Universidad de los Andes
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas
Departamento de Computación

Conceptos Básicos de La Orientación por Objetos

Prof. Domingo Hernández Hernández

Objetivos

Este cuaderno tiene la finalidad de introducir al lector o al estudiante que poseen conocimientos en el área de la computación a una nueva forma de pensar en el arte de programar, como es el caso de la Programación Orientada por objeto.

A lo largo del texto el lector estudiará los conceptos fundamentales que se requiere para diseñar problemas de la vida real con este nuevo enfoque.

Al final del curso los estudiantes deberán ser capaces de diseñar e implantar un pequeño programa donde apliquen todos los conceptos aprendidos.

Contenido

1.- Introducción al diseño orientado por objeto.
- ¿Qué es la orientación por objeto?
- ¿Qué es el desarrollo orientado por objetos?

2.- Historia y evolución de la orientación por objetos.

3.- Tipos abstractos de datos (separar el que del como)
- Ventajas de la abstracción de datos.
- Característica que debe satisfacer un lenguaje que soporte tipos abstractos de datos.

4.- Conceptos de la orientación por objetos
- ¿Qué es un objeto?
- ¿Qué se puede considerar como objeto?
- Identidad de un objeto.
- Componentes para la construcción de software de un objeto.
- Tipos de objetos.
- Abstracción
  .- Tipos de abstracción.
  .- Mecanismos de abstracción
- Operaciones y métodos.
- Generalización (métaclase o superclase)
- Agregación vs generalización.
- Encapsulamiento.
- Polimorfismo
  .- clasificación de polimorfismos.
- Herencia.
  .- Tipos de herencia.
- Encadenamiento dinámico.

5.- Estilo de programación para la orientación por objetos
- Reusabilidad.

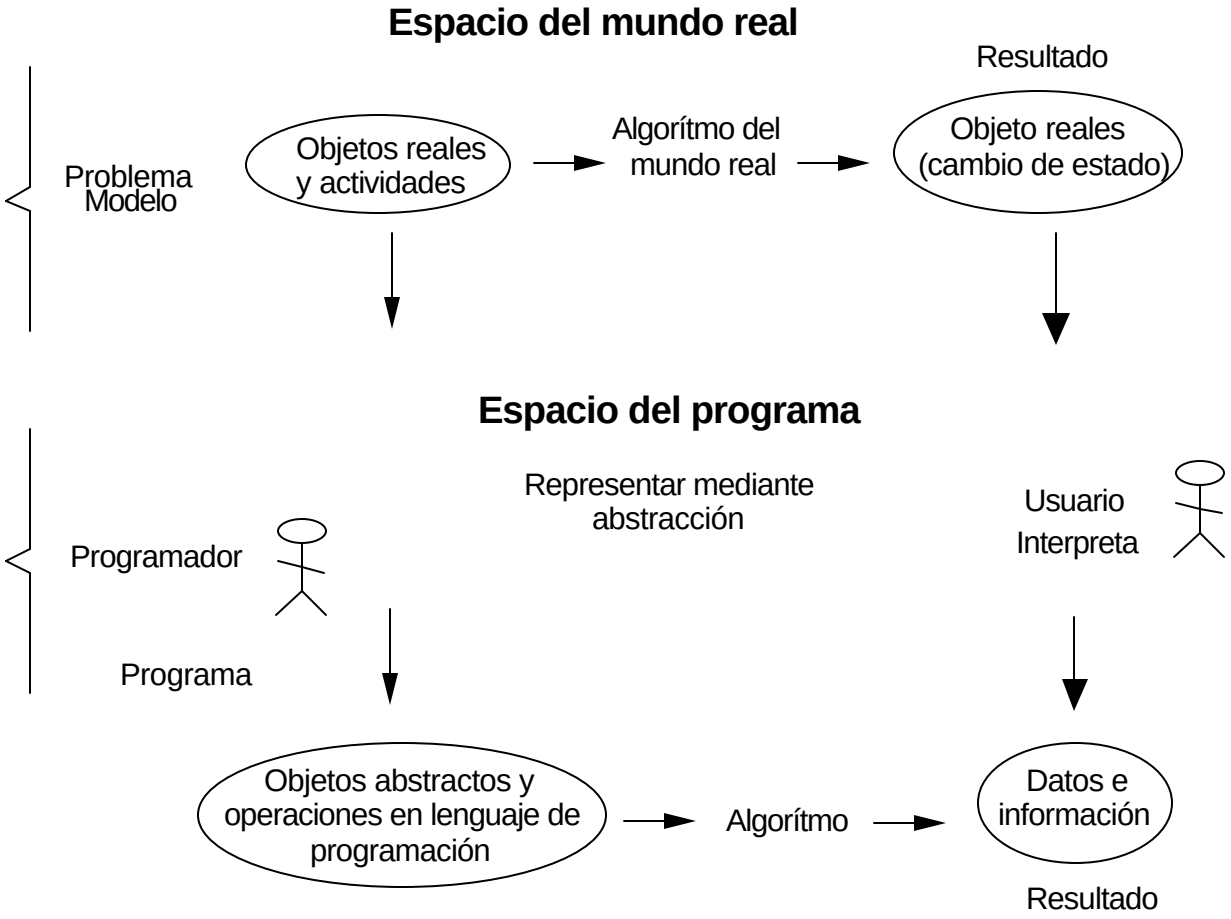
- Extensibilidad.
- Robustez.
- Validación de argumentos.
- Programación a gran escala.

6.- Metodologías para el diseño orientado por objetos.
Metodología de Abbolt, R y Booch, G.
Metodología de Loresen y otros.

1.- INTRODUCCION: DISEÑO ORIENTADO POR OBJETOS

El diseño orientado por objetos (DOO), como otras metodología de diseños orientados a la información crean una representación del dominio del problema en el mundo real y lo transforma en un dominio de soluciones que es el software. A diferencia de otros métodos, el DOO da como resultado un diseño el cual interconexiona los objetos de datos (elementos de datos) y las operaciones de procesamiento, de forma tal que encapsula la información y el procesamiento. Este encapsulamiento es el paradigma fundamental de la orientación por objetos.

Modelo de una tarea típica de programación



En la figura anterior se puede observar que los problemas de la vida real se pueden representar por medio de la abstracción en un programa de computación. En la vida real existen actividades y objetos que son manejados por procesos de la vida cotidiana del hombre, estos procesos producen nuevos objetos o cambios de estado en los objetos ya existente, que son interpretado por el usuario final. Cada uno de estos componentes de un problema de la vida real tiene su representación el espacio del programa, los objetos reales y actividades son representado en el espacio del programa por medio de objetos abstractos y operaciones en un lenguaje de programación, los procesos cotidianos son representados por los algorítmos que al ser ejecutados generan datos e información que serán interpretados por el usuario final.

La naturaleza única del diseño orientado por objetos se debe a su habilidad para construir basándose en tres conceptos importantes del diseño del software: **Abstracción**, **ocultamiento de la información y modularidad**. Todos los métodos de diseño buscan la creación de software que exhiban estas características fundamentales, pero solo el DOO
da un mecanismo que facilita al diseñador adquirir los tres conceptos anteriores sin complejidad o compromiso.

Wiener y Sincovec resumen el diseño orientado por objeto de la siguiente manera:

"Ya no es necesario para el diseñador de sistemas convertir el dominio del problema en estructura de datos y control predefinidas, presentes en un lenguaje de implantación. En vez de ello, el diseñador puede crear sus propios tipos abstracto de dato que pueden ser inventado por el diseñador. Además el diseño del software separa los detalles de representación de los objetos de los datos usados en el sistema.

¿ Qué es la orientación por objetos ?

Superficialmente el término "orientación por objeto" significa que organizaremos el software como una colección de objetos discretos que incorporan tanto estructuras de datos como procedimientos. Esto constrasta con la programación convencional, en la cual las estructura de datos y el comportamiento están solo aproximadamente conectados. Existen algunas disputas acerca de que características exactamente se requieren para una aproximación orientada por objetos; ellas generalmente incluyen cinco aspectos fundamentales: tipos abstractos de datos (encapsulamiento), polimorfismo, encadenamiento dinámico, herencia e identidad del objeto. Estos serán explicados en las secciones siguientes.

¿ Qué es el desarrollo orientado por objetos ?

El desarrollo orientado por objetos es una nueva forma de pensar acerca del software basado sobre abstracciones que existen en el mundo real. En este contexto, el desarrollo es referido a la primera parte del ciclo de vida del software: análisis, diseño e implantación.

La esencia del desarrollo orientado por objetos es la identificación y organización de conceptos del dominio de aplicación y del problema complejo de la herencia más que la traducción del diseño a un lenguaje particular.

El desarrollo orientado por objetos es un proceso conceptual independiente de un lenguaje de programación hasta su etapa final.

El desarrollo orientado por objeto es una nueva forma de pensar y no una técnica de programación.

Los grandes beneficios del diseño orientado por objetos es la ayuda a la especificación del sistemas de los desarrolladores y a la comunicación clara de los conceptos abstractos del sistema con los usuarios.

2.- HISTORIA Y EVOLUCION DE LA ORIENTACION POR OBJETOS

Desde la introducción del computador digital, los programadores han estado a la búsqueda de lenguajes de alto nivel. La raíz de la programación orientada por objetos puede ser vistas como un intento gradual de los diseñadores de lenguajes de programación de ayudar a construir software de manera más rápida por programadores individuales o equipos de programadores y ayudar al mantenimiento del software después que este se ha escrito.

A finales de los 50 uno de los problemas que existía en grandes desarrollos de programas en FORTRAN era que los nombres de las variables presentaban conflictos en diferentes partes del programa . A consecuencia de esto los diseñadores del lenguaje Algol deciden proveer "barreras" para separar nombres de variables dentro de segmentos de programas. Esto da nacimiento al Begin ... End en el bloque del Algol 60. Desde entonces los nombres de variables aparecen dentro de un bloque de área local; el uso de ellas no dará conflicto con el uso del mismo nombre de la variable en otro sitio del mismo programa. Por lo tanto este fué el primer intento de proveer protección o encapsulamiento dentro de un lenguaje de programación. Las estructuras de bloque son ahora muy usadas en una variedad de lenguajes tales como: C, Pascal y ADA.

A final de los años 60, los diseñadores del lenguaje SIMULA 67 (Dahl y Nygaard) toman el concepto de "bloque" del Algol 60 para dar el primer paso hacia el concepto futuro de "objeto". Aunque las raíces del SIMULA fue el Algol, este fue principalmente ideado como un lenguaje de simulación. Este fué el comienzo del uso de la encapsulación de datos dentro de un lenguaje de programación.

A principio de los 70 el concepto de abstracción de datos fue propuesto por un número de diseñadores de lenguaje con una orientación para manejar grandes programas. Los lenguajes tales como

Alphard (Wulf y Shaw.1976) comienzan el uso de objetos , clases y propiedades de herencia. Al mismo tiempo, una buena parte de los fundamentos y teorías matemáticas para tipos abstracto de datos comienzan a desarrollarse. Esto ayudó a establecer el concepto de tipo abstracto de datos, suministrando así una rigurosidad matemática para ser utilizada en la orientación por objeto.

Símilarmente al concepto de abstracción de datos y herencia, en los lenguajes tales como TAXIS (Mylopoulos y Wong 1980) quién también provee un puente para el surgimiento del concepto de herencia, el cual ha sido muy popular en inteligencia artificial basadas en redes semánticas.

A principio de los 80 estos conceptos comienzan a emerger juntos hacia un concepto uniforme para encapsulamiento de datos, propiedades de herencia y elementos de datos activos.

En 1986, la OOPSLA (Object-oriented Programing Systems and Lenguage) fue la primera gran conferencia dedicada completamente a la orientación por objetos. Adicionalmente pasado algunos años; comienzan a estar disponibles lenguajes orientados por objetos tales como C++, Objetive C, Smalltalk-80 y lenguajes orientado a bases de datos como el Gemstone.

3.- Abstracción de datos.

Dijkstra y algunos otros autores han expresado que la complejidad que la mente humana puede manejar en cualquier instante de tiempo es considerablemente mucho menor que la incorporada en el cuerpos de muchos de los programas que ella (la mente) puede construir. Así el problema clave en el diseño e implantación de grandes sistemas programados es reducir la cantidad de complejidad o detalles que pueden considerarse cuando se quiere. Un forma para lograrlo es vía el proceso de abstracción.

Una de las más significativas ayudas para la abstracción usada en la programación es la utilización de subrutinas independientes, esto quiere decir que la subrutina no le importa en que punto del programa se invoque, ella realizará su función siempre y cuando se le pasen los parámetros que ella necesite. En el punto donde el programador decide invocar una subrutina él puede tratarla como una "caja negra". Ella ejecuta una función abstracta específica por medio de un algoritmo sin prescripción. Así el nivel donde se invoca, separa los detalles relevantes del **Qué** de los detalles irrelevantes del **Cómo**. Símilarmente en el nivel de implantación es usualmente innecesario complicar el **Como** por considerar el **Porqué**; la razón exacta por la cuál es invocada una subrutina es a menudo desconocida por su implantador.

Así como en los procedimientos, el concepto básico de tipo abstracto de datos se utilizada para separar la abstracción vista por los usuarios de los detalles de su implantación, de tal manera que la implantación de la misma abstracción pueda ser sustituida libremente por otra, esto quiere decir que el código que utiliza estos tipos abstractos de datos no debe ser afectado por cambios en la implantación de los mismos. Esto se alcanza por medio de la encapsulación de los datos del objeto con el conjunto de operaciones que los manipula y las restricciones que el usuario impone a esas operaciones.

La abstracción de datos ha surgido rápidamente con el paso de pocos años. El compartimiento masivo de los datos contemplado por la tecnología de los manejadores de bases de datos y las

limitaciones que presentan los lenguajes de programación han sido un estímulo fuerte para el cambio de la tecnología actual. En un intento por liberar a los usuarios y programadores los detalles de trabajos los sistemas de computación se dirigen en dos direcciones:

1.- En el área de los lenguajes, las investigaciones desarrolladas permiten operaciones más naturales sobre los tipos de datos que se parecen más a objetos del mundo real.

Muchos lenguajes de programación modernos ofrecen más facilidades algorítmicas para la definición de tipos. Esto significa que se proveen módulos o mecanismos de ocultamiento de información para poder definir que nuevos tipos de datos.

2.- En el área de bases de datos se han realizado investigaciones para promover métodos más naturales para la manipulación de grandes bases de datos.

Para aplicaciones de bases de datos los tipos abstractos de datos juegan tres papeles interrelacionados. Primero, ellos pueden ser empleados para la extensión de tipos del sistema o del lenguaje, para proveer más tipos con atributos complejos. Por ejemplo tipos tales como: sexo, hora, color, nombre, etc; los cuales son comúnmente utilizado en el dominio de las bases de datos pero no siempre provistos como tipos propios de esos sistemas, pero ellos pueden ser implantados convenientemente vía tipos abstracto de dato. Segundo, los tipos abstracto de dato se pueden construir para representar objetos de alto nivel. En este caso, se utilizan para representar entidades dinámicas en oposición al los datos inertes; en esta consideración los tipos abstracto de dato proveen una forma conveniente para implantar restricciones de integridad y procedimientos que se dispararán para gobernar un objeto a tiempo de ejecución. Y finalmente los tipos abstracto de dato pueden representar relaciones entre entidades vía abstracciones de agregación y generalización. Agregación es una abstracción en la cual una relación entre objetos se representa por un objeto agregado de alto nivel o tipo. La generalización es una abstracción en la cual un conjunto de objetos con propiedades similares es representado por un objeto genérico.

Las investigaciones en el área de lenguaje y de bases de datos poseen temas comunes y apuntan hacia el desarrollo de niveles más significativos para entender el mundo del discurso, suprimiendo el enlace entre la representación de los datos en la máquina y el ocultamiento la de las propiedades físicas de la implantación de las propiedades lógicas que el usuario observa. El objetivo de ambas áreas es modelar situaciones complejas en una forma útil e inteligente.

¿ Qué es la Abstracción de Datos ?

El diccionario describe la palabra abstracción como una idea general que se concentra sobre las cualidades esenciales de algún objeto del mundo real más que sobre la realización concreta del mismo.

La abstracción consiste en enfocar los aspectos esenciales inherentes a una entidad e ignorar las propiedades accidentales. En el desarrollo de sistemas esto significa que se deben estudiar los objetos antes de decidir como implementarlos. El uso de la abstracción preserva la libertad de movimiento para tomar decisiones pues evita la acometida prematura de los detalles.

El uso de la abstracción durante el análisis solo conduce al dominio de los conceptos de la aplicación; no pueden hacerse diseño y no tomarse decisiones de implantación antes de que el problema sea entendido.

El problema esencial de la abstracción es el modelaje de situaciones del mundo real sin restringirse a los objetos y operaciones que son ofrecido en los lenguajes de programación convencionales.

Abstracción vista como un proceso:

Consiste realmente en separar las propiedades esenciales de un objeto sistema, fenómeno o problema y omitir las propiedades no esenciales.

Abstracción vista como un producto:

Es una descripción simplificada o especificación de un sistema en el que se enfátiza algunos de los detalles o propiedades y se suprimen otros. El conjunto de propiedades esenciales es tratado como un todo.

Tipos de abstracción:

- Abstracción funcional
- Abstracción de datos

Abstracción funcional:Un conjunto finito de funciones, instrucciones y subprogramas que son tratados como un todo.

Abstracción de datos: Un grupo de funciones u operaciones que actúan sobre una clase particular de objetos. Ejemplo: Estructura de datos.

Mecanismos de abstracción

1.- Clasificación
2.- Generalización
3.- Agregación

4.- TIPOS ABSTRACTOS DE DATOS (separar el qué del cómo)

" Si se posee un tipo de dato llamado T que se define como una clase de valores y una colección de operaciones sobre esos valores y si las propiedades de esas operaciones son especificadas solamente con axiomas, entonces T es un tipo abstracto de dato o una abstracción de datos " (Guttag,

John y Comm.A.C.M. volumen 20 # 6 1977). Una implantación correcta del tipo abstracto de dato cumple con todos los axiomas especificados para él.

La especificación por axiomas algebraicos para el tipo de dato T se compone de:

1.- Una especificación sintáctica: En esta sección, se listan las funciones u operaciones que actúan sobre las instancias de T, definiéndose los nombres, dominios y rango de dichas funciones. Estas funciones pueden clasificarse de la siguiente manera:

Operación Constructor: Esta operación produce una nueva instancia para el tipo de dato abstracto. En sistemas orientados por objetos tal operación es llamada **crear** (create) o **nuevo** (new) y provee al usuario de una capacidad para generar dinámicamente nuevos objetos y asignarle valores actuales a sus propiedades.

Operación Destructor: Esta operación permite al usuario descartar instancias de objetos que no son deseadas. En algunos sistemas la desincorporación de objetos no deseados puede ser automátiaca.

Operación de Accesos Esta operación produce elementos que solo son propiedades del tipo de dato existente. Tales operaciones pueden proveer, por ejemplo, capacidad para acceso asociativo, es decir, la habilidad de localizar objetos sobre los valores básicos de sus propiedades.

Operación de Transformación Esta operación produce nuevos elementos de tipos abstracto de dato partiendo de los elementos existentes y posiblemente de otros argumentos.

2.- Una especificación semántica Está compuesta del conjunto de axiomas en forma de ecuaciones, que dicen como operan cada una de las operaciones especificados sobre las otras operaciones.

Ejemplos de estructuras de datos que se pueden especificar e implantar como un tipo de dato abstracto son:

- Cadenas de caracteres
- Pilas
- Colas
- Arboles
- Grafos
- Archivos
- Conjunto
- etc.

Un ejemplo de especificación formal de un tipo de dato abstracto en función de los axiomas algebraicos visto anteriomente. La especificación del tipo abstracto de dato llamado pila.

Tipo de dato: Pila (elemento: tipo)
Pila: Tipo de dato abstracto
Elemento: Elemento que se almacenará en la Pila

Tipo: Tipo del elemento que se almacenará (entero, caracter, cadena,...)

Especificación Sintáctica

Nombre ↓ de la operación	(Dominio) ↓ del argumento	Rango ↓ del resultado	Tipo ↓ de operación
CREA_PILA	( )	PILA	Constructor
INSERTAR_EN_PILA	(PILA,ELEMENTO)	PILA	Transformación
ELI_ELEM_PILA	(PILA)	PILA	Transformación
TOPE_PILA	(PILA)	Elemento, o U = Indefinido.	Acceso
PILA_VACIA	(PILA)	VALOR BOOLEANO	Acceso
ELIMINA_PILA	(PILA)	Pila Eliminada, o ERROR	Destructor

Especificación Semántica

Declaración P: Pila, ELEM : elemento;

INSERTAR_EN_PILA (CREA_PILA ( ), ELEM) = P

ELI_ELEM_PILA (CREA _PILA ( )) = P

ELI_ELEM_PILA (INSERTAR_EN_PILA (P, ELEM)) = P

TOPE_PILA (CREA _PILA ( )) = U indefinido

TOPE_PILA (INSERTAR_EN_PILA (P , ELEM)) = ELEM

PILA_VACIA (CREA_PILA ( )) = VERDADERO

PILA_VACIA (INSERTAR_EN_PILA (P, ELEM)) = FALSO

Con la especificación sintáctica se puede ver claramente cuales son las operaciones válidas sobre la estructura y cuales son los datos que cada una regresa, luego que se ha efectuado una operación, para mayor claridad véase la operación ELIM_ELEM_PILA que se necesita para operar la pila y devuelve como resultado la misma pila, pero disminuida en un elemento para el caso de que existan elementos en la pila. Si en la pila no existen elemento la operación ELIM_ELEM_PILA no realiza ningún proceso sobre la pila y la devuelve tal como entro.

Con la especificación semántica se observa el efecto que tienen cada una de las operaciones especificadas sobre las otras.

Por ejemplo: ¿ Qué sucede si se desea saber si la pila esta vacía, habiéndose creada eficientemente?

Esto es PILA_VACIA (CREAR_PILA ( )) el resultado es verdadero porque la pila ha sido recientemente creada y por ello esta vacía. En el caso de TOPE_PILA(CREAR_PILA ( )) el resultado es un valor especial "U" que significa no definida para expresar que no puede devolverse elemento alguno pues la pila esta vacía. Este valor especial debe estar definido a la implantación el tipo abstracto de dato.

otro ejemplo: Especificación del tipo abstracto de dato llamado Lista.

Tipo de dato: Lista (elemento: tipo)
Lista: Tipo de dato abstracto
Elemento: Elemento que se almacenará en la lista
Tipo: Tipo del elemento que se almacenará (entero, caracter, cadena,...)

Especificación Sintáctica

Nombre ↓ de la operación	**(Dominio)** ↓ del argumento	**Rango** ↓ Resultado	**Tipo** ↓ de operación
CREA_LISTA	( )	LISTA	Constructor
INSERTAR_EN_LISTA	(LISTA,ELEMENTO)	LISTA	Transformador
ELI_ELEM_LISTA	(LISTA)	LISTA	Transformador

PRI_LISTA	(LISTA)	Elemento, o U = indefinido	Acceso
CONSULTA_LISTA	(LISTA)	Elemento, o U = indefinido	Acceso
ANULA_LISTA	(LISTA)	VALOR BOOLEANO	Destructor
SIG_LISTA	(LISTA)	Elemento o U = indefinido	Acceso

Especificación Semántica

Declaración L: Lista , ELEM : elemento;

INSERTAR_EN_LISTA (CREA_LISTA ( ), ELEM) = L

ELI_ELEM_LISTA (CREA _LISTA ( )) = U

ELI_ELEM_LISTA (INSERTAR_EN_LISTA (L, ELEM)) = L

PRI_LISTA (CREA _LISTA ( )) = U indefinido

CON_LISTA (INSERTAR_EN_LISTA (L , ELEM)) = ELEM

Tomemos otro ejemplo para la especificación del tipo abstracto de dato llamado conjunto.

Tipo de dato: Conjunto (elemento: tipo)
Conjunto: Tipo de dato abstracto
Elemento: Elemento que se almacenará en el conjunto
Tipo: Tipo del elemento que se almacenará (entero, caracter, cadena,...)

Especificación Sintáctica

Nombre ↓	**(Dominio)** ↓	**Rango** ↓	**Tipo** ↓
de la operación	del argumento	Resultado	de operación

CREA_CONJUNTO	( )	CONJUNTO	Constructor
INSERTAR_EN_CONJUNTO	(CONJUNTO,ELEMENTO)	CONJUNTO	Transformador
CONJUNTO_VACIO	(CONJUNTO)	VALOR BOOLEANO	Acceso
MIEMBRO	(CONJUNTO,Elemento)	VALOR BOOLEANO	Acceso
UNION	(CONJUNTO,CONJUNTO)	CONJUNTO	Transformador
INTERSECCION	(CONJUNTO,CONJUNTO)	CONJUNTO	Transformador
SUBCONJUNTO	(CONJUNTO,CONJUNTO)	VALOR BOOLEANO	Acceso
CARDINALIDAD	(CONJUNTO)	INTEGER	Acceso
ELI_CONJUNTO	(CONJUNTO)	VALOR BOOLEANO	DESTRUCTOR

Especificación Semántica

Declaración C: Conjunto, ELEM : elemento;

INSERTAR_EN_CONJUNTO (CREA_CONJUNTO ( ), ELEM) = C
CONJUNTO_VACIO (CREA_CONJUNTO ( )) = VERDADERO
CONJUNTO_VACIO(INSERTAR_EN_CONJUNTO(C,ELEM)=FALSO

la abstracción de datos por especificación permite:

-Ocultamiento de información.
-Enfoque más formal para la especificación de modelos.
-Reduce complejidad.
-Facilita verificación y razonamiento.
-Agregar datos diferentes a los del lenguaje utilizado.

Estos conceptos seran tratados en la sección 5 de este capítulo

Un tipo abstracto de dato define una clase de objeto, la cual tiene una misma estructura de dato para todas las operaciones aplicables sobre ellas y además define propiedades formales para tales operaciones.

Ventajas de Abstracción de datos

1.- La implementación es irrelevante para el diseño preliminar.

2.- La especificación es un acuerdo entre desarrolladores y usuarios. El lenguaje de comunicación es más sencillo.

3.- El usuario no tiene nada que ver con la implementación de módulos.

4.- Independencia de cada abstracción:

-La implementación se puede cambiar sin efecto alguno sobre los programas que la usan.
Los programas solo necesitan conocer su comportamiento.

5.- Sistemas fáciles de construir, modificar y mantener, pues cada abstracción puede ser trabajada en forma independiente.

6.- Reduce la complejidad global del sistema.

7.- Puede adelantarse parte de la implementación, al construirse los tipos de datos antes de que las estructuras de datos básicas del sistema sean definidas.

8.- La barrera entre especificación y diseño preliminar se elimina casi completamente.

Ejercicio:

Realice la especificación formal de tipo abstracto de datos para las siguientes estructuras.

a.- Cola
b.- Cadena de Caracteres
c.- Arbol Binario
d.- Archivo de Acceso Directo

Uno de los rasgos más importantes de la orientación por objeto es la de soportar tipos abstractos de datos, lo cual define conjuntos de objetos similares con colecciones de operadores asociados, independientemente de la implantación.

Características que deben satisfacer un lenguaje que soporte tipos abstractos de datos :

1.- Clases de objetos. Todo datum (objeto) debe estar dentro de un tipo de abstracto de datos.

2.- Ocultamiento de información. El acceso y la modificación a los objetos solo se realiza a través de la interfaz externa y por las operaciones definidas por el mismo tipo abstracto de dato. Los detalles de implantación interna como estructura de datos y elementos de almacenamiento usados para implantar el objeto de un tipo abstracto de datos y sus operaciones no son visibles para el usuario final que accede y manipulan los objetos.

3.- Completitud. Las operaciones están correctamente asociadas con un tipo de dato abstracto y la definición completa del comportamiento del tipo abstracto de dato pueden ser proyectado por el programador.

En efecto, 1 y 2 podrán ser forzadas si el lenguaje soporta tipos abstractos de datos. El principio de completitud es imposible de forzar y se deja al usuario.
El contraste entre los sistemas convencionales y los sistemas orientados por objetos usando tipo abstracto de dato se ilustra en las figura 2 y 3.

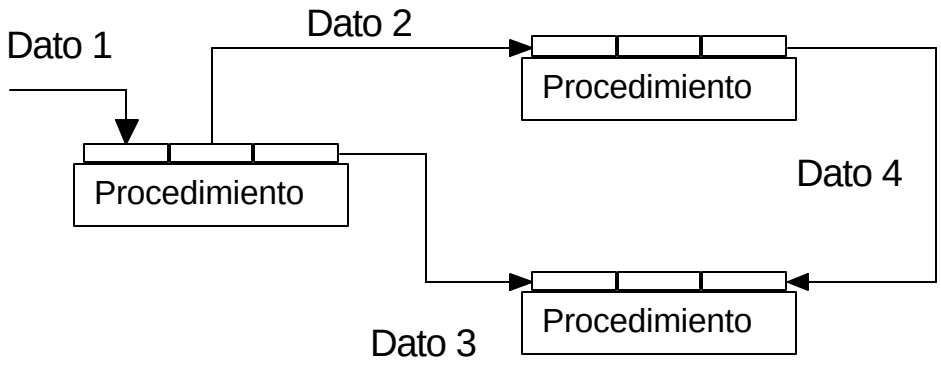


Fig. 2 Sistema convencional

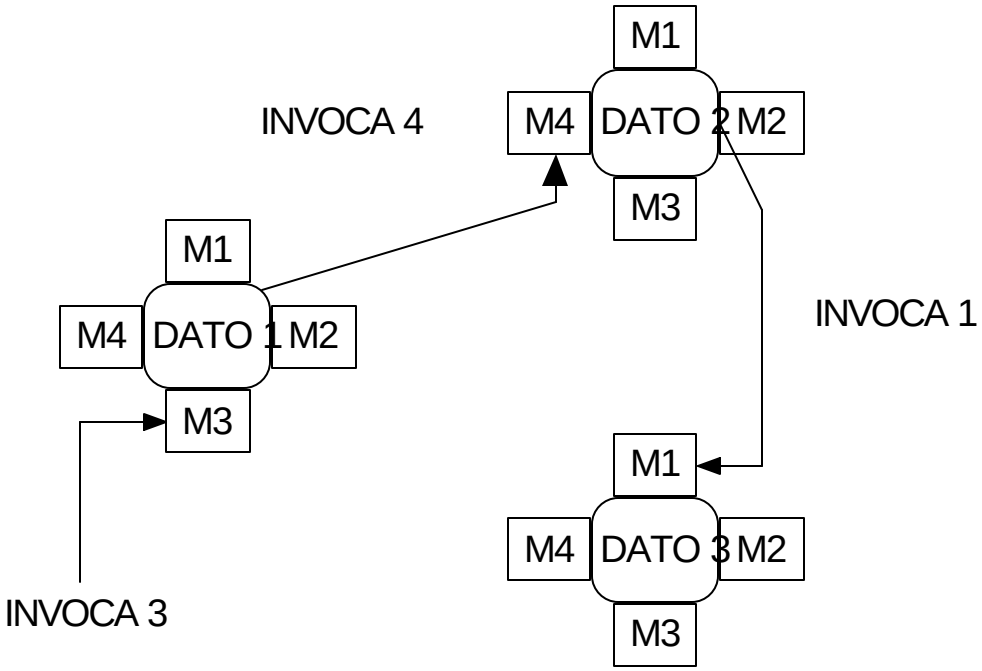


Fig.3 Sistema Orientado por objeto

En un lenguaje convencional los programas son una colección de procedimientos o funciones a los que se les pasan parámetros. Cada procedimiento manipula esos parámetros, algunas veces los actualizan a ellos y posiblemente retornan un valor. Por lo tanto la ejecución de la unidad (es decir los procedimientos) es control.

En un sistema orientado por objeto el universo es visto como una colección de objetos independientes, que se comunican cada uno con cada otros a través de procedimientos llamados **mensajes** en el ambiente de orientación por objeto. En efecto, los objetos son entidades activas y los mensajes son entidades pasivas que juntos con sus argumentos son pasados de un objeto a otro. Para ejecutar un objeto se requieren que los mensajes estén presentes y actúen sobre él. Por lo tanto los objetos son centrales.

El rasgo de ocultamiento de la información de los tipo abstracto de dato significa que los objetos tienen una interfaz que pueden ser implantadas de forma pública o privada. La idea básica de esto es simple, considere un tipo base tal como el entero en un lenguaje de programación convencional como el C o el Pascal. Estos lenguajes proveen un número finito de operaciones tales como: +, * y - que se pueden realizar sobre enteros, ademas de algunas otras operaciones adicionales tales como el resto y el cociente. Estas operaciones tienen bien definidas el comportamiento del objeto entero. Los usuarios simplemente evocan esos operadores para manipular a los enteros aunque existen algunos lenguajes de bajo nivel tales como el C que permite la manipulación de los tipos bases de dos formas: evocando las operaciones o operando directamente sobre los bit del tipo base. En algunos otros lenguajes tales como el Pascal, la representación interna de la cadena de bit está completamente oculta. El programa que manipula enteros puede ser fácilmente portable y compilable sobre sistemas que usen diferentes representaciones internas para enteros.

Un lenguaje que incorpore tipos abstracto de dato extienden ese contexto a todos los objetos o datos del programa. El mecanismo de la abstracción obliga a que el acceso y actuación de objetos con sus respectivas operaciones de manipulación estén encapsuladas de aquí que solo puede ejecutarse una operación que fué definida para un objeto particular a través de la interfaz.

4.- Concepto sobre la Orientación por objeto

¿ Qué es un Objeto ?

Un objeto es una entidad física o abstracta que tiene un comportamiento antes ciertos estímulos, tanto externos como de otros objetos específicos que se encuentran dentro del sistema.

¿ Qué se puede considerar como objeto ?

- Persona
- Equipo Hardware
- Materiales
- Información
- Software
- Procesos

-Procedimientos

Ejemplo de objeto:

Objeto Físico: Horno

Operaciones asociadas: Encendido/Apagado/Cargado/Descargado.

Objeto Abstracto: Cola

Operaciones: Agregar/Eliminar/Verificar vacía/Verificar si llena/Primero cola/Siguiente cola/

Identidad de un objeto

Cada objeto tiene su propia identidad que lo distingue de los demás objetos. En otras palabras, dos objetos distintos no son iguales aunque todos los valores de sus atributos sean idénticos.

En el mundo real un objeto simple existe, pero dentro de un lenguaje de programación cada objeto tiene un único nombre por el cual puede hacerse referencia. La identidad de un objeto significa que los objetos son distiguibles por su existencia inherente y no por las propiedades descriptivas que ellos pueden tener. Por ejemplo dos manzanas con el mismo color, forma y textura son aún manzanas individuales.

La identificación de un objeto concierne al proceso por medio del cual tanto conceptos abstractos (es decir clases) y objetos concretos (es decir instancias) son identificables de manera única. Por ejemplo, el nombre de una clase debe ser el identificador único de la clase y las distintas instancias de objetos se le debe asignar un único identificador de objeto. En el modelo de Entidad Relación Extendido todos los tipos entidades y relaciones deben tener un nombre único y en cada tipo de definición los nombres de los atributos deben ser distintos. En el nivel de implamtación las bases de datos relacionales distinguen entre entidades de un tipo dado por el significado de sus valores claves. Esos valores claves son a menudo utilizado para localizar el objeto en el medio de almacenamiento físico y su rápida localización por medio de funciones hash, índices o algún otro método de acceso. Sin embargo los valores de los campos claves representan un aspecto del estado de un objeto el cual puede cambiar con el tiempo mientras que el identificador de un objeto debe permanecer immutable a pesar de cambios en el estado del objeto, posición o estructura. En los lenguajes de programación, las instancias de clases pueden ser identificada por los nombres de la variable y eso da una medida de immutabilidad dentro del alcance de un programa simple. Sin embargo proveer a las instancias de una clase un identificador único es un problema de mucha dificultad debido a que esas instancias persisten más allá del tiempo de vida de un programa.

Uno de las primeras interrogantes que se debe de responder cuando se adopta un enfoque de orientación por objeto para el diseño de bases de datos es ¿ Cómo se encontrará el objeto ?. En efecto

la tarea real es encontrar conjuntos o clases de objetos con una estructura y comportamiento común. La solución que se ha dado para resolver este problema es que cada clase y sus respectivas instancias (objetos) el sistema le asigna un identificador único independiente del valor del objeto y de la localización del objeto en el medio de almacenamiento físico.

Componentes para la construcción de Software de un objeto.

Cuando un objeto se transforma en una realización de software, consta de una interfaz, una estructura de datos privada y unos procesos llamados operaciones o métodos que son los únicos que pueden transformar legítimamente la estructura de datos de un objeto diccionario

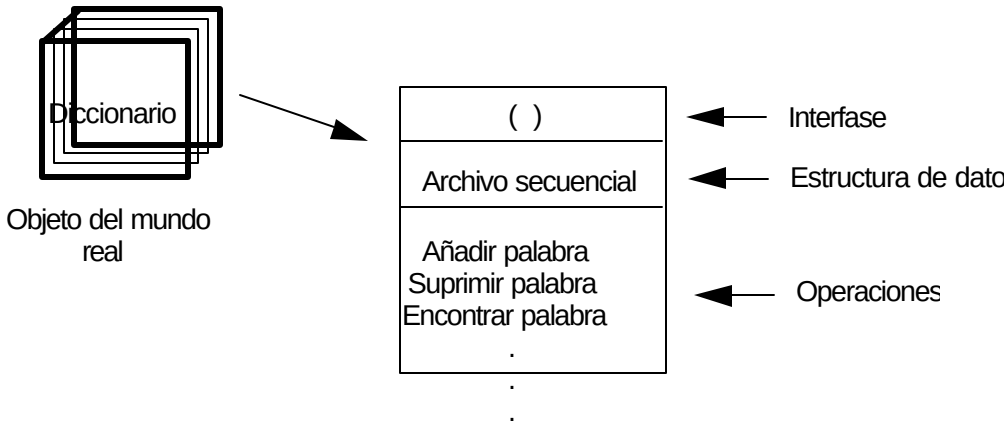


Fig. 4 Representación de un objeto.

Tipos de objetos

1.- Objetos reales

- **1.1** Objeto físico: Son objetos extremamente perceptibles ejemplo(vista, olfato,gusto,oído)
- **1.2** Objetos psíquicos: Son internamente perceptibles. Ejemplo: dolor, sentimiento, imágenes.

2.- Objeto ideales no son perceptibles.

- **2.1** Relaciones o asociaciones entre objetos.
- **2.2** Pensamientos
- **2.3** Matemáticos, lógicos semiótico

1.- Clasificación:

El proceso de clasificación es el enfoque central de la orientación por objeto y concierne a la agrupación de objetos con propiedades (estructuras de datos o atributos) y comportamiento (operaciones) similares dentro de una clase.

Una clase es la abstracción que describe propiedades importantes para una aplicación.

Los objetos que pertenecen a una clase se describen colectivamente por la definición de una clase, esto significa que en lugar de describir los objetos individuales la orientación por objeto concentra en un patrón tanto el estado como el comportamiento que es común a todos los objetos de la clase. También es posible clasificar objetos en subclases basado sobre semánticas adicionales. El estado de un objeto es representado por sus propiedades y su comportamiento por un conjunto de procedimientos que están encapsulado con las propiedades. Esta clase de estructura que abarca tantos propiedades como comportamiento es la unidad natural de la abstracción en los sistemas de orientación por objeto y puede ser utilizarse para modelar tanto entidades objetos como relaciones entre los objetos.

Cada clase describe una posibilidad infinita de un conjunto individual de objetos. Cada objeto que pertenezca a una clase es llamado instancia de la clase.

La instanciación es lo inverso a la clasificación y concierne a la generación de los distintos objetos de una clase. La distinción entre una clase y sus instancias es similar a la distinción entre una definición de tipo y la declaración de una variable en un lenguaje de programación convencional. Sin embargo la mayoría de los sistemas orientados por objeto crean dinámicamente instancias por envío de mensajes "Nuevo" y "Crear" una clase.

El ejemplo siguiente de un tipo entidad Hotel ilustra las nociones de clases e instancias:

```
Type
        Tipo_opcion(Piscina,Sauna,Tennis,Golf);
Class Hotel
        Propiedades
                Nombre: Cadena;
                Dirección: Cadena;
                Dueño: Compañia;
                Director: Persona;
                Facilidades: Set (Tipos_opcion):
        Operaciones
                Create ();
                Reserva_hab(Habitación:integer; Huesped: Person;
                                Fecha_llegada,Fecha_partida:Tipo_fecha)
end Hotel.
Class Compañía
        Propiedades
```

Nombre, Oficina_central, Teléfono: Cadena;

.....

Operaciones

.....

end Compañia.

Class Persona

Propiedades

Nombre, Dirección, Teléfono: Cadena;

.....

Operaciones

.....

end Persona.

Una instancia de la clase hotel puede tomar la forma de la siguiente figura:

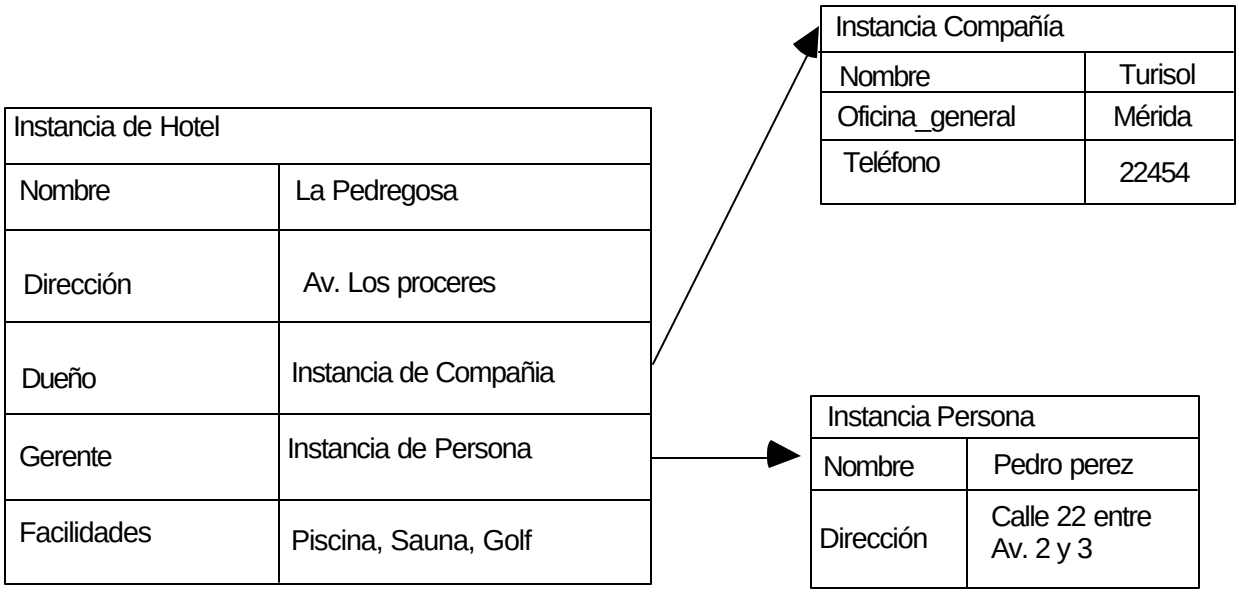


Fig. 5 Ejemplo de la clase Hotel

En este ejemplo definimos una clase Hotel la cual tiene cinco propiedades : Nombre, Dirección, Dueño, Gerente, y facilidades. Note que esas propiedades pueden ser también definidas en término de otras clases. Por ejemplo, dueño del Hotel es una instancia de la clase Compañía, Gerente es una instancia de la clase Persona, y las facilidades que ofrece el hotel son definidas como una instancia de la clase conjunto con el tipo de parámetro actual Tipo_opción.

Cada instancia de la clase tiene sus propios valores para cada atributo; pero puede compartir los nombres de los atributos con otras intancias de clases. La siguiente figura 6 muestra dos clases con sus respectivas instancias.

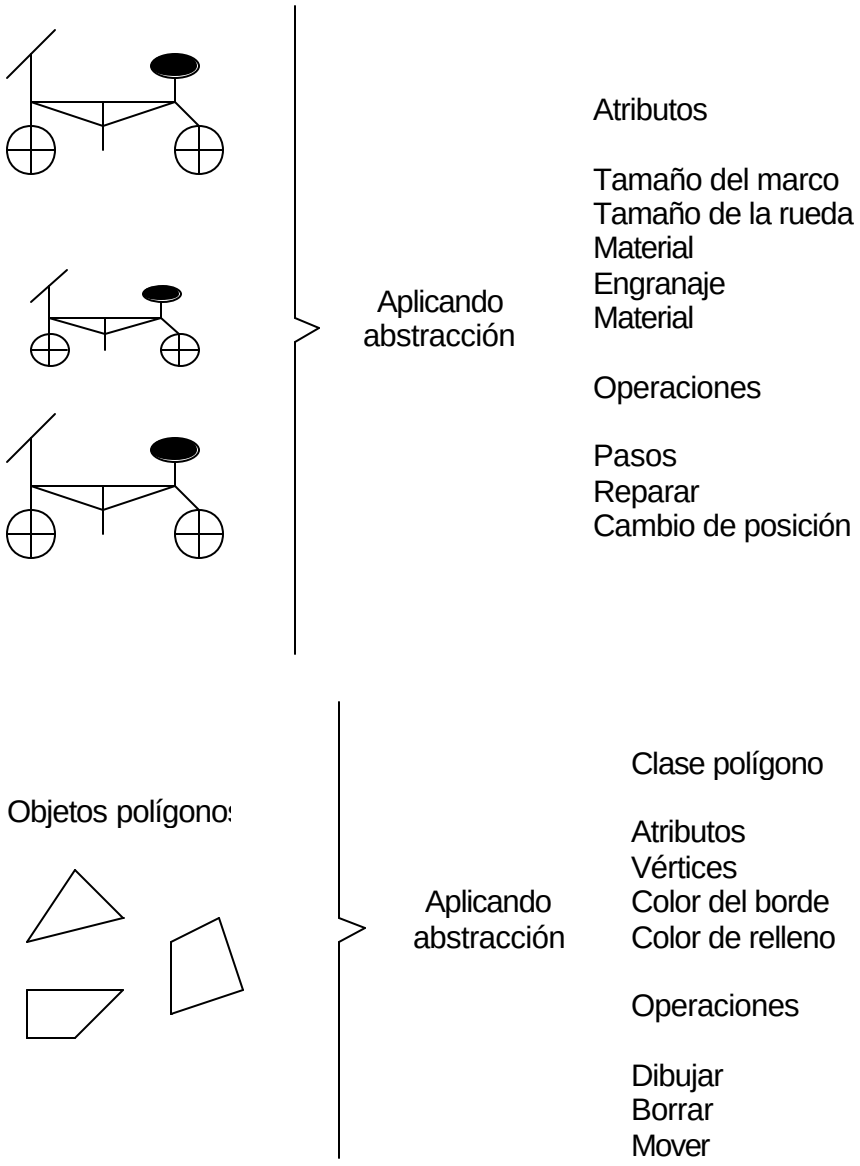


Fig. 6 clase Bicicleta y polígonos

La figura 7 muestra la clase Persona, que tiene los atributos nombre y edad donde nombre es un string y edad es un entero. Un objeto de la clase Persona tiene los valores José Hernández para nombre y 24 años para el valor edad, y el otro objeto tiene nombre María Chacón y edad 52 años.

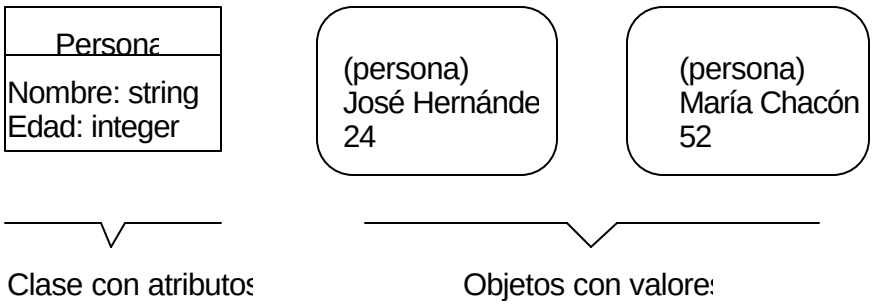


Fig 7 Clase persona y sus valores

Operaciones y métodos

Una operación es una función o transformación que puede ser aplicada sobre los objetos en una clase. Por ejemplo: abrir, cerrar, ocultar, desplegar, son operaciones sobre una clase ventana.

Cada operación tiene un objeto destino con un argumento implícito. El comportamiento de la operación depende de la clase destino.

Si la misma operación puede aplicarse sobre diferentes clases. Se dice que es polimórfica, que significa que la misma operación toma distintas formas en diferentes clases. Un método es una implantación de una operación para una clase. Por ejemplo, la clase archivo puede tener una operación de impresión. Pueden implantarse diferentes métodos para imprimir archivos de fotos digitalizadas. Todos estos métodos ejecutan lógicamente la misma tarea la cual es imprimir un archivo, por lo tanto puede hacerse referencia a ello por la operación genérica imprimir siempre y cuando cada método pueda ser implantado por un segmento de código diferente.

Una operación puede tener argumentos, parametrizan la operación pero no afectan la elección del método. El método depende de la clase del objeto final (pocos lenguajes de orientación por objetos tal es el caso de CLOS permiten que la elección de métodos dependan de algunos argumentos, pero tal generalidad conduce a una considerable complejidad semántica que no ha sido explorada aún)

El comportamiento de todos los métodos de una operación debe tener consistencia interna. Esto es, debe evitarse el uso del mismo nombre para dos operaciones que son semánticamente diferentes, igualmente si ellos son aplicados sobre distintos conjuntos de clases. Por ejemplo, usar el nombre "Invertir" tanto para describir la inversión de matrices como para la inversión de las figuras geométricas en un proyecto grande de programación puede causar confusión.

Notación para representar una clase.

Una clase se representa por una caja la cual puede tener 3 regiones. La primera región contiene el nombre de la clase, la segunda región contiene la lista de atributos, cada nombre de atributo puede estar seguido por detalles opcionales tales como tipo de atributo y valores por defectos.

La tercera región contiene los nombres de las operaciones. Cada nombre de operación puede ser seguido por detalles opcionales tales como listas de argumentos y tipos de resultados. Los atributos

y operaciones puede o no mostrarse; esto depende del nivel de detalle deseado. la figura siguiente muestra esto con mayor explicación.

Nombre de la clase
Nombre_atributo1: tipo dato1= valor defecto1 Nombre_atributo2: tipo dato2= valor defecto2 . . .
Nombre_operación1(lista de argumento):tipo_resultado1 Nombre_operación2 (lista de argumento):tipo_resultado2 . . .

Fig. 9 Representación de una clase

En la figura 10 muestra la clase Persona y la clase Objeto Geométrico. La clase Persona con los atributos nombres y edad y las operaciones cambia_trabajo y cambia_dirección. La clase objeto geométrico con los atributos color y posición y las operaciones mover_objeto con un parámetro llamado delta de tipo vector, selección con un parámetro llamado P de tipo puntero y cuyo valor devuelto es un valor booleano y Rotar cuyo parámetro es el ángulo de rotación.

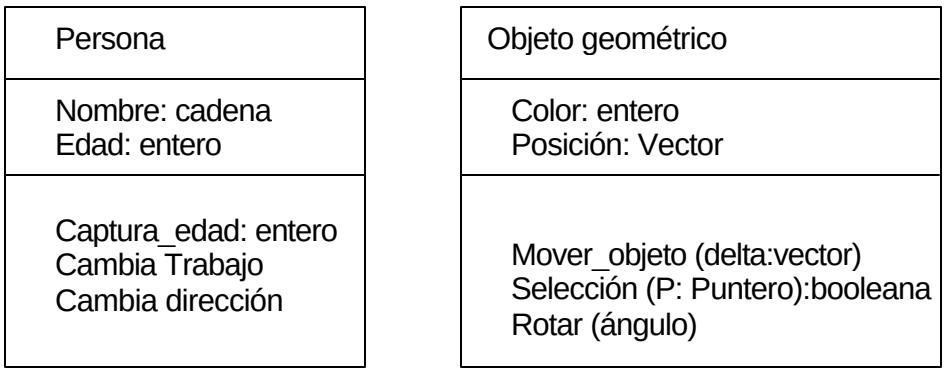


Fig 10 Clase Persona y Clase Objeto Geométrico

¿ Cómo implantar una clase en C++ ?

En C++ una simple clase puede definirse con la siguientes palabras reservadas: Struct, unión o clase combinada con funciones denominadas funciones miembros de la clase y datos llamados datos miembros. Usualmente a una clase se le asigna un nombre. Ese nombre identifica un nuevo tipo de dato que se puede utilizar para declarar instancias u objetos de ese tipo de clase.

Funciones miembros:

Una función miembro es una función que ha sido declarada dentro de la definición de la clase. Las funciones miembros son conocidas como métodos en otros lenguajes de orientación por objeto. Existen dos formas de adicionar una función miembro para una clase:

-Definición de la función dentro de la clase.
-Declaración de la función dentro de la clase, y la definición de la función se encuentra fuera de la clase.

Datos miembros son los datos que la clase conoce.

Ejemplo:
Declaremos la clase Persona que se definió anteriormente.

```
Class Persona
    /* Declaración de los datos miembros */
    Char Nombre(30), dirección(80)
    int edad;
    /* Declaración de las funciones miembros */
    int captura_edad ( );
    void cambia_trabajo ( );
    void cambia_dirección ( );
```

fig 11a

```
Class persona
    /* Declaración de datos miembros */
    Char Nombre(30), dirección (80);
    int edad;
    /* Declaración de funciones miembros */
    int captura_edad ( )
    Return(edad);
    void cambia_trabaajo ( )
    void cambia_dirección ( )
```

fig 11b

Observe que en la declaración de la clase de la fig. 11 a, las funciones solo están declaradas dentro de la clase, y en la fig b la función captura_edad está definida dentro de la clase persona.

La mayor diferencia entre las clases de C++ y las estructuras de C consiste en la accesibilidad a los miembros. En C++ los miembros de una estructura (struct) se pueden accesar libremente por alguna expresión o función que esté a su alcance, también con C++ se puede controlar el acceso a los miembros de las estructuras (struct) y clases (class) por la declaración individual de los miembros como: públicos, privados o protegidos.

¿ Cómo se declara una función miembro fuera de la clase ?

Por ejemplo considere la clase punto:

```
Class punto {
        int x,y;
        Boolean visible;
        int getx ( );
        };
int punto : : getx ( ) {
        Return (x);
        };
```

Los **: :** son conocidos como el alcance del operador resolución, lo que indica al compilador donde comienza la función.

Las funciones miembros definidas fuera de la clase pueden hacerse en líneas (si se dan ciertas condiciones), pero teniendo que preguntar esa explicidad con la palabra reservada **inline.**

Note cuidadosamente el uso del alcance de la clase punto para la definición de la función getx. La clase llamada punto se necesita para la compilación de la operación getx debido a que esta operación pertenece a dicha clase. Las operaciones declaradas dentro de la clase no necesitan del operador **: :** porque la ellas pertenece claramente a la clase.

Generalización (Metaclase o superclase)

La generalización según [Smith y Smith 1977] es una abstración en el cual un conjunto de objetos con propiedades similares se representa por una clase genérica. Dentro del marco de la orientación por objeto es quizas el más importante mecanismo para modelar el mundo real, permitiendonos ir gradualmente desde lo especifico a lo general. La generalización nos permite movernos desde observaciones de propiedades especificas de objetos hasta un modelo que representa esos objetos por una clase genérica. Existen dos maneras en el cual la generalización puede efectuarse:

1.- Las propiedades comunes y funciones comunes de un grupo de objetos con tipos similares son agrupados para formar un nuevo tipo de clase genérica.

2.- Sub-tipos de un tipo de objeto dado pueden ser definidos utilizando predicados para las restricciones de los valores de los atributos.

En las secciones anteriores hemos descrito a los objetos como una instancia de una clase. Una clase contiene la descripción de esos objetos (estructura de datos y comportamiento). Con lo anterior mencionado cabe preguntarse aquí, ¿pueden unas clases ser instancias de otras clases?

La respuesta es sí, pero algunos de los lenguajes de orientación por objeto tienen conflictos para resolver esto, sin embargo existen otros lenguajes de orientación por objeto que proporcionan una solución muy confusa a este problema.

El primero de los lenguajes que trabajan con generalización fué Smalltalk, donde una clase es una instancia particular de una tipo de clase llamada metaclase o superclase. Por lo tanto, una generalización es una clase cuyas instancias también son clase en base a una o más propiedades comunes.

La generalización define la relación entre una clase y una o más versiones refinadas de ella misma. La clase inicial refinada es llamada metaclase o superclase y cada versión redefinida es llamado subclase. Por ejemplo, en la figura 12

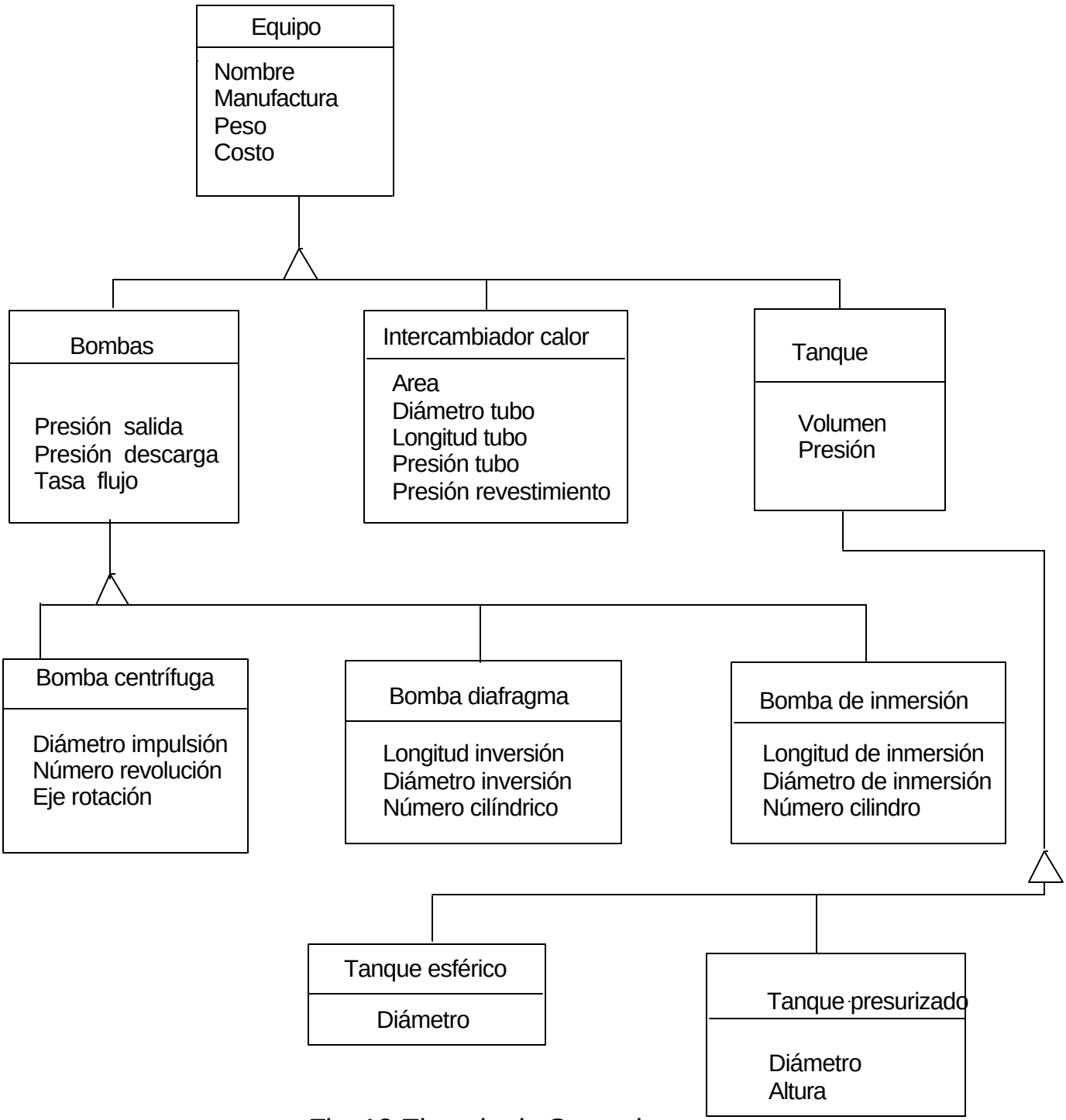


Fig. 12 Ejemplo de Superclases

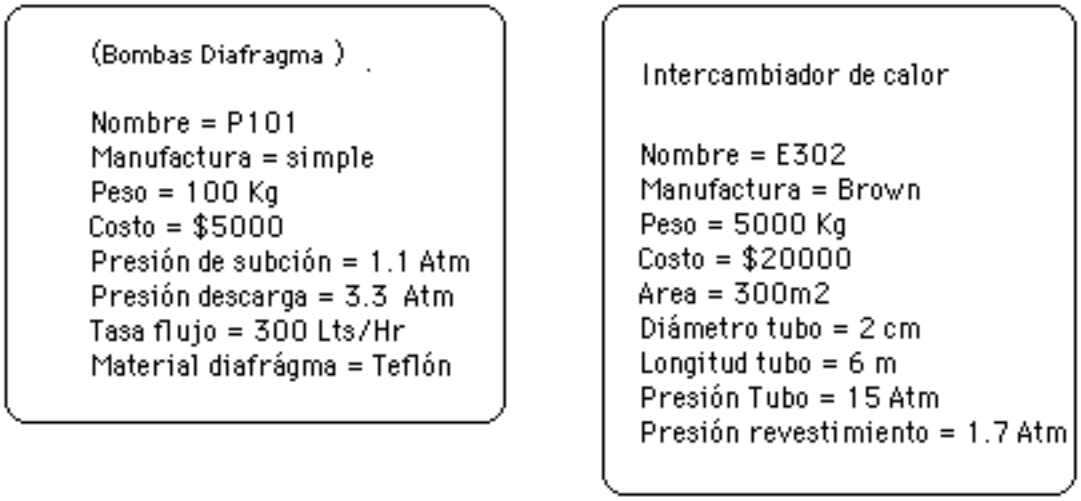


Fig. 13 Algunas instancias del modelo de la fig 12

Equipo es la superclase de Bomba y Tanque. Los atributos y operaciones comunes al grupo de subclases están ligadas a la superclase y son compartidas por cada una de las subclases. Cada subclase hereda rasgo de la superclase. Por ejemplo, los atributos de Bomba tales como manufactura, peso y costo son heredados de la superclase equipo. La generalización es algunas veces llamado relación is a (es una) porque cada instancia de una subclase es una instancia de la superclase.

La generalización y la herencia se cruzan transitivamente en un número arbitrario de niveles. Los términos ancestro y descendiente referidos por la generalización de clases cruzan múltiples niveles. Una instancia de una subclase es simultáneamente una instancia del ancestro de las clases. El estado de una instancia incluye un valor para todos los atributos de todas las clases ascentros. Algunas operaciones sobre una clase ancestro pueden aplicarse sobre sus instancias. Cada subclase no sólo hereda todos los rasgos de sus ancestros sino que tienen sus propios atributos y operaciones específicas, por ejemplo, Bomba adiciona los atributos: tasa de flujo, el cual no es compartido por otro miembros de equipo.

La notación gráfica para la generalización es un triángulo que conecta a las superclases con la subclase.

Las palabras escritas cerca del triángulo en el diagrama son los discriminantes. Un discriminante es un atributo de enumeración de tipo, que indican cuales propiedades de un objeto es la abstracción inicial de la relación de generalización. Solo una propiedad debe ser discriminante a la vez.

Uso de la generalización

La generalización es un constructo muy utilizado para el modelado conceptual y la implantación. La generalización facilita el modelaje por estructuración de clases en función de la diferenciación entre lo que es semejante y lo que es diferente a cerca de las clases. La herencia de operación es de mucha ayuda durante la implantación porque conduce a la reutilización de código.

Los términos de herencia y generalización se refiere a la misma idea y a menudo son utilizados intercambiablemente, pero nosotros utilizaremos el término generalización para referirnos a las relaciones entre las clases, mientras que el término herencia se refiere al mecanismo de compartir atributos y operaciones.

Agregación

La agregación según [Smith and Smith, 1977] es una abstracción en la cual una relación entre objetos es representada por un objeto agregado de alto nivel o un tipo.

La agregación es una relación que se define como "es parte de quien" o "es parte de" en la cual el objeto se representa como una asociación de componentes de la misma especie. Esta relación "es parte de quién" ocurre muy frecuentemente en aplicaciones de bases de datos y representa una situación donde una clase se ensambla o agrega a objetos componentes. Cada instancia de una clase ensamblada está comprendida de un conjunto de instancias compuestas y por esta razón a menudo es referenciado como un objeto compuesto. También podremos adicionar la noción de dependencia como consecuencia de la relación "es parte de quién". Un objeto dependiente es un objeto cuya existencia depende de la existencia de otro objeto. Ejemplo:

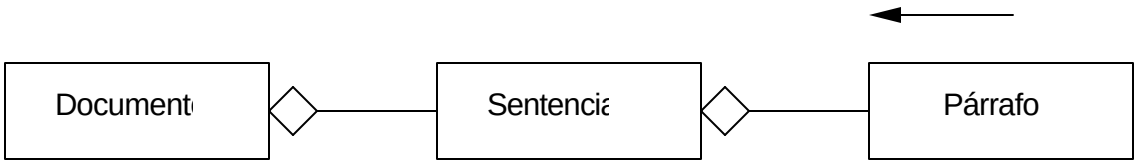


Fig 14 Diagrama para representar la agregación

La agregación se representa por un diamante que índica cual es el objeto de alto nivel que se forma por los objetos de más bajo nivel de la misma especie. En la figura 14 se observa que el objeto sentencia es parte del objeto documento y el objeto párrafo es parte del objeto sentencia.

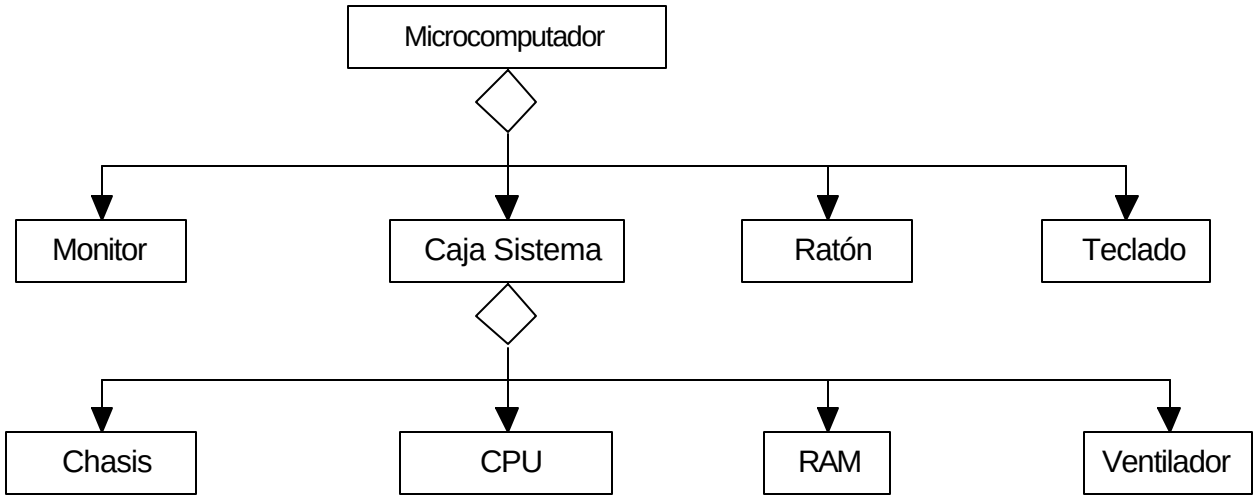


Fig 15 Ejemplo de representación de la agregación

La relación "es parte de quien" es completamente diferente a la relación de herencia. Esto no se aprecia facilmente y los dos conceptos algunas veces se confunden, para ilustrar esta diferencia estudiemos el ejemplo de la figura 16:

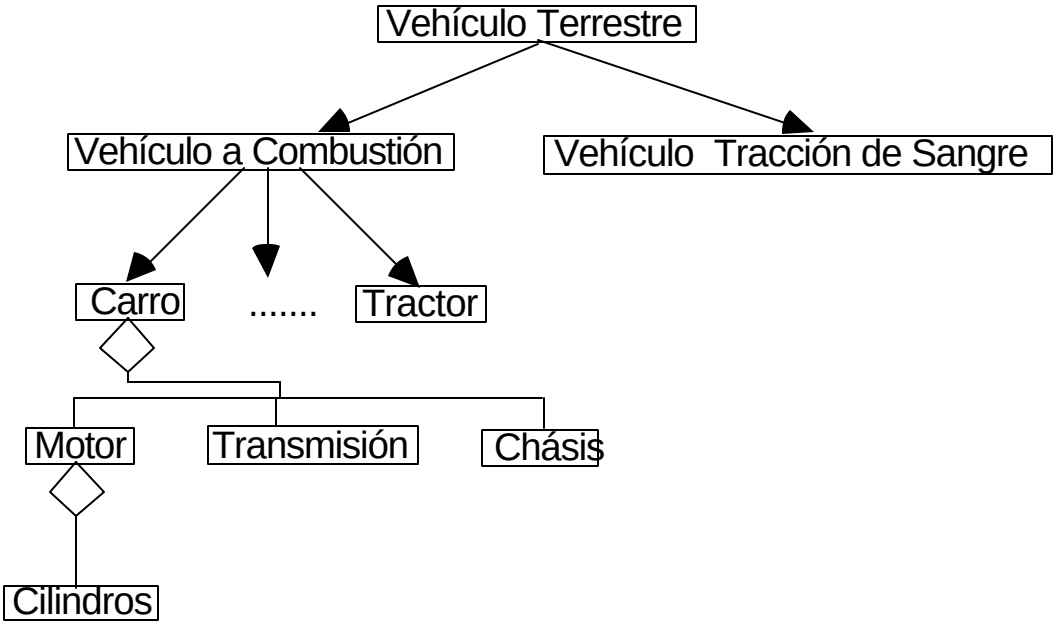


Fig. 16 Agregación y Herencia.

Observe que la clase Carro hereda las propiedades y operaciones de la clase vehículo terrestre. Sin embargo un objeto carro no "es parte de" la clase Vehículo terrestre, ni es asociada con una instancia de Vehículo. Aunque un carro puede ser considerado que esta formado por tres grandes componentes: el chasis, el motor y la transmisión. Cada uno de estos componentes es una clase con sus

propias características y funciones, y cada una de ellas posee sus propios componentes constituyentes tal como Cilindro es parte de motor. Cada instancia de chasis, motor y transmisión pertenece exclusivamente a una instancia de carro. Las clases chasis, motor y transmisión esta relacionadas con carro a través de la relación "es parte de" y su existencia depende de la existencia de una instancia de carro que las relaciones.

Agregación vs Generalización.

La agregación no es lo mismo que la generalización. La agregación relaciona instancias, involucrando dos objetos distintos, uno es parte del otro. La generalización relaciona clases y es una manera de estructurar un objeto simple. La superclase o la subclase hacen referencia a las propiedades del objeto simple. Con la generalización, un objeto es simultáneamente una instancia de una superclase y una instancia de una subclase. La confusión surge porque ambos agregación y generalización dan origen a un árbol de clausura transitiva. Un árbol de agregación está compuesto de instancias de objeto, que son todos parte del objeto a formar; un árbol de generalización está compuesto de clases que describen a un objeto. La agregación es a menudo relación "parte de"; la generalización es llamada relación "es una".

La figura 20 muestra la agregación y la generalización para el caso de una lámpara de escritorio.

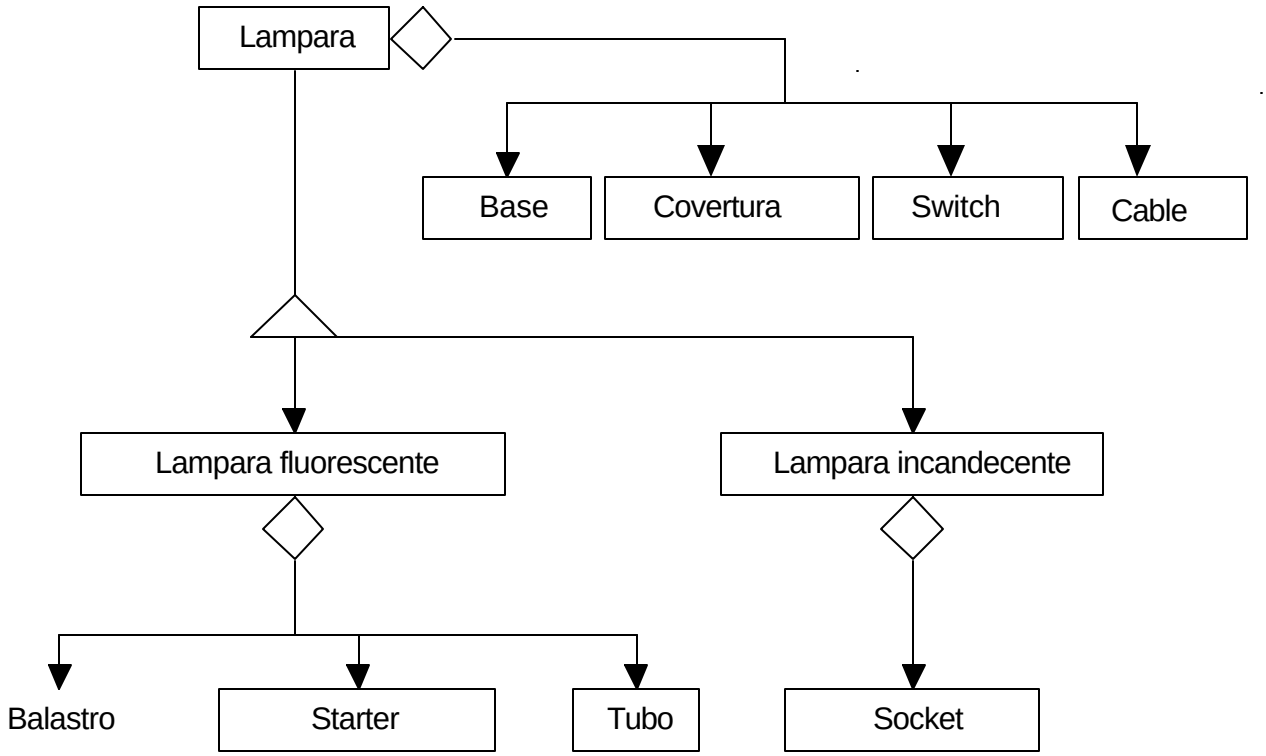


Fig 20 Ejemplo de Agregación y Generalización

Base, cobertura, swicth son las partes de la lámparas y pueden clasificarse en algunas subclases diferentes tales como fluorescente e incandescente. Cada subclase puede tener sus propias partes distintas, por ejemplo una lámpara fluorescente tiene un ballastro, un tubo, y un starter; una lámpara incandescente tiene un socket.
Agregación es algunas veces llamado relación " Y " y la generalización relación " O "

Encapsulamiento

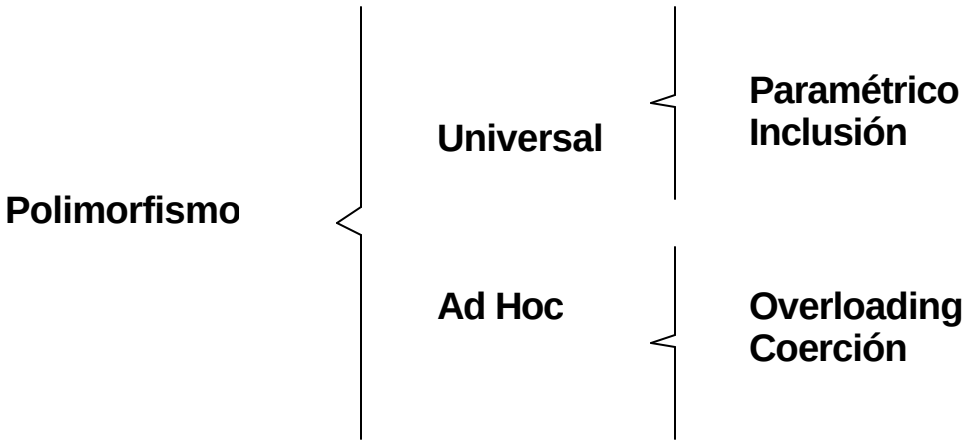
El encapsulamiento también se llama información oculta, consiste en la separación de los aspectos externos de un objeto, el cual es llamado por otro objeto de los detalles de su implantación interna.
El objeto llamador no debe enterarse como está implantado internamente el objeto llamado.

La encapsulación impide a un programa ser interdependiente de los pequeños cambios que el diseñador realice a un objeto. La implantación de un objeto puede ser cambiada sin afectar la aplicación que lo usa. Se puede desear cambiar la implantación de un objeto para mejorar la ejecución, ocultar fallas, consolidar código para la portabilidad. El encapsulamiento no es único de los lenguajes orientado por objeto, pero la habilidad de estos para combinar estructura de datos y comportamiento en una entidad simple hace que la encapsulación sea más clara y poderosa que los lenguajes convencionales, quienes separan la estructura de dato del comportamiento.

Polimorfismo

Polimorfismo significa que la misma operación puede comportarse diferentemente sobre distintas clases. Por ejemplo, la operación "mover" ejemplo puede comportarse diferentemente sobre una clase llamada Ventana y una clase llamada Piezas_ajedrez. Una operación es una acción o transformación que un sujeto ejecuta sobre un objeto, por ejemplo, justifica a la derecha, desplegar, mover. Una implantación específica de un operación para una cierta clase es llamada método. Un operador orientado por objeto o polimórfico puede tener más de un método implantado.

Clasificación de Polimorfismo



Polimorfismo Paramétrico: Se obtiene cuando una función trabaja uniformemente sobre un rango de tipos; esos tipos normalmente exhiben una estructura común y puede comportarse de manera distinta para cada tipo.

Polimorfismo de Inclusión: Es un polimorfismo utilizado por modelos de subtipos y herencia. En este tipo de polimorfismo un objeto puede pertenecer a clases diferentes que no necesariamente son disjuntas.
El polimorfismo paramétrico y de inclusión están clasificados en una categoría mayor llamada polimorfismo universal, el cual contrasta con el polimorfismo no universal o el polimorfismo ad hoc.

Polimorfismo por Overloading: En este caso el mismo nombre de la variable se utiliza para denotar diferentes funciones, y el contexto se utiliza para decidir cual función se debería ejecutar para una invocación particular del nombre.

Puede imaginarse que para el procesamiento de un programa eliminemos el overloading por asignación de nombres distintos a las funciones diferentes, en este caso tendríamos programas con muchos nombres de funciones. El overloading es justamente una abreviación sintáctica conveniente, que permite poseer diferentes funciones con un mismo nombre.

Polimorfismo por Coerción: Es una operación semántica que convierte argumentos a los tipos esperado por una función, en una situación que de otra forma resultaría en un tipo de error. La coerción puede estar dada

estáticamente, insertándose automáticamente entre argumentos y funciones a tiempo de compilación o pueden tener que determinarse dinámicamente, con pruebas a tiempos de ejecución sobre los argumentos.

Herencia Jerárquica

Un poderoso concepto que caracteriza a los lenguajes de orientación por objeto es la herencia. La herencia consiste en el compartir atributos y métodos entre clases basándose en una relación jerárquica. Una clase puede definirse ampliamente y redefinirse sucesivamente en subclases más refinadas. Cada subclase que se incorpora, hereda todas las propiedades de su superclase y adiciona sus propias y únicas propiedades. Las propiedades de la superclase no necesariamente deben de repetirse en cada subclase. La habilidad para sacar factor común de las propiedades de una clase dentro de una superclase común y la herencia de propiedades desde la superclase pueden ayudar enormemente en el diseño.

Por ejemplo, si tenemos una compañía que está dividida en dos subclase llamadas Compañía con fines de lucro y organizaciones sin fines de lucro. Ambos tipos de organizaciones contienen información almacenadas como: nombre de la compañía, dirección de la compañía y personal. La subclase Organización sin fines de lucro tiene una información especializada como: garantías gubernamentales, apoyo comercial, enlaces internacionales, etc. Similarmente, la compañía con fines de lucro tiene una especialización tal como: deducción de impuesto. Por lo tanto, para capturar las operaciones comunes o comportamiento construiremos una clase genérica llamada Compañía que posee dos clases, compañías comerciales y organización sin fines de lucro. En la figura 21 se observa lo explicado.

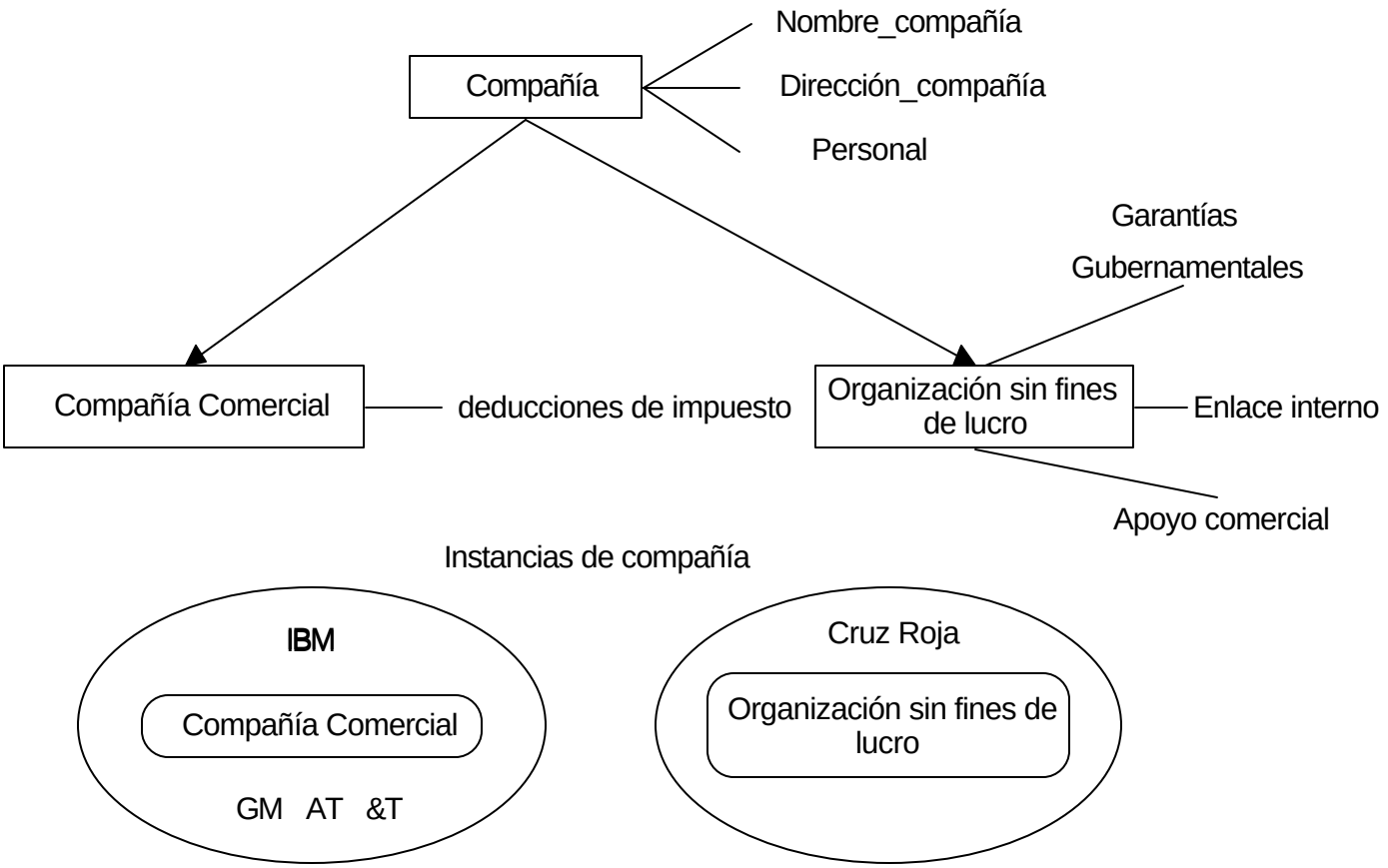


Fig. 21 Herencia Jerárquica

Compañía comercial y Organización sin fines de lucro son subclase de la clase Compañía. La clase Compañía es una superclase de compañía comercial y Organización sin fines de lucro. La subclase y las superclases tienen una relación transitiva, esto significa lo siguiente: si "X" es una subclase y además es superclase de "Y" , y "Y" es una subclase y además superclase de "Z" entonces "X" también es una subclase y además superclase de "Z". Por ejemplo si en el gráfico anterior se anexa una subclase llamada compañía de semiconductores a compañía comercial, entonces por transitividad Compañías de semiconductores es una subclase de Compañía. Esto significa que compañía de semiconductores hereda el compor-tamiento y forma de representación de Compañía a través de Compañía comercial. La figura 22 expresa gráficamente lo anterior.

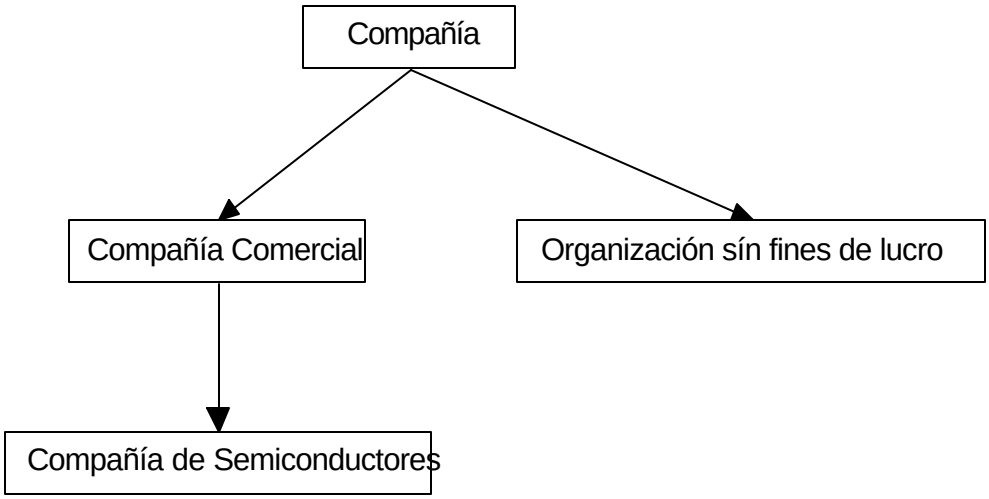


Fig. 22 Herencia estructural y de comportamiento.

Como se observa en el la figura 22 existe dos aspectos importantes de herencia:

- Herencia Estructural.
- Herencia de Comportamiento (herencia de métodos).

Herencia Estructural: Significa que las instancias de una clase tal como Organizaciones sin fines de lucro, la cual es una subclase de compañía, heredará las instancias variables de Compañía tales como: nombre, dirección, etc.

Herencia de Comportamiento: La clase Compañía tiene métodos tales como: Adicionar subsidiarias, Evaluación de presupuesto, etc; las cuales son heredadas por las subclases Compañía comercial y Organización sin fines de lucro.Esto significa que se puede mandar un mensaje al método Adicional subsidiarias para una instancia llamada NPO de Organización sin fines de lucro y ejecutar el método Adicional subsidiaria en Compañía con NPO como objeto destino.

De lo anterior se observa que una clase define la estructura y el comportamiento de una colección de objetos.

El comportamiento se especifica en el método asociado con las instancias de la clase, recordando que los métodos son operaciones que pueden recuperar o actualizar el estado de un objeto. El estado de un objeto está almacenado en las variables de instancia.

En la herencia jerárquica un método definido por una clase es heredado por una subclase de ésta, por lo tanto la herencia de métodos es parte de

la interfaz de manipulación de instancia de una subclase. Por ejemplo si se tiene una clase Ventana con la subclase Ventana con borde y Ventanas de texto. Los métodos tales como mover_ventana cambia tamaño_ventana y rotar_ventana puede invocarse sobre las instancias de Ventana de texto y Ventana con bordes. La subclase Ventana sin texto posee métodos más especializados para editar y modificar el tipo de caracteres y tamaño de caracteres (editar, font, size). Esos métodos no están definidos sobre las instancias Ventana. La subclase ventana con borde pueden también tener sus propios métodos (editar, font, etc), que no tienen ninguna relación con los otros métodos con los mismos nombres en otras clases.

Otro ejemplo de herencia se observa en la figura 23.

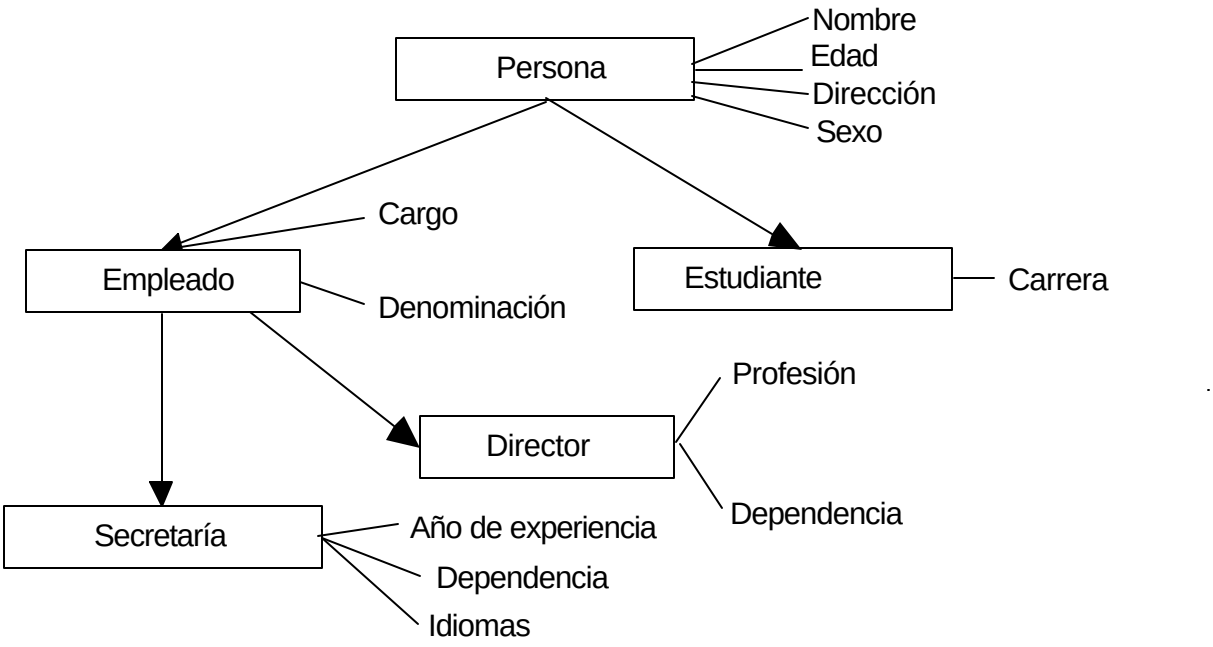


Fig. 23 Ejemplo de herencia.

Herencia Múltiple: La herencia múltiple permite a una clase tener más de una superclase y heredar todos los rasgos de todos sus padres. Esto permite mezclar información de dos o más fuentes. Es una forma más complicada de generalización que de simple herencia, la cual restringe las clases a una jerarquía de árbol. La ventaja de la herencia múltiple es que

aumenta el poder en una clase específica o incrementa la posibilidad de reutilizarse. La desventaja es que se pierde toda simplicidad tanto conceptual como de implantación.

Definición de Herencia Múltiple: Una clase puede heredar rasgos de más de una superclase. Una clase con más de una superclase es llamada clase junta. Un rasgo de una clase ancestro que se encuentra más de una vez a lo largo de una ruta solo se hereda una vez. Los conflictos entre definiciones paralelas de clases crean ambiguedades que son resueltas en la implantación. En la práctica, tales conflictos deben evitarse o resolverse explícitamente.
En la figura 24 se explica la herencia múltiple con un ejemplo:

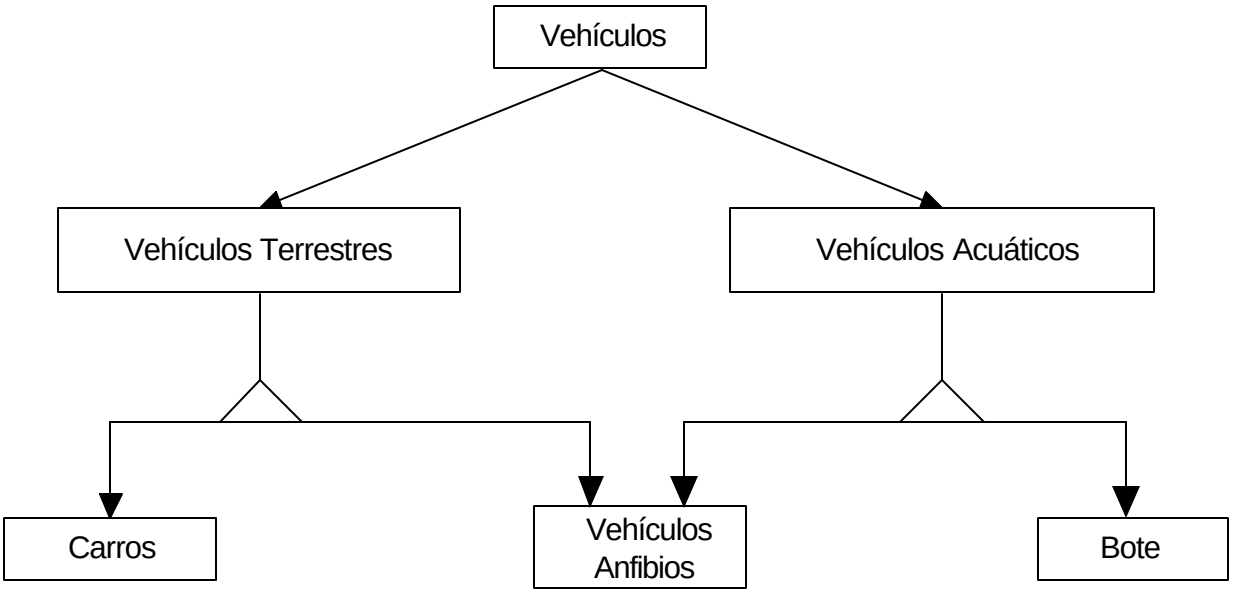


Fig. 24 Herencia múltiple para clases juntas.

En la figura 24 Vehículo_anfibio es tanto un Vehículo terrestre como un Vehículo acuático. Por lo tanto Vehículo_anfibio es una clase junta. A este tipo de herencia se le conoce como herencia múltiple por el sobrelapamiento de clase, esto quiere decir que una subclase hereda tanto la estructura de datos como el comportamiento de dos o más superclases.

Si una clase puede refinarse en algunas dimensiones distintas e independientes, entonces debe utilizarse generalizaciones múltiples. Recuérdese que el contenido del método objeto es conducido por esas relevancias para una solución eficiente a una aplicación.

En la figura 25 se muestra un ejemplo de multiherencia para clases disjuntas:

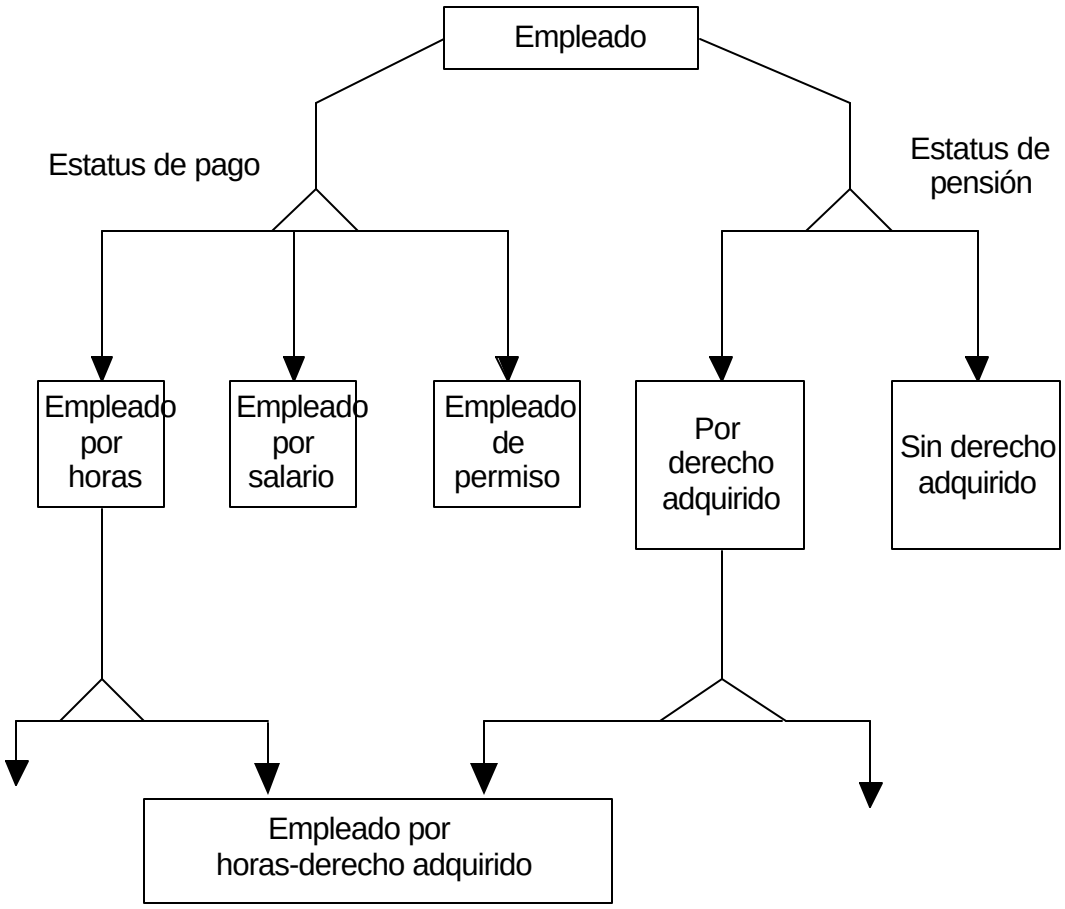


fig. 25 Herencia múltiple para clases disjuntas.

En la figura 25, la clase Empleado es independiente de las especializaciones, estatus de pago y estatus de pensión, por esta razón se muestra con dos generalizaciones separadas.

Las subclases de una generalizaciones pueden o no ser disjuntas. Por ejemplo, en la figura 24 Vehículo de tierra y Vehículo de agua se superponen porque algunos vehículos pueden viajar sobre agua y tierra, como es el caso de los vehículos anfibios de la fuerza naval. Por otro lado se observa de la figura 25 que Empleado por horas, Empleado por salario, Empleado por permiso son disjuntos.

Encadenamiento Dinámico:

Una de las ventajas que promueve el estilo de programación orientada por objeto es la característica del encadenamiento dinámico, también llamado encadenamiento tardío. En efecto, no se tendrían sistemas orientados por objeto sin esa poderosa capacidad.

Simplemente, la declaración encadenamiento dinámico significa que el sistema encadenará una rutina a un selector para un método particular que está implantado sobre un objeto clase. La capacidad del encadenamiento a tiempo de ejecución es necesario por:

1.- El mismo mensaje o nombre del método (es decir, selector) puede utilizarse por diferentes clases (overloading)

2.- Una variable objeto de una clase puede ser no conocida hasta el tiempo de corrida (por ejemplo cuando el lenguaje es menos fuerte en tipos y las variables pueden ser objetos de diferentes tipos en un mismo programa).

Un ejemplo prototipo para ilustrar la ventaja del encadenamiento dinámico es la escritura de un mensaje que se aplica a todos los elementos de una colección heterogénea de objetos. Asumiremos tener una pila que pueda contener algunas especies de objetos y estamos interesados en imprimir toda la pila. Por lo tanto, asumiremos que un método de impresión con una particular implantación se asocia con cada clase (de aquí esos métodos de impresión son distintos y sin relación alguna). Asumiremos que la pila es ST y se implanta como un arreglo de 1 hasta el tope de objetos, el pseudocódigo para imprimir todos los elementos de la pila es for P:=1 to tope do print(ST [i]), de aquí que cada objeto ST [i] será ejecutado con su apropiado método de impresión, dependiendo de la clase a la cual el pertenece; esto se ilustra en la figura 26.

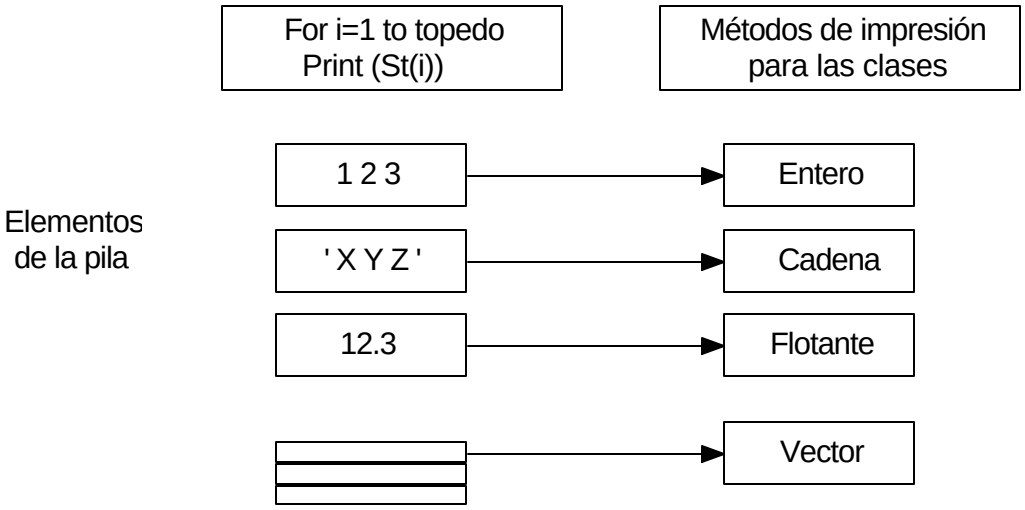


Fig. 25 Encadenamiento Dinámico a tiempo de ejecución.

El puntero al objeto es el que decide que pieza de código se ejecuta para imprimir el mensaje.

Estilo de Programación

Así como algunos jugadores profesionales de ajedrez pueden realizar algunas otras actividades tales como: cocinar o esquiar, ellos pueden dar testimonio de que existe una gran diferencia entre el conocimiento de algo y lo que ellos realizan muy bien como es el caso de jugar ajedrez y realizar las otras actividades.

Escribir un programa orientado por objeto no es diferente al ejemplo anteriormente planteado, no es suficiente conocer la estructura básica de la orientación por objeto, para lograr un ensamblaje inteligente de un programa, es necesario tener experiencia de programador para seguir el principio de realizar programas que perdurarán más allá de una necesidad inmediata.

Un buen estilo es muy importante en todo tipo de programación pero lo es más en una programación orientado por objeto pues el principal beneficios de la orientación por objeto es producir programas reusables, extensibles y de fácil comprensión.

Estilo de programación para la orientación por objeto

Realizar un buen programa es más que satisfacer los requerimientos funcionales. Programas que siguen guías de diseños tienen más posibilidades de ser correctos, reusables, extensibles y rápidos de depurar. Además, las guías de estilo que se utilizan para programación convencional también pueden aplicarse a programas con orientación por objeto, adicionándoles facilidades tales como la herencia, que es una propiedad particular de los lenguajes orientados por objeto y que requieren nuevas guías para su programación.

Presentaremos una guía para el estilo de programación orientada por objeto:

* Reusabilidad
* Extensibilidad
* Robustez
* Programación a gran escala

Reusablidad:

Un software reusable reduce diseño, codificación y costos de pruebas por amortización de esfuerzo sobre algún otro diseño. La reducción del tamaño del código también simplifica la entendibilidad, aumentando la posibilidad de que el código sea correcto. La reusabilidad es posible en los lenguajes convencionales, pero en los lenguajes orientados por objetos incrementan enormemente la posibilidad de reusabilidad de código.

Bondades de la reusabilidad:

* Compartir nuevo código escrito dentro de un proyecto de programación.
* Reuso de código previamente escrito sobre un nuevo proyecto de programación.

A estas dos bondades se aplica la misma guía de estilo de reusabilidad. El comportamiento de código dentro de un proyecto es un asunto de descubrir la secuencia de código redundante en el diseño y la utilización de las facilidades que ofrecen los lenguajes de programación tales como procedimientos o métodos para compartir la implantación. Esta bondad de comportamiento del código da como resultado casi siempre la
producción de programas pequeños, fáciles de depurar y de fácil interacción con el diseño.

La planificación para el reuso futuro de los módulos se debe de tomar con más previsión, lo que representa una investigación profunda. Es muy poco probable que una clase aislada sea utilizada en múltiples proyectos. Los programadores probablemente reusarán cuidadosamente ideas de subsistemas de salida tales como tipos abstractos de datos, paquetes gráficos, librerías de análisis numérico, etc.

Reglas de estilo para la Reusabilidad:

* Mantener métodos coherentes.

Un método es coherente si el ejecuta una función simple o un grupo de funciones estrechamente relacionadas. Si dos o más cosas no están relacionadas, entonces debemos de picarlos en métodos pequeños y separados.

* Mantener métodos pequeños.

Si un método es grande entonces se debe picar en métodos pequeños. Un método que exceda una página es probablemente muy largo. Para picar un método en partes pequeñas usted puede dividirlo inteligentemente en partes reusables o en partes iguales cuando el método completo no es reusable.

* Mantener métodos consistentes.

Métodos similares deberán usar los mismos nombres, condiciones, orden en los argumentos, tipos de datos, valores retornados y condiciones de errores. Mantener estructuras paralelas cuando sea posible. Por ejemplo en la librería del lenguaje C existen dos funciones inconsistentes para la salida de un string como lo son las funciones Puts y Fputs. La función Puts escribe un string para salida estandar seguida por una línea nueva de caracteres y el Fputs escribe un string en un archivo especificado sin una nueva línea de caracteres. Debe evitarse este tipo de inconsistencia.

* Separar los métodos de verificación de los métodos implementables (métodos operativos).

Los métodos de verificación toman decisiones, mezclan argumentos y recojen el contexto global. Los métodos de verificación deben chequear estatutos y errores. Los métodos de verificación son a menudo aplicaciones dependientes, simples de escribir y fáciles de entender.

Los métodos implementados ejecutan operaciones específicas sin decidir. Si el método implementado encuentra un error, el solo debe de retornar un estatus y no debe tomar ninguna acción. Los métodos implementables ejecutan cálculos específicos sobre argumentos y a menudo contienen algoritmos complicados.

Los métodos implementados no accesan el contexto global, contienen valores por defectos o interruptores de flujo de control.

No debe combinarse verificación e implantación en un método simple.

Por ejemplo:

Un método para la escala de una ventana tiene asignado un factor de 2. Este es un método de verificación de factor de escala . La ventana llama a un método implementado que construye ventana en cualquier factor, en este momento dicho método tomará el valor de escala dado para construir la ventana. Si usted decide cambiar el factor de escala a otro valor por ejemplo = 1.5, debe hacer la modificación del parámetro del método de verificación sin cambiar el método implantado que actualmente esta trabajando.

* Proveer una cobertura uniforme.

Si las condiciones de entrada pueden ocurrir en varias combinaciones, entonces hay que escribir un método para cada una de las combinaciones posibles. Por ejemplo, si se escribe un método para tomar el último dentro de una lista, también debe escribir un método para tomar el primer elemento.

* Ensanchar el método tan grande como sea posible.

Trate de generalizar tipos de argumentos, precondiciones y restricciones, asumiendo como trabajará el método y el contexto en el cual el método operará. Tome acciones significativas sobre valores vacíos, valores extremos, valores fuera de rango. A menudo un método puede ser más general con un pequeño incremento en el código.

* Evite la información global.

Minimice las referencias externas. La imposición de referencia hacia un objeto global requieren contexto sobre el uso del método. A menudo la información puede ser pasada como argumentos. Otra manera consiste en almacenar información global como parte del objeto tal que otros métodos puedan acceder a esa información.

* Evite los Modos.

Funciones que cambian drásticamente su comportamiento dependiendo del contexto sobre el que actúan son duras de reusar. Trate de reemplazar a estas funciones modelos. Por ejemplo un procesador de texto requiere las operaciones insertar y reemplazar. Una aproximación es colocar un

modo para insertar o remplazar texto dependiendo del modo actual. Una mejor aproximación sería utilizar dos operaciones, insertar y reemplazar que realicen la operación sin la colocación de un modo.

Extensibilidad

Los siguientes principios incrementan la extensibilidad .

* Clases encapsuladas.

Una clase está encapsulada si su estructura interna se encuentra oculta de otras clases. Sólo los métodos implantados sobre esas clases podrán acceder a su estructura interna. Algunos compiladores son bastantes elegantes para optimizar operaciones y tienen acceso directo a la implantación pero los programadores no la tienen.

* Ocultamiento de las estructuras de datos.

No se puede exportar la estructura de datos desde un método. Las estructuras de datos interna están especificadas para un método.

Si la exportamos limitamos la flexibilidad para cambiar los algorítmos y las estructuras de datos más tarde.

* Evitar declaraciones encajonadas sobre objetos.

Las declaraciones encajonadas pueden ser usadas para probar atributos internos de un objeto, pero no deben ser utilizadas para seleccionar el comportamiento basado sobre el tipo objeto.

* Distinguir las operaciones públicas y privadas.

Las operaciones públicas son visibles fuera de la clase y tiene interfaz pública. De aquí que una operación pública es usada por otras clases, por lo cual trae como consecuencia que sea costoso cuando se desea cambiar la interfaz de una operación pública. Por eso algunas operaciones públicas deben ser cuidadosamente definidas. Las operaciones privadas son internas a la clase y son utilizadas para ayudar a implementar las operaciones públicas. Las operaciones privadas pueden borrarse o puede cambiarse su interfaz con un impacto limitado sobre otros métodos de la clase.

¿ Por qué clasificar las operaciones como públicas y privadas ?

* Los usuarios no necesitan preocuparse de los detalles internos de la clase.
* Los métodos privados dependen sobre decisiones internas de implementación.
* Los métodos privados pueden depender sobre precondiciones o estados de información creados por otros métodos en la clase. Un método privado aplicado fuera del contexto puede generar resultados incorrectos.
* Los métodos privados adicionan modularidad. Los detalles internos de un método sólo afectan métodos sobre la clase, no a otros métodos.

Similarmente, atributos y asociaciones deben ser clasificadas como públicas y privadas. Además, los atributos y asociaciones públicas pueden clasificarse como solo lectura o escritura fuera de la clase a la que pertenecen.

Robustez

Debe procurarse escribir métodos eficientes pero no la expensa de la robustez, el método no debería fallar si se escribe un parámetro apropiado. Robustez contra fallas internas pueden ser un compromiso contra la eficiencia. Robustez contra errores del usuario nunca deberían de sacrificarse.

***** Protección contra errores.

El software debería de protegerse a sí mismo contra entradas incorrectas del usuario, nunca debería causar el colapso del método. Algunos métodos que aceptan entradas del usuario deberían de validar las entradas que causan preocupación.

El diseño de métodos debe considerar dos grandes condiciones de error :

- Errores del usuario que son identificados durante el análisis.
- Reporte sobre condiciones que existen en el dominio del problema.

Ejemplo:

Una máquina automática de un taller debe procesar y reportar errores acerca de la tarjeta del scanner y las líneas de comunicación. El responsable de ese error es parte del análisis. Otra forma de errores están relacionadas con los aspectos de programación de un método. Esos errores de bajo nivel incluyen errores en el sistema operativo, tales como errores de localización de memoria, errores de entrada/salida y fallas de hardware. Sus programas debería chequear esos errores y al menos tratar de terminar la ejecución de su programa lo más elegante posible.

Tratar de que las vigilancias contra fallas de programación sea la mejor posible y dar una buena información de diagnóstico si una falla total ocurre.

Durante el desarrollo, a menudo vale la pena insertar aserciones internas dentro del código para descubrir fallas, una vez realizado el chequeo estas aserciones son removidas de la versión final. Un lenguaje fuertemente orientado por objeto provee grandes protecciones contra incompatibilidad de tipos, pero las aserciones pueden ser manualmente insertadas en cualquier lenguaje.

***** Optimizar después que el programa corra.

No optimize el programa hasta que no tenga el trabajo. A menudo los programadores gastan mucho esfuerzo tratando de mejorar porciones de código que son utilizados con muy poca frecuencia.

Usted debe medir el rendimiento de su programa antes de optimizar, esto es, estudie su aplicación para saber cuales medidas son las importantes, tales como: operaciones con los peores

tiempos , frecuencia de uso de las operaciones. Si una operación puede ser implantada en más de una manera debe de tomar en cuenta el compromiso de las alternativas con lo relativo a uso de la memoria, velocidad y simplicidad de implantación. En general la optimización del programa no debe de llegar más allá del rompimiento de sus compromisos con la extensibilidad, reusabilidad y fácil entendimiento.

Si sus métodos están encapsulados ellos pueden ser remplazados por versiones optimizadas sin afectar el resto del programa.

Validación de Argumentos

* Las operaciones externas que están disponibles para el usuario de una clase deben chequear rigurosamente sus argumentos para prevenir fallas, pero los métodos internos pueden asumir que sus argumentos son válidos por razones de eficiencia. Para los métodos públicos se debe tener más cuidado en el chequeo de la validez de sus argumentos, porque estos métodos son utilizados por usuarios externos que probablemente violen las restricciones sobre sus argumentos.
* No incluya argumentos que no puedan ser validados, por ejemplo: La famosa función scanf del Unix lee una línea de entrada en un buffer interno sin chequear el tamaño del buffer, esto ha sido explotado por los programas que escriben virus para forzar un desborde del buffer del sistema. Usted no debe utilizar operaciones cuyos argumentos no pueden ser validados.

* Evite límites predefinidos cuando utilice memoria dinámica para crear estructura de dato para una aplicación. Por lo tanto no debe de colocarse límites. Los lenguajes de orientación por objeto tienen excelentes facilidades para la manipulación de la memoria dinámica.

Programación a gran escala

La programación a gran escala se refiere a la escritura de programas complejos con un equipo de programación. El desarrollo de tales proyectos requieren del manejo de las propiedades de la Ingeniería del Software.

La siguiente guía debería ser considerada antes de iniciar un proyecto a gran escala..

* <u>No inicie prematuramente la programación.</u>

Es importante que primero complete la idea genérica del proceso antes del confrontamiento con las peculiaridades de la implementación del objetivo. Todas las metodologías de desarrollo del software enfatizan la importancia del diseño.

* <u>Mantener métodos entendibles.</u>

Un método es entendible si alguna otra persona diferente al creador del método puede entender el código. También esto es importante para el creador del método porque después de haber pasado un largo periodo de tiempo el puede volver a entender como fué que el construyó el método. lo anterior se logra creando métodos pequeños y coherentes.

* Realice métodos legibles.

Los nombres de variables deben de ser significativos para incrementar la legibilidad. Escribir unos caracteres extras con sentido lleva consigo una economía porque el tiempo empledado por otros programadores en entender los nombres de esas variables será menor. Utilice variables temporales en lugar de expresiones demasiadas complejas. No utilice la misma variable temporal para dos propósitos distintos dentro de un método.

* Use exactamente el mismo nombre como en el modelo de objeto.

La elección del nombre a utilizar dentro de un programa debe de ser exactamente el mismo que el encontrado en el modelo del objeto. Un programa puede necesitar que se introduzcan nombres adicionales por razones de implantación, esto es excelente pero la uniformidad de los nombres debe de preservarse. Esta práctica mejora el seguimiento, la documentación y la entendibilidad para el software.

* Seleccione los nombres cuidadosamente.

Asegúrese que el nombre utilizado describa a la operación, a la clase o a los atributos con solo ver la etiqueta. Siga un patrón uniforme en la colocación de los nombres. Por ejemplo usted puede seguir el patrón "verbo-objeto" para colocar los nombres a sus operaciones, esto quiere decir que si deseamos colocar un nombre a una operación que adiciona elemento, el nombre de esa operación podrá ser adicionar_elemento. No use el mismo nombre para operaciones que son semánticamente distintas.

* Use guía de Programación.

El equipo del proyecto debe utilizar las guías de programación o estándares de programación que tienen disponibles las compañías. Si esta guía no existe, el equipo se software debe crear guía que dirijan las salidas tales como los nombres de las variables de control, métodos de documentación y documentación en línea.

* Empaquetar en módulos. Grupo clase con funciones similares.

Ejemplo:

Módulo	Clase
Dibujar	línea, barras, ...etc.
Geometría	Polígonos, círculos,...etc.
Ventanas	Menos, botones, panelas, etc.

* Documentar clases y métodos.

La documentación de un método describe cual es su propósito, su función, el contexto sobre el cual actúa, entradas, salidas, asunciones y precondiciones acerca del estado del objeto. Usted debe describir el algoritmo, incluyendo el porqué este método será seleccionado.

* Publicar la especificación.

La especificación es un contrato entre el productor y los consumidores de la clase. Una vez escrita la especificación, el productor no puede romper el contrato, hacer esto afectaría al consumidor. La especificación solo contiene declaraciones. El usuario de un método debe ser capaz de usar el método, pues fue el quien indicó las especificaciones, algunos lenguajes tales como el ADA y C++ soportan la separación entre la especificación y la implantación.
Ejemplo de una especificación:

Descripción de la clase

Nombre de la clase: Círculo
Versión: 1.0
Descripción: Es una elipse cuyo eje mayor y menor son iguales.
Superclase: Elipse
Rasgos

Atributos públicos

Centro: Punto /* Localización del centro */
Radio: Real /* Radio */

Métodos públicos

Dibujar (ventana) /* Dibuja el círculo en la ventana */
Intersección_líneas (línea): Conjunto de puntos /* Encuentra la intersección de una línea con el círculo, devuelve un conjunto de puntos.

Area ( ): Real /* Cálcula el área del circulo */

Métodos Privados

Ninguno.

Descripción del Método

Método círculo: Intersección–líneas (línea:Línea) : Conjunto de puntos.

Descripción: Dado un círculo y una línea este método encuentra su intersección, retornando un conjunto de 0 - 2 puntos de intersección. Si la línea es tangente al círculo el conjunto contiene un único punto.

Entradas: Círculo: Círculo /* Círculo que será intersectado con la línea */
Línea: Línea /* Línea que va a ser entersectada con el círculo */
Salidas: Un conjunto de puntos intersectados. El conjunto puede contener 0,1 ó 2 ptos.
Efectos especiales: Ninguno.

Descripción de la Operación

Operación intersección_Línea (Línea: Línea) : Conjunto_de_puntos

Clase original: Figuras_geométricas.

Descripción: Retorna un conjunto de puntos intersección entre el objeto geométrico y una línea. El conjunto puede contener 0,1,2 puntos. Cada punto tangente solo aparece una vez.

Estatus: Operación abstracta en la clase origen.

Entradas: Figura: Figura_geométrica /* figura que debe ser intersectada con la línea */
Línea: Línea /* Línea que va a ser intersectada */

Salidas: Un conjunto de puntos de intersección, el conjunto puede contener 0 ó más puntos.

Efectos especiales: Ninguno.

Errores: -Si la figura no se intersepta se devuelve un conjunto vacío.
-Si las figuras no se interseptan se retorna un conjunto vacío.
-Si la línea es tangente al círculo, se retorna un punto tangente.
-Si el radio del círculo es cero se devuelve el punto simple si el punto se encuentra sobre la recta.

5.- Metodología para el diseño orientado por objeto.

Aunque los conceptos han sido establecidos hace algún tiempo, los primeros intentos para describir una metodología para crear diseño orientado por objeto emergen en los primeros años de la década de 1980 con Abbolt, R. J y Booch, G . estos dos autores defienden que el DOO debe de comenzar con una descripción en un lenguaje natural de la estrategia de la solución para la realización

del software de un problema del mundo real. A partir de esta descripción el diseñador puede aislar los objetos y las operaciones.

En 1983 Booch,G propone los siguientes pasos para lograr el diseño orientado por objetos.

1.- Definir el problema
2.- Desarrollar una estrategia informal para la realización del software.
3.- Formalizar la estrategia utilizando los siguientes pasos:
    - Identificar los objetos y sus atributos.
    - Identificar las operaciones que pueden aplicarse a los objetos.
    - Establecer interfaz para mostrar las relaciones entre los objetos y las operaciones.
    - Decidir los aspectos del diseño detallado que harán una descripción de la implantación para los objetos.
4.- Repetir los pasos 2, 3, hasta que se cree un diseño completo.

Los pasos 1 y 2 son los mismos pasos que se ejecutaron durante el análisis de requerimiento del software, con la metodología tradicional. El paso 3 es donde realmente comienza el diseño orientado por objeto, en este paso se identifican los objetos, operaciones y sus interrelaciones de forma que pueda derivarse un diseño.

La orientación por objeto enfátiza la importancia de la identificación precisa de los objetos y sus propiedades, los que han de ser manipulados por un programa. Sín esta identificación cuidadosa es casi imposible ser preciso en la determinación de las operaciones que han de ejecutarse sobre los objetos.

Metodología de Lorensen para DOO.

El método anteriormente presentado está orientado hacia el desarrollo de software en lenguajes de programación tales como el ADA. El método no enfoca explícitamente varios de los conceptos importantes de la orientación por objeto como lo son la herencia y los mensajes que hacen el DOO poderoso.

En 1986 Willian Lorensen desarrolla una metodología para el diseño orientado por objeto a partir del desarrollo de software en lenguajes de programación que soportan directamente la abstracción, herencia, mensajes y todos los otros conceptos del DOO.

El método propuesto por Lorensen tiene como objetivo primario el de definir y caracterizar las abstracciones, de forma que se tengan una definición de todos los objetos importantes, métodos (operaciones) y mensajes.

Lorensen propone la siguiente metodología para el diseño orientado por objeto.

1.- Identificar las abstracciones de datos para cada subsistema.

Estas abstracciones de datos son las clases del sistema. Trabajando con los documentos de requerimientos, el proceso de abstracciones debe de ejecutarse de la forma más descendente posible, aunque muchas veces las abstracciones se mencionan explícitamente en los requerimientos. Frecuentemente las clases se corresponden con objetos físicos del sistema que se está modelando. Si este no es el caso es útil hacer uso de las analogías obtenidas de las experiencias del diseñador. Este es el paso más difícil en el proceso del diseño y la selección de estas abstracciones influyen en la arquitectura global del sistema.

2.- Identificar los atributos de cada abstracción

Los atributos se convierten en variables de las instancias (métodos para manipular los datos) para cada clase. Muchas veces si las clases se corresponden con objetos físicos, son obvias las variables requeridas para las instancias. Pueden necesitarse otras variables de instancias para responder a preguntas sobre otros objetos del sistema.

3.- Identificar las operaciones de cada abstracción.

Las operaciones son los métodos (procedimientos) de cada clase. Algunos métodos acceden y actualizan variables de instancias, mientras que otros ejecutan operaciones propias de las clases. A este nivel no se especifican los detalles de la implantación del métodos, sólo se especifican las funcionalidades. Si la nueva abstracción hereda de otra clase, se debe de inspeccionar los métodos de esa clase. Debe dejarse el diseño interno de los métodos hasta la etapa del diseño detallado.

4.- Identificar las comunicaciones entre los objetos

En este paso se definen los mensajes que los objetos mandan a los otros. Aquí se deben definir una correspondencia entre los métodos y los mensajes que llaman a los métodos.

5.- Probar escenarios.

Los escenarios, que consisten en mensajes a los objetos, prueban la habilidad del diseño para cumplir con los requerimientos del sistema.

6.- Aplicar la herencia donde sea necesario.

Si el proceso de abstracción de datos en el paso 1 se ejecuta en forma descendente, introducir allí la herencia; sin embargo, si las abstracciones se crea en forma ascendente (frecuentemente debido a que los requerimientos denominan directamente a las abstracciones), se debe aplicar aquí directamente la herencia antes de entrar a otro nivel de abstracción. El objetivo reutilizar tanto como se pueda los datos y/o los métodos que hayan sido diseñado.

En este paso emergen frecuentemente datos comunes, operaciones variables comunes de la instancia y métodos que pueden combinarse en una nueva clase. Esta clase puede tener o no significado

como objeto por si mismo. Si su único propósito es coleccionar variables y métodos de instancias comunes, se le llama una clase abstracta.

El diseñador debe de repetir estos pasos en cada nivel de abstracción. A través de sucesivo refinamiento del diseño la visión del diseñador cambia dependiendo de la necesidad de cada momento. Cada nivel de abstracción se implanta en un nivel inferior, hasta que se alcance un punto en el que la abstracción se corresponda con un elemento primitivo del diseño.

Ejemplo de DOO aplicando la metodología anterior:

Se desea desarrollar una herramienta CAD para ver y manipular distintas primitivas en dos dimensiones esta herramienta CAD tiene las siguientes requerimientos descritos en forma narrativa.

La herramienta debe permitir al usuario crear y manipular polígonos de 2 dimensiones, cónicas y giros en un terminal gráfico a color con un dispositivo de entrada como el ratón, el usuario debe poder mover, rotar, escalar y colorear las cónicas.

Debemos saber que una cónica es una curva implícita de segundo orden de la forma

$$Ax^2 + Bxy + C\ y^2 + Dx + Ey + F = 0$$

Las cónicas son: Círculos, elipses, hipérbolas y parábolas al aplicar el método anterior se debe de recordar que el método se repite para

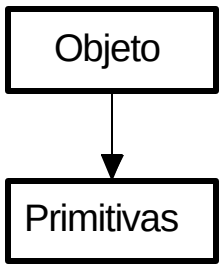
cada nivel de abstracción hasta llegar a un punto donde el nivel de abstracción se corresponde con un elemento primitivo del diseño.

Paso 1

1.- Identificar las abstracciones de datos de cada subsistema.

Las abstracciones para sistemas geométricos son particularmente fáciles de derivar. Los requerimientos dicen que la herramienta tendrá tres primitivas diferentes (polígonos de 2D, cónicas y giros).

Nuestra primera jerarquía de clase es como la siguiente:



La clase más alta del diagrama llamada objeto, que ejecuta funciones que son adecuadas a todas las clases. La clase primitiva es una clase abstracta que actúa como un titular del lugar para los datos y métodos que puedan ser usados por las clases más bajas de la jerarquía.

2.- Identificar los atributos de cada abstracción

Un atributo de todas las primitivas es obvio, el color. Si nosotros pensamos en objetos geométricos en general, puede venirnos a la mente otros atributos tales como posición, orientación, textura, escala, opacidad, etc. Sabemos que los requerimientos del usuario son mover, escalar, rotar las primitivas, por lo que es adecuado utilizar variables de instancias para posición, factores de escala y orientación.

3.- Identificar operaciones de cada abstracción

Los documentos de los requerimientos explicitan frecuentemente las operaciones mencionadas. Aquí nuestros requerimientos para la herramienta especifican (mover, escalar, rotar), también necesitaremos métodos para acceder para acceder a las variables de instancias.

Método u operaciones para las primitivas

Método	**Significado**
Crear-Primitiva	Crea una primitiva
Asignar-Posición	Asigna posición x, y
Obtener-Posición	Obtiene la posición x, y
Añadir_Posición	Incrementa la posición x, y
Obtener_Orientación	Obtiene el ángulo de rotación
Añadir-Orientación	Incrementa el ángulo de rotación
Asignar-Escala	Asigna el factor de escala x, y
Obtener-Escala	Obtiene la escala x, y
Añadir-Escala	Incrementa escala x, y
Asigna-Color	Asigna el color
Obtener-Color	Obtiene el color

4.- Identificar la comunicación entre los objetos

Ahora se puede especificar el protocolo de comunicación entre objetos asociando mensajes con los métodos u operaciones definidos anteriormente.

Mensaje	**Significado**
Posición =	Asignar posición
Posición ?	Obtener posición

Posición +	Añadir posición
Orientación =	Asignar orientación
Orientación ?	Obtener orientación
Orientación +	Añadir orientación
Escala =	Asignar escala
Escala ?	Obtener escala
Escala +	Añadir escala
Color =	Asignar color
Color ?	Obtener color

5.- Probar el diseño con escenario

Se hace coincidir los requerimientos con escenarios sencillos, que muestren cómo se cumplirán cada uno de los requerimientos.

Crear una primitiva:
    Primitiva nueva! nombre = una primitiva
Mover:
    una primitiva posición = (1,2)
Rotar:
    una primitiva orientación = 30
Escalar:
    una primitiva escala (10,1)
Asignar color
    una primitiva (1,0,0)

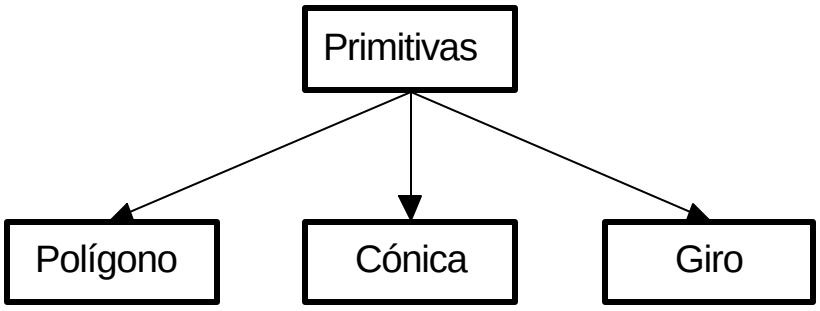
6.- Aplicar la herencia donde sea apropiado.

Puesto que hasta ahora sólo tenemos un nivel de herencia, este paso no es aplicable en este momento.

Paso 2

1.- Identificar las abstracciones de datos para cada subsistema.

Los polígonos, cónicas y giros son tipos de primitivas. La jerarquía de este nivel de abstracción es la siguiente:



2.- Identificar los atributos de cada abstracción.

Nos concentraremos solo en las cónicas. Los seis coeficientes que especifican una cónica se convierten en las variables de instancia.

3.- Identificar las operaciones de cada abstracción.

Añadiremos nuevas operaciones al conjunto y recuperamos los seis coeficientes.

Método u operación	**Significado**
-Asignar coeficientes	Asigna coeficientes
-Obtener coeficientes	Obtiene los coeficientes
-Crear cónica	Crea una cónica

4.- Identificar la comunicación entre los objetos.

-Nueva!	Crea_cónica
-Coeficiente =	Asigna coeficientes
-Coeficientes ?	Obtener coeficientes

5.- Probar el diseño con escenario

Crear una cónica

conicanueva! Nombre = Unacónica

Modificar una cónica

unacónica coeficiente = (1,0,3,2,5,3,0)

Mover

unacónica posición = (1,2)

Rotar

unacónica orientación = 30

Escalar

unacónica escalar = (10,1)

Asignar color

unacónica = (1,0,0)

6.- Aplicar la herencia donde sea necesario.

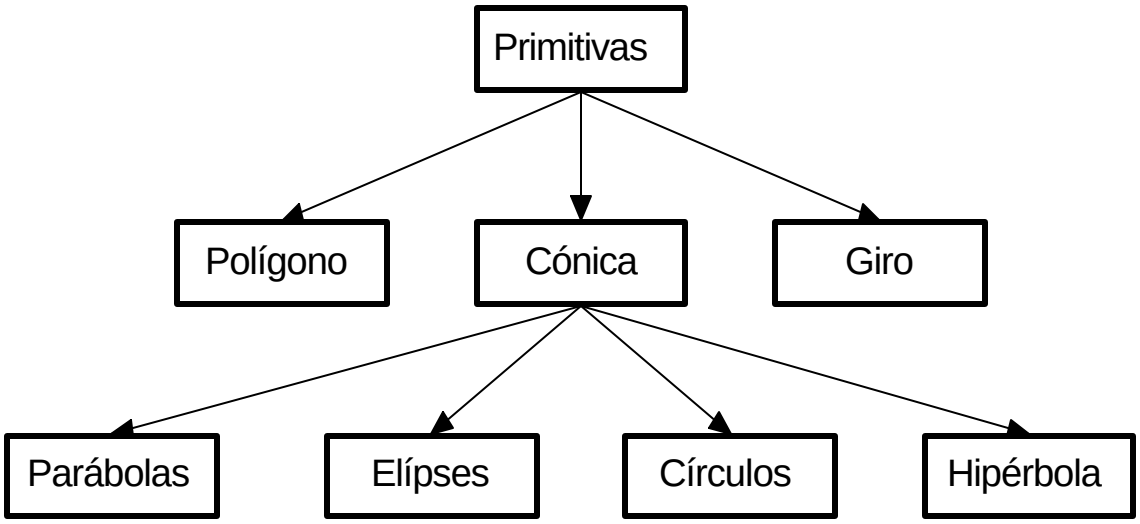
Ninguna herencia es apropiada,puesto que estamos procediendo descendentemente.

Paso 3

Por ahora un paso más en el diseño completará el proceso.

1.- Identificar las abstracciones de datos para cada subsistema.

Seis coeficientes especifican una cónica, pero esto no es una forma conveniente para especificar la forma para los usuarios, sin embargo observamos que los círculos y elipses tienen parámetros que un usuario puede especificar fácilmente ya sea numéricamente o gráficamente. En este nivel podemos añadir también hipérbolas y parábolas, por lo que introducimos un nuevo nivel de abstracción que es el siguiente:



2.- Identificar los atributos de cada abstracción

En este caso lo haremos solo para el círculo. Un atributo adicional del círculo es el radio.

3.- Identificar las operaciones para cada abstracción.

Asignar radio Asignar el radio del círculo

Obtener radio Obtiene el radio del círculo

El círculo es una restricción de una cónica, por lo que damos un método de creación que mantiene la restricción.

Crear_círculo Crea un círculo

Puesto que un círculo hereda métodos de una cónica, debemos examinar los métodos de la cónicas para ver si existe alguno que podamos suprimir. Por ejemplo: en la ecuación de una cónica, el círculo tiene algunos de los coeficientes a cero. Si el círculo hereda el método asigna_coeficiente de una cónica, podemos crear lo que pensamos que es un círculo, pero realmente es una cónica general. Necesitamos suprimir el método de Asignar_coeficiente de la cónica. Sin embargo, Obtener_coeficiente de la cónica permanece aún válido.

Asignar_coeficientes Asigna coeficientes de la cónica para un círculo.

4.- Identificar la comunicación entre objetos.

Nueva	Crear círculo
Radio =	Asigna el radio
Radio ?	Obtiene el radio
Asignar_coeficientes	Asignar coeficientes

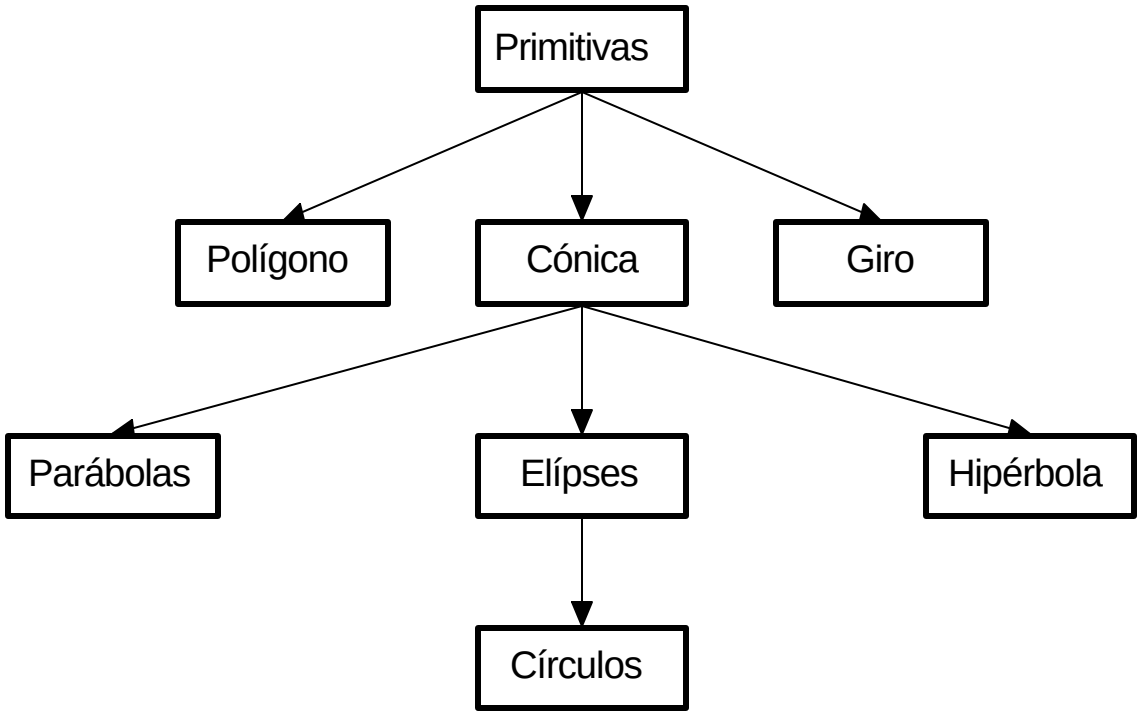
5.- Probar el diseño con escenario.

Crear un círculo
Círculo nueva! nombre = uncírculo
Modifica una cónica
Uncírculo radio = 10
Mover
Uncírculo posición = (1,2)
Rotar
Uncírculo orientación = 30

Escalar
Uncírculo escala = (10,1)
Poner_color
Uncírculocolor = (1,0,0)

6.- Aplicar la herencia donde sea necesario.

Después de repetir los pasos para una elipse se observa que un círculo es una especialización de una elipse. Aunque esto parezca obvio en este ejemplo, muchas veces en un diseño, es solo después de haber diseñado algunas clases cuando nos damos cuenta de tal realización. Al aplicar la herencia obtenemos la siguiente abstracción.



Ejercicio:

Usted es responsable del desarrollo de un sistema de correo electrónico que ha de ser implantado sobre una red de Pc. El sistema correo-eletrónico facilitara al usuario la creación de cartas que han de enviarse a otros usuarios o distribuirse según una lista específica de direcciones.Las cartas pueden ser leídas, copiadas, almacenadas, etc. El sistema de correo electrónico hará uso de las capacidades existentes de procesamiento de palabras para crear las cartas. Usando esta descripción como punto de partida, aplicar la técnica de DOO.

Proceso de análisis por objeto.

Esta es la metodología que seguiremos durante el curso debido a que maneja todos los conceptos de la orientación por objeto y es una metodología bastante fácil de aplicar.

Pasos a seguir para realizar el análisis por objeto:

1.- Identificar todos los objetos del sistema.

2.- Asociar las operaciones con los objetos.

3.- Identificar el tipo o categoría de cada objeto, por medio de la clasificación de su papel en el sistema, identificar clases y subclases.

4.- Identificar cada operación que se lleva a cabo en el sistema.

Para los pasos:

1.- Atender a nombres y frases para definir cuáles son los objetos.

3.- Cuando la forma de manejar o tratar (operaciones) y los objetos son similares, se colocan en grupos que se llaman clases y subclases de objetos. Generalmente, las operaciones asociadas a un miembro de la clase se aplican para los otros miembros de la clase; por lo tanto se reduce la complejidad al analizar el sistema.

Una vez realizados los pasos 1-4 se sigue con la abstracción de datos que consiste en una especificación en detalles de los objetos abstractos del sistema.
Ejemplo:

Un sistema de concordancia permite realizar un estudio de frecuencia para palabras en un documento, a través de un archivo que se llama Concordancia. El sistema lleva cuenta de las veces que aparece cada palabra en el documento y opcionalmente indica en que parte del documento aparece la palabra.

El sistema analiza línea por línea y extrae cada palabra de esa línea y la edita. Si la palabra no está en el archivo de concordancia la agrega; si ya está incluida, entonces incrementa el contador de palabras. Se mantiene un registro de las líneas donde cada palabra fue encontrada. Cuando el documento ha sido completamente analizado el usuario selecciona si quiere los resultados por pantalla y/o por impresora.

Aplicando el proceso de análisis por objeto se tendrá lo siguiente:

1.- Identificar todos los objetos del sistema

- Documento
- Palabra
- Registros_líneas
- Usuario
- Pantalla
- Archivo_concordancia
- Línea de texto
- Contador_Palabras
- Impresora
- Opción salida

2.- Identificar las operaciones asociadas con los objetos

Nombre del objeto	**Operaciones Asociadas**
Palabra	Extraer palabra
	Si_esta

	No_esta
Documento	Almacenar_documento
	Eliminar_documento
	Leer_línea_Documento
Contador_Palabra	Incrementar_contador
Pantalla	Escribir_resultado
Archivo_concordancia	Buscar_palabra
	Almacenar_palabra
	Almacenar_frecuencia
	Almacenar_registro_línea
	Eliminar_archivo
Línea_Texto	Extraer_palabra
	Editar_línea
Registro_línea	Almacenar_dirección
Opción_salida	Leer_opción
Impresora	Escribir_resultado

3.- Identificar clases y subclases.

Opción salida (clase)

-Pantalla
(subclase)
-Impresora

-Documento (clase)

-Línea_texto
(subclase)
-Palabra

Archivo_concordancia (clase)

-Contador_palabra
-Registro_línea (subclase)
-palabra

4.- Identificar cada operación que lleva a cabo el sistema
-Analizar línea por línea
-Extraer palabras
-Agregar palabras
-Mantener registro de direcciones
-Desplegar resultados

VENTAJAS Y DESVENTAJAS DE LA PROGRAMACION ORIENTADA POR OBJETOS

VENTAJAS

1.- El ocultamiento de la información y la abstracción de dato incrementan la confiabilidad y ayudan a desaclopar los procedimientos y la especificación representacional de la implantación.

2.- El encadenamiento dinámico incrementa la flexibilidad por permitir la adición de nuevas clases de objetos (tipos de datos) sin tener que modificar el código existente.

3.- La herencia unida al encadenamiento dinámico permite que el código sea reusable. Esto incrementa la productividad del programador.

DESVENTAJAS

Los lenguajes de orientación por objetos tienen pocas características que son consideradas desventajas.

1.- Costo de tiempo de ejecución del encadenamiento dinámico (el envío de mensaje toma más tiempo que el llamado de una función). Algunos estudios muestran que una buena implantación con mensajes tarda 1.75 veces más que la ejecución de una función estándar.

2.- La implantación con lenguajes de orientación por objetos es más compleja que la de un lenguaje procedural.

3.- El programador debe leer con frecuencia extensas librerías de clases.

BIBLIOGRAFIA

L. Cardelli, P. Wegner On Understanding Types, Data Abstraction, and Polymorphism . ACM Computing Surveys. 17 (4) : Dec. 1985

W. Kim, F. H. Lochovsky Object-Oriented Concepts, Database, and Applications .
ACM PRESS New York, 1989

W. Lorensen Object - Oriented Modeling and Design. Prentice Hall, Cliffs, Jersey 1991

P.P. Chen The Entity Relationship model ACM Transaction Database Systems, vol 1 no 1 1976

W. Kent Limitations of Record Based Information Models ACM Transactions on Data Base System, Vol 4, no 1 1979

J. M. Smith, D. C. Smith Database Abstration: Aggregación and Generalization ACM Transaction Data Base Systems vol 2, no 10 1986

J. Banerjee, W. Query in Object Oriented Database Acm Transaction on Data Base System

C. Delobel, M. Kifer, Y. Masunaga Deductive and Object-Oriented Database Proceeding Springer-Verlarg Munich Germany 1991

Kamram Parsay, Mark Chinell Inteligent Database Jhon Wiley 1989.

Golgerg A. Smalltalk 80 El lenguaje y su Implantación Addison Wesley 1983

Ramez Elmari , Shankant Navathe Fundamentals of Database Systems Benjamin Cuming 1989.

Alfred V. Aho, Jeffrey Ullman Compiladores Principios Técnicas y Herramientas Addison Wesley 1990

APENDICE