeto entero **4.**
#(4 3 2 1)**at:4**	El mensaje **at: 4** es enviado al arreglo que contiene **(4 3 2 1).** El selector es **at:** y **el** argumento **es** el entero **4.** El resultado devuelto es el objeto entero 1, que corresponde con el objeto asociado con el índice **4** en el arreglo.
5 between: 3 **and:** 12	El mensaje **between:3 and: 12** es enviado al entero 5. El selector es between:and: y los argumentos son los enteros **3** y **12** respectivamente. El resultado devuelto es el objeto **true** ya que 5 está en el rango 3 a 12 inclusive.

En este caso, el selector es un mensaje formado por la concatenación conjunta de todas las palabras clave en el mensaje; por ejemplo, **between:and:.** Las mismas palabras clave pueden aparecer en diferentes mensajes selectores, sin embargo, la concatenación de palabras clave forma un selector único.

Evaluación de Expresiones Mensaje.

El receptor o el argumento en las expresiones mensaje pueden ser a su vez un mensaje o una expresión. Esto da lugar a complejas expresiones mensaje y a la necesidad de un orden de evaluación. Por ejemplo, el siguiente mensaje expresión contiene mensajes unarios, binarios, y con palabras clave.

4 factorial gcd: 4 * 6

Muchos lenguajes, basan la evaluación de expresiones en prioridades asignadas a diferentes operadores. Por ejemplo, a la multiplicación (*) se le asigna usualmente una prioridad mayor que a la suma (+). En Smalltalk las reglas de evaluación, sin embargo, están basadas en el tipo de mensajes (unarios, binarios, y palabras clave) involucrados en la expresión. El orden de aplicación es el siguiente:

1. Expresiones con **paréntesis**
2. Expresiones **unarias** (evaluadas de izquierda a derecha)
3. Expresiones **binarias** (evaluadas de izquierda a derecha)
4. Expresiones con **palabras clave**

Note que todas las operaciones binarias tienen el mismo nivel de prioridad.

El uso completo de paréntesis en una expresión mensaje elimina la ambigüedad en el orden de evaluación. Cada uno de los ejemplos que siguen muestran una expresión y la expresión con paréntesis equivalente.

Expresión	Expresión con paréntesis
2 factorial negado	(2 factorial) negado
3+4*6+3	((3+4)*6)+3
15 gcd: 32//3	15 gcd: (32//3)
2 factorial + 4	(2 factorial) + 4
4 factorial gcd: 4*6	(4 factorial) gcd: (4*6)

Mensajes en cascada

Los mensajes en cascada son una manera precisa de especificar que muchos mensajes son enviados al mismo receptor. Un **mensaje en cascada** consiste de series de expresiones mensaje separadas por puntos y coma (;), donde la primer expresión del mensaje especifica el receptor común. Por ejemplo, imagine que deseamos modificar los primeros tres elementos de un arreglo unarreglo. El mensaje **at:** índice **put:** unvalor modifica un elemento del arreglo, Podemos enviar sucesivamente a unarreglo los tres mensajes que se muestran a continuación:

unarreglo **at: 1 put:** 3. unarreglo **at:** 2 **put:** 8. unarreglo **at:** 3 **put:** 5

alternativamente se puede usar la expresión de mensajes en cascada

unarreglo **at: 1** put: 3; **at:** 2 **put:** 8; **at:** 3 **put:** 5

No se especifica receptor para el segundo y tercer mensaje **at:put:** implícitamente el receptor es el mismo que el mensaje que precede al primer punto y coma. El resultado de evaluar una expresión en cascada es el resultado de enviar el último mensaje en la cascada. En este caso, como **at:put:** regresa el valor modificado, el resultado regresado podría ser 5. Para regresar el arreglo modificado como un resultado, el mensaje **yourself** debe ser agregado a la cascada.

unarreglo **at: 1 put:** 3, **at:** 2 **put:** 8, **at:** 3 **put:** 5 **yourself**

Cuando un objeto recibe el mensaje **yourself** se regresa el objeto (o receptor) mismo.

Asociación dinámica y sobrecarga de mensajes

El mismo mensaje puede ser interpretado de diferentes maneras en objetos diferentes. Por ejemplo, considere los siguientes casos:

5+ 100

(200 @ 200) + 1 00

Ambos ejemplos usan el mensaje + 100 pero los objetos receptores reaccionan al mensaje de maneras muy diferentes. En el primer ejemplo, el receptor es el entero 5 y el selector + es interpretado como una suma entera, en el segundo, el receptor es el punto coordenadas x e y iguales a 200 (el selector binario @ cuando es enviado a un entero crea una instancia inicializada de la clase Point de Smalltalk). En esta expresión, el selector + es interpretado como una indicación de suma definida sobre los puntos, así es que regresa el punto con coordenadas x e y iguales a 300.

Como discutimos antes, es el receptor del mensaje quien determina como se interpretan los mensajes. Esto significa que el mismo mensaje enviado a objetos diferentes producirá resultados diferentes. Por ejemplo, podríamos usar el selector genérico **printstring** para generar presentaciones impresas de puntos, rectángulos, etc.. Por ejemplo:

unpunto **printstring**	Imprime un punto en la forma x@y; ejemplo 100@200
unrectangulo **printstring**	Imprime un rectángulo en la forma "punto0rigen esquina: puntoesquina"; por ejemplo, 100@ 100 esquina: 200@ 200.

El método actual de impresión invocado por una expresión como unobjeto **printstring** es determinado por el tipo del mensaje recibido.

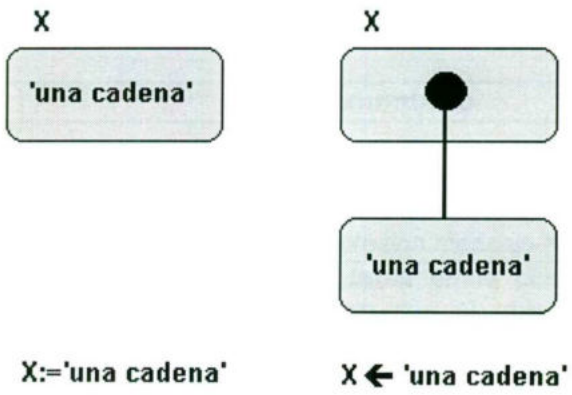
Cuando el mismo selector es aceptado por clases diferentes, decimos, en la terminología del lenguaje de programación que el selector es sobrecargado.

Variables y Asignamientos

Los nombres de variables en Smalltalk son simples identificadores que consisten de una secuencia de letras y dígitos que inician con una letra. Aunque ellos tienen la misma sintaxis en otros lenguajes, las variables en Smalltalk son muy diferentes. Todas las variables en Smalltalk son objetos apuntador o variables apuntador. En Smalltalk, un asignamiento tomaría la forma:

X := 'UNA CADENA'

y podemos decir que la variable x es asociada (o apunta a) el objeto 'una cadena'. (ver la sig. figura).



En Smalltak los nombres de variables son usados para hacer referencia a un objeto apuntado por la variable. Las expresiones de asignamiento son usadas

compilación, en contraste no se especifica tipo alguno cuando se declara una variable Smalltalk. Por ejemplo, una variable declarada como de tipo entero no puede tomar subsecuentemente un valor carácter. Por el contrario, las variables Smalltalk, no tienen control de tipos. Los objetos en Smalltalk tienen tipos pero no las variables que hacen referencia a ellos.

Creación y Destrucción de objetos.

Otra diferencia importante entre algunos lenguajes de programación estructurada y Smalltalk es el método de creación y destrucción dinámicos de objetos. En dichos lenguajes, es necesario retirar explícitamente objetos cuando ellos ya no son necesarios, en Smalltalk este proceso es manejado de manera automática. El ejemplo siguiente ilustra la creación dinámica de una arreglo de tamaño 10 que se almacena en la variable **unatabla.** El proceso de creación de Smalltalk envía un mensaje **new:** a la clase **Array.**

Smalltalk
| una tabla | unaTabla:=Array **new: 10**

Estructuras de control con paso de mensajes.

Las estructuras de control se implantan en términos de objetos y paso de mensajes. En particular, las estructuras de control de Smalltalk involucran una clase de objeto conocido como un **bloque.** Por el momento, un bloque puede ser concebido de manera sintáctica, como el análogo a la construcción **begin ... end**; es decir, simplemente como una manera de agrupar un conjunto de proposiciones. Un bloque literal consiste de una secuencia de expresiones separadas por puntos y delimitados por paréntesis cuadrados; por ejemplo:

[x:=3.y:=4]

El resultado regresado cuando se evalúa un bloque es el objeto regresado por la evaluación de la última expresión en el bloque, El bloque vacío [ ] regresa el objeto especial **nil** cuando se evalúa.

A continuación se da un resumen de las estructuras de control en Smalltalk.

Smalltalk

Asignamiento
variable := expresión

Selección Condicional
expresión lógica
  ifTrue:[
        bloque verdad]
  ifFalse:[
        bloque falso]

expresión lógica
 ifTrue:[
        bloque verdad]

expresión lógica.
 ifFalse: [
        bloque falso]

Repetición condicional

[expresión lógica] **whiletrue:**[
      cuerpo del ciclo]

[expresión lógica] **whileFalse:**[
      cuerpo del ciclo]

Repeticiones de longitud fija(enteros únicamente)

valor inicial **to:** valor final **do:**
 [variable de control del ciclo |
      cuerpo del ciclo]

valor de repetición **timesRepeat:**[
      cuerpo del ciclo]

Selección Condicional.

Las estructuras de control para la selección condicional son expresadas usando bloques como se muestra a continuación, Veamos un ejemplo que muestra el cálculo del mayor y menor números de una pareja de números enteros dados.

Smalltalk

```
número1 < número2
i(True:[
            máximo := número2.
            mínimo := número1]
ifFalse: [
            máximo := numero 1.
            mínimo := número2]
```

El valor regresado por la selección condicional es el valor del bloque evaluado. En este ejemplo, será regresado el objeto asociado a **mínimo** ya que una expresión de asignamiento de la forma **mínimo := número2** es la última proposición en los dos bloques de la selección condicional.

Como la selección condicional es implantada en términos de expresiones mensaje, una selección condicional puede ser inmergida dentro de una expresión mensaje. Por ejemplo, la siguiente expresión calcula el valor absoluto de un número entero dado.

numero := numero >=0 **if False:[**número **negated] ifTrue:**[numero]

Repetición condicional

Smalltalk brinda una forma de repetición condicional equivalente a la proposición **while ... do.** Veamos un ejemplo de su uso.

```
suma := 0.
numero := 1.
[numero <= 100] whiletrue:[
            suma := suma + numero.
            numero := numero + 1].
suma
```

Este programa debe interpretarse como sigue:

1. El mensaje **whileTrue**:[...] es enviado al bloque **[numero := 100].**

2. En respuesta al mensaje **whiletrue:** se evalúa por si mismo el bloque receptor**[numero <= 100].**

3. Si el bloque evaluado regresa el objeto **true,** el bloque argumento de **whiletrue:** es evaluado y el mensaje **whiletrue:** se envía nuevamente al bloque **[numero<100],** y se repiten los pasos 1,2 y 3.

Si el bloque de evaluación regresa el objeto **false,** el bloque argumento de **whiletrue** no es evaluado nuevamente y el mensaje **whiletrue:** regresa el objeto especial **nil** como el resultado.

Note que el resultado regresado por un mensaje **whiletrue:** es siempre **nil.** Para regresar el resultado requerido cuando se evalúa un fragmento de código Smalltalk, evaluamos la variable **suma** inmediatamente después del ciclo while.

El efecto de enviar un mensaje **whiletrue:** a un bloque es evaluar repetidamente el bloque argumento con el bloque receptor de los mensajes evaluados por true. Esto brinda una estructura condicional que se repite cero o más veces.

El funcionamiento del mensaje **whilefalse:** es similar al del mensaje **whiletrue:,** el efecto de enviar un mensaje **whilefalse:** es evaluar repetidamente el bloque argumento, mientras que el mensaje recibido por el bloque sea evaluado como falso. El resultado que regresa **whilefalse:** siempre es el objeto especial **nil.** El ejemplo anterior se puede reescribir usando ahora un **whilefalse.**

```
suma := 0.
numero := 1.
[numero > 100] whilefalse:[
            suma := suma + numero.
            numero := numero + 1 ]
suma
```

Repetición de un bloque un número fijo de veces

El manejo de ciclos iterativos de manera determinista está dado por el mensaje **to: finalvalue do: ablock,** que está definido para los enteros. El argumento de la palabra clave do: es un sólo argumento bloque. Los argumentos bloque son declarados al principio del bloque y separados de las expresiones en el bloque por una barra Cada nombre argumento en el bloque debe ser precedido sintácticamente por dos puntos. El ejemplo anterior para calcular la suma de los primeros 100 enteros puede ser recodificado como sigue:

```
suma := 0.
1 to: 100 by: 2 do: [índice|
        suma := suma + índice].
suma
```

El mensaje **to: finalValue by:** stepvalue **do:** ablock definido para los enteros es una variante del mensaje **to: do:** que especifica qué tanto debe ser incrementado el índice para controlar la iteración. Para valores positivos de paso, la evaluación repetida termina cuando el índice de ciclo es más grande que el valor final. Para pasos negativos, el ciclo termina cuando es menor que el valor final.

Como veremos después, esta es una manera simple de agregar métodos a los enteros para que soporten estructuras de control adicionales. Por ejemplo, podríamos agregar al un bloque **downto:** finalvalue **do: aBlock,** donde el argumento del bloque es decrementado en 1, en lugar de ser incrementado, en cada evaluación del bloque.

Una forma simple de ciclo determinístico está dado por el mensaje **timesrepeat:** que evalúa el bloque sin argumentos un número fijo de veces. El número de evaluaciones es especificado por el receptor que debe ser un entero. Por ejemplo, la expresión

```
5 timesrepeat [...]
```

que evalúa el bloque cinco veces.

En Smalltalk para hacer la iteración un número finito de veces se utiliza la instrucción **do:** que se aplica a muchas clases de objetos. Por ejemplo, podemos escribir

```
(1 to: 10) do: [:loopIndex| ... code]
#(red green blue) do [:loopIndex| ... code... ]
#(5'hi' 1.5) do: [:loopIndex|... code... ]
aset do: [:loopIndex| ... code... ]
anOrderedCollection do: [:loopIndex|... code... ]
```

En Smalltalk, las estructuras de control son implantadas mediante paso de mensajes entre objetos. Consecuentemente, podemos implantar el **do:** para cada clase diferente de objeto sobre la cual queremos iterar. Esta es una gran ventaja porque las estructuras de control pueden ser construidas no sólo para iterar sobre rangos simples de enteros, sino también puede usar los elementos de estructuras de datos como arreglos, listas, árboles, cuentas de banco, o elementos de un circuito.

Un ejemplo: Prueba de primalidad

Para ilustrar los conceptos recién vistos sobre estructuras de control, mostraremos un ejemplo mayor, en el que desarrollaremos un programa para probar si un número dado es primo. Para nuestros propósitos, un número es **primo** si es positivo y divisible únicamente entre si mismo y entre 1. El algoritmo usado rechaza todos los números pares mayores - de 2. Para números impares mayores que 3, se divide el entero sucesivamente entre divisores impares de ensayo hasta que se encuentra un número que divide de manera exacta, en cuyo caso el número original no es primo, o bien se alcanza al mismo número que se divide con lo que finalmente se determina que el número es primo.

```
esprimo
|candidato divisor|
candidato := self
candidato <= 0 ifTrue: [^truel.
(candidato >=1 & (candidato <= 3)) ifTrue:[^true].
(candidato \\ 2)= 0 ifTrue: [^false].
divisor := 3.
[divisor*divisor <= candidato] whiletrue:[
            (candidato \\ divisor) = 0
                        ifTrue:[^false]
                        ifFalse:[divisor := divisor + 2] ].
^true
```

En Smalltalk, una expresión precedida por un carácter '^' es llamada una **expresión return.** Una expresión return indica que el resultado de evaluar la expresión que sigue al "^" es el resultado que se regresa al código de evaluación una vez que se término.

Se introducen dos selectores no familiares: & y \\. El selector & denota la operación **and,** su resultado es cierto si tanto el receptor como el argumento son verdaderos. El selector \\ definido sobre los enteros regresa el residuo entero cuando el receptor es dividido entre el argumento.

Como en otros lenguajes, las variables deben ser declaradas antes que ellas puedan ser usadas. Pero como se explicó antes, las variables no tienen tipos, de aquí que su declaración sea innecesaria. La forma

```
| candidato divisor |
```

declara **candidato** y **divisor** como **variables temporales.** Estas variables existen solo mientras exista el fragmento de código a ser evaluado. Todas las variables temporales son asignadas al objeto temporal **nil.**

En Smalltalk, el receptor del mensaje **esPrimo** es llamado **self.** Se hace un asignamiento explícito de **self** a una variable local, vía el asignamiento **candidato := self,** sin embargo, esto es en realidad superfluo porque **self** puede ser reverenciado en cualquier lugar en el método.

Estructuras de control definidas por el usuario

No hay analogía en Smalltalk para las proposiciones **repeat ... until** o case que se tienen en otros lenguajes. Sin embargo, como las estructuras de control se implantan enviando mensajes a objetos, es posible que el programador agregue estructuras de control al sistema. A continuación introduciremos algunos ejemplos sobre estructuras de control avanzadas que se pueden crear en Smalltalk. Sin embargo, no entraremos en demasiados detalles sobre su implantación.

Muy a menudo queremos aplicar una función a cada elemento o estructura de datos tal como arreglo, Vista o árbol. Por ejemplo, es tradicional sumar los elementos de un arreglo extrayéndolos sucesivamente. De manera alterna, podríamos desear imprimir los valores almacenados en un árbol binario o recorrer los nodos del árbol en algún orden específico tal como post-orden.

Cada una de estas tareas nos exige generar sucesivamente los elementos de una estructura de datos para aplicarles una función. A tales formas de control se les llama generadores. los programadores en Lisp llaman a estas funciones **mapping tuactiotti.** El mensaje **do:aBlock,** cuando es enviado a un arreglo receptor, envía sucesivamente cada objeto en el arreglo como el argumento a un bloque con un solo argumento, que sintácticamente se coloca después de la palabra **do.** Por ejemplo, el código

```
suma := 0
tabla do: [:elemento| suma := suma + elemento]
```

suma los elementos en el arreglo **tabla.** Conforme se genera cada objeto en el arreglo, este es asignado al argumento **elemento** del bloque y se evalúa la expresión mensaje **suma := suma + elemento.**

Son posibles muchas otras variantes de esta forma. Veamos un ejemplo. En lugar de aplicar una función a cada objeto de un arreglo, tomaremos aquellos objetos que satisfacen alguna restricción. Tal vez queremos colectar en un arreglo nuevo sólo los elementos distintos de cero que están en el arreglo existente. Esto obliga a generar cada elemento del arreglo original, probar si éste es cero o no, y si no lo es entonces se le agrega al nuevo arreglo. La expresión

```
#(0 3 0 4 2 0 0) select: [:elemento ~= 0]
```

regresa al arreglo #(3 4 2) corno resultado; es decir, colecta los elementos no cero en una instancia de la misma clase que el receptor. Un último ejemplo es:

#(1 2 3 4 5) collect:[:elemento|elemento **squared]**

La cual regresa #(1 4 9 16 25).

3.4 CLASES

Diseño de una clase nueva

Las clases son el mecanismo de abstracción fundamental en Smalltalk. Sirven para agrupar objetos con características similares.

El primer paso en el diseño de una clase nueva es desarrollar una especificación para la clase; es decir, definir el protocolo de mensajes o **vista exterior** de la nueva clase. La especificación debe brindar toda la información requerida por los usuarios de la clase pero únicamente esa información. La especificación de una clase debe completarse antes de hacer las consideraciones de implantación. El desarrollo de la especificación consiste de los tres pasos siguientes:

1. Listar los nombres de las operaciones requeridas
2. Especificar con todo detalle los parámetros de cada operación
3. Especificar la semántica de cada operación de manera informal

Para ejemplificar este proceso, trabajaremos en las siguientes secciones con un ejemplo para el manejo de números complejos. Supondremos que únicamente queremos crear, sumar y multiplicar números complejos así como tener acceso y modificar sus partes real e imaginaria.

Protocolo de clase contra protocolo de instancia

El protocolo de mensajes de una clase es descrito en dos partes: el **protocolo de clase** y el **protocolo de instancia.**

protocolo de clase: Es una descripción del protocolo que entiende una clase.

protocolo de instancia: Es una descripción del protocolo que entienden las instancias de una clase.

El **protocolo de clase** describe mensajes que son enviados directamente a la clase. Típicamente, el protocolo de clase contiene mensajes para crear e inicializar instancias nuevas de una clase. Las clases se pueden pensar como moldes para la creación de instancias. Por ejemplo, es responsabilidad de la clase **Complex** crear instancias nuevas de la clase. La expresión 1

Complex **newWhitReal: 1.0 andimaginary:** 3.5

envía el mensaje **newWhithReal: 1.0 andimaginary:** 3.5 a la clase **Complex.** El objetivo es regresar una nueva instancia de la clase **Complex;** así, el protocolo de la clase debe ser especificado como sigue:

instance creation

newWitbReal: realpart **andImaginary:** imaginarypart

Regresa una instancia de la clase **Complex** con parte real **realPart** y parte imaginaria **imaginarypart.**

Se usan **categorías de métodos** en Smalltalk para agrupar métodos que brindan funciones similares. Por ejemplo, los métodos de clase **newWithReal:andImaginary:** pueden ser agrupados en una categoría con el nombre instance creation. Los nombres de categorías no tienen significado semántica, se usan únicamente para propósitos de documentación. Internamente, los nombres de categorías son usados por el ambiente de programación para agrupar métodos relacionados entre si.

El **protocolo de instancia** es el protocolo de mensajes soportado por las instancias de una clase: es decir, esta formado por los mensajes que se pueden enviar a una instancia de la clase. Por ejemplo, el mensaje para sumar dos números complejos es parte del protocolo de la instancia. En la expresión

complex1 + complex2

el receptor del mensaje + **complex2** es la instancia **complex1.** Un subconjunto del protocolo de instancia para la clase **Complex** es el siguiente:

accessing

realpart
Regresa el componente real del receptor

imaginarypart
Regresa el componente imaginario del receptor

realPart:realValue
Asigna a componente real del receptor el valor **realValue.** Regresa el receptor modificado

arithmetic

+ acomplex
Regresa un número complejo igual a la suma del receptor y el argumento acomplex

*** acomplex**
Regresa un número complejo igual al producto de receptor y el argumento acomplex

Una vez que se han completado los protocolos de clase y de instancia de la clase **Complex** debemos ser capaces de escribir código que manipule números complejos, a pesar de que no hayamos considerado aún cómo representar estos números ni cómo implantar sus operaciones. Por ejemplo, la siguiente expresión mensaje podría crear dos números complejos y luego evaluar otros dos, uno igual a la suma de los originales y el otro igual a su producto. Es una buena práctica intentar "programar" con una clase nueva tan pronto como su protocolo ha sido especificado. Muchas veces, este proceso revela deficiencias en el protocolo. Por supuesto, es mejor descubrir esos problemas en la etapa de especificación que cuando la clase ha sido implantada.

```
|complex1 complex2 complexsum complexproduct |
complex1 := Complex newWithReal: 2.5 andimaginary: 3.1.
complex2 := Complex newWithReal: 1.0 andImaginary: 0.5.
complexsum := complex1 + complex2.
complexproduct := complex1 * complex2
```

Implantación de la descripción de una clase

La **vista interno** de una descripción de clase Smalltalk, desde el punto de vista de la implantación, se puede concretizar ejecutando los siguientes pasos:

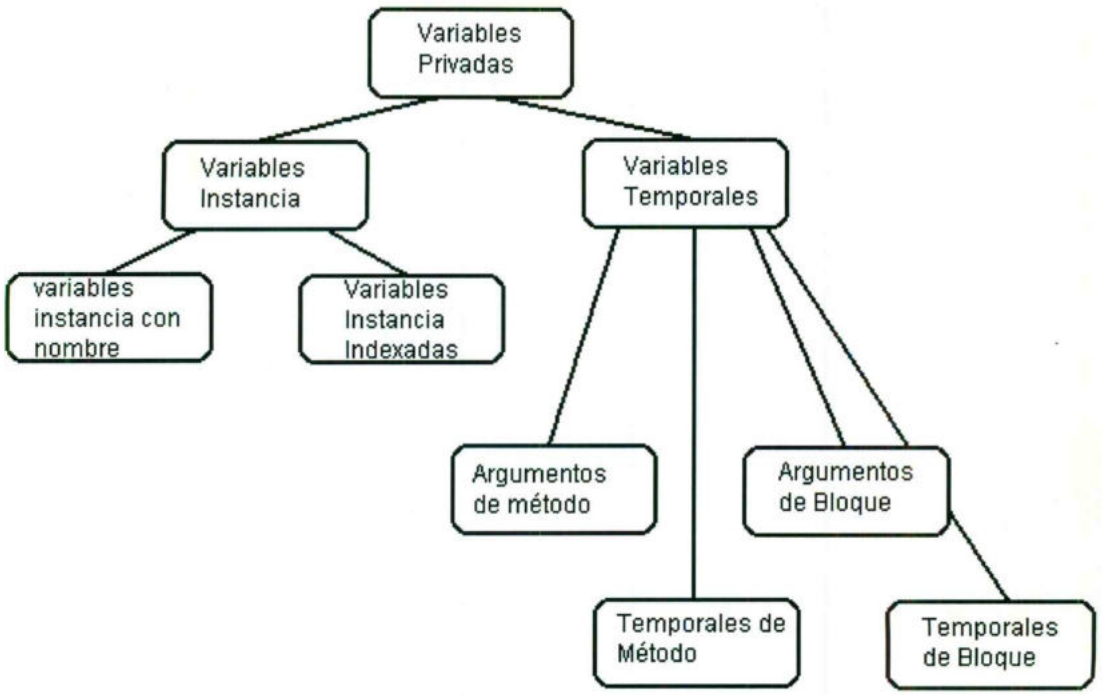
probado de manera inmediata. El sistema brinda una plantilla para guiar la adición de nuevas clases y métodos.

Variables y su alcance

En Smalltalk, como en los lenguajes tradicionales, es importante entender que las variables tienen un ámbito específico donde se pueden utilizar, las reglas que norman este aspecto son los mecanismos de alcance del lenguaje cuya función es controlar el acceso a las variables, indicar cuando se les debe asignar espacio en memoria y cuando se les debe despojar del mismo. Smalltalk brinda dos tipos básicos de variables: **variables privadas** y **variables compartidas.** Las variables **privadas** son accesibles sólo desde dentro del objeto, mientras que las variables **compartidas** pueden ser compartidas por varios objetos. Las variables privadas empiezan con una letra minúscula, mientras que las variables compartidas empiezan con una letra mayúscula.

Variables privadas

Las variables privadas incluyen tanto a las **variables instancia** corno a las **variables temporales.** Las **variables instancia** de un objeto son las partes o componentes del objeto -ellas son directamente accesibles únicamente por ese objeto. Las variables instancia presentan dos variantes: **variables instancia con nombre,** reverenciadas por nombre, y las **variables instancia indexadas,** referidas por un índice entero.



Las **variables instancia con nombre** del receptor de un mensaje pueden ser reverenciadas en cualquier método instancia de una clase del receptor o sus subclases. Como es imposible hacer referencia directa a las variables instancia con nombre de un objeto diferente de "self', el acceso a las variables instancia con nombre desde otros objetos sólo puede ser obtenido enviando mensajes apropiados al objeto. Es un error común intentar referirse a las variables instancia dentro de un método clase, sólo las instancias tienen acceso a las variables instancia.

Las **variables instancia indexadas** son variables instancia sin nombre en las instancias de una clase. Se puede tener acceso a ellas enviando un mensaje a la instancia con un índice que especifica cuál variable instancia (indexada) se desea. Por ejemplo, las clases del sistema **Array** y **String** tienen variables instancia indexadas. Cada instancia de una clase con variables instancias indexables puede tener un número arbitrario de elementos así como variables instancia pertenecientes a diferentes clases. El número exacto de elementos se especifica cuando se crea el objeto enviando el mensaje **new:size** a la clase. Las variables instancia individuales pueden ser reverenciadas usando mensajes **at: y at:put.** Considere los siguientes ejemplos:

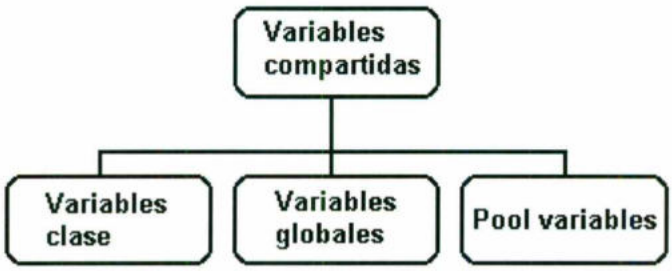
tabla := Array **new:** 20	Regresa una instancia de la clase Array de tamaño 20; es decir, un arreglo con **20** variables instancia indexadas
tabla **at:** 3 **put:** 'abcde'	La tercer variable instancia de **tabla** hace ahora referencia al objeto 'abcde'.
(tabla **at:** 3) **at:** 2	La expresión **tabla at:3** regresa el objeto **'abcde'.** Este objeto es una cadena que por si misma tiene variables instancia indexadas -ella recibe el mensaje **at:2** y regresa el carácter **$b.**

Las clases que tienen variables instancia indexadas pueden tener también variables instancia con nombre. Por ejemplo, la clase Set tiene una instancia size que hace referencia al número de objetos en un conjunto. Muchas clases del sistema y del usuatio sólo tienen instancia con nombre.

Las **variables temporales** incluyen los **argumentos de métodos, variables temporales de métodos argumentos de bloque y variables temporales de bloque**. Las variables temporales de métodos deben ser declaradas explícitamente debajo del patrón del mensaje, los argumentos de método y de bloque son declarados implícitamente (el contexto indica que ellas son variables) las variables temporales de un bloque deben ser declaradas explícitamente después del bloque de argumentos (estas no se permiten antes de la versión 2.4). El alcance de los **argumentos de métodos** y **las variables temporales de métodos** está limitado a los métodos en los que ellos son definidos. El alcance de los **argumentos bloque** depende de la versión del sistema en uso. Antes de la versión 2.4, los argumentos bloque no estaban restringidos y se podía tener acceso a ellos desde el exterior del bloque que contiene al método; por ejemplo, bloques distintos con el mismo nombre de argumento de bloque reverenciaban los mismos argumentos temporales del método. En la versión 2.4 y posteriores, los argumentos de bloques (y variables temporales de bloque) son locales al bloque en el que ellos están definidos. Los bloques anidados pueden hacer referencia a argumentos externos al bloque así como a variables temporales sólo si ellas no están localmente redefinidas. Las variables temporales de métodos y bloques pueden ser cambiadas mediante proposiciones de asignamiento pero los argumentos de métodos y bloques no pueden recibir asignamientos.

Variables compartidas

Las variables compartidas incluyen **variables globales, variables clase, y "pool variables"**. La diferencia entre ellas está dada por la forma como se pueden compartir.



Variables globales	Son compartidas por todos los objetos.
Variables clase	Son compartidas por todas las instancias de una clase. Ellas pueden ser reverenciadas dentro de cualquier clase o método de instancia de la clase (o subclases).
Pool variables	Son compartidas por un conjunto definido de clases en el sistema. Las **Pool variables** son almacenadas en diccionarios conocidos como **pool dictionaries.** Las variables en un "pool dictionary" se pueden poner a la disposición de una clase declarando el nombre del "pool dictionary" en la lista pool dictionary de la definición de la clase.

Las **variables clase** se usan principalmente para permitir que algunas constantes estén disponibles a todas las instancias de una clase (y sus subclases). Por ejemplo, en el sistema Smalltalk la variable **Pi** pertenece a la clase **Float.**

Las **variables globales** se almacenan en una instancia especial de la clase **Dictionary** llamada **Smalltalk.** Los diccionarios son colecciones de asociaciones entre nombres (o llaves) y valores. En Smalltalk se tiene un gran número de variables globales predefinidas. Por ejemplo, **Display** es una instancia especial de la clase gráfica **Forma** que se refiere a la pantalla de desplegado actual.

mayúsculas, mientras que las privadas no. Las variables Muiti-palabra privadas se escriben siempre con cada palabra excepto la primera con letra mayúscula, sin espacio alguno entre las

palabras. Recuerde que el nombre de clase para los números complejos fue escrita **Complex,** mientras que el selector para la creación de instancias de la clase Complex fue **newWithReal:andimaginary:.** Los selectores y palabras llave dentro de los selectores empiezan con letras minúsculas.

La pseudo-variable self

Supongamos que agregamos el método **esprímo** desarrollado previamente. El mecanismo actual para hacer esto será discutido en la sección que trata con el "Browser" de Smalltalk. Los enteros pueden ser entonces probados usando mensajes de la forma

7 **esprimo**
256 **esprimo**

Para permitir la referencia a un receptor particular en uso cuando se evalúa un método, Smalltalk brinda la pseudo-variable **self.** Como la pseudo-variable, **self** no puede ser cambiada con asignamientos dentro de un método queda asociada al receptor cuando inicia el método de evaluación. Si el método de la instancia **esprimo** hubiera sido involucrado con una expresión tal como **7 esprimo,** entonces **self** podría hacer referencia a la instancia entera 7,

Los métodos pueden ser recursivos

La pseudo-variable **self** nos brinda el medio para hacer referencia al receptor de un mensaje dentro de un método. Esto implica que podemos enviar mensajes adicionales al receptor (o más comúnmente un nuevo receptor basado en el original) desde dentro del método y consecuentemente invocar al mismo método recursivamente.

El siguiente ejemplo ilustra una definición recursiva del método **factorial** definido sobre los enteros.

```
factorial
    "Regresa el factorial del receptor"
    self = 0
        ifTrue: [^1]
        ifFalse:    [^self*(self - 1) factorial]
```

3.5 HERENCIA

La herencia en Smalltalk se basa en la noción de manejo de **subclases:** es decir, se define una clase como subclase de otra. Las clases en el sistema Smalltalk son acomodadas en una sola jerarquía de herencia con la clase mas general **Objeto** en el tope. Una clase puede tener cualquier número de subclases, pero cada clase tiene una sola **superclase.** La habilidad para heredar de una sola superclase es restrictiva, pero como la biblioteca de clase de Smalltalk es implantada de esta manera, ignoremos la posibilidad de múltiples superclases o **herencia múltiple** por el momento.

La pseudo-variable super

La pseudo-variable **super** permite el acceso a métodos en la cadena de superclases inclusive cuando el método ha sido redefinido en la clase. Corno **self super,** hace referencia al receptor de] método. Sin embargo, cuando se usa **super,** la búsqueda empieza en la superclase de la clase que contiene la definición del método. Hay que **ser** cuidadoso -no es siempre lo mismo iniciar la búsqueda en **la** superclase que en el receptor.

El mensaje modificado **turn:** para ConstrainedPen que hace uso de **super** se muestra a continuación,

turn:degrees
"La dirección del receptor es girada en el sentido de las manecillas del reloj una cantidad igual al argumento (interpretado en grados). Este argumento se trunca automáticamente a un múltiplo de 90 grados."

super **turn:**(degrees **roundto:** 90)

Consideremos ahora el método **go:distance.** A primera vista, podríamos pensar que este método puede ser heredado directamente de la clase **Pen** porque una dirección de la pluma se restringe a movimientos verticales y; horizontales. Sin **embargo,** un examen más cuidadoso del método revela que invoca al mensaje **goto:** El mensaje **goto:** será enviado al receptor del mensaje **go:.** es decir a **ConstrainedPen.** Pero este mensaje fue ocultado previamente para las plumas restringidas de manera que se produzca un error al enviarlo. Este ejemplo ilustra **el** tipo de problemas que se pueden presentar con la herencia de un método si el método de herencia "sino invoca métodos que han sido ocultados en la subclase. Para lograr el efecto deseado, debemos re-implantar **go:** en ConstrainedPen como una copia de **go:** en **Pen,** pero con una referencia **self goto:** que reemplace a **super goto:.**

go: distance
"Mueve al receptor siguiendo su dirección, un número de bits igual al argumento distancia. Si la pluma está abajo, se dibuja una línea usando la forma de punta del receptor. De otra manera nada se dibuja."

```
langle newdirection|
angle := direction degreesToRadians.
newdirection := angle cos @ angle sin.
super goto- newdirection * distance + location
```

Al final de este párrafo se resumen las jerarquías de herencia para las clases **Pen** y **ConstrainedPen.** Note que son posibles algunos, otros cambios. Por ejemplo, podemos querer un método **dirección** que permita establecer de manera absoluta la dirección. los métodos **east, west,** y **south** pueden ser implantadas usando esta nueva operación. Podríamos desear también operaciones adicionales como **turnleft, turnright,** y **turnback.** El método **turnleft,** por ejemplo, podría consistir simplemente del código "self **turn:-90",** Usando self en lugar de "super" podríamos asegurar que los cambios futuros (si los hubiera) para **turn:** en **ConstrainedPen** van a ser reflejados en los nuevos métodos. Por supuesto, muchas de esas operaciones pueden tener sentido para plumas estándard, Esto sugiere que algunos de los métodos en ConstrainedPen podrían ser llevados a la clase **Pen.** Cuando las clases afectadas son ambas definidas por el usuario, esta es una mejora natural que se debe hacer. Cuando las clases del sistema son afectadas, se necesita más deliberación.

Class ConstrainedPen

class name ConstrainedPen
superciass Pen

class methods

examples

example
"ilustra el uso de plumas restringidas"
| quill |
quill := ConstrainedPen **new.**
quill **home; place:** 300@300; **down.**
4 **times repeat:** [quill **go: 1 00; turnleft**]

"ConstrainedPen **example"**

instance methods

moving

goto: apoint

"Este mensaje no es apropiado para el objeto"
self **shouldnotimplement**

go: distance
"Mueve el receptor en su dirección actual un número de bits igual a su argumento"

| angle newdirection |
angle:=direction **degreesToRadians.** newdirection:=angle **cos @** angle **sin.**
super **goto:** newdirection * distance + location

south
"La dirección del receptor es puesta apuntando a la parte inferior de la pantalla."
direction := 90

east
"la dirección del receptor es puesta apuntando a la derecha de la pantalla." direction := 0

west
"La dirección del receptor es puesta apuntando a la izquierda de la pantalla"
direction := 180

turn: degrees
"La dirección del receptor es girada en el sentido delas manecillas del reloj una cantidad igual al argumento degrees. El argumento se restringe a múltiples de 90 grados por redondeo."
super turn:(degrees **roundto:** 90)

turnleft
"La dirección del receptor es girada a la izquierda 90 grados."
super **turn: -90**

turnright
"La dirección del receptor es girada a la derecha 90 grados."
super **turn: 90**

Clases abstractas

Una **clase abstracta** especifica un protocolo pero es incapaz de implantarlo completamente porque sus subclases pueden tener diferentes representaciones.

Como una clase abstracta no implanta completamente su protocolo, no se pueden crear instancias abstractas de clase. El papel de una clase abstracta es especificar el protocolo común a todas sus subclases, las subclases de hecho implantan aquellos elementos que no forman parte del comportamiento común.

Por ejemplo, la clase **Magnitude** es una clase abstracta usada para describir objetos que pueden ser comparados en una dimensión lineal. Las subclases de **Magnitude** son las clases **Character, Date, number,** y **Time.** El protocolo común especificado por la clase **Magnitude** refleja el hecho de que todas las instancias de una de estas subclases pueden ser comparadas entre sí usando operadores relacionases.

Como las representaciones de instancias de las subclases **Character, Date, Number** y **Time** son claramente diferentes, cada subclase brinda su propia implantación para opciones que son dependientes de la representación. Las operaciones que hacen referencia a su representación directamente son

operaciones primitivas, si la representación fuera cambiada, ellas podrían requerir modificaciones. La implantación de operaciones primitivas debe ser la responsabilidad de las subclases.

Para mensajes donde es responsabilidad de una subclase brindar la implantación, una clase abstracta implanta el método para generar un mensaje de error. El protocolo **subclassrespondibiiity** soportado por la clase **Object** puede ser usado para generar un mensaje que indica que una subclase podría haber eliminado su implantación de este método. Esto es útil cuando se agrega una nueva subclase y el programador olvida implantar todo el protocolo especificado por la superclase abstracta. Los métodos en las clases abstractas pueden ser reimplantados por la subclase que deba tener el control

self **subclassresponsability**

Ninguna operación primitiva se puede implantar en términos de otra práctica y/o operación no-prímitiva y por consiguiente puede ser implantada una vez en la clase abstracta. Por ejemplo, en el caso de magnitudes, sólo la operación primitiva "<" es implantada por las subclases. Operaciones tales como ><= son no primitivas porque ellas se pueden implantar en términos de <. Ellas necesitan ser implantadas sólo una vez en la clase abstracta, como se muestra a continuación.

comparando

```
< amagnitude
    "Responde si el receptor es menor que el argumento."
    ^self subclassResponsibility

<= amagnitude
    "Responde si el receptor es menor o igual que el argumento"
    ^(self > aMagnitude) not

> amagnitude
    "Responde si el receptor es mayor que el argumento"
    ^amagnitude < self

>= amagnitude
    "Responde si el receptor es mayor o igual que el argumento."
    ^(self < amagnitude) not
```

Las clases abstractas tienen un papel importante en Smalltalk y en programación orientada a objetos. Como hemos visto, ellas permiten que un protocolo común a una colección de clases sea identificado rápidamente. Eligiendo adecuadamente las superclases abstractas, por ejemplo, es fácil determinar qué operaciones son comunes a todos los tipos de números, todos los tipos de enteros, etc. Otro beneficio es que ellas pueden ser usadas para maximizar el nivel de reutilización de código mediante la herencia.

MODULO IV: EJEMPLOS EN SMALLATLK

A continuación se tratarán algunas líneas de código y ejemplos de programas escritos en smalltalk con la finalidad de que el lector comprenda como son interpretados por el lenguaje y el (los) resultado(s) que se obtengan.

Los **objetos simples**, son los bloques básicos de construcción en el lenguaje smalltalk, por ejemplo:

'Este es un string'

es un objeto muy similar a los objetos tipo string en cualquier otro lenguaje. Ahora, analicemos otros objetos que tienen algunas diferencias en otros lenguajes:

1234	**'un Integer'**
$A	**'El simple carácter A'**
#(123)	**'Un arreglo de tres objetos enteros'**

Observemos el último ejemplo, es un objeto que contiene otros objetos. Veamos más ejemplos:

#('arreglo' 'de' 'cuatro' 'elementos')
#('arreglo' 'de' 5 'strings' 'y' 2 'enteros')

como podemos ver en el último ejemplo, todos los objetos contenidos en un objeto no necesariamente deben ser del mismo tipo o tamaño. Este es un ejemplo del poder que tiene el lenguaje smalltalk. Consideremos un ejemplo más complejo:

#(1 ('dos' 'tres') 4)

es un objeto del tipo arreglo que contiene tres elementos. El segundo elemento es otro arreglo de dos strings y esto es soportado por el lenguaje.

Los **mensajes simples** en smalltalk, son similares a las llamadas de funciones en otros lenguajes, por ejemplo observemos la siguiente instrucción, compuesta de un mensaje simple:

20 factorial

En la instrucción se envía el mensaje factorial al objeto 20 y nos retornaría el entero largo 2432902008176640000.

Ejemplo:

'Dime el numero de caracteres' size

Un mensaje esta compuesto por un objeto receptor, un selector de mensaje y los argumentos. En el ejemplo el string es el objeto receptor, size es el mensaje y no existen argumentos. El resultado de la instrucción es el tamaño del string: 28; o bien consideremos

#(1 3 5 7) at: 2

el arreglo es el receptor, at: es el mensaje y, el 2 es el argumento; el resultado es el elemento del arreglo en la posición número dos: 3.

Los mensajes en smalltalk que no tienen argumentos son denominados **mensajes Unarios**:

```
#('arreglo' 'de' 'strings') size
'dime tu nombre' asUpperCase
'hola como estas' reversed
#( 4 'cinco' 6 7) reversed
$A asciiValue
65 asCharacter
```

Los **mensajes Keyword** son aquellos que tienen uno o mas argumentos:

```
'hola como estas' at: 6
'Hola' includes: $o
'hola' at: 1 put: $H
'El lenguaje Smalltalk' copyFrom: 4 to: 9
```

En los últimos dos ejemplos los mensajes at:put: y copyFrom:to: son divididos por los argumentos. Cada argumento es dividido por los dos puntos (:). Los arreglos funcionan de manera similar:

```
#(9 8 7 6 5) at: 3
#(1 (2 3) 4 5) includes: #(2 3)
#(1 0 4 5) at: 2 put: #(2 3)
#(9 8 7 6 5) copyFrom: 1 to: 2
```

En estos ejemplos observarás otro punto importante en la programación orientada a objetos: diferentes clases de objetos responden de diferente manera al mismo mensaje.

Los **mensajes aritméticos** en smalltalk son muy similares a los de otros lenguajes, observemos:

3 + 4

el objeto entero 3 es el receptor, + es el mensaje, el entero 4 es el argumento y el entero 7 es el resultado.

Los mensajes aritméticos que se utilizan en smalltalk son:

5 * 7	Multiplicación
5 // 2	División de enteros
4 \\ 3	Residuo de la división de enteros
2 / 6	División de números racionales

Las expresiones aritméticas en smalltalk difieren de muchos lenguajes en el orden de evaluación. En smalltalk las expresiones se evalúan estrictamente de izquierda a derecha y no por la jerarquía de los operadores. Por ejemplo, si evaluamos la expresión:

3 + 4 * 2

El resultado seria 14 y no 11. La operación que se realiza primero, es la suma de 3+4 y después la multiplicación del resultado de la suma por el numero 2 (7 * 2). Bien, si queremos que primero realice la multiplicación, debemos utilizar paréntesis para ordenar las operaciones.

3 + (4 * 2)

y entonces el resultado seria 11.

Otro tipo de mensajes que encontramos en el lenguaje smalltalk son los de **tipo binario**, los cuales al igual que los operadores aritméticos son estrictamente evaluados de izquierda a derecha, por ejemplo:

'hola', 'como estas'

el resultado seria la concatenación de los dos strings dado que tienen el argumento coma (,) y el segundo string se une al primero.

Para accesar a los valores almacenados en arreglos utilizaremos sentencias similares a las siguientes:

```
x:= a at: i.
a at: i+1 put: y.
a at: i+1 put:(a at: i)
```

La primera instrucción asigna a la variable x el valor del arreglo a en la posición i, en la segunda se asigna al arreglo a en la posición i+1 el valor de y, y en la tercera se asigna al arreglo a en la posición i+1, el valor del arreglo a en la posición i.

La instrucción IF.

```
a<b
   ifTrue: [a:=a+1].

stream atEnd
   ifTrue: [stream reset]
   ifFalse: [c: stream next]
```

El resultado de la comparación es un resultado booleano. En la primera condición se incrementa el valor de la variable a en 1. En la segunda si, se cumple la condición del final de archivo del apuntador (stream) se envía el mensaje reset al objeto stream y lo regresa a la primera posición, en caso contrario recorre el apuntador stream una posición hacia adelante.

Instrucciones de Iteración.

```
[i<10]
   whileTrue:[
         sum:=sum + (a at: i).
         i:=i+1].

1 to: 10 do: [:i |
   a at: i put: 0]
```

En el primer ejemplo las dos instrucciones serán ejecutadas mientras el valor de la variable i sea menor de 10. En el segundo, se asignará un cero al arreglo a de la posición 1 a la 10.

El siguiente programa nos muestra la frecuencia de los caracteres alfabéticos en una cadena:

```
| s c f k |

f:= Array  new: 26.

s:=Prompter prompt: 'Introduzca una línea de texto' default: ' '.
1 to: 26 do: [ :i |
      f at: i put: 0].

1 to: s size do: [ :i |

      c:=(s at:i) asLowerCase.

      c is Letter
         ifTrue: [
            k:=c asciiValue
               - $a asciiValue
               +1.
            f at: k put: (f at: k) + 1
            ]
      ].
^f
```

Primeramente el programa pide a el usuario que introduzca una línea de texto. Cada carácter es analizado y acumulado (+1) en la posición del alfabeto que le corresponde. Cabe mencionar que todos los caracteres son tratados como minúsculas. En el ejemplo anterior es implementado un algoritmo que se pudiera aplicar en cualquier lenguaje de programación, pero ahora veamos el mismo ejemplo utilizando el poderoso built-in proporcionado en smalltalk.

```
| s f |
s:=Prompter prompt: 'Introduzca una línea de texto' default: ' '.
f:= Bag new.
s do:[ :c | c isLetter ifTrue: [f add: c asLowerCase] ].
^f
```

Bien, ahora veamos un ejemplo en cual declaremos variables temporales

```
| temp indice factoriales |
factoriales:= #(3 4 5 6).
indice:= 1.
factoriales size timesRepeat:[
   temp:= factoriales at: indice.
   factoriales at: indice put: temp factorial.
   indice:= indice +1].
^factoriales
```

en el ejemplo declaramos tres variables temporales temp, indice y factoriales, las cuales puede almacenar cualquier tipo de objeto. Asignamos a la variable factoriales un arreglo de cuatro elementos (3,4,5,6) para después reemplazar cada uno de ellos con el resultado de su factorial (6,24,120,720).

Otro ejemplo en el cual logramos mezclar los mensajes en cascada y las iteraciones de una serie de instrucciones en un numero determinado.

```
"dibuja una flor con hojas en forma de polígonos"
| sides length |
Window turtleWindow: 'Turtle Graphics'.
sides:= 5.
length:= 240 // sides.
Turtle
   black;
   home;
   north.
sides timesRepeat: [
   Turtle go: length.
   sides - 1 timesRepeat: [
      Turtle
         turn: 360 // sides;
         go: length] ]
```

El programa anterior nos dibuja una flor de cinco hojas y cada una de ellas tiene forma de polígono.

Otro poderosa herramienta que tiene la técnica de programación es la recursividad. Esta es con frecuencia utilizada cuando un algoritmo o una estructura de datos son definidos en términos de si mismos. Bien, ahora observemos como es creado el método factorial (no confundirse con la sintaxis del mensaje factorial utilizado anteriormente) definido para la clase de los enteros.

```
factorial
    "Pregunta para recibir el número del factorial"
    self > 1
      ifTrue: [^(self - 1) factorial * self ].
    self < 0
      ifTrue: [^self error: 'Factorial negativo' ].
    ^1
```

para probar el método anterior escribiremos la siguiente instrucción

```
#(0 1 2 3 4 10 15 20) collect: [ :n | n factorial ]
```

Otro ejemplo clásico en el que se utiliza la recursividad es el cálculo de la serie fibonacci de un número n, el cual a continuación, escribiremos el código del método fibonacci para los números enteros.

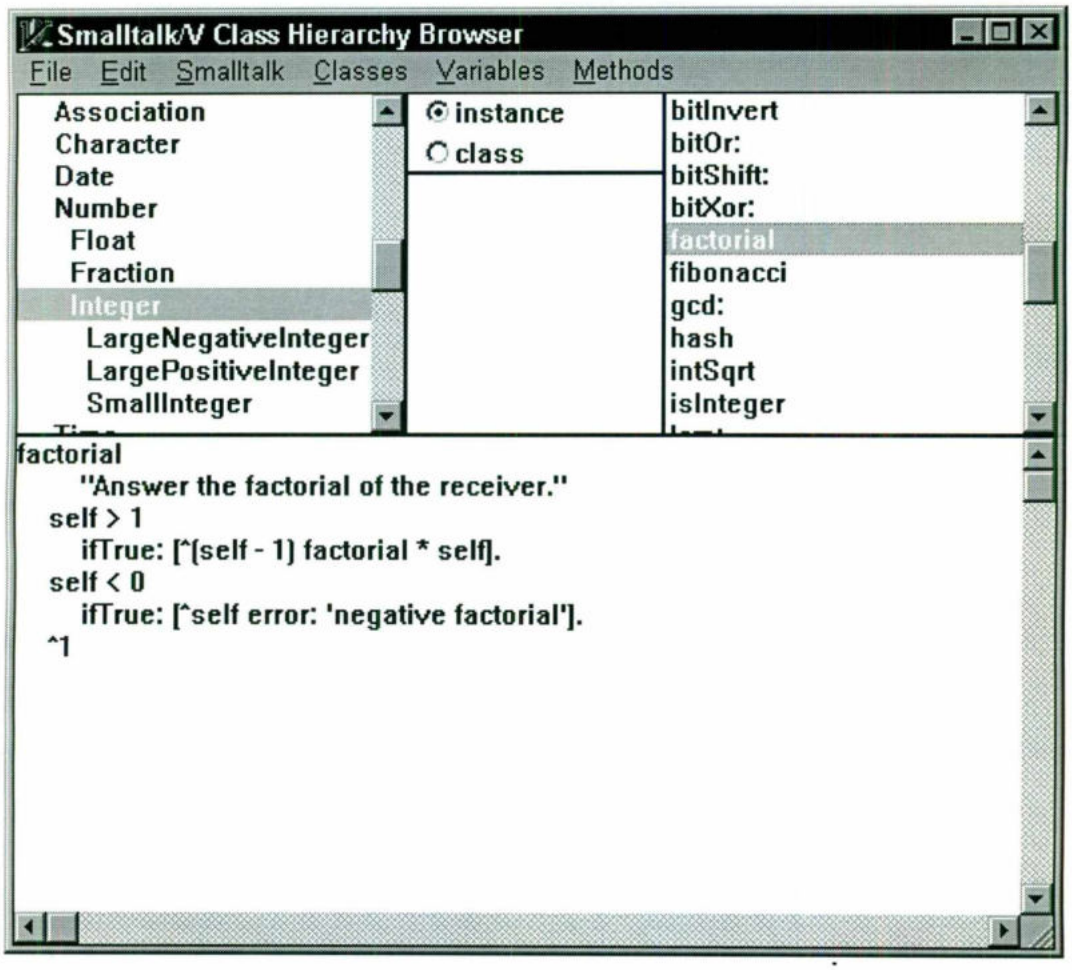
```
fibonacci
    "Cual es el número."
    ^self < 3
     ifTrue: [1]
     ifFalse: [(self - 1) fibonacci + (self - 2) fibonacci ]
```

de igual manera para probar el resultado del método fibonacci creado con el código anterior escribiremos la siguiente instrucción:

```
#(1 2 3 4 5 6 7 10 20) collect: [:m | m fibonacci ]
```

Para poder crear un nuevo método o una nueva clase en el lenguaje smalltalk, mencionaremos la herramienta denominada Class Hirerarchy Browser incluido en el lenguaje y, el cual trataremos de explicar su funcionamiento.

El Class Hirerarchy Browser del lenguaje smalltalk, nos permite crear, examinar, modificar clases y métodos para las mismas.



Hablar del Class Hirerarchy Browser seria desfazarnos un poco del tema, solo mencionaremos la manera de crear un nuevo método para una clase existente. Bien, al momento de seleccionar el objeto clase (Integer) listado en la ventana superior izquierda, activar la opción New Method de la opción Methods. Esto nos crea en la ventana inferior un esqueleto de programa para escribir nuestro método, en el cual escribiremos el nombre del mismo, los comentarios, la declaración de variables y por último el bloque de instrucciones que comprende el método. (Para un mejor entendimiento del funcionamiento del Class Hirerarchy Browser se recomienda ver los manuales del lenguajes de programación Smalltalk/V)

ANEXO I

Programación Estructurada versus el paradigma de la POO

El paradigma orientado a objetos sugiere una metodología que difiere del proceso tradicional de análisis y diseño estructurado de muchas maneras. El análisis y diseño estructurado se basa en procesos que manipulan los datos introducidos para generar un resultado. Los primeros pasos en la programación estructurada son:

- La creación de diagramas preliminares de contexto y de flujo de datos.
- La siguiente fase consiste en la creación de un modelo esencial del sistema especificando lo que se debe hacer.

En éste modelo se considera la combinación del modelo ambiental y el de comportamiento. Este último es construido principalmente de los modelos de los proceso esenciales y de la interacción que existe entre ellos. Por consiguiente, una parte esencial de la metodología estructurada es la modelación de los procesos o métodos que son utilizados para la transformación de los datos en resultados.

En las últimas dos décadas las metodologías de desarrollo han sido utilizadas para crear software en muchas áreas. La reutilización de código no ha sido de primera consideración en el diseño y desarrollo de los grandes sistemas.

Algunos investigadores han dado a conocer los modelos de desarrollo orientado a objetos, los cuales permiten una gran variedad de usos. La efectividad de un buen sistema en ésta metodología consiste en la capacidad de una interacción entre los objetos, los cuales deberán exponer las especificaciones de su implementación.

Como una diferencia básica entre la programación orientada a objetos y la programación estructurada, son los procedimientos básicos o los métodos algoritmicos.

La POO describe a un sistema en términos de los objetos involucrados mientras que los métodos de la programación tradicional lo describen en términos de su funcionalidad.

Una ventaja de la programación estructurada para la solución de un problema, son los grandes niveles de detalle de la descomposición funcional. En ésta

técnica solucionamos el problema algoritmicamente de una manera paso a paso, en el cual cada paso de un proceso describe una solución para el problema a un cierto nivel de abstracción. La fase de depuración de sistemas es más fácil de describir utilizando diagramas, en los cuales los modelos más grandes son organizados jerárquicamente y donde cada uno de ellos representa una función o un subproblema de la solución.

Cuando utilizamos la técnica de la POO para la solución de problemas, nuestro primer objetivo no su funcionalidad sino los objetos que se involucrarán en la solución computacional. La facilidad para identificar los objetos es su similitud con la vida real. Una vez que estos han sido plenamente identificados, nuestra siguiente tarea es conocer sus características y las relaciones que existen entre ellos. Los objetos son caracterizados por sus estado y las operaciones que pueden desarrollar en cada estado. Generalmente los objetos tiene componentes y el estado de un objeto es por lo tanto caracterizado por el estado de sus componentes. Cada clase de objeto soporta un conjunto de operaciones que pueden ser aplicadas para modificar o interrogar sus estado.

Como resumen, un sistema orientado a objetos puede ser descrito como **un conjunto de objetos comunicándose entre ellos para lograr un resultado.**

Hacia la metodología de desarrollo orientada a objetos

En pocos años se puede concluir que la metodología de desarrollo orientada a objetos surja como un producto comprobado. Algunos investigadores han propuesto varios métodos para el desarrollo de software orientado a objetos.

En conclusión se ha llegado a dos técnicas, las cuales analizaremos en seguida:

La técnica denominada "Manejo de Objetos" plantea las siguientes preguntas:

¿Qué estructura hace la representación del objeto?
¿Qué operaciones pueden ser desarrolladas por el objeto?

Otra técnica llamada "Manejo de responsabilidad" se basa en los siguientes cuestionamientos:

¿De que acciones el objeto puede ser responsable?
¿Que información comparte el objeto?

La conclusión a la cual se ha llegado mediante las dos técnicas mencionadas anteriormente es:

> Las ventajas para el manejo de datos para el diseño orientado a objetos se basa principalmente en la estructura de los datos en un sistema. Esto resulta de la incorporación de la información estructural en la definición de las clases, lo cual hace que se viole el encapsulamiento de los datos.
>
> El método de manejo de responsabilidad enfatiza en el encapsulamiento tanto de la estructura como del comportamiento del objeto. Enfocándonos en las responsabilidades contractuales de una clase, el diseñador deberá ser hábil para posponer las consideraciones de la implementación durante ésta fase.

Otra técnica propuesta para la metodología del desarrollo orientado a objetos involucra la construcción de tres tipos de modelos formales:

a) Un modelo de información
b) Un conjunto de modelos de estado que indican los ciclos de vida y,
c) Un conjunto de modelos de proceso

En el modelo de información se identifican las entidades conceptuales del mundo real que estamos modelando. Además ayuda a la descripción del mundo real en términos de objetos, atributos o características de los objetos, y las relaciones o asociaciones entre dos o más conjuntos de entidades abstractas como objetos.

Por lo que respecta a los ciclos de vida de los objetos y las relaciones, deberán ser representadas en términos de los estados que indican la condición de un objeto, eventos que indican los cambios a la siguiente fase en el ciclo de vida, y acciones (procesos) que toman efecto cuando un objeto de algún tipo es cambiado de estado y llega al nuevo estado. El modelo de estado representa el comportamiento de un objeto o relaciones.

Los modelos de proceso sugieren la separación de los diagramas de flujo de los datos para cada estado en cada modelo de estado.

Bibliografía:

INSIDE SMALLTALK
VOLUME 1
WILF R. LALONDE / JOHN R. PUGH
PRENTICE HALL

ANALISIS Y DISEÑO ORIENTADO A OBJETOS
JAMES MARTIN / JAMES J. ODELL
PRENTICE HALL

MANUAL DEL LENGUAJE DE PROGRAMACION
SMALLTALK/V WINDOWS

C++ AND THE OOP PARADIGM
BINDU R. RAO
CAP GEMINI AMERICA SERIES
MC GRAW HILL

SMALLTALK PROGRAMMING FOR WINDOWS
DAN SHAFER WITH SCOTT HERNDON AND LAURENCE ROZIER
PRIMA PUBLISHING

INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS
THYMOTHY BOUDT
ADDISON WESLEY IBEROAMERICANA

METODOS ORIENTADO A OBJETOS
IAN GRAHAM
ADDISON WESLEY IBEROAMERICANA