UDS

Mi Universidad

LIBRO

Nombre de la materi: PROGRAMACION

Nombre de la Licenciatura: ISC

Cuatrimestre: 9°

Periódo

Marco Estratégico de Referencia

Antecedentes históricos

Nuestra Universidad tiene sus antecedentes de formación en el año de 1979 con el inicio de actividades de la normal de educadoras "Edgar Robledo Santiago", que en su momento marcó un nuevo rumbo para la educación de Comitán y del estado de Chiapas. Nuestra escuela fue fundada por el Profesor Manuel Albores Salazar con la idea de traer educación a Comitán, ya que esto representaba una forma de apoyar a muchas familias de la región para que siguieran estudiando.

En el año 1984 inicia actividades el CBTiS Moctezuma Ilhuicamina, que fue el primer bachillerato tecnológico particular del estado de Chiapas, manteniendo con esto la visión en grande de traer educación a nuestro municipio, esta institución fue creada para que la gente que trabajaba por la mañana tuviera la opción de estudiar por las tardes.

La Maestra Martha Ruth Alcázar Mellanes es la madre de los tres integrantes de la familia Albores Alcázar que se fueron integrando poco a poco a la escuela formada por su padre, el Profesor Manuel Albores Salazar; Víctor Manuel Albores Alcázar en julio de 1996 como chofer de transporte escolar, Karla Fabiola Albores Alcázar se integró en la docencia en 1998, Martha Patricia Albores Alcázar en el departamento de cobranza en 1999.

En el año 2002, Víctor Manuel Albores Alcázar formó el Grupo Educativo Albores Alcázar S.C. para darle un nuevo rumbo y sentido empresarial al negocio familiar y en el año 2004 funda la Universidad Del Sureste.

La formación de nuestra Universidad se da principalmente porque en Comitán y en toda la región no existía una verdadera oferta Educativa, por lo que se veía urgente la creación de una institución de Educación superior, pero que estuviera a la altura de las exigencias de los

jóvenes que tenían intención de seguir estudiando o de los profesionistas para seguir preparándose a través de estudios de posgrado.

Nuestra Universidad inició sus actividades el 18 de agosto del 2004 en las instalaciones de la 4ª avenida oriente sur no. 24, con la licenciatura en Puericultura, contando con dos grupos de cuarenta alumnos cada uno. En el año 2005 nos trasladamos a nuestras propias instalaciones en la carretera Comitán – Tzimol km. 57 donde actualmente se encuentra el campus Comitán y el corporativo UDS, este último, es el encargado de estandarizar y controlar todos los procesos operativos y educativos de los diferentes campus, así como de crear los diferentes planes estratégicos de expansión de la marca.

Misión

Satisfacer la necesidad de Educación que promueva el espíritu emprendedor, aplicando altos estándares de calidad académica, que propicien el desarrollo de nuestros alumnos, Profesores, colaboradores y la sociedad, a través de la incorporación de tecnologías en el proceso de enseñanza-aprendizaje.

Visión

Ser la mejor oferta académica en cada región de influencia, y a través de nuestra plataforma virtual tener una cobertura global, con un crecimiento sostenible y las ofertas académicas innovadoras con pertinencia para la sociedad.

Valores

- Disciplina
- Honestidad
- Equidad
- Libertad

Escudo

El escudo del Grupo Educativo Albores Alcázar S.C. está constituido por tres líneas curvas que nacen de izquierda a derecha formando los escalones al éxito. En la parte superior está situado un cuadro motivo de la abstracción de la forma de un libro abierto.

Eslogan

"Mi Universidad"

ALBORES

Es nuestra mascota, un Jaguar. Su piel es negra y se distingue por ser líder, trabaja en equipo y obtiene lo que desea. El ímpetu, extremo valor y fortaleza son los rasgos que distinguen.

PROGRAMACION

OBJETIVO:

La programación, acortada como programación, es el proceso de diseñar, codificar, depurar y mantener el código fuente de programas computacionales. El propósito de la programación es crear programas que exhiban un comportamiento deseado.

INDICE

4. Objetivo(s) General(es) de la Asignatura:

OBJETIVO GENERAL:

Aplicar los conceptos avanzados de la programación orientada a objetos.

UNIDAD I

CONCEPTOS BÁSICOS

1.1.- ¿Que es la programación orientada a objetos?.
1.2.- Importancia de la programación orientada a objetos (POO).
1.3.- Clases y objetos: un primer contacto.
1.4.- Objetos.
1.5.- Características de la programación orientada a objetos.

UNIDAD II

INTRODUCCIÓN A LAS CLASES

2.1.- Diagramas de clases y objetos.
2.2.- Clases.
2.3.- Técnicas de creación e inicialización de objetos.
2.4.- Introducción a la herencia.
2.5.- Reglas para la construcción de clases.
2.6.- Introducción a las funciones amigas.

UNIDAD III

SOBRECARGA DE FUNCIONES Y OPERADORES

3.1.- Funciones.
3.2.- Operadores.
3.3.- Clases abstractas y herencia.
3.4.- Abstracción de generalización y especialización de clases.
3.5.- Clases abstractas.
3.6.- Herencia.

UNIDAD IV

POLIMORFISMO

4.1.- Ligadura.
4.2.- Funciones virtuales.
4.3.- Polimorfismo.

6.-Actividades de aprendizaje

Frente al docente:

- Retroalimentación de conceptos, ejemplos.
- Aplicaciones y proyecto.
- Realización de ejercicios en clases.

Independientes:

- Elaboración de ensayo.
- Elaboración de apuntes.
- Lectura obligatoria.

Criterios y procedimientos de evaluación y acreditación:

Trabajos escritos	10%
Actividades web escolar	20%
Actividades áulicas	20%
Examen	50%
Total	100%
Escala de calificaciones	7-10
Mínima aprobatoria	7

Introducción

Un autómata es cualquier mecanismo capaz de realizar un trabajo de forma autónoma (un reloj, una caja de música, la cisterna del WC, un radiador con termostato).

Todos estos aparatos tienen en común que, una vez conectados, pueden realizar su función sin más intervención externa. También comparten el hecho de que son bastante simples. Unos autómatas más flexibles serían un organillo, un video, una lavadora, ya que al menos su repertorio de acciones posibles es más variado. El ejemplo del organillo es revelador ya que en él aparecen las mismas fases que en el desarollo de un programa: una pieza de música es "diseñada" por un composistor, codificada en un soporte físico y ejecutada por una máquina.

En estor términos, un ordenador es un autómata de cálculo gobernado por un programa, de tal modo que diferentes programas harán trabajar al ordenador de distinta forma. Un programa es la codificación de un algoritmo, y un algoritmo es la descripción precisa de una sucesión de instrucciones que permiten llevar a cabo un trabajo en un número finito de pasos.

Así, un ordenador es probablemente el más flexible de los autómatas, ya que la tarea a ejecutar puede ser descrita por cualquier algoritmo que el usuario esté dispuesto a codificar.

¿Qué es la programación orientada a objetos?

Lo que pretendemos con este capítulo es que usted entienda los conceptos que aquí se le van a exponer, no importa que no entienda cómo llevarlos a la práctica, o dicho de un modo más técnico, cómo implementarlos en un lenguaje concreto (C, Java, Visual Basic, FoxPro, etc.). De esto ya nos encargaremos más adelante. El concepto de *Sistema de Programación Orientado al Objeto* -Object Oriented Programming System (OOPS)-, y que nosotros llamamos OOP, agrupa un conjunto de técnicas que nos permiten desarrollar y mantener mucho más fácilmente programas de una gran complejidad. Veamos ahora cuales son los conceptos relacionados con la OOP.

Conceptos básicos

En este capítulo veremos cuáles son los principales conceptos relacionados con la OOP. Estudiaremos los conceptos de *Clase, Objeto, Herencia, Encapsulación* y *Polimorfismo*. Estas son las ideas más básicas que todo aquel que trabaja en OOP debe comprender y manejar constantemente; es por lo tanto de suma importancia que los entienda claramente. De todos modos, no se preocupe si al finalizar el presente capítulo, no tiene una idea clara de todos o algunos de ellos: con el tiempo los irá asentando y se ira familiarizando con ellos, especialmente cuando comience a trabajar creando sus propias clases y jerarquías de clases.

Importancia de la programación orientada a objetos (POO)

La importancia

• Facilita la creación de software de calidad: potencia en mantenimiento, la extensión y la reutilización.

• Basada en el mono de pensar del hombre y en el modo de operar de la máquina.

• El elemento básico no es la función (programación estructurada), si no un ente denominado objeto.

Pensar en términos de cómo se piensa objetos es muy parecido a en objetos como lo haríamos en la vida real. Por ejemplo vamos a pensar en un coche para tratar de modernizarlo en un esquema de poo. Diríamos que el coche es un elemento principal que tiene una serie de características, como podría ser el color, el modelo o la marca.

El objeto es algo estudio real o nombre imaginario. Lenguaje natural características automóvil funciones

• Tipo

• Color

• Cilindraje

• Modelo

• Marca movilidad lenguaje técnico nombre atributos o propiedades métodos Composición de un objeto

• Tiempo de vida: duración de un objeto en un programa. los objetos se crean mediante la instalación y dejan de existir cuando son destruidos.

• Estado: definido por sus atributos.

• Comportamiento: definida por sus métodos.
Las clases

• Abstracciones que representan a un conjunto de objetos con un comportamiento e interfaz común.

• Plantilla para creación de objetos.

• Cuando se crea un objeto (instalación se ha de especificar de qué clase es el objeto instanciado, para que el compilador comprenda sus características.

Estructura de clase atributos: variables que representan el estado de los objetos. Métodos: funciones mediante cuales se representa el comportamiento de los objetos. Estos métodos modifican los valores de los atributos y representan las capacidades del objeto (servicios)

Clases y objetos: un primer contacto

Definición de Clase

Una *clase*, es simplemente una abstracción que hacemos de nuestra experiencia sensible. El ser humano tiende a agrupar seres o cosas -objetos- con características similares en grupos -clases-. Así, aun cuando existen por ejemplo multitud de vasos diferentes, podemos reconocer un vaso en cuanto lo vemos, incluso aun cuando ese modelo concreto de vaso no lo hayamos visto nunca. El concepto de *vaso* es una abstracción de nuestra experiencia sensible. Quizás el ejemplo más claro para exponer esto lo tengamos en las taxonomías; los biólogos han dividido a todo ser (vivo o inerte) sobre la tierra en distintas clases. Tomemos como ejemplo una pequeña porción del inmenso árbol taxonómico: Ellos, llaman a cada una de estas parcelas *reino*, *tipo, clase, especie, orden, familia, género*, etc.; sin embargo, nosotros a

todas las llamaremos del mismo modo: *clase*. Así, hablaremos de la clase animal, clase vegetal y clase mineral, o de la clase félidos y de las clases leo (león) y tigris (tigre). Cada clase posee unas cualidades que la diferencian de las otras. Así, por ejemplo, los vegetales se diferencian de los minerales -entre otras muchas cosas- en que los primeros son seres vivos y los minerales no. De los animales se diferencian en que las plantas son capaces de sintetizar clorofila a partir de la luz solar y los animales no. Como vemos, el ser humano tiende, de un modo natural a clasificar los objetos del mundo que le rodean en clases; son definiciones estructuralistas de la naturaleza al estilo de la escuela francesa de Saussure. Prosigamos con nuestro ejemplo taxonómico y bajemos un poco en este árbol de clases. Situémonos en la clase *felinos* (felis), aquí tenemos varias subclases (géneros en palabras de los biólogos): león, tigre, pantera, gato, etc. cada una de estas subclases, tienen características comunes (por ello los identificamos a todos ellos como felinos) y características diferenciadoras (por ello distinguimos a un león de una pantera), sin embargo, ni el león ni la pantera en abstracto existen, existen leones y panteras particulares, pero hemos realizado una abstracción de esos rasgos comunes a todos los elementos de una clase, para llegar al concepto de león, o de pantera, o de felino.

La *clase león* se diferencia de la *clase pantera* en el color de la piel, y comparte ciertos atributos con el resto de los felinos -uñas retráctiles por ejemplo- que lo diferencian del resto de los animales. Pero la clase león, también hereda de las clases superiores ciertas cualidades: columna vertebral (de la clase vertebrados) y es alimentado en su infancia por leche materna (de la clase mamíferos). Vemos cómo las clases superiores son más generales que las inferiores y cómo, al ir bajando por este árbol, vamos definiendo cada vez más (dotando de más cualidades) a las nuevas clases. Hay cualidades que ciertas clases comparten con otras, pero no son exactamente iguales en las dos clases. Por ejemplo, la clase hombre, también deriva de la clase vertebrado, por lo que ambos poseen columna vertebral, sin embrago, mientras que en la clase hombre se halla en posición vertical, en la clase león la columna vertebral está en posición horizontal. En OOP existe otro concepto muy importante asociado al de clase, el de "*clase abstracta*". Una clase abstracta es aquella que construimos para derivar de ella otras clases, pero de la que no se puede instanciar. Por ejemplo, la clase mamífero, no existe como tal en la naturaleza, no existe ningún ser que sea tan solo mamífero (no hay ninguna instanciación directa de esa clase), existen humanos, gatos, conejos, etc. Todos ellos son mamíferos, pero no existe un animal que sea solo mamífero. Del mismo modo, la clase que se halla al inicio de la jerarquía de clases, normalmente es creada sólo para que contenga aquellos datos y métodos comunes a todas las clases que de ella derivan: Son clases abstractas.

En árboles complejos de jerarquías de clases, suele haber más de una clase abstracta. Este es un concepto muy importante: el de "*clase abstracta*". Como hemos dicho, una *clase abstracta* es aquella que construimos para derivar de ella otras clases, pero de la que no se puede instanciar. Por ejemplo, la clase mamífero, no existe como tal en la naturaleza, no existe ningún ser que sea tan solo mamífero (no hay ninguna instanciación directa de esa clase), existen humanos, gatos, conejos, etc. Todos ellos son mamíferos, pero no existe un animal que sea solo mamífero. Por último, adelantemos algo sobre el concepto de objeto. El león, como hemos apuntado antes, no existe, igual que no existe el hombre; existen leones en los circos, en los zoológicos y, según tenemos entendido, aún queda alguno en la sabana africana. También existen hombres, como usted, que está leyendo este libro (hombre en un sentido neutro, ya sea de la subclase mujer o varón), o cada uno de los que nos encontramos a diario en todas partes. Todos estos hombres comparten las características de la clase hombre, pero son diferentes entre sí, en estatura, pigmentación de la piel, color de ojos, complexión, etc. A cada uno de los hombres particulares los llamamos "objetos de la clase hombre". Decimos que son objetos de tipo hombre o que pertenecen a la clase hombre. Más técnicamente, *José Luis Aranguren* o *Leonardo da Vinci* son instanciaciones de la clase hombre; instanciar un objeto de una clase es crear un nuevo elemento de esa clase, cada niño que nace es una nueva instanciación a la clase hombre.

Objetos

Definición de Objeto

Según el Diccionario del Uso del Español de María Moliner (Ed. Gredos, 1983), en la tercera acepción del término *objeto* podemos leer: "Con respecto a una acción, una operación mental, un sentimiento, etc., cosa de cualquier clase, material o espiritual, corpórea o incorpórea, real o imaginaria, abstracta o concreta, a la cual se dirigen o sobre la que se ejercen." No se asuste, nuestra definición de objeto, como podrá comprobar es mucho más fácil. En OOP, un objeto es un **conjunto de datos y métodos**; como imaginamos que se habrá quedado igual, le vamos a dar más pistas. Los datos son lo que antes hemos llamado características o atributos, los métodos son los comportamientos que pueden realizar. Lo importante de un sistema OOP es que ambos, datos y métodos están tan intrínsecamente ligados, que forman una misma unidad conceptual y operacional. En OOP, no se pueden desligar los datos de los métodos de un objeto. Así es como ocurre en el mundo real. Vamos ahora a dar una serie de ejemplos en los que nos iremos acercando paulatinamente a los objetos informáticos. Los últimos ejemplos son para aquellos que ya conocen Java y/o C; sin embargo, estos ejemplos que exigen conocimientos informáticos, no son imprescindibles para entender plenamente el concepto de clase y el de objeto. Observe que aunque los datos y los métodos se han enumerado verticalmente, no existe ninguna correspondencia entre un dato y el método que aparece a su derecha, es una simple enunciación.

Ejemplo 1

Tomemos la clase león de la que hablamos antes y veamos cuales serían algunos de sus datos y de sus métodos.

Color Desplazarse
Tamaño Masticar
Peso Digerir
Uñas retráctiles Respirar
Colmillos Secretar hormonas
Cuadrúpedo Parpadear

Ejemplo 2

Nuestros objetos (los informáticos), como hemos comentado antes, se parecen mucho a los del mundo real, al igual que estos, poseen propiedades (datos) y comportamientos (métodos). Cojamos para nuestro ejemplo un cassette. Veamos cómo lo definiríamos al estilo de OOP.

Peso Rebobinar la cinta
Dimensiones Avanzar la cinta
Color Grabar
Potencia Reproducir
etc. etc.

Ejemplo 3

Pongamos otro ejemplo algo más próximo a los objetos que se suelen tratar en informática: un recuadro en la pantalla. El recuadro pertenecería a una clase a la llamaremos **marco**. Veamos sus datos y sus métodos.

Coordenada superior izquierda Mostrarse
Coordenada inferior derecha Ocultarse
Tipo de línea Cambiar de posición
Color de la línea
Color del relleno
etc. etc.

Ejemplo 4

Vayamos con otro ejemplo más. Este es para aquellos que ya tienen conocimientos de informática, y en especial de lenguaje C o Java. Una clase es un nuevo tipo de dato y un objeto cada una de las asignaciones que hacemos a ese tipo de dato.

int nEdad = 30;

Hemos creado un objeto que se llama nEdad y pertenece a la clase *integer* (entero).

Para crear un objeto de la clase marco lo haríamos de forma muy parecida: llamando a la función creadora de la clase marco:
Marco oCajaMsg := new Marco();

No se preocupe si esto no lo ve claro ahora, esto es algo que veremos con detenimiento más adelante. Para crear un objeto de tipo Marco, previamente hemos tenido que crear este nuevo tipo de dato. Aun cuando ni en Java ni en C un integer sea internamente un objeto, bien podría serlo; de hecho no hay ninguna diferencia conceptual entre la clase integer y la clase Marco. La primera es una de las clases que vienen con el lenguaje y la segunda la creamos nosotros. Pero del mismo modo a como nEdad es una instancia de la clase integer, oCajaMsg es una instancia de la clase Marco (De hecho, en Java existe la clase Integer, que si es una clase "de verdad" (tanto a nivel externo como interno) y de la que se instancias objetos de tipo integer.). Como se ve, básicamente la OOP lo que nos permite es ampliar los tipos de datos con los que vamos a trabajar: esto que dicho así, resulta tan tan simple, si lo piensa un poco, verá que es de una potencia nunca vista antes. Un objeto lo podemos considerar como un array multidimensional, donde se almacenan los datos de ese objeto (las coordenadas, el color, etc. en el ejemplo del marco) y sus métodos, que no serían más que punteros a funciones que trabajan con esos datos.

De hecho, y para ser más precisos, en el array de un objeto no se guardan los punteros a los métodos de ese objeto, sino los punteros a otros arrays donde se hallan estas funciones, para ahorrar de este modo memoria. También se guardan punteros a las clases superiores si las hubiese para buscar allí los métodos que no encontremos en la clase a la que el objeto pertenece: si un método no es hallado en una clase, se supone que debe pertenecer a la clase padre (clase superior). A esto le llamamos herencia, pero de esto hablaremos extensamente más adelante.

Ejemplo 5

Como hemos dicho, una clase es un nuevo tipo de dato y objetos son cada una de las asignaciones que hacemos a ese tipo de dato. En C un objeto lo podemos considerar como un *Struct*. Esta estructura albergaría los datos del objeto, y los punteros a las funciones que formarían el conjunto de sus métodos, más otros punteros a las clases superiores (padre).
Cada una de las instanciaciones (asignaciones) de variables que hagamos a un Struct, equivaldrá a crear un nuevo objeto de una clase. Las clases aparecen en C como Structs muy mejoradas. De hecho, fueron los Structs quienes sugirieron y dieron lugar a las clases y son sus antecesores informáticos. Para que entendamos hasta donde llega esta similitud, le informaremos que existe un programa que recibe un código fuente hecho en C++ (C con OOP) y devuelve otro código fuente equivalente al
anterior, pero en C normal.

Características de la programación orientada a objetos

Características:

La **abstracción**: Consiste en la generalización conceptual de un determinado conjunto de objetos y de sus atributos y propiedades, dejando en un segundo término los detalles concretos de cada objeto. ¿Qué se consigue con la abstracción? Bueno, básicamente pasar del plano material (cosas que se tocan) al plano mental (cosas que se piensan).

Herencia

Esta es la cualidad más importante de un sistema OOP, la que nos dará mayor potencia y Productividad, permitiéndonos ahorrar horas y horas de codificación y de depuración de errores. Es por ello que me niego a considerar que un lenguaje es OOP si no incluye herencia, como es el caso de Visual Basic (al menos hasta la versión 5, que es la última que conozco).

Como todos entendemos lo que es la herencia biológica, continuaremos con nuestro ejemplo taxonómico del que hablábamos en el epígrafe anterior. La clase león, como comentábamos antes, hereda cualidades -métodos, en lenguaje OOP- de todas las clases predecesoras -padres, en OOP- y posee métodos propios, diferentes a los del resto de las clases. Es decir, las clases van especializándose según se avanza en el árbol taxonómico. Cada vez que creamos una clase heredada de otra (la padre) añadimos métodos a la clase padre o modificamos alguno de los métodos de la clase padre. Veamos qué hereda la clase león de sus clases padre:

Vertebrados --> Espina dorsal
Mamíferos --> Se alimenta con leche materna
Carnívoros --> Al ser adultos se alimenta de carne

La clase león hereda todos los métodos de las clases padre y añade métodos nuevos que forman su clase distinguiéndola del resto de las clases: por ejemplo el color de su piel.

Observemos ahora algo crucial que ya apuntábamos antes: dos subclases distintas, que derivan de una misma clase padre común, pueden heredar los métodos de la clase padre tal y como estos han sido definidos en ella, o pueden modificar todos o algunos de estos métodos para adaptarlos a susnecesidades. En el ejemplo que exponíamos antes, en la clase león la alimentación es carnívora, mientras que en la clase hombre, se ha modificado éste dato, siendo su alimentación omnívora. Pongamos ahora un ejemplo algo más informático: supongamos que usted ha construido una clase que le permite leer números enteros desde teclado con un formato determinado, calcular su IVA y almacenarlos en un fichero. Si desea poder hacer lo mismo con números reales (para que admitan decimales), solo deberá crear una nueva subclase para que herede de la clase padre todos sus métodos y redefinirá solo el método de lectura de teclado. Esta nueva clase sabe almacenar y mostrar los números con

formato porque lo sabe su clase padre. Las cualidades comunes que comparten distintas clases, pueden y deben agruparse para formar una clase padre -también llamada **superclase**-. Por ejemplo, usted podría **derivar** las clases *presupuesto*, *albarán* y *factura* de la superclase *pedidos*, ya que estas clases comparten características comunes. De este modo, la clase padre poseería los métodos comunes a todas ellas y sólo tendríamos que añadir aquellos métodos propios de cada una de las subclases, pudiendo reutilizar el código escrito en la superclase desde cada una de las clases derivadas. Así, si enseñamos a la clase padre a imprimirse, cada uno de los objetos de las clases inferiores sabrá automáticamente y sin escribir ni una solo línea más de código imprimirse.

¡Genial! estará pensando usted, desde luego que lo es. La herencia como puede intuir, es la cualidad más importante de la OOP, ya que le permite reutilizar todo el código escrito para las superclases re-escribiendo solo aquellas diferencias que existan entre éstas y las subclases. Veamos ahora algunos aspectos más técnicos de la herencia: A la clase heredada se le llama, *subclase* o *clase hija*, y a la clase de la que se hereda *superclase* o *clase padre*.

Al heredar, la clase heredada toma directamente el comportamiento de su superclase, pero puesto que ésta puede derivar de otra, y esta de otra, etc., una clase toma indirectamente el comportamiento de todas las clases de la rama del árbol de la jerarquía de clases a la que pertenece.

Se heredan los datos y los métodos, por lo tanto, ambos pueden ser redefinidos en las clases hijas, aunque lo más común es redefinir métodos y no datos. Muchas veces las clases –especialmente aquellas que se encuentran próximas a la raíz en el árbol de la jerarquía de clases– son abstractas, es decir, sólo existen para proporcionar una base para la creación de clases más específicas, y por lo tanto no puede instanciarse de ellas; son las clases virtuales. Una subclase hereda de su superclase sólo aquellos miembros visibles desde la clase hija y por lo tanto solo puede redefinir estos. Una subclase tiene forzosamente que redefinir aquellos métodos que han sido definidos como abstractos en la clase padre o padres.

Normalmente, como hemos comentado, se redefinen los métodos, aun cuando a veces se hace necesario redefinir datos de las clases superiores. Al redefinir un método queremos o bien sustituir el funcionamiento del método de la clase padre o bien ampliarlo.

En el primer caso (sustituirlo) no hay ningún problema, ya que a la clase hija la dotamos con un método de igual nombre que el método que queremos redefinir en la clase padre y lo implementamos según las necesidades de la clase hija. De este modo cada vez que se invoque este método de la clase hija se ejecutará su código, y no el código escrito para el método homónimo de la clase padre. Pero si lo que queremos es ampliar el funcionamiento de un método existente en la clase padre (lo que suele ser lo más habitual), entonces primero tiene que ejecutarse el método de la clase padre, y después el de la clase hija. Pero como los dos métodos tienen el mismo nombre, se hace necesario habilitar alguna forma de distinguir cuando nos estamos refiriendo a un método de la clase hija y cuando al del mismo nombre de la clase padre. Esto se hace mediante el uso de dos palabras reservadas, las cuales pueden variar dependiendo del lenguaje que se utilice, pero que normalmente son:

this (en algunos lenguajes se utiliza la palabra reservada *Self*) y **super**: **this**

Con esta palabra, podemos referirnos a los miembros de la clase.
De hecho, siempre que dentro del cuerpo de un método nos refiramos a cualquier miembro de la clase, ya sea una variable u otro método, podemos anteponer *this*, auque en caso de no existir duplicidad, el compilador asume que nos referimos a un miembro de la clase.

Algunos programadores prefieren utilizar *this* para dejar claro que se está haciendo referencia a un miembro de la clase. **super** al contrario que *this*, *super* permite hacer referencia a miembros de la clase padre (o a los ancestros anteriores, que no hayan sido ocultados por la clase padre) que se hayan redefinido en la clase hija. Si un método de una clase hija redefine un miembro –ya sea variable o método– de su clase padre, es posible hacer referencia al miembro redefinido anteponiendo *super*.

Veamos un ejemplo (suponemos que solo los datos especificados de la clase A son visibles desde la clase B):

Datos Visible Métodos Visible
AD1 No AM1 No
AD2 Si AM2 Si
AD3 Si Am3 Si
La clase B hereda de la clase A:

Datos Heredado Métodos Heredado
AD2 Si AM2 Si
AD3 Si AM3 Si
BD1 No BM1 No
BD2 No BM2 No
Si en la clase B quisieramos redefinir el método AM2 de la clase A ampliándolo tendríamos que referirnos a él mediante *super* del siguiente modo:

```
public void AM2::B
{
super.AM2(); // Invocamos la ejecución del método AM2::A
```



```
..... // Resto del la implementación de AM2::B
.....
}
```

::B indica que el método AM2 está dentro de la clase B, al igual que AM2::A indica que es método AM2 de la clase A. Las palabras en azul indican el alcance del método y el valor que este devuelve (no se preocupe si no lo entiende ahora). En rojo hemos indicado el centro del tema que estamos tratando.

Veamos otro caso: supongamos que desde el método BD1 (BD1::B) queremos invocar el método AM2 de la clase B (AM2::B) y que desde el método BD2 (BD2::B) queremos invocar el método
AM2 de la clase A (AM2::A).

```
public void BD1::B
{
AM2(); // Invocamos la ejecución del método AM2::B
this.AM2() // Lo mismo que la línea anterior, pero usando this
..... // Resto del la implementación de BD1::B
.....
}
public void BD2::B
{
super.AM2(); // Invocamos la ejecución del método AM2::A
..... // Resto del la implementación de BD2::B
.....
}
```

Encapsulación

Hemos definido antes un objeto como un conjunto de datos y métodos. Dijimos también, que los métodos son procedimientos que trabajan con los datos del objeto. Veamos esto ahora con más detenimiento. Cuando definíamos el concepto de objeto un poco más arriba, dimos varios ejemplos de objetos; el tercero se refería a un marco (un recuadro) que podía visualizarse en nuestra pantalla de ordenador. La clase marco tenía las siguientes características:

Coordenada superior izquierda Mostrarse
Coordenada inferior derecha Ocultarse
Tipo de línea Cambiar de posición
Color de la línea
Color del relleno

Evidentemente, podríamos (y así deberíamos hacerlo) derivar la *clase marco* de la *superclase visual*, pero para simplificar lo más posible y centrarnos en lo que ahora tratamos, vamos a considerar esta clase como totalmente independiente de las demás.

Los datos de la clase son siempre los mismos para todos los objetos de esa clase, e igualmente los métodos, pero para cada instanciación de la clase -cada uno de los objetos pertenecientes a esa claselos valores de esos datos serán distintos; los métodos trabajarán con cada uno de estos valores de los datos dependiendo del objeto del que se trate.

Veámoslo gráficamente, supongamos que la tabla es nuestro monitor en modo texto 80x24 y los recuadros son dos objetos de la clase marco.

Hemos puesto en cada una de las esquinas las coordenadas de los vértices de los objetos marco. Estas coordenadas son, desde luego, aproximadas y se las hemos escrito en el siguiente formato: primero la ordenada y luego la abcisa (<Ypos>,<Xpos>), situando el origen de coordenadas en el ángulo superior izquierdo de la pantalla con origen en 0,0. De hecho, para definir un cuadrado en un plano cartesiano solo es necesario definir los dos vértices superior izquierdo e inferior derecho. De este modo, podemos apreciar cómo ambos objetos poseen los mismo datos:

Coordenada superior izquierda 7, 5 14,50
Coordenada inferior derecha 22,12 20,60
Color de la línea Verde Verde
Introducción a la OOP © Grupo EIDOS

Tipo de línea del recuadro Doble Simple

Los datos son los mismos, pero los valores que toman esos datos son diferentes para cada objeto. Ahora, podemos aplicar los métodos a los datos. Estos métodos producirán distintos resultados según a qué datos se apliquen. Así, el *objeto marco 1*, al aplicarle el método **Mostrarse**, aparece en la parte izquierda, rectangular verticalmente y con línea doble, mientras que el *objeto marco 2*, al aplicarle el mismo método, aparece en la parte izquierda, cuadrado y con línea simple.

Si quisiéramos ahora aplicarle el método **Cambiar de posición** al *objeto marco 1*, este método debería seguir los siguientes pasos y en este orden.

Llamar al método **Ocultarse** para este objeto.
Cambiar los datos de coordenadas para este objeto.
Llamar al método **Mostrarse** para este objeto.

Vemos así cómo un método de una clase puede llamar a otros métodos de su misma clase y cómo puede cambiar los valores de los datos de su misma clase. De hecho es así como debe hacerse: los datos de una clase sólo deben ser alterados por los métodos de su clase; y no de forma directa (que es como cambiamos los valores de las variables de un programa). Esta es una regla de oro que no debe olvidarse: **todos los datos de una clase son privados y se accede a ellos mediante métodos públicos**.

Veamos cómo se realiza esta acción según los dos modos comentados. Tomemos como ejemplo un objeto perteneciente a la clase *marco*, modificaremos su dato nY1 (coordenada superior izquierda) de dos modos distintos: directamente y mediante el método PonerY1().
Cambio directo: oCajaGeneral.nY1 = 12;

Cambio mediante invocación de método: oCajaGeneral.PonerY1(12);
Es más cómodo el primer método, ya que hay que escribir menos para cambiar el valor del dato y además, a la hora de construir la clase, no es necesario crear un método para cambiar cada uno de los datos del objeto. Sin embargo, y como ya hemos comentado, la OOP recomienda efusivamente que se utilice el segundo procedimiento. La razón es bien simple: una clase debe ser una estructura cerrada, no se debe poder acceder a ella si no es a través de los métodos definidos para ella. Si hacemos nY1 público (para que pueda ser accedido directamente), estamos violando el principio de encapsulación.

Esta forma de trabajo tiene su razón de ser: en algunos casos, pudiera ser que el método que cambia un dato realiza ciertas acciones o comprobaciones que el programador que está usando un objeto creado por otra persona no conoce, con lo que al manipular los datos del objeto directamente, podemos estar provocando un mal funcionamiento del mismo.

Para evitar que se puedan modificar datos que el usuario del objeto no debe tocar, algunos de ellos se hacen de *solo-lectura*, es decir, se puede saber su valor, pero no alterarlo. A estos datos los llamamos *ReadOnly* -LecturaSolo-. Sin embargo, hay una excepción a esta regla: se pueden hacer públicos todos los datos que la clase no utiliza. Y usted se preguntará, si la clase no los utiliza ¿Para qué los quiere? Hay un caso especial en el que al usuario de la clase se le proporciona una "bolsa" ("cargo" en Clipper, "tag" en Delphi) para que él almacene allí lo que quiera. De hecho, este recurso no es muy ortodoxo, ya que que lo que la teoría dice es que hay que heredar de la clase y añadir lo que uno necesite, pero es un recurso muy práctico, muy cómodo, y tampoco hay que ser más papistas que el Papa.

Dejemos ahora un poco de lado a los datos para centrarnos en otros aspectos de los métodos. ¿Cómo requerimos la actuación de un método? Enviando un **Mensaje** al objeto.
Al enviar un mensaje, se ejecuta un método, el cual puede llamar a su vez a otros métodos de su clase o de cualquier otra clase o bien cambiar los valores de los datos de ese objeto. Si el programador tiene que alterar los valores de los datos de un objeto deberá mandar un mensaje a ese objeto; lo mismo sucede cuando un objeto tiene que cambiar los valores de los datos de otro objeto.

Como podemos apreciar, un objeto es como una caja negra, a la que se le envía un mensaje y éste responde ejecutando el método apropiado, el cual producirá las acciones deseadas. un objeto, una vez programado es solo manipulable a través de mensajes.

A este intrínseco vínculo entre datos y métodos y al modo de acceder y modificar sus datos es a lo que llamamos **Encapsulación**. Gracias a la encapsulación, una clase, cuando ha sido programada y probada hasta comprobar que no tiene fallos, podemos usarla sin miedo a que al programar otros objetos estos puedan interferir con los primeros produciendo efectos colaterales indeseables que arruinen nuestro trabajo; esto también nos permite depurar (eliminar errores de programación) con suma facilidad, ya que si un objeto falla, el error solo puede estar en esa clase, y no en ninguna otra. Si usted ha programado con técnicas tradicionales sabrá apreciar lo que esto vale.

UNIDAD II

INTRODUCCIÓN A LAS CLASES

2.1.- Diagramas de clases y objetos.

Una clase es una descripción de conjunto de Una clase es una descripción de conjunto de objetos que comparten los mismos atributos, objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica. Operaciones, métodos, relaciones y semántica.

• Las clases son gráficamente representadas por Las clases son gráficamente representadas por cajas con compartimentos para: cajas con compartimentos para: – Nombre de la clase, atribu Nombre de la clase, atributos y operaciones / métodos tos y operaciones / métodos – Responsabilidades, Reglas, Responsabilidades, Reglas, Historia de Modificaciones, Historia de Modificaciones, etc.

• Los diseñadores desarrollan clases como conjuntos Los diseñadores desarrollan clases como conjuntos de compartimentos que crecen en el tiempo de compartimentos que crecen en el tiempo agregando incrementalmente aspectos y agregando incrementalmente aspectos y funcionalidades. funcionalidades.

Ejemplo HelloWorld
Ejemplo HelloWorld clase HelloWorld paint()
nombre operaciones Abstracción para HelloWorld Abstracción para HelloWorld clase HelloWorld paint() nombre nota g.drawString ("HelloWorld", 0, 10) operaciones
Ejemplo: " Ejemplo: "Hello, World Hello, World "

import java.awt.Graphics; class HelloWorld extends java.applet.Applet { public void paint (Graphics g) { g.drawString ("Hello, World!", 10, 10); } }

2.2.- Clases.

Diagrama de Clase Diagrama de Clase Applet HelloWorld paint() generalización dependencia Graphics. Diagramas de Clase Diagramas de Clase

• Muestra un cjto de elementos que son estáticos, Muestra un cjto de elementos que son estáticos, como las clases y tipos, junto con sus como las clases y tipos, junto con sus contenidos y relaciones contenidos y relaciones.

• Es un grafo de elementos clasificadores Es un grafo de elementos clasificadores conectados por varias relaciones estáticas conectados por varias relaciones estáticas.

• Clasificador Clasificador --> Class, Interface, DataType. > Class, Interface, DataType.

• Clase. Alcance. Referencia. Clase Abstracta. Clase. Alcance. Referencia. Clase Abstracta.
• Orden: [stereotype] nbre [stringPropiedades] Orden: [stereotype] nbre [stringPropiedades]

Ejemplo: Clase Dispositivo Ejemplo: Clase Dispositivo

• Define e implementa las operaciones para Define e implementa las operaciones para config, transmitir y recibir transmitir y recibir informac informac. hacia y desde el puerto serie . hacia y desde el puerto serie.

• HCom: handler handler al dispositivo. al dispositivo.

• Puerta: nombre del puerto serie puerta: nombre del puerto serie

• Velocidad: velocidad de la comunicación. velocidad: velocidad de la comunicación.

• Paridad: tipo de paridad paridad: tipo de paridad

• BitStop bitStop: cantidad de bits de : cantidad de bits de stop

• <> Dispositivo() crea y abre el dispositivo <> Dispositivo() crea y abre el dispositivo retornando un retornando un handler handler.

• <> RecuperarDispositivo RecuperarDispositivo() inf. BD para () inf. BD para config., LeerBloque LeerBloque() información del puerto () información del puerto.

• <>ConfigurarDispositivo ConfigurarDispositivo(), GrabBloquePuerto GrabBloquePuerto()

Diagramas de Clase Diagramas de Clase

• Atributo Atributo:

• Visibilidad nbre : exprTipo [= valor] [{prop}] visibilidad nbre : exprTipo [= valor] [{prop}]
• Visibilidad: public+, protected #, private visibilidad: public+, protected #, private - (no default) (no default).

• Prop: {changeable} (default), {frozen}. Multiplicity []. prop: {changeable} (default), {frozen}. Multiplicity [].

• Atributos de clase subrayados. Comienzan con minúscula Atributos de clase subrayados. Comienzan con minúscula.

• Operación Operación:

• Visibility nbre (parámetros) [:TipoRetorno] [{prop}] visibility nbre (parámetros) [:TipoRetorno] [{prop}]

• Prop: {query}, {sequential}, {guarded}, {concurrent}, prop: {query}, {sequential}, {guarded}, {concurrent}, {abstract} {abstract}

• Parámetros: [in|out|inout] nbre : TipoExp = valorDefault parámetros: [in|out|inout] nbre : TipoExp = valorDefault

• Operaciones de clase subrayadas. Operaciones de clase subrayadas.

2.3.- Técnicas de creación e inicialización de objetos.

Creando una persona

Con nuestra **clase Persona** ya escrita podemos crear cuantas personas queramos simplemente **instanciando objetos de esta clase**. Recordemos que los objetos se crean con ayuda de un **constructor**, el cual es un método especial que reserva la memoria para todos los atributos del objeto además de darnos la opción de inicializarlos. Un constructor tiene las siguientes características:

- Debe tener como identificador el mismo de la clase
- No puede retornar datos, por lo que no debe llevar un tipo de retorno en su declaración
- No puede ser heredado
- Debe tener visibilidad pública, ya que siempre se usa fuera de la clases.

En el código de la clase que escribimos tenemos el siguiente constructor que pide algunos valores para inicializar el objeto:

```
1.  // Constructor declarado en la definición de la clase como función prototipo
2.  Persona(const std::string& nombre,int edad, float peso, float estatura);
3.
4.  // Implementación fuera de la definición de la clase
5.  Persona::Persona(const string& nombre,int edad, float peso, float estatura){
6.    this -> nombre = nombre;
7.    this -> edad = edad;
8.    this -> peso = peso;
9.    this -> estatura = estatura;
10. }
```

La sintaxis para construir un objeto es la siguiente:

```
1. // Para un manejo automático de la memoria
2. NombreClase nombre_objeto;  // constructor por defecto, la llamada va sin paréntesis
3. NombreClase nombre_objeto(arg1, arg2,..., argn); // constructor con parámetros
4.
5. // Usando un apuntador y reservando memoria explícitamente
6. NombreClase* nombre_objeto = new NombreClase();
7. NombreClase* nombre_objeto = new NombreClase(arg1, arg2,..., argn);
```

Error común: para usar el constructor por defecto con manejo automático de la memoria hay que evitar poner los paréntesis ya que de no ser así el compilador creerá que intentamos declarar el prototipo de una función.

Hay dos maneras de usar el constructor: una que permite crear un objeto con manejo automático de la memoria que ocupa, y otra que nos permite utilizar un apuntador para manejar la memoria manualmente:

- **Objetos temporales**: la primera manera nos otorga un objeto que podremos usar de manera muy similar a cualquier tipo de dato primitivo del lenguaje, por ejemplo un entero `int`. El objeto creado de esta manera **permanecerá en memoria según el ámbito en que haya sido declarado**, es decir, que una vez que el programa deje su ámbito, automáticamente se destruye el objeto y se libera la memoria que ocupaba.

  - **Ventaja**: nos podemos despreocupar por el manejo de la memoria.
  - **Desventaja**: hay que olvidarse de poder hacer polimorfismo con herencia (en notas posteriores discutiremos a fondo este punto).

- **Apuntadores a objeto**: la segunda manera proporciona la memoria necesaria para el objeto y nos otorga la dirección en un apuntador. Con esta alternativa debemos **pedir la memoria explícitamente** con la palabra reservada `new` y seremos responsables de liberarla cuando sea necesario, en otras palabras, nos toca **destruir el objeto manualmente**.

  - **Ventaja**: tenemos la posibilidad de usar técnicas avanzadas de programación (por ejemplo, polimorfismo).
  - **Desventaja**: debemos lidiar con la complejidad que implica el uso de apuntadores.

Ahora veamos cómo **instanciar** una persona:

```
#include "Persona.h" //debemos incluir el archivo donde hicimos la declaración de la clase
int main(){
  Persona persona("Verónica", 22, 60, 1.65); //objeto temporal
  Persona* persona2; //apuntador a Persona
  persona2 = new Persona("Verónica",22,60,1.65);//asignación de la memoria para el objeto
  delete persona2; //liberación de la memoria apuntada por persona2
  return 0;
}
```

Como podemos observar se han creado dos objetos diferentes, de las dos manera explicadas anteriormente:

- **Persona**: es un objeto creado en el ámbito de la función `main()`, existirá mientras la ejecución del programa no deje el bloque de dicha función; al salir de ella la memoria es liberada automáticamente. Para acceder a los miembros del objeto (métodos y atributos) se utlizará el **operador punto "."**:

```
persona.saluda();
```

- **Persona2**: es un apuntador a objeto al cual se le ha asignado la dirección de memoria para manipular un objeto de la **clase Persona**. La liberación de la memoria depende de nosotros; así como la hemos pedido con `new`, debemos liberarla usando `delete`, si no lo hacemos es posible que esa memoria quede ocupada sin poder ser reasignada, al menos hasta que el programa deje de ejecutarse. La manera de acceder a los miembros del objeto es mediante el **operador flecha "->"**:

```
persona2->saluda();
```

2.4.- Introducción a la herencia.

Herencia Herencia Object Panel Component Container interface ImageObserver Applet HelloWorld

Algunas cuestiones sobre constructores

En nuestra clase de ejemplo únicamente tenemos un constructor, sin embargo, cabe destacar que podemos escribir más de uno y así tener la posibilidad de construir objetos de distintas maneras, utilizando la **sobrecarga de métodos**. Las dos posibilidades que hasta ahora conocemos son **constructor por defecto** y **constructor con parámetros**, analicemos un poco ambas alternativas:

- **constructor por defecto**: no permite datos como argumentos, sin embargo puede inicializar atributos. Normalmente se utiliza para crear objetos de manera rápida y tener los valores por defecto en sus atributos.

- **constructor con parámetros**: nos otorga la posibilidad de dar valores para inicializar el objeto al ser creado, los argumentos pueden ser de cualquier tipo de dato. Este tipo de constructor proporciona muchas alternativas de inicialización de objetos, por ejemplo un **constructor de copia** que recibe como argumento la referencia de otro objeto de la misma clase y copia los atributos al nuevo.

  Una cualidad interesante de C++, o mejor dicho, de los compiladores de C++ es la creación automática de funciones miembro en las clases, entre ellas el constructor. Cuando olvidamos escribir un constructor, el compilador nos proporciona uno por defecto, por eso es que una clase sin constructor puede ser compilada sin problemas. Aunque es de gran ayuda esta característica, es mejor escribirlos uno mismo y asegurarse de que los objetos se inicialicen correctamente.

2.5.- Reglas para la construcción de clases.

DESCRIPCIÓN DE LOS GRAFOS CONCEPTUALES Y DEL DIAGRAMA DE CLASES DE UML

Los grafos conceptuales son diagramas que modelan los conceptos y relaciones conceptuales de un sistema, son de fácil entendimiento para el interesado, y permiten un manejo computacional. La notación gráfica de los grafos conceptuales representa los conceptos

identificados en el dominio del problema de un sistema a través de rectángulos; las relaciones conceptuales por medio de círculos u óvalos y los enlaces establecidos entre conceptos y relaciones conceptuales se dibujan como flechas. (Véase Figura 1).

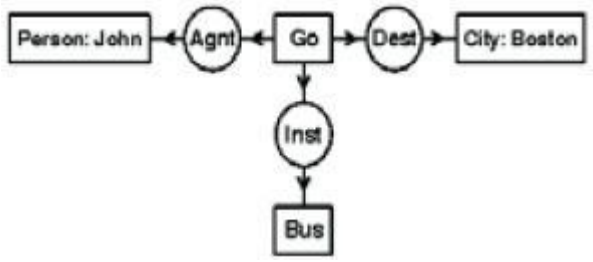


Figura 1. Forma gráfica del grafo conceptual para la oración: John va a Boston en bus. (Tomada de Sowa, 1984)

La forma gráfica de los grafos conceptuales tiene equivalencias en formatos legibles por máquina, tales como el CGIF (Conceptual Graph Interchange Format) y el KIF (Knowledge Interchange Format), los cuales contribuyen a darle utilidad computacional a estos grafos, puesto que existen herramientas actualmente que permiten obtener estas equivalencias desde editores de dichos grafos (Sowa, 1984).

A mediados de la década de los noventa, surgió el Unified Modeling Language (Lenguaje Unificado de Modelamiento o UML), como uno de los principales estándares para el desarrollo de software. El diagrama de clases de UML muestra los objetos relevantes del dominio del problema de un sistema específico y los agrupa en clases (Object Management Group, 2003). Para ello identifica en cada objeto los atributos que le pertenecen, las operaciones que se pueden realizar con él y sus relaciones con otros objetos (asociaciones, generalizaciones, dependencias, agregaciones y composiciones) con sus cardinalidades y el rol que desempeña en cada relación. Desde sus inicios, el diagrama de clases ha permitido el modelamiento de los conceptos del dominio de un problema determinado, pero en una notación que está dirigida a analistas entrenados en su uso, constituyéndose en un diagrama cuyo nivel de abstracción es bajo. La simbología básica del diagrama de clases de UML se muestra en la figura 2.

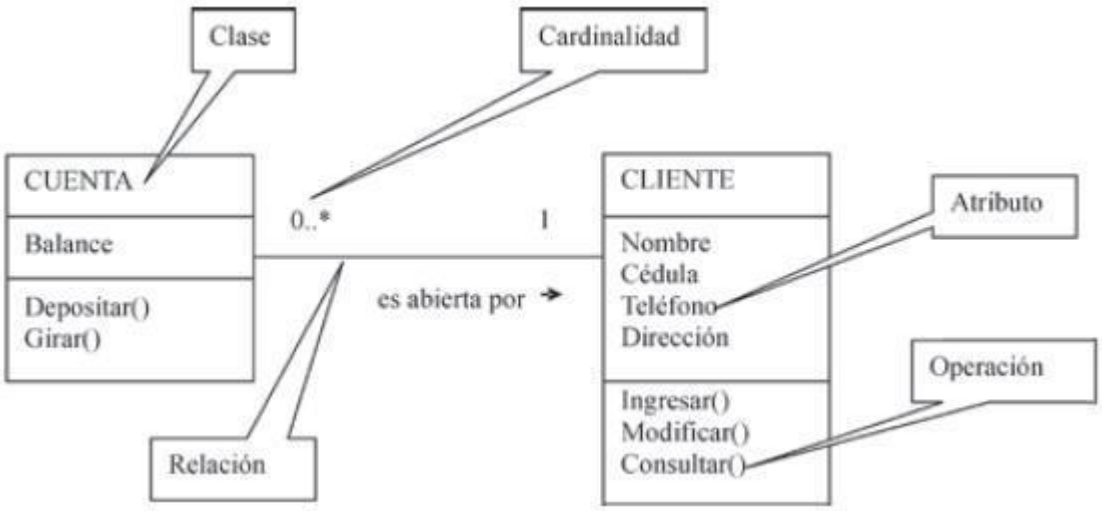


Figura 2. Simbología del diagrama de clases de UML

En la especificación del diagrama de clases de UML (Object Management Group, 2003) se describen cada uno de los elementos del diagrama de clases, los elementos con los cuales es posible su relación, su notación y sus representaciones opcionales.

En la siguiente sección se realiza un análisis crítico de algunos trabajos realizados para establecer conversiones entre esquemas conceptuales generados en el proceso de diseño de sistemas informáticos.

ANTECEDENTES

Entre los principales trabajos realizados para lograr conversión entre modelos conceptuales se pueden mencionar:

Snook y Butler (2001) proponen una adaptación de una notación de diseño gráfica (UML) para la especificación formal y la construcción de una herramienta de traducción automática, agregando detalles semánticos y definiendo el significado de cada elemento del diagrama. Implementan, además, un prototipo para trasladar las características diagramáticas de UML al lenguaje B (Abrial, 1996). B es un lenguaje de especificación formal, y el objetivo de este trabajo es utilizar algunas de las características de los diagramas de clases para realizar especificaciones formales de forma fácil. Las restricciones de los componentes del diagrama de clases son descritas en una forma restringida de la notación del lenguaje B. Esta traducción no agrega información a la especificación de diagramas en UML, sino que proporciona una forma textual alternativa. La iniciativa de agregar detalles semánticos a los modelos conceptuales puede conducir a una forma de encontrar deficiencias en tal diagrama y realizar refinamientos al modelo. La conversión que se plantea en Snook y Butler (2001) no contribuye a mejorar la comprensión del diagrama de clases por parte del interesado; esto se

produce porque una especificación en un lenguaje formal como B requiere de un entrenamiento mucho mayor para su comprensión, puesto que el nivel de abstracción de este lenguaje es incluso mucho más bajo que el diagrama de clases de UML.

La herramienta de conversión descrita en Timm y Gannod (2005) fue creada para tomar una especificación UML y convertirla en una representación equivalente en OWL-S (que es un lenguaje enfocado a agentes de software, creado para describir conceptos y relaciones semánticas de servicios Web usando ontologías). Para ello, utiliza XMI (un estándar de comunicación entre aplicaciones basado en el lenguaje de marcado extendido XML) y sobre é l aplica las transformaciones con un lenguaje basado en este estándar. Estos autores buscan la forma de generar un modelo de servicios OWL-S tomando una representación XML basada en un diagrama de actividades de UML. El mecanismo utilizado para validar qué tan correcta es la conversión realizada es la herramienta de especificación de ontologías Protégé. Este tipo de conversión se ocupa de la legibilidad por parte de la máquina y poco o nada contribuye a la legibilidad por parte de seres humanos; además, el nivel de abstracción se conserva bajo, lo cual no permite la validación por parte de los interesados.

Creasy y Ellis (1993) presentan una herramienta informática (basada en grafos conceptuales) para solucionar el problema de integración de esquemas conceptuales generados en el diseño de sistemas de información complejos. Los lineamientos teóricos de esta propuesta están orientados al diagrama entidad-relación, no al diagrama de clases que es completamente objetual y además es reconocido como el estándar aceptado por el OMG (Object Management Group, 2003). Pese a trabajar con grafos conceptuales, esta propuesta no eleva de manera sustancial el nivel de abstracción, pues se limita a elaborar una descripción del diagrama entidad-relación con la terminología de este diagrama, la cual es poco entendible para los interesados. Por otra parte, esta metodología no utiliza los roles semánticos para especificar las relaciones entre los conceptos del dominio, haciendo más compleja la traducción de esquemas conceptuales a lenguaje natural.

Coad y Yourdon (1990) establecieron un método que parte del análisis del modelo verbal y a partir de tal análisis identifican los principales elementos del diagrama de clases, diagrama principal en el modelamiento y desarrollo de un producto de software orientado por objetos. Esta metodología identifica los principales sustantivos presentes en el modelo verbal, información valiosa para la detección de las ' clases' y 'objetos' del modelo, y los principales verbos, que pueden suministrar información sobre las relaciones entre las diferentes entidades del modelo. Esta conversión parte de un modelo más comprensible por el interesado hacia uno de notación más elaborada y compleja, que es un proceso inverso al planteado en este artículo. Como la entrada de esta conversión es el lenguaje natural, es necesario llevar a cabo procesos de desambiguación antes de generar el diagrama de salida. En esta misma línea de trabajo, Harmain y Gaizauskas (2000) proponen el CM-Builder, una herramienta CASE para la obtención del diagrama de clases a partir de las especificaciones de requisitos en lenguaje natural. En este caso, dada la existencia de un diagrama de clases, no es posible realizar el proceso contrario, con el fin de realizar una validación del mismo por parte del interesado.

NL-OOPS (Natural Language Object – Oriented Product System), es un sistema propuesto por Mich (1996) e incorpora un sistema de procesamiento del lenguaje natural denominado LOLITA (Large-scale Object-based Language Interactor, Translator and Analyser). LOLITA contiene un conjunto de funciones para el análisis del lenguaje natural y a través de este análisis detecta ambigüedades en el texto. Como resultado final NL-OOPS entrega, en una estructura de árbol, un conjunto de las clases candidatas y sus posibles instancias, atributos y métodos (Mich y Garigliano, 2002). Esta solución es sólo una contribución a la complementación del diagrama de clases, ya que no posee la representación de las relaciones entre las clases del modelo convertido.

Fliedl (1999, 1999a, 2002, 2002a y 2003) acompañado de un grupo de ingenieros de sistemas (Kop y Mayr, 2002), (Mayr y Kop, 2002) desarrollaron el KCPM (Klagenfurt Conceptual Predesing Model) como parte del proyecto NIBA. El objetivo del proyecto es facilitar el desarrollo de productos de software a través de la traducción automática del lenguaje natural a esquemas conceptuales. Para lograr este objetivo cuenta con los siguientes elementos:

• Un analizador sintáctico.

• Un intérprete semántico basado en la teoría de roles Theta (Agente, Experimentador, Tema, Meta, Fuente, Locación) (Gruber, 1965), y en lineamientos teóricos para determinar las funciones de las palabras que acompañan a un verbo (Fillmore, 1968).

• Una herramienta KCPM para la traducción y análisis (sintáctico y semántico) de oraciones.

• Una herramienta para el cálculo de puntos de función.

El proceso de conversión descrito en el proyecto NIBA es completamente contrario al planteado en este artículo, ya que va de lenguaje natural a esquemas conceptuales y requiere desambiguación. En la sección siguiente se generan las reglas de transformación entre diagramas de clases y grafos conceptuales de Sowa, partiendo de las ventajas y desventajas de las propuestas existentes y de los aspectos comunes de los diagramas involucrados en la conversión.

BREVE DESCRIPCIÓN DE LA PROPUESTA

Los antecedentes analizados en la sección anterior no suministran una solución que posibilite a los interesados lograr una comprensión del diagrama de clases, que le permita definir si efectivamente ese diagrama representa lo que está tratando de expresar. Sin embargo, en la mayoría de esos trabajos se emplean reglas de conversión para lograr la transformación de un modelo a otro.

Entre las limitaciones de los trabajos expuestos en la sección anterior se pueden nombrar los siguientes:

• En la mayoría de los casos la conversión se presenta a partir de lenguaje natural, pero no se establece qué ocurre cuando el diagrama de clases ya existe y se requiere que el interesado realice una validación del mismo. Esta es una situación típica durante la captura de los requisitos.

• En los casos en que se parte de diagramas se llega a una especificación formal o a un diagrama de igual o incluso menor nivel de abstracción que, por ende, requiere mayor entrenamiento para el entendimiento humano.

Para aprovechar las bondades de las propuestas existentes y dejar de lado las limitaciones expuestas, en este artículo se propone un método para realizar la conversión entre el diagrama de clases y los grafos conceptuales de Sowa. Esta conversión se realizará directamente sobre el diagrama de clases generado por el analista en la fase de análisis del ciclo de vida de un producto de software. El objetivo de esta conversión es encontrar una forma de expresar una sintaxis tan compleja como lo es el diagrama de clases de UML en una notación más comprensible por los interesados en la construcción de un sistema informático; con ello se busca involucrar al interesado en la validación del diagrama de clases, especialmente tratando de establecer si dicho diagrama realmente representa de manera adecuada el dominio del problema que se pretende solucionar con la pieza de software.

El primer paso para lograr la conversión será encontrar las partes de ambos diagramas que expresan aspectos comunes. Por ejemplo, las clases del diagrama de clases muestran los principales conceptos de un sistema informático modelado al igual que los conceptos de los grafos conceptuales. De igual manera, las relaciones entre las clases de un diagrama de clases suministran cierta información para identificar las relaciones conceptuales de los grafos de Sowa y los enlaces establecidos entre conceptos y relaciones conceptuales.

Para llevar a cabo la conversión de diagramas se definen los siguientes principios:

• Toda clase, atributo y operación del diagrama de clases debe ser un concepto del grafo conceptual de Sowa.

• Las relaciones entre clases, atributos y operaciones del diagrama de clases se pueden expresar en términos de otros conceptos y los roles theta o casos semánticos del grafo conceptual de Sowa (agente, experimentador, locación, destino, instrumento, etc.). Adicionalmente, se definen las siguientes reglas de transformación del diagrama de clases a grafos conceptuales de Sowa:

Regla 1: clase-atributo

Un atributo de una clase genera los siguientes elementos en el grafo conceptual

• Un concepto dado por el nombre del atributo.

• El concepto 'tener'.

• Un concepto dado por el nombre de la clase.

• Una relación 'beneficiario' entre los conceptos 'tener' y el nombre de la clase.

• Una relación 'tema' entre los conceptos ' tener' y el nombre del atributo.

Una frase que se puede extraer de la definición de la clase cliente, que se aprecia en la figura 3 es: 'El cliente es el beneficiario del concepto tener, cuyo tema es código' o, más concretamente, 'El cliente tiene código'. El grafo conceptual correspondiente a esta frase se muestra en la figura 4.

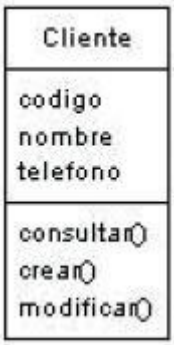


Figura 3. Definición de la clase cliente

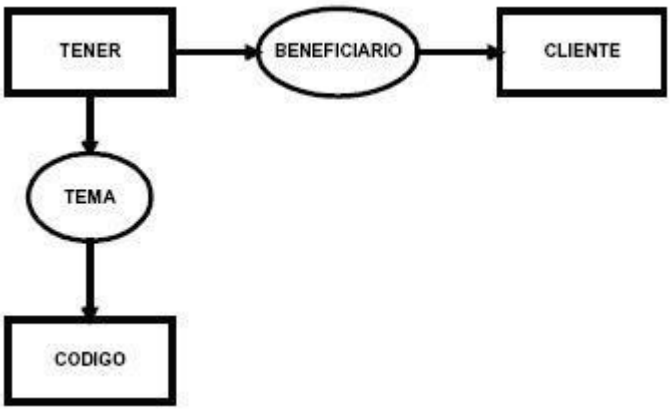


Figura 4: Grafo conceptual de la frase 'El cliente tiene código'

Regla 2: clase-operación

Una operación de una clase genera los siguientes elementos en el grafo conceptual:

• Un concepto dado por el nombre de la clase.

• Un concepto dado por el nombre de la operación.

• El concepto 'usuario', sugerido al interesado para que interactivamente defina si ese concepto existe o si tiene otro nombre.

• Una relación 'agente' entre los conceptos ' usuario' y el nombre de la operación.

• Una relación 'tema' entre los conceptos dados por el nombre de la operación y el nombre de la clase.

De la definición de la clase cliente del ejemplo anterior se puede afirmar que 'el usuario es el agente del concepto consultar, cuyo tema es el cliente', o simplemente que 'el usuario consulta al cliente'. El grafo conceptual para representar lo anterior sería el que aparece en la figura 5.

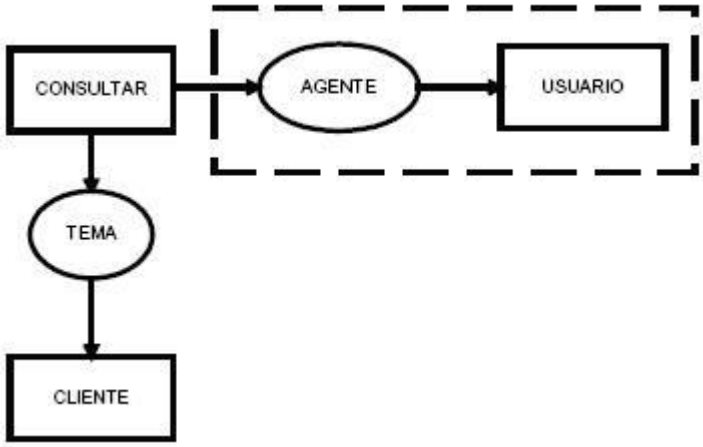


Figura 5. Grafo conceptual de la frase 'El usuario consulta al cliente'

Regla 3: clase-relación-clase

Dos clases relacionadas como en el caso de la Figura 6, generan los siguientes elementos:

• Dos conceptos dados por el nombre de las clases.

• Un concepto dado por el nombre de la relación.

• Una relación 'agente' entre el nombre de la relación y el nombre de la clase que tiene cardinalidad 1.

• Una relación 'tema' entre el nombre de la relación y el nombre de la clase que tiene cardinalidad *.

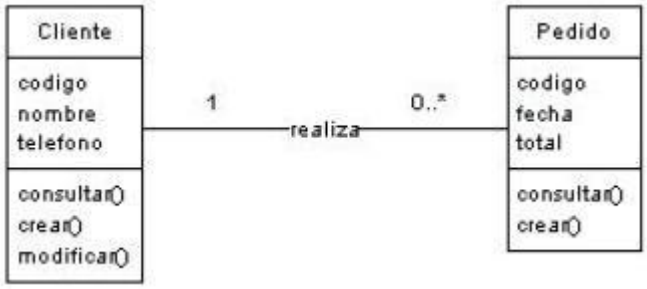


Figura 6. Relación entre las clases cliente y pedido

El grafo conceptual correspondiente se muestra en la figura 7. Es de notar que, si la relación es 1 a 1, entonces el sentido de la relación definirá la relación conceptual de cada una de las clases. Si la relación es una agregación o composición, como en la figura 8, entonces la clase de agregación y la clase de composición no son agentes sino beneficiarios.

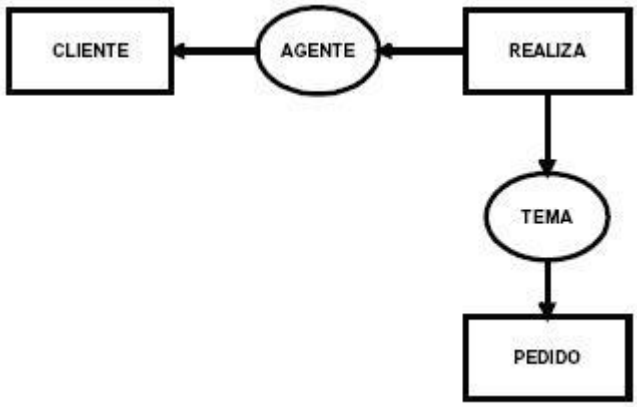


Figura 7: Grafo conceptual para la relación entre dos clases

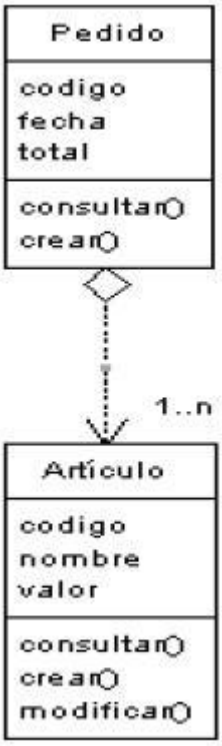


Figura 8: Relación de agregación entre dos clases

Según la regla anterior, el grafo conceptual generado para esta porción de diagrama de clases es el que aparece en la figura 9:

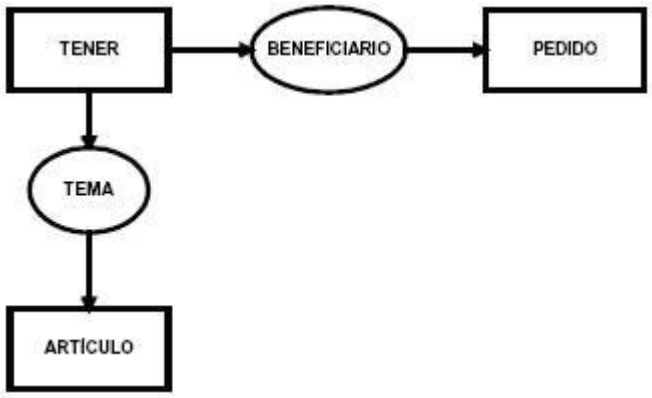


Figura 9: Grafo conceptual para la relación de agregación entre las clases pedido y artículo

Regla 4: Metarregla para correferencia

En cada diagrama generado, los conceptos que se hayan generado desde el mismo nombre de clase se unen con una línea punteada que expresa la correferencia en el grafo conceptual, es decir, la forma de reconocer que este concepto es el mismo en dicho grafo.

En la siguiente sección se muestra la implementación de las reglas y se ejemplifica con un caso de estudio.

APLICACIÓN Y CASO DE ESTUDIO

Las reglas definidas en la sección anterior se implementaron en el prototipo de una aplicación con las siguientes características:

• La entrada al proceso es un diagrama de clases elaborado en la herramienta CASE ArgoUML, disponible en la página http://argouml.tigris.org/. Para ingresar el diagrama en la aplicación, se requiere el código XMI generado por el ArgoUML.

• Las reglas se programaron en XQuery, un lenguaje de consulta que explora archivos en XML y genera nuevos archivos en XML que pueden ser leídos por otras herramientas. Para la programación de dichas reglas se empleó la herramienta XQEngine, disponible en la página http://xqengine.sourceforge.net/.

La salida del proceso es un archivo en formato XML que se puede leer en la herramienta Charger, disponible en la página http://sourceforge. net/projects/charger/, la cual permite la edición de grafos conceptuales de Sowa.

Parte de la consulta en XQuery para la extracción de los atributos de las clases (Regla 1) se muestra a continuación. Se suprimieron las partes del código que generan los elementos del XML en Charger, para facilitar la legibilidad del mismo.

```
<grafo_conceptual>
    {for $a in //Foundation.Core.Class
    return
        <reglas_para_clases>
        {for $b in $a/Foundation.Core.Classifier.feature/Foundation.Core.Attribute/Foundation.Core.ModelElement.name
        return
            <reglas_para_atributos>
                <relacion nombre="beneficiario">
                    <concepto1>tener</concepto1>
                    <concepto2>{$a/Foundation.Core.ModelElement.name/text()}</concepto2>
                </relacion>
                <relacion nombre="tema">
                    <concepto1>tener</concepto1>
                    <concepto2>{$b/text()}</concepto2>
                </relacion>
            </reglas_para_atributos>}
        </reglas_para_clases>}
</grafo_conceptual>
```

De manera similar, el código en XQuery que corresponde a la extracción de las operaciones es el siguiente:

```
<grafo_conceptual>
    {for $a in //Foundation.Core.Class
    return
        <reglas_para_clases>
        {for $c in $a/Foundation.Core.Classifier.feature/Foundation.Core.Operation/Foundation.Core.ModelElement.name
        return
            <reglas_para_operaciones>
                <relacion nombre="agente">
                    <concepto1>{$c/text()}</concepto1>
                    <concepto2>usuario</concepto2>
                </relacion>
                <relacion nombre="tema">
                    <concepto1>{$c/text()}</concepto1>
                    <concepto2>{$a/Foundation.Core.ModelElement.name/text()}</concepto2>
                </relacion>
            </reglas_para_operaciones>}
        </reglas_para_clases>}
</grafo_conceptual>
```

En ambos casos, Foundation.Core.Class es la etiqueta del código XMI de ArgoUML que almacena la información de la clase y es superior en jerarquía a las demás etiquetas que comienzan con Foundation y que se emplean para recopilar la información de atributos y operaciones.

Empleando esta aplicación se ejemplifica la conversión de un diagrama de clases correspondiente a un sistema de pedidos de artículos; se supone que este diagrama fue generado por un analista en la fase de análisis y su imagen en ArgoUML se puede apreciar en la figura 10.

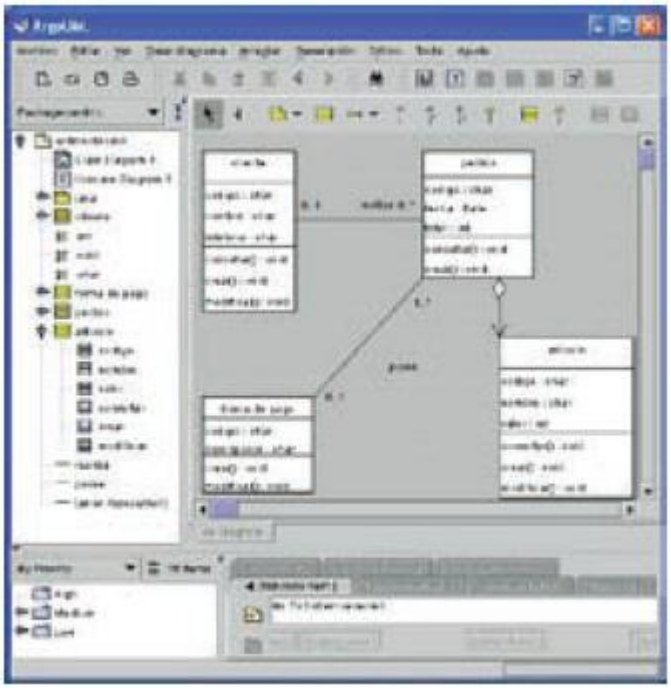


Figura 10. Diagrama de clases

Ahora, cada clase, atributo, operación y relación del diagrama de clases se representan como conceptos del grafo conceptual, utilizando la relación conceptual apropiada (rol semántico) dependiendo de la regla aplicada, para terminar el proceso de conversión con la unión de los conceptos del grafo. A manera de ejemplo, véanse las figuras 11 a 15, generadas con el Charger a partir del archivo en XMI del diagrama de clases de la figura 10.

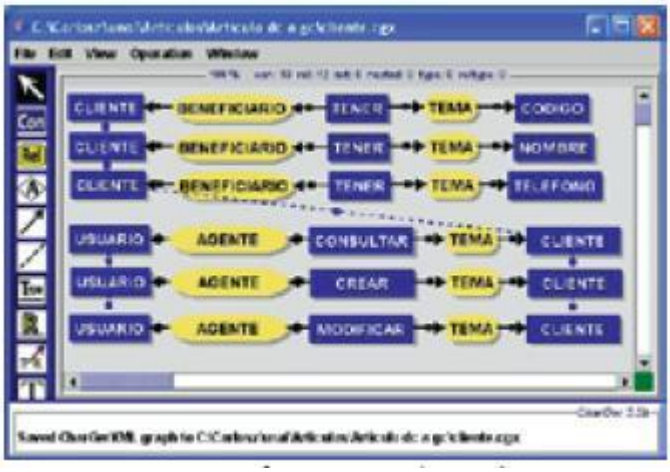


Figura 11. Información de atributos y operaciones de la clase cliente

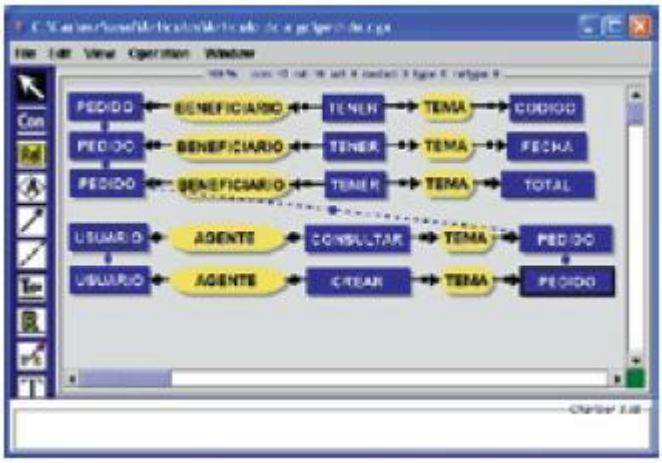


Figura 12. Información de atributos y operaciones de la clase pedido

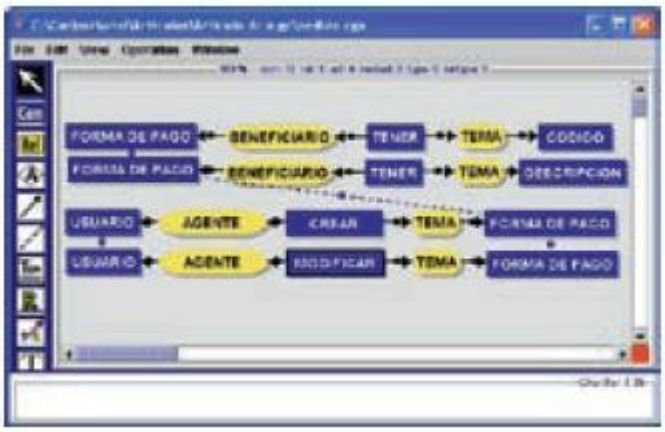


Figura 13. Información de atributos y operaciones de la clase forma de pago

Figura 14. Información de atributos y operaciones de la clase artículo

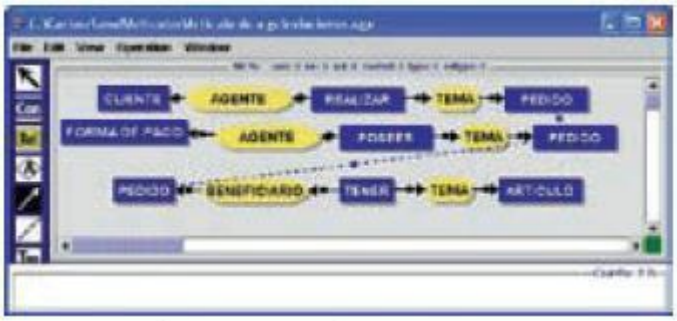


Figura 15. Información de las relaciones entre las clases

Nótese que en cada una de estas figuras se empleó una línea punteada para unir los rectángulos que se refieren a los mismos conceptos. Esta notación se adoptó a partir del estándar definido por Sowa (1984) para los grafos conceptuales.

En este artículo se propuso un mecanismo para permitir la conversión de diagramas de clases a grafos conceptuales, esquemas de nivel más alto de abstracción y, consecuentemente, de más fácil comprensión por los interesados humanos; en estos grafos, sin embargo, no se renuncia a la posibilidad de comunicación entre máquinas debido a sus formas textuales equivalentes, que pueden ser legibles por máquina. Uno de los objetivos y motivaciones para realizar la conversión es la validación, por parte del interesado, de los esquemas conceptuales elaborados en las diferentes fases de desarrollo de un producto de software.

Se presentan cuatro reglas para definir la conversión a grafos conceptuales de las relaciones entre clases y sus atributos, clases y sus operaciones y clases con otras clases del diagrama de clases; estas reglas se aplican a un caso de estudio referente a un sistema de pedidos de artículos, para obtener los grafos conceptuales de Sowa.

Las reglas enunciadas pretenden un acercamiento con el lenguaje del interesado, a diferencia de otros trabajos del estado del arte, que se limitan a convertir a grafos de Sowa esquemas conceptuales, pero conservando el lenguaje del analista. Sin embargo, y pese a que la legibilidad del diagrama de clases expresada en términos de los grafos conceptuales mejora, se puede apreciar que la extensión de los diagramas es mucho mayor, dado que se sigue el estándar de correferencia enunciado por Sowa (1984).

Entre los trabajos a realizar se pueden enumerar los siguientes:

• La conversión de diagramas de clases más complejos que incluyan todos los tipos de relaciones definidas en la especificación de UML, además de la conversión a grafos conceptuales de otros esquemas comúnmente utilizados en la definición, análisis y diseño de sistemas informáticos.

• La complementación de la aplicación que implementa las reglas.

• La definición de un esquema de comunicación más cercano aun que los grafos conceptuales al lenguaje que utilizan los interesados, de forma que se pueda lograr una comunicación más rápida y efectiva que permita la validación del diagrama de clases.

2.6.- Introducción a las funciones amigas.

Introducción a las funciones amigas

En algunas situaciones es necesario que una función tenga acceso a los miembros privados de una clase sin que esta función sea un miembro de esa clase. Para esto están las funciones amigas.

Una función amiga no es miembro de una clase, pero tiene acceso a los elementos privados la clase. Las funciones amigas se usan comúnmente para la sobrecarga de operadores y la creación de funciones de E/S.

Veamos una breve introducción a las funciones amigas mediante un ejemplo:

```
#include <iostream.h>
class miclase{
        int n, d; //miembros privados
public:
    miclase(int i, int j) { n=i; d=j;}
    friend int esfactor (miclase ob); //se declara esfactor como funcion amiga
                                      //esta funcion tendra acceso a las variables privadas
n y d
                                    //no es una función miembro de miclase, es amiga
nada más
                                   //por tanto no se implementa dentro de la miclase

};

//aqui declaramos la funcion esfactor, recordemos que es amiga de miclase
//por lo tanto aquí podemos acceder a los miembros privados de miclase como n y d
```

```
int esfactor(miclase ob){
        if(!(ob.n % ob.d) //se accede a n que es un miembro privado de ob
            return 1;

        else


            return 0;

}

main(){
miclase ob1(10,2), ob2(13,3);
if (esfactor(ob1)) cout <<"2 es factor de 10\n";
else cout << "2 no es factor de 10\n";
if (esfactor(ob2)) cout <<"3 es factor de 13\n";
else cout << "3 no es factor de 13\n";
return 0;
}
```

Hay que tener en cuenta que una función amiga no es miembro de la clase que es amiga, por lo tanto la siguiente instrucción dentro del main sería erronea:

ob1.esfactor() ya que ob1 no tiene una funcion miembro dentro de su clase llamada esfactor, esta función es sólo amiga.

SOBRECARGA DE FUNCIONES Y OPERADORES

3.1.- Funciones.

FUNCIONES VIRTUALES Y FUNCIONES ABSTRACTAS

En programación orientada a objetos (POO), una función virtual o método virtual es una función cuyo comportamiento, al ser declarado "virtual", es determinado por la definición de una función con la misma cabecera en alguna de sus subclases. Este concepto es una parte muy importante del polimorfismo en la POO. El concepto de función virtual soluciona los siguientes problemas:

En POO, cuando una clase derivada hereda de una clase base, un objeto de la clase derivada puede ser referido (o coercionado) tanto como del tipo de la clase base como del tipo de la clase derivada. Si hay funciones de la clase base redefinidas por la clase derivada, aparece un problema cuando un objeto derivado ha sido cohercionado como del tipo de la clase base. Cuando un objeto derivado es referido como del tipo de la base, el comportamiento de la llamada a la función deseado es ambiguo.

Distinguir entre virtual y no virtual sirve para resolver este problema. Si la función en cuestión es designada "virtual", se llamará a la función de la clase derivada (si existe). Si no es virtual, se llamará a la función de la clase base.

EJEMPLO

Por ejemplo, una clase base Animal podría tener una función virtual come. La subclase Pez implementaría come() de forma diferente que la subclase Lobo, pero se podría invocar a come() en cualquier instancia de una clase referida como Animal, y obtener el comportamiento de come() de la subclase específica.

Esto permitiría a un programador procesar una lista de objetos de la clase Animal, diciendo a cada uno que coma (llamando a come()), sin saber qué tipo de animales hay en la lista (Función Virtual). Tampoco tendría que saber cómo come cada animal, o
cuántos tipos de animales puede llegar a existir. El siguiente, es un ejemplo en C++:

```
# include <iostream>
class Animal
{
public:
virtual void come() { std::cout << "Yo como como un animal
genérico.\n"; }
virtual ~Animal() {}
};
class Lobo : public Animal
{
public:
void come() { std::cout << "¡Yo como como un lobo!\n"; }
virtual ~Lobo() {}
};
class Pez : public Animal
{
public:
void come() { std::cout << "¡Yo como como un pez!\n"; }
virtual ~Pez() {}
};
class OtroAnimal : public Animal
```

```
{
virtual ~OtroAnimal() {}
};
int main()
{
Animal *unAnimal[4];
unAnimal[0] = new Animal();
unAnimal[1] = new Lobo();
unAnimal[2] = new Pez();
unAnimal[3] = new OtroAnimal();
for(int i = 0; i < 4; i++) {
unAnimal[i]->come();
}
for (int i = 0; i < 4; i++) {
delete unAnimal[i];
}
return 0;
}
```

```
Salida con el método virtual come:
Yo como como un animal genérico.
¡Yo como como un lobo!
¡Yo como como un pez!
Yo como como un animal genérico.
Salida sin el método virtual come:
Yo como como un animal genérico.
Yo como como un animal genérico.
Yo como como un animal genérico.
Yo como como un animal genérico.
```

FUNCIONES VIRTUALES

El polimorfismo en tiempo de ejecucion es logrado por una combinacion de dos características: 'Herencia y funciones virtuales". Una función virtual es una función que es declarada como 'virtual' en una clase base y es redefinida en una o mas clases derivadas. Ademas, cada clase derivada puede tener su propia version de la funcion virtual. Lo que hace interesantes a las funciones virtuales es que sucede cuando una es llamada a través de un puntero de clase base (o referencia).

En esta situacion, C++ determina a cual version de la funcion llamar basándose en el tipo de objeto apuntado por el puntero. Y, esta determinacion es hecha en 'tiempo de ejecucion'. Además, cuando diferentes objetos son apuntados, diferentes versiones de la funcion virtual son ejecutadas.

En otras palabras es el tipo de objeto al que esta siendo apuntado (no el tipo del puntero) lo que determina cual version de la función virtual sera ejecutada. Ademas, si la clase base contiene una funcion virtual, y si dos o mas diferentes clases son derivadas de esa clase base, entonces cuando tipos diferentes de objetos estan siendo apuntados a traves de un puntero de clase base, diferentes versiones de la funcion virtual son ejecutadas. Lo mismo ocurre cuando se usa una refrencia a la clase base.

Se declara una funcion como virtual dentro de la clase base precediendo su declaracion con la palabra clave virtual. Cuando una funcion virtual es redefinida por una clase derivada, la palabra clave 'virtual' no necesita ser repetida (aunque no es un error hacerlo).

Una clase que incluya una funcion virtual es llamada una 'clase polimórfica'. Este término también aplica a una clase que hereda una clase base conteniendo una funcion virtual.

Examine este corto programa, el cual demuestra el uso de funciones virtuales:

```
// Un ejemplo corto que usa funciones virtuales
#include <iostream>
using namespace std;
class base {
public:
virtual void quien() {cout << "Base" << endl;} // especificar una
clase virtual
};
class primera_d : public base {
public:
// redefinir quien() relativa a primera_d
void quien() {cout << "Primera derivacion" << endl;}
};
class segunda_d : public base {
public:
// redefinir quien relativa a segunda_d
void quien() {cout << "Segunda derivacion" << endl;}
};
int main()

{
base obj_base;
base *p;
primera_d obj_primera;
segunda_d obj_segundo;
p = &obj_base;
p->quien(); // acceder a quien() en base
p = &obj_primera;
p->quien(); // acceder a quien() en primera_d
```

```
p = &obj_segunda;
p->quien(); // acceder a quien() en segunda_d
return 0;
}
```

Este programa produce la siguiente salida:

Base
Primera derivacion
Segunda derivación

Examinemos el programa en detalle para comprender como funciona: En 'base', la funcion 'quien()' es declarada como 'virtual'. Esto significa que la funcion puede ser redefinida en una clase derivada. Dentro de ambas 'primera_d' y 'segunda_d', 'quien()' es redefinida relativa a cada clase. Dentro de main(), cuatro variables son declaradas: 'obj_base', el cual es un objeto del tipo 'base'; 'p' el cual un un puntero a objetos del tipo 'base'; 'obj_primera' y 'obj_segunda', que son objetos de dos clases derivadas. A continuacion 'p' es asignada con la direccion de 'obj_base', y la funcion quien() es llamada.

Como quien() es declarada como virtual, C++ determina en tiempo de ejecucion, cual version de quien() es referenciada por el tipo del objeto apuntado por 'p'. En este caso, 'p' apunta a un objeto del tipo 'base', asi que es la version de 'quien()' declarada en 'base' la que es ejecutada. A continuacion, se le asigna a 'p' la direccion de 'obj_primera'. Recuerde que un puntero de la clase base puede referirse a un objeto de cualquier clase derivadas. Ahora, cuando quien() es llamada, C++ nuevamente comprueba para ver que tipo de objeto es apuntado por 'p' y basado en su tipo, determina cual version de quien() llamar.

Como 'p' apunta a un objeto del tipo 'obj_primera', esa versión de quien() es usada. De ese mismo modo, cuando 'p' es asignada con la direccion de 'obj_segunda', la version de quien() declarada en 'segunda_d' es ejecutada.

RECUERDE: "Es determinado en tiempo de ejecucion cual version de una funcion virtual en realidad es llamada. Ademas, esta determinacion es basada solamente en el tipo del objeto que esta siendo apuntado por un puntero de clase base." Una funcion virtual puede ser llamada normalmente, con la sintaxis del operador estandar de 'objeto.' Esto quiere decir que en el ejemplo precedente, no seria incorrecto sintácticamente acceder a quien() usando esta declaracion: obj_primera.quien(); Sin embargo, llamar a una funcion virtual de esta manera ignora sus atributos polimórficos. Es solo cuando una función virtual es accesada por un puntero de clase base que el polimorfismo en tiempo de ejecucion es logrado.

A primera vista, la redefinicion de una funcion virtual en una clase derivada parece ser una forma especial de sobrecarga de funcion. Sin embargo, este no es el caso. De hecho, los dos procesos son fundamentalmente diferentes. Primero, una funcion sobrecargada debe diferir en su tipo y/o numero de parametros, mientras que una funcion virtual redefinida debe tener exactamente el mismo tipo y numero de parametros.

De hecho, los prototipos para una funcion virtual y sus redefiniciones debe ser exactamente los mismos.

Si los prototipos difieren, entonces la funcion simplemente se considera sobrecargada, y su naturaleza virtual se pierde. Otra restriccion es que una funcion virtual debe ser un miembro, no una funcion 'friend', de la clase para la cual es definida. Sin embargo, una funcion virtual puede ser una funcion 'friend' de otra clase. Tambien, es permisible para funciones destructores ser virtuales, pero esto no es asi para los constructores.

"Cuando una funcion virtual es redefinida en una clase derivada, se dice que es una funcion 'overriden' (redefinida)" Por las restricciones y diferencias entre sobrecargar funciones normales y redefinir funciones virtuales, el termino 'overriding' es usado para describir la redefinicion de una funcion virtual.

FUNCIONES VIRTUALES PURAS Y CLASES ABSTRACTAS

Una funcion virtual que no es redefinida en una clase derivada es llamada por un objeto de esa clase derivada, la version de la funcion como se ha definido en la clase base es utilizada. Sin embargo, en muchas circunstancias, no habra una declaración con significado en una funcion virtual dentro de la clase base.

Por ejemplo, en la clase base 'figura' usada en el ejemplo anterior, la definicion de 'mostrar_area()' es simplemente un sustituto sintetico. No computara ni mostrara el area de ningún tipo de objeto. Como vera cuando cree sus propias librerias de clases no es poco comun para una funcion virtual tener una definicion sin significado en el contexto de su clase base.

Cuando esta situacion ocurre, hay dos manera en que puede manejarla. Una manera, como se muestra en el ejemplo, es simplemente hacer que la funcion reporte un mensaje de advertencia. Aunque esto puede ser util en ocasiones, no es el apropiado en muchas circunstancias. Por ejemplo, puede haber funciones virtuales que simplemente deben ser definidas por la clase derivada para que la clase derivada tenga algún significado.

Considere la clase 'triangulo'. Esta no tendria significado si 'mostrar_area()' no se encuentra definida. En este caso, usted desea algun metodo para asegurarse de que una clase derivada, de hecho, defina todas las funciones necesarias. En C++, la solucion a este problema es la 'funcion virtual pura.' Una 'funcion virtual pura' es una función declarada en una clase base que no tiene definicion relativa a la base.

Como resultado, cualquier tipo derivado debe definir su propia versión -- esta simplemente no puede usar la version definida en la base. Para declarar una funcion virtual pura use esta forma general:

virtual 'tipo' 'nombre_de_funcion'(lista_de_parametros) = 0; Aqui, 'tipo' es el tipo de retorno de la funcion y 'nombre_de_funcion' es el nombre de la funcion. Es el = 0 que designa la funcion virtual como pura. Por ejemplo, la siguiente version de 'figura', 'mostrar_area()' es una funcion virtual pura.

3.2.- Operadores.
Sobrecargando funciones y operadores.

La sobrecarga es uno de los mecanismos más utilizados del C++ pues reporta una gran cantidad de beneficios a la hora de diseñar las prestaciones de nuestras funciones de miembro. Existen dos tipos fundamentales de sobrecarga: la sobrecarga de funciones y la sobrecarga de operadores.

Sobrecarga de funciones

La sobrecarga de funciones consiste, básicamente, en crear funciones con el mismo nombre dentro de una clase pero con distinto tipo de argumentos de tal forma que, al llamar a la función el compilador se encargará de escoger la adecuada mediante la comparación de la lista de argumentos pasados en la invocación. Un sencillo ejemplo sería pensar en dos funciones llamadas *ImprimirMensaje* Podemos definirlas así:

```
void ImprimirMensaje(void);
void ImprimirMensaje (char *texto);
```

Como podéis observar, se llaman igual pero tienen argumentos distintos. Mientras que la primera no está preparada para recibir ningún tipo de datos, la segunda sí que lo está y esperará recibir una cadena de caracteres. Si ahora implementamos las funciones de esta forma:

```
void NombreClase::ImprimirMensaje(void)
{
        cout << "\n Esta es la función sin argumentos";
}

void NombreClase::ImprimeMensaje(char *texto)
{
        cout << "\ n Esta es la función con argumentos
                y ha recibido el mensaje " << texto;
}
```

Si creáramos una instancia a la clase que contiene a esas dos funciones y llamáramos a la función

ImprimeMensaje();
Obtendríamos por pantalla:

Esta es la función sin argumentos

Si, con esa misma instancia, llamáramos a la función

ImprimeMensaje("Macedonia Magazine");
Lo que obtendríamos sería:

Esta es la función con argumentos y ha recibido el
mensaje Macedonia Magazine

Como podéis observar, es muy sencillo sobrecargar funciones (más que sobrecargar operadores) y pueden ofrecer, sin un coste computacional demasiado elevado, prestaciones excelentes para trabajar dada la gran flexibilidad que proporcionan al programador. He aquí el ejemplo codificado formalmente:

```
#include <iostream.h>

class SobrecargaFunciones
{
public:
void ImprimeMensaje(void);
void Imprime Mensaje(char* texto);
};

void SobrecargaFunciones::ImprimirMensaje(void)
{
        cout << "\n Esta es la función sin argumentos";
}

void SobrecargaFunciones::ImprimeMensaje(char *texto)
{
        cout << "\n Esta es la función con argumentos
                y ha recibido el mensaje " << texto;
}

int main(void)
{
SobrecargaFunciones sobrecarga;

sobrecarga.ImprimeMensaje();
sobrecargaImprimeMensaje("Macedonia Magazine");
}
```

Sobrecarga y constructores

La sobrecarga de constructores se presenta como una de las aplicaciones más directas y comunes de aplicación de este mecanismo del C++, es más, la sobrecarga de constructores es algo que está "a la orden del día" para cualquier programador de C++ pues permite inicializar las instancias de muy distintas formas y dotar al objeto de unos valores iniciales acordes a la finalidad a la que vamos a llevar dicho objeto. Es muy común, pues, ver clases con muchos constructores. Por ejemplo:

```
class Personaje
{
public:
        Personaje(char *nombre);
        Personaje();
}
```

Aquí, podríamos decidir inicializar a la instancia con un nombre que el usuario ha introducido por teclado o bien, crear nosotros la instancia Personaje con un nombre por defecto llamando al constructor sin ningún tipo de argumentos.

Sobrecarga de operadores

La sobrecarga de operadores no es un mecanismo que los programadores de C++ recién iniciados se aventuren a utilizar ya que tardan un tiempo en "pillarle el truco" pero es masivamente utilizado en cualquier biblioteca de C++ como la **MFC** de Microsoft.

Pero, ¿en qué consiste una sobrecarga de operadores?. Sobrecargar operadores sirve para que las operaciones tales como la suma (+), resta (-), multiplicación (*), asignación (=), incremento (--), etc se comporten de forma diferente al trabajar con nuestros objetos, más exactamente, con los objetos preparados para soportar la sobrecarga de operadores. Imaginar que tenemos dos objetos de tipo *Cadena*. La clase *Cadena* está preparada para manejar una cadena de caracteres y darla funcionalidad a través de diversos métodos. Si nosotros quisiéramos sumar dos objetos de tipo *Cadena*, bastaría con tener sobrecargado el operador suma (+) para los objetos de dicha clase con lo que la siguiente sentencia sería válida:

```
Cadena cadena1, cadena2, cadenaResult;
cadenaResult = cadena1 + cadena2;
```

Con esto podríamos conseguir, por ejemplo, que el objeto *cadenaResult* almacenara la concatenación de las cadenas de caracteres que soportan los objetos *cadena1* y *cadena2* respectivamente.

Cómo implementar la sobrecarga de operadores

Para sobrecargar un operador, debemos de definir una función de sobrecarga de operador en la clase que nos interese (aunque también la podemos hacer de carácter global... y eso podría traer algún que otro "galimatías" si no tenéis cuidado). Un operador que se sobrecarga se define siempre con el nombre de la clase seguido de la palabra clave *operator* y del operador. Después del operador pondremos, entre paréntesis, los tipos de datos con los que queremos que funcione nuestro operador. Por ejemplo, si quisiéramos sobrecargar el operador suma de la clase Cadena, deberíamos de incluir, en la definición de la clase, la siguiente definición:

Cadena **operator**+(Cadena);

Con esto estaríamos declarando que estamos sobrecargando el operador suma (+) para que sea capaz de trabajar con argumentos (sumandos) que sean instancias a la clase cadena. Obviamente, para que esto funcione en la parte izquierda del operador asignación debe de haber otro objeto de tipo de Cadena, es decir, no podemos hacer esto:

```
int valor;
valor = cadena1 + cadena2;
```

Recordemos que estamos sobrecargando el operador suma para que trabaje con objetos que sean instancias de la clase Cadena.

3.3.- Clases abstractas y herencia.

CLASES ABSTRACTAS E INTERFACES

La posibilidad de definir relaciones de herencia entre clases, dando lugar a relaciones de subtipado entre las mismas, nos ha permitido en el Tema 3 definir el polimorfismo de métodos (es decir, que un mismo método tuviese distintas definiciones, y además los objetos fuesen capaces de acceder a la definición del método adecuada en tiempo de ejecución).

Estas mismas relaciones de subtipado entre clases daban lugar a una segunda situación menos deseable. Siempre que declaramos un objeto como perteneciente a un tipo (por ejemplo, al declarar una estructura genérica o al definir funciones auxiliares), restringimos la lista de métodos (o la interfaz) que podemos utilizar de dicho objeto a los propios de la clase declarada. En este caso estábamos perdiendo información (en la forma de métodos a los que poder acceder) que podrían sernos de utilidad.

Una posible solución a este problema la ofrecen los métodos abstractos. Un método abstracto nos da la posibilidad de introducir la declaración de un método (y no su definición) en una clase, e implementar dicho método en alguna de las subclases de la clase en que nos encontramos.

De este modo, la declaración del método estará disponible en la clase, y lo podremos utilizar para, por ejemplo, definir otros métodos, y además no nos vemos en la obligación de definirlo, ya que su comportamiento puede que sea todavía desconocido.

Una consecuencia de definir un método abstracto es que la clase correspondiente ha de ser también abstracta, o, lo que es lo mismo, no se podrán crear objetos de la misma (¿cómo sería el comportamiento de los objetos de la misma al invocar a los métodos abstractos?), pero su utilidad se observará al construir objetos de las clases derivadas.

Además, la posibilidad de declarar métodos abstractos enriquecerá las jerarquías de clases, ya que más clases podrán compartir la declaración de un método (aunque no compartan su definición). La idea de un método abstracto puede ser fácilmente generalizada a clases que sólo contengan métodos abstractos, y nos servirán para relacionar clases (por relaciones de subtipado) que tengan una interfaz (o una serie de métodos) común (aunque las definiciones de los mismos sean diferentes en cada una de las subclases).

El presente Tema comenzará con la Sección 4.1 (“Definición de métodos abstractos en POO. Algunos ejemplos de uso”) donde presentaremos más concretamente la noción de método abstracto e ilustraremos algunas situaciones en las que los mismos pueden ser de utilidad; la propia noción de método abstracto nos llevará a la noción de clase abstracta que introduciremos en la misma Sección.

La Sección 4.2 nos servirá para repasar la idea de l polimorfismo y para mostrar la dependencia de los métodos abstractos en la presencia del mismo; aprovecharemos también para introducir las sintaxis propias de Java y C++ de métodos y clases abstractas.

En la Sección 4.3 mostraremos cómo se puede generalizar la noción de método abstracto para llegar a la de clase completamente abstracta. También presentaremos la noción de “interface” en Java, partiendo de la idea de que una “interface” está basada en la presencia de métodos abstractos, pero poniendo énfasis también en el hecho de que en Java las “interfaces” permiten implementar ciertas situaciones que a través de clases abstractas no serían posibles.

La Sección 4.4 nos servirá para repasar la notación propia de UML para los conceptos introducidos en el Tema (aunque ya la habremos introducido antes), del mismo modo que la Sección 4.5 y la Sección 4.6 las utilizaremos para recuperar las sintaxis propias de dichas nociones en C++ y Java.

MÉTODOS ABSTRACTOS: DEFINICIÓN Y NOTACIÓN UML Definición: un método abstracto es un método de una clase (o también de una "interface" en Java) que no tiene implementación o definición (es decir, sólo tiene declaración). Sus principales usos son, en primer lugar, como un "parámetro indefinido" en expresiones que contienen objetos de dicha clase, que debe ser redefinido en alguna de las subclases que heredan de dicha clase (o que implementan la "interface"). En segundo lugar, sirve para definir "interfaces abstractas" de clases (entendido como partes públicas de las mismas) que deberán ser definidas por las subclases de las mismas. Por tanto, lo primero que debemos observar es que al hablar de métodos abstractos hemos tenido que recuperar las nociones de herencia y subclases (Secciones 2.3, y ss.), así como de redefinición de métodos (Sección 2.6), e, implícitamente, de polimorfismo (Tema 3).

CLASES ABSTRACTAS: DEFINICIÓN Y VENTAJAS DE USO Definición: una clase abstracta es una clase de la cual no se pueden definir instancias (u objetos). Por tanto, las clases abstractas tendrán dos utilidades principales: 1. En primer lugar, evitan que los usuarios de la clase puedan crear objetos de la misma, como dice la definición de clase abstracta. De este modo, en nuestro ejemplo anterior, no se podrán crear instancias de la clase "Articulo". Éste es 4 un comportamiento deseado, ya que si bien "Articulo" nos permite crear una jerarquía sobre las clases "Tipo4", "Tipo7" y "Tipo16", un objeto de la clase "Articulo" como tal no va a aparecer en nuestro desarrollo (todos los artículos tendrán siempre un IVA asignado). Sin embargo, es importante notar que sí se pueden declarar objetos de la clase "Articulo" (que luego deberán se construidos como de las clases "Tipo4", "Tipo7" ó "Tipo16"). 2.

En segundo lugar, permiten crear interfaces que luego deben ser implementados por las clases que hereden de la clase abstracta. Es evidente que una clase abstracta, al no poder ser instanciada, no tiene sentido hasta que una serie de clases que heredan de ella la implementan completamente y le dan un significado a todos sus métodos. A estas clases, que son las que hemos utilizado a lo largo de todo el curso, las podemos nombrar clases concretas para diferenciarlas de las clases abstractas.

De la propia definición de clase abstracta no se sigue que una clase abstracta deba contener algún método abstracto, aunque generalmente será así. En realidad, el hecho de definir una clase cualquiera como abstracta se puede entender como un forma de evitar que los usuarios finales de la misma puedan crear objetos de ella; es como una medida de protección que el programador de una clase pone sobre la misma. Sin embargo, lo contrario sí es cierto siempre: si una clase contiene un método abstracto, dicha clase debe ser declarada como abstracta. Si hemos declarado un método abstracto en una clase, no podremos construir objetos de dicha clase (ya que, ¿cuál sería el comportamiento de dicho método al ser invocado desde un objeto de la clase? Estaría sin especificar, o sin definir).

Por lo tanto, como resumen a los dos anteriores párrafos, si bien que un método esté declarado como abstracto implica que la clase en la que se encuentra debe ser declarada como abstracta (en caso contrario obtendremos un error de compilación), que una clase sea abstracta no implica que alguno de los métodos que contiene haya de serlo (únicamente implica que no se pueden crear objetos de la misma).

Por este motivo, en el ejemplo anterior, el hecho de declarar el método "getParteIVA(): double" como abstracto tiene como consecuencia que la clase "Articulo" deba ser definida como abstracta. La notación UML para clases abstractas consiste en escribir en letra cursiva el nombre de dicha clase, como se puede observar en el diagrama anterior (en nuestro ejemplo, "Articulo"). Por lo tanto, ya no podremos crear objetos de la clase "Articulo" en nuestra aplicación. Sin embargo, aunque una clase sea abstracta, podemos observar cómo puede contener atributos ("nombre: string" y "precio: double"), constructores ("Articulo(string, double)") o métodos no abstractos ("getNombre(): string", ...). 5 Una vez más, insistimos en que la única consecuencia de declarar una clase como abstracta es que evitamos que los usuarios de la clase puedan definir instancias de la misma (aunque sí pueden declararlas). Salvo esto, es una clase que puede contener atributos, constantes, constructores y métodos (tanto abstractos como no abstractos).

La siguiente pregunta podría ser formulada: ¿Qué utilidad tiene un constructor de una clase abstracta, si no se pueden crear objetos de la misma? La respuesta a dicha pregunta es doble: 1. En primer lugar, para inicializar los atributos que pueda contener la clase abstracta (en nuestro ejemplo anterior, el "nombre: string", o el "precio: double"). 2. En segundo lugar, y volviendo a uno de los aspectos que enfatizamos al introducir las relaciones de herencia ("La primera orden que debe contener un constructor de una clase derivada es una llamada al constructor de la clase base"), el constructor de una clase abstracta será de utilidad para que lo invoquen todos los constructores de la las clases derivadas. En realidad, éstas deberían ser las únicas invocaciones a los mismos que contuviera nuestro sistema, ya que no se pueden crear objetos de las clases abstractas.

3.4.- Abstracción de generalización y especialización de clases.

AUMENTANDO LA REUTILIZACIÓN DE CÓDIGO GRACIAS A LOS MÉTODOS ABSTRACTOS

Como ventajas de uso de las clases abstractas hemos señalado ya que permiten al programador decidir qué clases van a poder ser instanciables (se van a poder crear objetos de ellas) y cuáles no (es decir, van a servir sólo para hacer de soporte para programar nuevas clases por herencia). También hemos señalado que los métodos abstractos nos permiten declarar métodos sin tener que definirlos, y de este modo "enriquecer" la parte visible ("public", "protected" o "package") de una clase, dotándola de más métodos (que no es necesario definir hasta más adelante).

Estos métodos declarados como abstractos pueden ser también utilizados para definir los métodos restantes de la clase abstracta, permitiéndonos así reutilizar código para diversas clases. Veámoslo con un ejemplo. Gracias a la declaración del método "getParteIVA(): double" (como método abstracto) dentro de la clase "Articulo", hemos enriquecido la lista de métodos disponibles en dicha clase. Esto nos permite dar una nueva definición ahora para alguno de los métodos de la clase "Articulo" que haga uso de los métodos abstractos añadidos ("getParteIVA(): double").

En nuestro caso concreto, vamos a empezar por observar la definición que habíamos dado del método "getPrecio(): double" en las clases "Articulo", "Tipo4", "Tipo7" y "Tipo16" antes de declarar la clase 6 "Articulo" como abstracta (mostramos la definición de los mismos en Java, que no difiere de la que se podría dar en C++ salvo los detalles propios de la sintaxis de cada lenguaje):

```
//Clase Articulo public double getPrecio (){ return this.precio; } //Clase Tipo4 public double getPrecio (){ return (super.getPrecio() + this.getParteIVA()); } //Clase Tipo7 public double getPrecio (){ return (super.getPrecio() + this.getParteIVA()); } //Clase Tipo16 public double getPrecio (){ return (super.getPrecio() + this.getParteIVA()); }
```

Los siguientes comentarios surgen al observar las anteriores definiciones: 1. La definición del método "getPrecio(): double" en la clase "Articulo" no resulta de especial utilidad, ya que cualquier objeto que utilicemos en nuestra aplicación pertenecerá a una de las clases "Tipo4", "Tipo7" ó "Tipo16". Esto resultará más obvio cuando declaremos la clase "Articulo" como abstracta y el método "getPrecio(): double" propio de la misma sólo pueda ser accedido desde las subclases (en nuestra aplicación no aparecerán objetos propios de la clase "Articulo") 2. El segundo hecho que podemos resaltar es que el método "getPrecio(): double" ha sido definido en las clases "Tipo4", "Tipo7" y "Tipo16" del mismo modo, es decir, accediendo al valor del atributo "precio: double" de la clase "Articulo" (a través del método de acceso "getPrecio(): double" de la clase "Articulo") y sumándole a dicha cantidad el resultado de llamar al método "getParteIVA(): double".

En nuestro nuevo diagrama de clases, al declarar "getParteIVA(): double" como método abstracto, este método aparece en a clase (abstracta) "Articulo": 7 +Articulo(entrada : string, entrada : double) +getNombre() : string +setNombre(entrada : string) : void +getPrecio() : double +getParteIVA() : double Por tanto, dicho método va a ser visible para los métodos restantes de la clase "Articulo", y lo pueden utilizar en sus definiciones. Realizamos ahora una nueva modificación sobre la misma, añadiendo un método abstracto "getTIPO(): double", que será definido en "Tipo4", "Tipo7" y "Tipo16" como un método de acceso a la constante de clase "TIPO: double". Este método no tiene por qué ser visible para los usuarios externos de la clase, por lo cual le añadimos el modificador de acceso "protected" (es suficiente con que sea visible en la clase y subclases).

Obtenemos el siguiente diagrama de clases UML para nuestra aplicación:
+Articulo(entrada : string, entrada : double)
+getNombre() : string +setNombre(entrada : string) : void
+getPrecio() : double +getParteIVA() : double #getTIPO() : double -nombre : string -precio : double Articulo +Tipo4(entrada : string, entrada : double)
+getPrecio() : double +getParteIVA() : double #getTIPO() : double -TIPO : double = 4 Tipo4
+Tipo7(entrada : string, entrada : double)
+getPrecio() : double
+getParteIVA() : double #getTIPO() : double -TIPO : double = 7 Tipo7
+Tipo16(entrada : string, entrada : double)
+getPrecio() : doublé
+getParteIVA() : double #getTIPO() : double -TIPO : double = 16 Tipo16 Pero ahora, sobre el diagrama anterior, podemos observar que el método "getPrecio(): double" admite la siguiente definición en la clase abstracta "Articulo": //Clase Articulo public double getPrecio (){ return (this.precio + this.getParteIVA()); }

La anterior definición es válida para las tres clases "Tipo4", "Tipo7" y "Tipo16", ya que tiene en cuenta el precio base de un artículo así como la parte correspondiente a su IVA.

De modo similar, podemos proponer ahora una definición unificada para el método "getParteIVA(): double" (por lo cual deja de ser abstracto) en la propia clase "Articulo" que sea válida para las clases "Tipo4", "Tipo7" y "Tipo16": //Clase Articulo public double getParteIVA (){ return (this.precio * this->getTIPO() / 100); 8 } Esta definición del método hace uso de un método abstracto ("getTIPO(): double") que en la clase "Articulo" no ha sido definido, sólo declarado. Esta situación no es "peligrosa" (desde el punto de vista del compilador) siempre y cuando no se puedan construir objetos de la clase "Articulo", ya que para esos objetos no habría un método definido "getTIPO(): double", pero como ya hemos comentado, el compilador nos asegura que no se puedan construir objetos de la clase "Articulo" (sólo de las clases "Tipo4", "Tipo7" y "Tipo16"). Por tanto, haciendo uso de las definiciones anteriores de "getParteIVA(): double", de "getPrecio(): double" y del método "getTIPO(): double", el nuevo diagrama de clases en UML se podría simplificar al siguiente:

+Articulo(entrada : string, entrada : double) +getNombre() : string +setNombre(entrada : string) : void +getPrecio() : double +getParteIVA() : double #getTIPO() : double -nombre : string -precio : double Articulo +Tipo4(entrada : string, entrada : double) #getTIPO() : double -TIPO : double = 4 Tipo4 +Tipo7(entrada : string, entrada : double) #getTIPO() : double -TIPO : double = 7 Tipo7 +Tipo16(entrada : string, entrada : double) #getTIPO() : double -TIPO : double = 16 Tipo16

Podemos hacer dos comentarios breves sobre el diagrama anterior: 1. En primer lugar, podemos observar como el número de métodos que aparecen en el mismo (excluyendo los constructores, tenemos 8 métodos, uno de ellos abstracto) es menor que el que aparecía en nuestra versión original del mismo sin hacer uso de clases y método abstractos (en aquel aparecían 9 métodos excluyendo los constructores). Esta diferencia sería aún mayor si tenemos en cuenta que las clases "Tipo4", "Tipo7" y "Tipo16" anteriormente contenían 2

métodos (“getPrecio(): double” y “getParteIVA(): double”) aparte de los constructores, mientras que ahora sólo aparece uno (“getTIPO(): double”). Cuanto mayor sea el número de clases que hereden de “Articulo”, mayor será el número de métodos que nos evitemos de redefinir. Por tanto, hemos simplificado nuestra aplicación, y además hemos conseguido reutilizar código, ya que métodos que antes poseían igual definición (“getPrecio(): double” en “Tipo4”, “Tipo7” y “Tipo16” y “getParteIVA(): double” en “Tipo4”, “Tipo7” y “Tipo16”) ahora han pasado a estar definidos una sola vez.

Desde el punto de vista de mantenimiento del código, hemos simplificado también nuestra aplicación. 2. En segundo lugar, el hecho de utilizar métodos y clases abstractas nos ha permitido mejorar el diseño de nuestro sistema de información. Primero, hemos podido hacer que la clase “Articulo” fuese definida como abstracta, impidiendo así que se creen objetos de la misma. En segundo lugar, el uso de métodos abstractos (“getTIPO(): double”) nos ha permitido cambiar la definición de 9 algunos otros métodos (finalmente de “getPrecio(): double” y “getParteIVA(): double”), reduciendo el número de métodos en nuestro sistema de información y simplificando el diseño del mismo.

En la Sección siguiente, a la vez que explicamos la necesidad del polimorfismo para poder hacer uso de métodos abstractos en nuestras aplicaciones, aprovecharemos para introducir la notación propia de C++ y Java con respecto al mismo.

3.5.- Clases abstractas.

NECESIDAD DEL POLIMORFISMO PARA EL USO DE MÉTODOS ABSTRACTOS

De las explicaciones y diagramas de clases anteriores en UML se puede extraer una conclusión inmediata. Para poder hacer uso de métodos abstractos, es estrictamente necesaria la presencia de polimorfismo de métodos, o, lo que es lo mismo, de enlazado dinámico de los mismos.

Imaginemos que no disponemos de enlazado dinámico (es decir, en enlazado en nuestro compilador es estático, y en tiempo de compilación a cada objeto se le asignan las definiciones de los métodos de los que hará uso). Supongamos que, haciendo uso del último diagrama de clases que hemos presentado en la Sección 4.1.3, realizamos la siguiente declaración (en Java, en C++ debería ser un puntero): Articulo art1; La anterior declaración puede ser hecha ya que, aun siendo “Articulo” una clase abstracta, se pueden declarar (no construir) objetos de dicha clase.

En condiciones de enlazado estático (de falta de polimorfismo), el compilador le habría asignado al objeto “art1”, independientemente de cómo se construya el mismo, las definiciones de los métodos que se pueden encontrar en la clase “Articulo”. Es decir, en el caso del método “getTIPO(): double”, se le habría asignado al objeto “art1” un método abstracto, sin definición. Por tanto, si queremos que nuestra aplicación se comporte de una forma coherente, todos los métodos abstractos deben ser polimorfos (y realizar enlazado

dinámico). Conviene recordar ahora que, si bien en Java (como introdujimos en la Sección 3.3), el compilador siempre realiza enlazado dinámico de métodos, y todos los métodos se comportan de modo polimorfo, en C++ (Sección 3.2), para que un método se comporte de manera polimorfa, éste debe ser declarado (en el archivo de cabeceras correspondiente) como "virtual", y el objeto desde el que se invoque al método debe estar alojado en memoria dinámica (es decir por medio de un puntero o referencia). Así que todos los métodos que sean declarados en Java y en C++ como abstractos, deberán satisfacer, al menos, los requisitos de los métodos polimorfos.

En lo que queda de esta Sección veremos qué más requisitos deben cumplir. 10 Antes de pasar a ver la sintaxis propia de Java y C++ para definir clases y métodos abstractos, conviene remarcar una diferencia sustancial entre ambos a la hora de considerar una clase como abstracta: En C++, una clase es abstracta si (y sólo si) contiene al menos un método abstracto. Por ser abstracta, no podremos declarar construir objetos de la misma. En Java, una clase es abstracta si (y sólo si) contiene en su cabecera el modificador "abstract".

Si contiene algún método abstracto, deberemos declarar también la clase con el modificador "abstract". Pero, a diferencia de C++, existe la posibilidad de que una clase no contenga ningún método abstracto, y sin embargo sea declarada como abstracta (haciendo uso del modificador "abstract"). De igual modo que en C++, el hecho de ser abstracta implica que no podremos crear objetos de la misma. 4.2.2 SINTAXIS DE MÉTODOS Y CLASES ABSTRACTAS EN C++ Pasemos a ilustrar la sintaxis propia de métodos y clases abstractas en C++. Para ello de nuevo retomamos el ejemplo de la clase "Articulo" y las clases "Tipo4", "Tipo7" y "Tipo16".

En primer lugar, veamos la codificación del mismo con respecto al siguiente diagrama UML (es decir, sin hacer uso de métodos ni clases abstractas):

+Articulo(entrada : string, entrada : double) +getNombre() : string
+setNombre(entrada : string) : void
+getPrecio() : double -nombre : string -precio : double Articulo
+Tipo4(entrada : string, entrada : double)
+getPrecio() : double +getParteIVA() : double -TIPO : double = 4 Tipo4
+Tipo7(entrada : string, entrada : double) +getPrecio() : double +getParteIVA() : double -TIPO : double = 7 Tipo7 +Tipo16(entrada : string, entrada : double) +getPrecio() : double +getParteIVA() : double -TIPO : double = 16 Tipo16 El código en C++ correspondiente al diagrama de clases sería el siguiente (omitimos el programa principal "main" por el momento): //Fichero Articulo.h #ifndef ARTICULO_H #define ARTICULO_H 1 class

Articulo{ private: char nombre [30]; double precio; public: 11 Articulo(char [], double); char * getNombre(); void setNombre(char []); virtual double getPrecio(); }; #endif //Fichero Articulo.cpp #include #include "Articulo.h" using namespace std; Articulo::Articulo(char nombre [], double precio){ strcpy (this->nombre, nombre); this->precio = precio; }; char *

Articulo::getNombre(){ return this->nombre; }; void Articulo::setNombre(char nuevo_nombre[]){ strcpy (this->nombre, nuevo_nombre); }; double Articulo::getPrecio(){ return this->precio; }; //Fichero Tipo4.h #ifndef TIPO4_H #define TIPO4_H #include "Articulo.h" class Tipo4: public Articulo{ private: const static double TIPO = 4.0; public:

Tipo4 (char [], double); double getPrecio(); double getParteIVA(); }; #endif 12 //Fichero Tipo4.cpp #include "Tipo4.h" Tipo4::Tipo4 (char nombre [], double precio): Articulo(nombre, precio){ }; double Tipo4::getPrecio(){ return (Articulo::getPrecio() + this->getParteIVA()); }; double Tipo4::getParteIVA(){ return (Articulo::getPrecio() * TIPO / 100); }; //Fichero Tipo7.h #ifndef TIPO7_H #define TIPO7_H #include "Articulo.h" class Tipo7:

public Articulo{ private: const static double TIPO = 7.0; public: Tipo7 (char [], double); double getPrecio(); double getParteIVA(); }; #endif //Fichero Tipo7.cpp #include "Tipo7.h" Tipo7::Tipo7 (char nombre [], double precio): Articulo(nombre, precio){ }; double Tipo7::getPrecio(){ return (Articulo::getPrecio() + this->getParteIVA()); }; double Tipo7::getParteIVA(){ return (Articulo::getPrecio() * TIPO / 100); }; //Fichero Tipo16.h 13 #ifndef TIPO16_H #define TIPO16_H #include "Articulo.h" class Tipo16: public Articulo{ private: const static double TIPO = 16.0; public: Tipo16 (char [], double); double getPrecio(); double getParteIVA(); }; #endif //Fichero Tipo16.cpp #include "Tipo16.h" Tipo16::Tipo16 (char nombre [], double precio): Articulo(nombre, precio){ }; double Tipo16::getPrecio(){ return (Articulo::getPrecio() + this->getParteIVA()); }; double Tipo16::getParteIVA(){ return (Articulo::getPrecio() * TIPO / 100); };

Las principales peculiaridades que se pueden observar en los ficheros anteriores han sido la necesidad de declarar el método “getPrecio(): double” como “virtual” en el fichero de cabeceras “Articulo.h”, y las llamadas desde unas clases a los métodos de otras (en particular, las llamadas al método “Articulo::getPrecio()” y las llamadas al constructor “Articulo(char [], double)” desde cada uno de los constructores de “Tipo4”, “Tipo7” y “Tipo16”). La primera modificación que propusimos en la Sección 4.1.1 consistía en definir el método “getParteIVA(): double” como abstracto e introducirlo en la clase “Articulo” (con lo cual, esta clase pasaría a ser también abstracta).

Veamos cómo quedaría tras esas modificaciones el fichero de cabeceras “Articulo.h”: #ifndef ARTICULO_H #define ARTICULO_H 1 class Articulo{ private: char nombre [30]; double precio; 14 public: Articulo(char [], double); char * getNombre(); void setNombre(char []); virtual double getPrecio(); virtual double getParteIVA()=0; }; #endif Pasamos a realizar algunos comentarios sobre el archivo de cabeceras “Articulo.h”: 1. En primer lugar, debemos destacar que el único archivo que ha sufrido modificación al declarar “getParteIVA(): double” como método virtual ha sido el fichero de cabeceras “Articulo.h”. Es más, la única modificación que ha sufrido este archivo es que ahora incluye la declaración “virtual double getParteIVA()=0;” Esto quiere decir que la clase “Articulo” no ha recibido ningún modificador adicional (veremos que esto no es así en Java), y que las clases restantes tampoco.

Simplemente debemos observar que las clases que redefinan “getParteIVA(): double” deben incluir en su archivo de cabeceras la declaración del mismo. 2. En segundo lugar, convendría observar más detenidamente la declaración del método: “virtual double getParteIVA()=0;” Podemos observar como el mismo incluye el modificador “virtual” que avisa al compilador de que dicho método será redefinido (y que ya introdujimos en la Sección 3.2). Recordamos que la declaración del método deberá ser incluida en todas las clases que lo redefinan (igual que sucede en los diagramas UML y en Java). En segundo lugar, conviene observar la “definición” del método.

Como el método es abstracto, y no va a ser definido en esta clase (en el fichero “Articulo.cpp”), se lo advertimos al compilador asignándole la “definición” “double getParteIVA()=0;”. El compilador así entiende que este método es abstracto, que por tanto su definición no va a aparecer en el fichero “Articulo.cpp”, y que serán las clases derivadas las que se encarguen de definirlo.

Veremos cómo en Java la notación es distinta. Para concluir, veamos ahora qué sucede cuando intentamos construir un objeto de la clase “Articulo”: #include #include "Articulo.h" #include "Tipo4.h" 15 #include "Tipo7.h" #include "Tipo16.h" using namespace std; int main (){ Articulo arti ("La historia interminable", 9); arti.getParteIVA(); Articulo * art1; art1 = new Articulo ("La historia Interminable", 9); system ("PAUSE"); return 0; } El anterior fragmento de código produce dos errores de compilación, como ya mencionamos en la Sección 4.1.2, advirtiéndonos de que no podemos construir un objeto de la clase “Articulo” (ni tampoco a través de punteros) ya que la misma contiene métodos que son abstractos (en este caso, “getParteIVA(): double”), y por tanto es abstracta.

El constructor de la clase “Articulo” pasa a tener utilidad únicamente para ser invocado desde los constructores de “Tipo4”, “Tipo7” y “Tipo16”, que siguen haciendo uso del mismo. Veamos ahora cómo quedaría la implementación en C++ del diagrama de clases en el que introdujimos un nuevo método abstracto “getTIPO(): double”, y los métodos “getPrecio(): double” y “getParteIVA(): double” pasaron a estar definidos en la clase abstracta

“Articulo”: +Articulo(entrada : string, entrada : double) +getNombre() : string +setNombre(entrada : string) : void +getPrecio() : double +getParteIVA() : double #getTIPO() : double -nombre : string -precio : double Articulo +Tipo4(entrada : string, entrada : double) #getTIPO() : double -TIPO : double = 4 Tipo4 +Tipo7(entrada : string, entrada : double) #getTIPO() : double -TIPO : double = 7 Tipo7 +Tipo16(entrada : string, entrada : double) #getTIPO() : double -TIPO : double = 16 Tipo16 //Fichero Articulo.h #ifndef ARTICULO_H #define ARTICULO_H 1 class Articulo{ private: char nombre [30]; 16 double precio; public: Articulo(char [], double); char * getNombre(); void setNombre(char []); double getPrecio(); double getParteIVA(); protected: virtual double getTIPO() = 0; }; #endif //Fichero Articulo.cpp #include #include "Articulo.h" using namespace std; Articulo::Articulo(char nombre [], double precio){ strcpy (this->nombre, nombre); this->precio = precio; }; char * Articulo::getNombre(){ return this->nombre; }; void Articulo::setNombre(char nuevo_nombre[]){ strcpy (this->nombre, nuevo_nombre); }; double Articulo::getPrecio(){ return this->precio + this->getParteIVA(); }; double

Articulo::getParteIVA(){ return this->precio * this->getTIPO()/100; }; //Fichero Tipo4.h #ifndef TIPO4_H #define TIPO4_H #include "Articulo.h" class Tipo4: public Articulo{ 17 private: const static double TIPO = 4.0; public: Tipo4 (char [], double); protected: double getTIPO(); }; #endif //Fichero Tipo4.cpp #include "Tipo4.h" Tipo4::Tipo4 (char nombre [], double precio): Articulo(nombre, precio){ }; double Tipo4::getTIPO(){ return (this->TIPO); }; //Fichero Tipo7.h #ifndef TIPO7_H #define TIPO7_H #include "Articulo.h" class Tipo7:

public Articulo{ private: const static double TIPO = 7.0; public: Tipo7 (char [], double); protected: double getTIPO(); }; #endif //Fichero Tipo7.cpp #include "Tipo7.h" Tipo7::Tipo7 (char nombre [], double precio): Articulo(nombre, precio){ }; double Tipo7::getTIPO(){ return (this->TIPO); }; 18 //Fichero Tipo16.h #ifndef TIPO16_H #define TIPO16_H #include "Articulo.h" class Tipo16: public Articulo{ private: const static double TIPO = 16.0; public:

Tipo16 (char [], double); protected: double getTIPO(); }; #endif //Fichero Tipo16.cpp #include "Tipo16.h" Tipo16::Tipo16 (char nombre [], double precio): Articulo(nombre, precio){ }; double Tipo16::getTIPO(){ return (this->TIPO); }; Algunos comentarios sobre el código anterior: 1. Conviene resaltar de nuevo la notación específica de C++ para declarar métodos abstractos (en este caso "getTIPO(): double"): "virtual double getTIPO() = 0;" Un método abstracto debe incluir el modificador "virtual" que nos permitirá acceder a él de modo polimorfo, así como la declaración "double getTIPO() = 0;" advirtiendo de que el mismo es abstracto. 2. En segundo lugar, destacar la definición que de los métodos "getPrecio(): double" y "getParteIVA(): double" hemos podido realizar en el nuevo entorno creado: double Articulo::getPrecio(){ return this->precio + this->getParteIVA(); }; double

Articulo::getParteIVA(){ return this->precio * this->getTIPO()/100; 19 }; Vemos que cualquiera de los dos ("getParteIVA(): double" directamente y "getPrecio(): double" indirectamente a través del primero) acceden al método (abstracto) "getTIPO(): double", cuya definición no ha sido dada todavía (y lo será en alguna de las clases derivadas). Aquí es donde el compilador debe desarrollar la tarea de asegurar que no podamos construir objetos de la clase abstracta "Articulo", ya que para los mismos no habría una definición de "getTIPO(): double", y comprobar que en las subclases que definamos de "Articulo" y de las cuales queramos construir objetos, el método "getTIPO(): double" sea definido.

Un programa "main" cliente del anterior sistema de clases podría ser el siguiente: #include #include "Articulo.h" #include "Tipo4.h" #include "Tipo7.h" #include "Tipo16.h" using namespace std; int main (){ Articulo * art1; art1 = new Tipo4 ("La historia Interminable", 9); Tipo7 * art2; art2 = new Tipo7 ("Gafas", 160); Articulo * art3; art3 = new Tipo16 ("Bicicleta", 550); cout << "El precio del primer articulo es " << art1->getPrecio() << endl; cout << "El precio del segundo articulo es " << art2->getPrecio() << endl; cout << "El precio del tercer articulo es " << art3->getPrecio() << endl; system ("PAUSE"); return 0; } En el mismo podemos observar cómo, si bien tiene sentido declarar objetos (o punteros a objetos) de la clase abstracta "Articulo", los mismos siempre son construidos por medio del constructor de alguna de las clases derivadas de "Articulo" en las que todos los métodos son definidos. También es posible observar que, si bien el método "getPrecio(): double" no está redefinido, internamente invoca al método "getTipo(): double" que sí lo está, y por tanto es necesario

que haya comportamiento polimorfo del mismo (de ahí la necesidad de declararlo como "virtual" y de invocarlo desde memoria dinámica).

SINTAXIS DE MÉTODOS Y CLASES ABSTRACTAS EN JAVA

Pasamos ahora a mostrar la sintaxis propia de Java para la definición de métodos abstractos y de clases abstractas. Para lo mismo, recuperamos el mismo ejemplo que en la Sección anterior. Tomamos como punto de partida la definición de las clases "Articulo", "Tipo4", "Tipo7" y "Tipo16" antes de incluir en los mismos métodos y clases abstractas, para poder luego observar mejor las diferencias:

//Fichero Articulo.java public class Articulo{ private String nombre; private double precio; public Articulo(String nombre, double precio){ this.nombre = nombre; this.precio = precio; } public String getNombre (){ return this.nombre; } public void setNombre (String nuevo_nombre){ this.nombre = nuevo_nombre; } public double getPrecio (){ return this.precio; } } //Fichero Tipo4.java public class Tipo4 extends Articulo{ private static final double TIPO = 4.0; public Tipo4(String nombre, double precio){ super (nombre, precio); } public double getPrecio (){ return (super.getPrecio() + this.getParteIVA()); } public double getParteIVA (){ 21 return (super.getPrecio() * TIPO / 100); } } //Fichero Tipo7.java public class Tipo7 extends Articulo{ private static final double TIPO = 7.0; public Tipo7(String nombre, double precio){ super (nombre, precio); } public double getPrecio (){ return (super.getPrecio() + this.getParteIVA()); } public double getParteIVA (){ return (super.getPrecio() * TIPO / 100); } } //Fichero Tipo16.java public class Tipo16 extends Articulo{ private static final double TIPO = 16.0; public Tipo16(String nombre, double precio){ super (nombre, precio); } public double getPrecio (){ return (super.getPrecio() + this.getParteIVA()); } public double getParteIVA (){ return (super.getPrecio() * TIPO / 100); } }

Veamos ahora cómo podemos conseguir que el método "getParteIVA(): double" pase a estar declarado, como método abstracto, en la clase "Articulo", que a consecuencia de este cambio también pasará a ser abstracta: //Fichero Articulo.java public abstract class Articulo{ 22 private String nombre; private double precio; public Articulo(String nombre, double precio){ this.nombre = nombre; this.precio = precio; } public String getNombre (){ return this.nombre; } public void setNombre (String nuevo_nombre){ this.nombre = nuevo_nombre; } public double getPrecio (){ return this.precio; } public abstract double getParteIVA(); }

Veamos en primer lugar las diferencias con respecto a la definición de la clase "Articulo" que teníamos antes: 1. En primer lugar, el método "getParteIVA(): double" ha sido declarado de la forma: "public abstract double getParteIVA();" Podemos observar cómo hemos añadido el modificador "abstract" que nos ha permitido no tener que definir el mismo; sólo hemos declarado su cabecera, sin especificar su comportamiento. 2. En segundo lugar, la clase, al contener un método abstracto, debe ser declarada también como abstracta, por medio del modificador "abstract": "public abstract class Articulo{...}"

No incluir el modificador “abstract” antes del nombre de la clase habría producido un error de compilación. Conviene recordar que, al incluirlo, estamos previniendo que se construyan objetos de la misma. Conviene ilustrar también las diferencias de la declaración de métodos y clases abstractas entre Java y C++: 1. Sobre la declaración de métodos abstractos: Sintaxis propia de Java: 23 “public abstract double getParteIVA();” Sintaxis propia de C++: “virtual double getParteIVA() = 0;” 2. Sobre la declaración de clases abstractas: Sintaxis propia de Java: “public abstract class Articulo{...}” Sintaxis propia de C++: “class Articulo{...}” Veamos ahora cómo quedaría en Java una posible implementación del diagrama de clases que introdujimos en la Sección 4.1.3: +Articulo(entrada : string, entrada : double) +getNombre() :

string +setNombre(entrada : string) : void +getPrecio() : double +getParteIVA() : double #getTIPO() : double -nombre : string -precio : double Articulo +Tipo4(entrada : string, entrada : double) #getTIPO() : double -TIPO : double = 4 Tipo4 +Tipo7(entrada : string, entrada : double) #getTIPO() : double -TIPO : double = 7 Tipo7 +Tipo16(entrada : string, entrada : double) #getTIPO() : double -TIPO : double = 16 Tipo16 //Fichero Articulo.java public abstract class Articulo{ private String nombre; private double precio; public Articulo(String nombre, double precio){ this.nombre = nombre; this.precio = precio; } public String getNombre (){ return this.nombre; } public void setNombre (String nuevo_nombre){ this.nombre = nuevo_nombre; } 24 public double getPrecio (){ return this.precio + this.getParteIVA(); } public double getParteIVA(){ return this.precio * this.getTIPO() / 100; } protected abstract double getTIPO(); } //Fichero Tipo4.java public class Tipo4 extends Articulo{ private static final double TIPO = 4.0; public Tipo4(String nombre, double precio){ super (nombre, precio); } protected double getTIPO(){ return this.TIPO; } } //Fichero Tipo7.java public class Tipo7 extends Articulo{ private static final double TIPO = 7.0; public Tipo7(String nombre, double precio){ super (nombre, precio); } protected double getTIPO(){ return this.TIPO; } } //Fichero Tipo16.java public class Tipo16 extends Articulo{ private static final double TIPO = 16.0; public Tipo16(String nombre, double precio){ super (nombre, precio); 25 } protected double getTIPO(){ return this.TIPO; } }

Se pueden observar las siguientes características de la implementación anterior: 1. En primer lugar, el método “getTIPO(): double” ha sido declarado como “abstract” en la clase “Articulo” por medio de la declaración: “protected abstract double getTIPO();” A consecuencia de esto, la clase “Articulo” también ha de ser declarada como “abstract”: “public abstract class Articulo{...}” 2. En segundo lugar, conviene apuntar una vez más que, desde los métodos “getParteIVA(): double” y “getPrecio(): double” propios de la clase abstracta “Articulo”, hemos podido hacer uso del método abstracto “getTIPO(): double”. Ambos métodos sólo podrán ser usados desde objetos que pertenezcan a clases en las cuales el método abstracto “getTIPO(): double” haya sido definido y que no hayan sido declaradas como abstractas.

Conviene recordar que, en Java, el modificador “abstract” puede ser añadido a clases en las que no haya ningún método abstracto, como simple medida de seguridad para prevenir nuestra clase de que sea instanciada (o de que se construyan objetos de la misma) por los usuarios finales. Por ejemplo, si recuperamos la clase “Articulo” tal y como la definimos al principio de esta Sección (es decir, sin contener ningún método abstracto), y le añadimos el

modificador “abstract” en la cabecera de la misma: public abstract class Articulo{ private String nombre; private double precio; public Articulo(String nombre, double precio){ this.nombre = nombre; this.precio = precio; } public String getNombre (){ return this.nombre; } public void setNombre (String nuevo_nombre){ 26 this.nombre = nuevo_nombre; } public double getPrecio (){ return this.precio; } } Se puede comprobar cómo no es posible crear objetos de la misma (obtendremos un error de compilación), a pesar de que todos sus métodos hayan sido declarados y definidos.

INTERFACES (EN JAVA) Y CLASES COMPLETAMENTE ABSTRACTAS

(EN C++) Hasta ahora hemos visto ejemplos sencillos es los que un único método en una clase se definía como abstracto y las clases que heredaban de la misma lo definían, pasando así a ser clases concretas. De este modo conseguíamos “enriquecer” la parte visible de una clase e incluso aumentar la reutilización de código, haciendo uso de métodos que no habían sido definidos pero sí declarados. El concepto anterior se puede extender un poco más allá, dando lugar a “clases completamente abstractas”, en las cuales todos los métodos están sin especificar, y que no poseen ni atributos ni constructores.

Estas “clases” en realidad no aportan estado ni comportamiento, simplemente declaraciones de métodos o cabeceras, con lo cual su utilidad no va especialmente dedicada a reutilizar código. En general, este tipo de “clases” se suelen entender más bien como una especie de declaración de un tipo de dato. La “clase” contiene una serie de cabeceras de métodos, y cualquier clase que herede de ella debe especificar la forma concreta de todos y cada uno de los métodos que contenga la clase (no sus declaraciones).

Se podría entender también que estas “clases” constituyen un contrato entre el programador de una clase y su cliente. El cliente le da al programador una “clase” que sólo especifica una serie de métodos, y el programador de la clase debe ser capaz de implementarlos de forma correcta. Lo que hemos dado en llamar “clases” (o “clases completamente abstractas”) en los párrafos anteriores, toma dos formas distintas en Java y en C++. En Java da lugar a la noción de “interface”, mientras que en C++ se puede lograr por medio del uso de “clases completamente abstractas”. 27 Definición: una “interface” en Java es un conjunto de métodos relacionados que no tienen definición (es decir, métodos abstractos). Una “interface” puede contener también declaraciones de constantes, aunque es una práctica que suele ser desaconsejada.

Definición: una “clase completamente abstracta” en C++ es una clase que contiene sólo declaraciones de métodos (es decir, métodos abstractos). De la definición de “clase completamente abstracta” se puede deducir que una “clase completamente abstracta” sólo poseerá archivo de cabeceras (es decir, un fichero “*.h”). Una pregunta natural ahora sería por qué son necesarios los “interfaces” en Java, cuando el lenguaje contiene ya clases abstractas que puede contener métodos abstractos. En los ejemplos que hemos visto hasta ahora, una clase siempre heredaba de otra única clase. Sin embargo, puede haber situaciones en las que nos interese que una clase herede al mismo tiempo de varias clases distintas. Por ejemplo, una clase “CircunferenciaDibujable” podría heredar al mismo tiempo de una clase

"Circunferencia" (que contenga los atributos y métodos necesarios para trabajar con objetos de tipo "Circunferencia") y de otra clase "Dibujable" (que contuviera métodos que permitan dibujar un objeto). La herencia múltiple no está permitida en Java (sí lo está en C++, pero por ejemplo tampoco lo está en C#).

Una de las razones para que no esté permitida la herencia múltiple en Java es que, si una clase pudiera heredar un mismo método definido en diversas clases de forma distinta, ¿qué definición (o comportamiento) del mismo debería heredar? Para evitar esa situación, una clase en Java sólo puede heredar de otra clase (sólo hay herencia simple). Sin embargo, una clase en Java no "hereda" de una "interface". En realidad, una "interface" no contiene ni atributos ni comportamiento, y por tanto sus propiedades no son "heredadas".

En terminología Java, se dice que una clase "implementa" una "interface", es decir, "implementa" la lista de métodos que contiene dicha "interface" (pero, insistimos, no hereda de dicha "interface"). Java sí permite que una clase "implemente" múltiples "interfaces", ya que al no tener éstas definidos sus métodos, se puede dar la situación en que una misma clase implemente dos "interfaces" que contienen un mismo método, pero como ninguna de las dos copias del método ha sido todavía definida, la definición válida del mismo será la provista en la clase. Por tanto, una de las razones para usar "interfaces" en Java (en lugar de clases abstractas) es que una misma clase puede implementar varios interfaces al mismo tiempo. Como C++ permite herencia múltiple, este comportamiento también puede ser conseguido en C++ simplemente heredando de varias clases que sean completamente abstractas. Veamos lo anterior con un sencillo ejemplo de la propia API de Java.

Observamos primero cómo se representa en UML una "interface" así como la relación de "implementación" (no de herencia) correspondiente:

28 +compareTo(entrada) : int «interface» Comparable +String() +String(entrada : String) +concat(entrada : String) : String +compareTo(entrada : String) : String +toString() : String String La clase "String" es una clase propia de la librería de Java con la que ya hemos trabajado que nos permite representar cadenas de caracteres (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html). Si observamos la especificación de la misma en la página web anterior, podremos observar cómo dicha clase "implementa" la "interface" "Comparable".

En notación UML, una "interface" se denota de modo similar a una clase, pero con el calificativo "<>" sobre el nombre para poder distinguirla. Su lista de métodos aparecerá siempre en cursiva, ya que todos ellos deben ser abstractos. En este caso, la "interface" "Comparable" sólo contiene un único método, de nombre "compareTo(T): int" (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html).

El tipo de dato "T" que admite como parámetro es lo que se conoce en Java como un tipo genérico. Por el momento no vamos a explicar su significado, pero debemos ser capaces de trabajar con las clases de la API que hacen uso del mismo. Las relaciones de implementación de una "interface" por parte de una clase se denotan por medio de una flecha (igual que las relaciones de herencia) pero en la cual la línea está punteada.

Como podemos observar en el diagrama anterior, el hecho de que la clase "String" implemente la "interface" "Comparable" quiere decir que debe definir el método "compareTo(String): int", que como podemos observar aparece en el diagrama UML de la misma (en caso contrario, el método "compareTo(T): int" seguiría siendo abstracto en la clase "String" y por tanto la clase sería abstracta, que no es el caso ya que el nombre de la misma no aparece en cursiva en el diagrama UML).

Veamos ahora la especificación que del método "compareTo(T): int" se hace en API de Java, tanto en la "interface" "Comparable" como la que se da del mismo método, ya definido, ("compareTo(String): int") en la clase "String": En la interface "Comparable" encontramos la siguiente especificación del método,que todavía es abstracto (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html#compareTo (T)):

"Compara este objeto con el objeto especificado como parámetro para comprobar su orden. Devolverá un entero negativo, un cero, o un entero positivo si el objeto es menor que, igual a, o mayor que el objeto especificado" 29 (Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.) En la clase "String" cuando ya el método "comparable(String): int" es definido, la especificación del mismo dice (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html#compareTo(java.l ang.String))): "Compara dos cadenas lexicográficamente. La comparación está basada en el valor Unicode de cada carácter en las cadenas ...

El resultado es un entero negativo si el objeto "this" lexicográficamente precede al argumento. El resultado es un entero negativo si el objeto "this" lexicográficamente sigue al argumento.

El resultado es cero si ambas cadenas son iguales; ..."

(Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings.

The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string.

The result is zero if the strings are equal; compareTo returns 0 exactly when the equals (Object) method would return true.) Como puedes observar, el comportamiento de la función "compareTo(String): int" es similar al que obtendrías con la función C++ "strcmp". Lo que pretendíamos ilustrar con el anterior ejemplo es que, si bien en la "interface" "Comparable" la especificación del método "compareTo(T): int" era todavía "abstracta", en el sentido de que sólo se indicaba cómo debía comportarse tal método de un modo genérico, en la clase "String" el mismo ya ha recibido una definición "concreta", ya que su comportamiento queda completamente especificado.

Esto nos permite entender mejor la diferencia entre el tipo de funciones que debe cumplir una "interface", es decir, facilitar una lista de métodos que luego las clases deben definir, y las funciones que debe definir una clase "concreta", que debe definir todos y cada uno de los métodos que desee implementar.

Veamos ahora alguno de los usos adicionales de las "interfaces" (o de las "clases completamente abstractas" de C++). Una "interface" permite la declaración de objetos de la misma. Veamos un ejemplo sobre el diagrama de clases anterior:

```
public static void main(String [] args)
{ Comparable c1, c2; c1 = new String ("caballo"); c2 = new String ("pellejo"); 30
System.out.println ("El resultado de comparar " + c1 + " con " + c2 + " es " +
c1.compareTo(c2)); }
```

Vemos cómo en el fragmento anterior de código hemos podido declarar tanto "c1" como "c2" como objetos de la "interface" "Comparable". Posteriormente los hemos podido construir como objetos, por ejemplo, de la clase "String", ya que "String" es una clase que implementa "Comparable". Como ambos objetos han sido declarados de la "interface" "Comparable", ahora podemos invocar sobre cualquiera de ellos al método "compareTo(String): int".

El resultado de ejecutar el anterior código sería (por ejemplo): El resultado de comparar caballo con pellejo es -13 El resultado nos está diciendo que la cadena "caballo" es menor, con respecto al orden lexicográfico, que la cadena "pellejo", ya que el valor devuelto por el método es menor que 0. Por cierto, el haber declarado "c1" y "c2" como de la "interface" "Comparable" quiere decir que ambos objetos poseen únicamente el método "compareTo(String): int" y a los propios de la clase "Object". Por ejemplo, observa lo que pasa al intentar invocar al método "concat(String): String" de la clase "String" sobre cualquiera de los objetos: //c1.concat(c2); //La orden provoca un error de compilación, ya que c1 es de tipo "Comparable" Lo anterior nos sirve para remarcar una vez más la importancia de declarar un objeto de una clase o "interface", ya que esto decide el conjunto de métodos que vamos a poder utilizar sobre el mismo.

De igual modo que podemos declarar objetos de una "interface", se puede utilizar una "interface" para declarar estructuras genéricas (como "arrays") o también para declarar el tipo de los argumentos que se pasan a una función o método.

Un último punto al que deberíamos prestar atención es al hecho de que, en Java, una misma clase puede implementar distintas "interfaces" (mientras que no puede heredar de distintas clases, situación que sí es posible en C++).

Si observamos de nuevo la especificación en la API de Java de la clase "String" (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html),

podemos ver cómo la misma implementa la "interface" "Comparable", y también hace lo propio con "CharSequence" (y "Serializable", de la que no nos ocuparemos ahora), por lo que el diagrama anterior UML de la clase "String" se podría ahora mostrar como:

31 +compareTo(entrada) : int «interface» Comparable +String() +String(entrada : String) +charAt(entrada : int) : char +concat(entrada : String) : String +compareTo(entrada : String) : String +length() : int +subSequence(entrada : int, entrada : int) : CharSequence +toString() : String String +charAt(entrada : int) : char +length() : int +subSequence(entrada : int, entrada : int) : CharSequence +toString() : String «interface» CharSequence

Vemos que la "interface" "CharSequence" declara algunos métodos (abstractos) útiles para calcular la longitud de una cadena ("length(): int"), capturar subsecuencias de una cadena ("subsequence(int, int): CharSequence"), devolver una carácter en una posición determinada ("charAt(int): char") y convertir un objeto a una cadena ("toString(): String").

Puedes encontrar más detalles sobre la misma en http://java.sun.com/j2se/1.5.0/docs/api/java/lang/CharSequence.html). La clase "String", al implementar dicha "interface", debe dar una definición concreta de los mismos (salvo que sea una clase abstracta, que no es el caso).

Por tanto, por el simple hecho de saber que la clase "String" implementa las "interfaces" "Comparable" y "CharSequence", conocemos ya parte de los métodos que tendrá la misma. En general, los "interfaces" nos proveen de una forma muy útil de organizar información, ya que sabiendo qué "interfaces" implementa una clase, conocemos algunos de los métodos que debe contener.

3.6.- Herencia.

Sobrecarga de operadores. Los operadores son un tipo de tokens que indican al compilador la realización de determinadas operaciones sobre variables u otros. La sobrecarga de operadores permite redefinir ciertos operadores, como '+' y '-', para usarlos con las clases que hemos definido. Se llama sobrecarga de operadores cuando reutilizando el mismo operador con un número de usos diferentes, y el compilador decide como usar ese operador dependiendo sobre qué opera.

La sobrecarga de operadores solo se puede utilizar con clases, no se pueden redefinir los operadores para los tipos simples predefinidos.

Los operadores lógicos && y || pueden ser sobrecargados para las clases definidas por el programador, pero no funcionaran como operadores de short circuit. Todos los miembros de la construcción lógica serán evaluados sin ningún problema en lo que se refiere a la salida. Naturalmente los operadores lógicos predefinidos continuaran siendo operadores de short circuit como era de esperar, pero no los sobrecargados.

Los operadores aceptan uno o varios operandos de tipo específico (alguno de los tipos básicos preconstruidos en el lenguaje), produciendo y/o modificando un valor de acuerdo con ciertas reglas. Sin embargo, C++ permite redefinir la mayor parte de ellos. Es decir, permite que puedan aceptar otro tipo de operandos (distintos de los tipos básicos) y seguir otro comportamiento, al tiempo que conservan el sentido y comportamiento originales cuando se usan con los operandos normales. Esta posibilidad recibe el nombre de sobrecarga del operador.

Permanencia de las leyes formales

El lenguaje C++ no impone ninguna restricción en la forma que adopte la sobrecarga de un operador. De forma que se puede conseguir que incluso operadores básicos como la suma (+), la asignación (=) o la identidad (==) adquieran para los tipos abstractos un carácter totalmente distinto del que adoptan para los tipos básicos.

No obstante, lo normal, lógico, y aconsejable, es mantener la máxima homogeneidad conceptual en el polimorfismo.

Es decir, que el efecto básico de los operadores (la imagen mental de su significado) se mantenga, aunque su implementación concreta varíe de una clase a otra. Por ejemplo, las definiciones de suma (+), asignación (=) e identidad (==) para los elementos de una clase C deberían garantizar que después de ejecutada la sentencia c = d; sobre instancias c y d de dicha clase, el resultado de (c == d) fuese true. El resultado de aplicar el constructor-copia para crear un objeto a partir de otro debería producir un nuevo objeto igual que el modelo.

Otro aspecto que debería mantenerse, es que los operadores que normalmente no tienen efectos laterales, se mantengan igualmente libres de tales efectos en sus versiones sobrecargadas. Lo anterior puede expresarse con otras palabras: debe procurarse que las propiedades formales de los operadores matemáticos se mantengan también en la versión sobrecargada. Por ejemplo, que la suma sea conmutativa y asociativa, que la identidad sea simétrica y transitiva. Al tratar la sobrecarga de los operadores relacionales se abunda en estos conceptos.

Sinopsis

A excepción de los que se detallan, el lenguaje C++ permite la sobrecarga los operadores estándar. La nueva versión del operador se diseña de forma que presente un comportamiento especial cuando los operandos sean instancias de clase. Por ejemplo, el operador de identidad = = podría ser definido en una hipotética clase Complejo para verificar la identidad de dos números complejos, al mismo tiempo que mantendría su uso normal cuando se utilizara con tipos básicos (int, float, char,).

Para distinguir unas de otras, a las versiones de los operadores preconstruidas en el lenguaje las denominamos versiones globales, mientras que a las definidas por el usuario, versiones sobrecargadas.

Operadores sobrecargables

El lenguaje C++ permite redefinir la funcionalidad de los siguientes operadores:

```
+    -    *    /    %    ^    &
|    ~    !    =    <    >    +=
-=   *=   /=   %=   ^=   &=   |=
<<   >>   >>=  <<=  ==   !=   <=
>=   &&   ||   ++   --   ->*  ,
->   []   ()   new   new[]  delete delete[]
```

Los operadores +, -, * y & son sobrecargables en sus dos versiones, unaria y binaria. Es decir: suma binaria +; más unitario +; multiplicación *; indirección *; referencia & y manejo de bits &.

Es notable que C++ ofrece casos de operadores sobrecargados incluso en su Librería Estándar. Por ejemplo, los operadores == y != para la clase type_info. Sin embargo, la posibilidad de sobrecarga no se extiende a todos los operadores (ver las excepciones).

Limitaciones

La sobrecarga de un operador no puede cambiar el número de operandos o la asociatividad y precedencia del mismo. En otras palabras: se puede modificar su funcionalidad pero no su gramática original. Por ejemplo, un operador unario no puede ser transformado en binario y viceversa.

- Los operadores globales no pueden ser sobrecargados.
- No pueden definirse nuevos tokens como operadores. En caso necesario deben utilizarse funciones. Por ejemplo, no puede definirse ** como un token para representar la exponenciación (a ** b); en todo caso utilizar algo así: pow(a, b).
- No es posible redefinir el sentido de un operador aplicado a un puntero. Por ejemplo, no es posible modificar el sentido del operador suma (+) entre un puntero y un entero (sentencia L.3).

```
class CL { /* ... */ };
CL c1, *cpt1 = &c1;
CL* cpt2 = cpt1 + 5;        // L.3
```

En otras palabras: no es posible modificar la aritmética de punteros sobrecargando sus operadores.

Excepciones

En la lista de los Operadores sobrecargables, puede verificarse que todos los operadores pueden ser sobrecargados, incluyendo new, new[ ], delete y delete[ ], excepto los siguientes:

- Selector directo de componente .
- Operador de indirección de puntero-a-miembro .*
- Operador de acceso a ámbito ::
- Condicional ternario ?:
- Directivas de preprocesado # y # #
- sizeof, typeid

Los operadores asignación =; elemento de matriz [ ] ; invocación de función ( ) y selector indirecto de miembro -> pueden ser sobrecargados solamente como funciones-miembro no estáticas y no pueden ser sobrecargados para las enumeraciones. Cualquier intento de sobrecargar la versión global de estos operadores produce un error de compilación.

Con la excepción de los anteriores (=, [ ], ( ) y ->), los operadores también pueden ser sobrecargados para las enumeraciones y pueden utilizarse funciones-miembro estáticas.

La función-operador

Los operadores C++ pueden considerarse funciones con identificadores un tanto especiales. Por ejemplo, cuando tenemos el operador de subíndice de matriz, x[y], donde x es un objeto de la clase X, el compilador lo traduce a la expresión: x.operator[](y). Es decir, lo interpreta como la invocación de un método de nombre operator[ ].

Este comportamiento del compilador puede hacerse extensivo al resto de operadores, de forma que, cuando se trata de miembros de clases, si @ representa un operador binario, la expresión a @ b es en realidad una forma abreviada de representar la invocación de una función-miembro (método): a.operator@(b), o de una función externa equivalente: operator@(a, b).

Igualmente, si @ representa un operador unario (por ejemplo, el operador preincremento ++, la expresión @ a es la forma abreviada de representar la invocación de una función-miembro que no acepte argumentos: a.operator@( ), o de una función externa equivalente que acepte un argumento: operator@(a).

Estas funciones, denominadas función-operador, determinan el tipo de los operandos; el Lvalue y orden de evaluación que se aplicará cuando se utilice el operador. Como consecuencia, la sobrecarga de un operador se realiza bajo la forma de sobrecarga de la función-operador y su definición determinará el nuevo comportamiento. Como en el caso general de sobrecarga de funciones, el compilador distinguirá las diferentes funciones-operador por el contexto de la llamada (número y tipo de los argumentos).

La palabra clave operator seguida del símbolo del operador conforma el identificador de la función-operador. Ejemplos:

```
<tipo-devuelto> operator +  (/*...*/) {/*...*/} ;
<tipo-devuelto> operator [] (/*...*/) {/*...*/} ;
<tipo-devuelto> operator -  (/*...*/) {/*...*/} ;
<tipo-devuelto> operator ->* (/*...*/) {/*...*/} ;
<tipo-devuelto> operator =  (/*...*/) {/*...*/} ;
<tipo-devuelto> operator= (/*...*/) {/*...*/} ;
```

Es indiferente dejar un espacio entre la palabra operator y el símbolo del operador (las dos últimas líneas son equivalentes). Además la identificación operador ↔ función-operador no es solo interna, también puede ser utilizada explícitamente en el código. Una función-operador invocada con los argumentos apropiados, se comporta en cualquier sentencia como un operador con sus operandos. Por ejemplo:

```
UnaClase c1, c2, c3;
...
c2 = c1;              // L.3: Ok.  asignación
c2.operator=(c1);     // L.4: Ok.  la misma asignación
c3 = c1 + c2;         // L.5: suma y asignación
c3.operator=(c1.operator+(c2));  // Ok. equivalente a L.5:
```

Las sentencias L.3 y L.4 son equivalentes. Aunque legal, la expresión L.4 no es la forma usual de invocar al operador de asignación =.

La función-operador no puede utilizar argumentos por defecto salvo en los casos que se autorizan expresamente. Tampoco pueden tener más o menos argumentos que los indicados en cada caso.

La función-operador puede ser miembro o friend (función externa) de la clase para la que se define. Que se utilice una u otra forma es, a veces, cuestión de preferencia personal, pero en otras viene obligada. En cualquier caso, las funciones-operador son buenas candidatas para ser declaradas funciones inline

- Se suelen declarar miembros de la clase los operadores unarios (de un solo operando), o los que modifican el primer operando (caso de los operadores de asignación). En estos casos el primer operando debe de ser necesariamente una instancia de esa clase, en concreto el objeto que constituye el argumento implícito. Por esta causa, salvo que sean declaradas funciones estáticas, puesto que el puntero this es incluido de forma implícita en la declaración, sólo hará falta incluir el segundo operando en la lista de argumentos de la función-operador.

- Se suelen declarar friend los operadores que aceptan varios operandos sin modificarlos (por ejemplo los operadores aritméticos y lógicos). En estos casos se exige que al menos uno de sus operandos (argumentos) sea del tipo de la clase para la que se define (las funciones-operador que redefinen los operadores new y delete son la excepción de esta regla).

Herencia y sobrecarga de operadores

A excepción del operador de asignación simple = todas las funciones-operador sobrecargadas en una clase antecesora son heredadas en las clases derivadas. Si B es base de la clase D, un operador @ sobrecargado para B puede ser sobrecargado más tarde para D. Es decir, pueden coexistir las siguientes definiciones:

```
B::operator@( ){ /* definicion para super-Clase */ }
D::operator@( ){ /* definicion para clase derivada */ }
```

POLIMORFISMO

4.1.- Ligadura.

La palabra "polimorfismo" significa "la facultad de asumir muchas formas", refiriéndose a la facultad de llamar a muchas funciones diferentes en una sola sentencia.

Funciones Virtuales

Una función virtual es un método de una clase base que puede ser redefinida en cada una de las clases derivadas, podrá utilizar los métodos redefinidos en esta clase derivada.

Una función virtual se define color cuando la palabra "virtual" antes de la declaración del método en la clase base.

Ejemplo:

```
// Punteros a clases derivadas

#include<iostream.h>
#include<conio.h>

class A
{    int a;
   public:
     void DefineA(int);
     int DameA(void);
     virtual void mostrar(void);
};

void A::DefineA(int x)
{    a= x;
}

int A::DameA(void)
{    return a;
}

void A::mostrar(void)
{    cout << "Clase A" << endl;
```

```
    cout << "a = " << a << endl;
}

class B : public A
{   int b;
  public:
    void DefineB(int);
    void mostrar(void);
    void mostrarA(void);
};

void B::DefineB(int x)
{   b= x;
}

void B::mostrar(void)
{   cout << "Clase B" << endl;
    cout << "b = " << b << endl;
}

void B::mostrarA(void)
{   cout << "Clase B" << endl;
    cout << "a = " << DameA() << endl;
}

void main (void)
{   A *pa;
    B *pb;

    clrscr();
    pb = new B;
    pb->DefineA(10);
    pb->DefineB(5);

    cout << "Desde pb :" << endl;
    pb->mostrar();

//    1ra. Parte: pa apunta a objeto de clase B

    pa = pb;
    pa->DefineA(25);

    pa->DefineB(25); INVALIDO DefineB no es un método
                     // de la clase A
```

```
    cout << "pa apunta a objeto de clase B:" << endl;
    pa->mostrar(); //ejecuta el método definido en B
//    2da. Parte : pb apunta a un objeto de clase A

    pa = new A;
    pa->DefineA(35);
    cout << "Dese pa :" << endl;
    pa->mostrar(); //ejecuta el método definido en A

//    pb = pa; INVALIDO
    pb = (B*)pa;

    pb->DefineB(82);

    cout << "pb apunta a objeto de clase A:" << endl;
    pb->mostrarA();
    pb->mostrar(); //ejecuta el método definido en A
    pb->DefineA(56);
    pb->mostrarA();
}

/*    Al ejecutar el programa se obtiene:

      Desde pb :
      Clase B
      b = 5
      pa apunta a objeto de clase B:
      Clase B
      b = 5
      Desde pa :
      Clase A
      a = 35
      pb apunta a objeto de clase A:
      Clase B
      a = 35
      Clase A
      a = 35
      Clase B
      a = 56
*/
```

Observe el siguiente programa:

```
#include<iostream.h>

class A
{  public:
     virtual void Imprime(void);
};

void A::Imprime(void)
{    cout << "Estoy en A" << endl; }

class B: public A
{  public:
     void Imprime(void);
};

void B::Imprime(void)
{    cout << "Estoy en B" << endl; }

class C: public A
{  public:
     void Imprime(void);
};

void C::Imprime(void)
{    cout << "Estoy en C" << endl;
}

class D: public A
{  public:
     void Imprime(void);
};

void D::Imprime(void)
{    cout << "Estoy en D" << endl;
}

class DD: public D
{  public:
```

```
    virtual void Imprime(void);
};

void DD::Imprime(void)
{   cout << "Estoy en DD" << endl;
}

//  Prototipos:
void Funcion1(void);
void Funcion2(void);
void MuestraObjeto(A *);

void main(void)
{   Funcion1();
    Funcion2();
}

void Funcion1(void)
{   A *pa[5], *aux;
    cout<<"Ejecución de la función 1: "<<endl;
    pa[0]= new A;
    pa[1]= new B;
    pa[2]= new C;
    pa[3]= new D;
    pa[4]= new DD;
    for (int i=0; i<5; i++)
    pa[i]->Imprime();
}

void Funcion2(void)
{   A Oa; B Ob; C Oc; D Od; DD Odd;

    cout<< endl <<"Ejecución de la función 2: "<< endl;

    MuestraObjeto(&Oa);
    MuestraObjeto(&Ob);
    MuestraObjeto(&Oc);
    MuestraObjeto(&Od);
    MuestraObjeto(&Odd);
}

void MuestraObjeto(A *O)
{   O->Imprime();
}
```

```
/*  Al ejecutar el programa se obtiene :
    Ejecución de la función 1:
    Estoy en A
    Estoy en B
    Estoy en C
    Estoy en D
    Estoy en DD
    Ejecución de la función 2:
    Estoy en A
    Estoy en B
    Estoy en C
    Estoy en D
    Estoy en DD
*/
```

Se puede apreciar que todos los enlaces se hicieron en tiempo de ejecución. Además se puede apreciar que una función virtual en una clase base continúa siéndolo cuando es heredada.

Implementación de las Funciones Virtuales

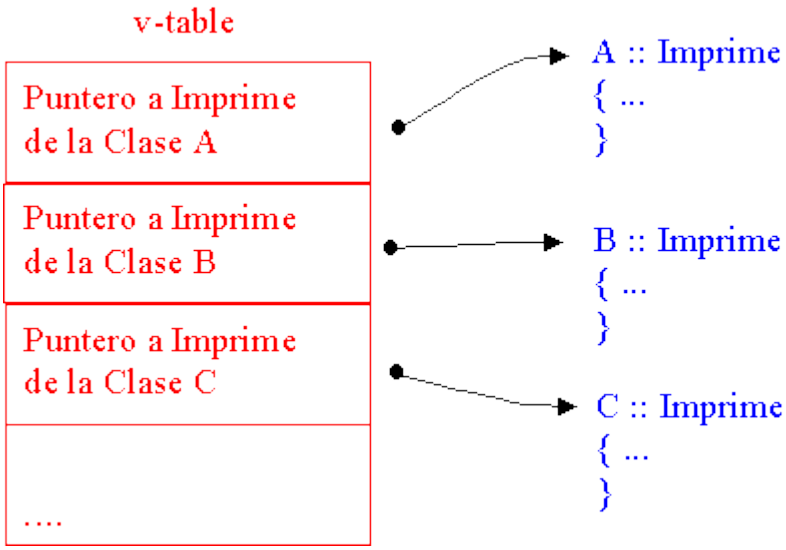
Al definir una función virtual, el compilador no podrá identificar la función que va a ser llamada por el puntero, ya que puede ser cualquiera de varias funciones diferentes. Por lo tanto, el compilador debe añadir código que permita evaluar la sentencia en tiempo de ejecución, que es cuando recién se puede conocer el tipo de objeto al que apuntará y por consiguiente saber a qué función hay que invocar.

Esto se conoce como "ligadura dinámica" o "ligadura retrasada".

Esto es una implementación muy diferente al de las funciones clásicas de C y C++ ya que la llamada a la función en estos casos, es convertida durante la compilación en un salto a una dirección fija que coincida con el inicio del código de la función. Esto último se denomina "ligadura estática".

La ligadura en C++ se implementa a través de una tabla de funciones virtuales llamada "v-table". Esta tabla está formada por un arreglo de puntero a funciones.

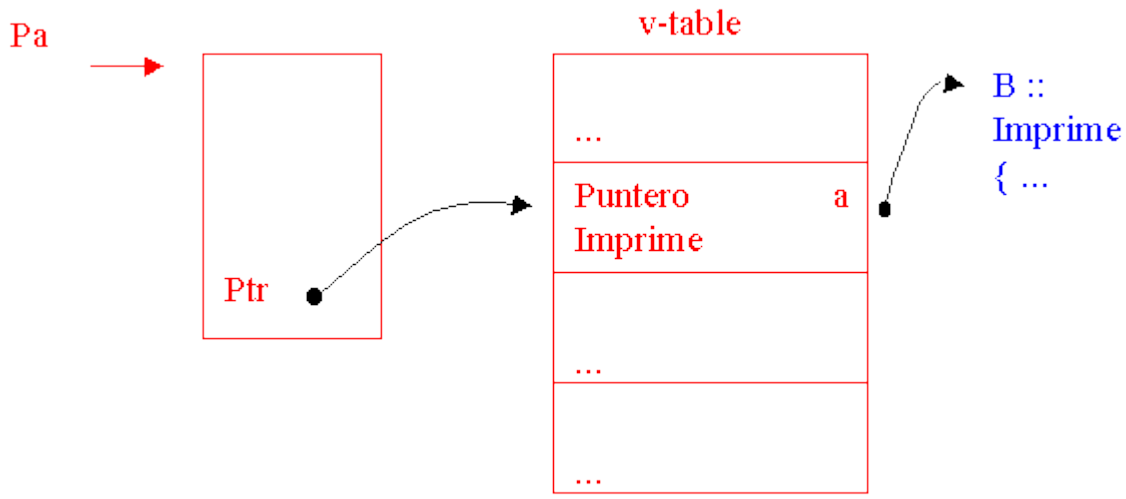
En el último ejemplo, la implementación llevó a un esquema similar al siguiente:



Cuando se crea la variable referenciada por un puntero a un objeto con funciones virtuales, esta variable contendrá un puntero oculto que apuntará a una entrada a v-table dependiendo del tipo de variable referenciada, esto es:

```
A *pa;
pa = new B;
```

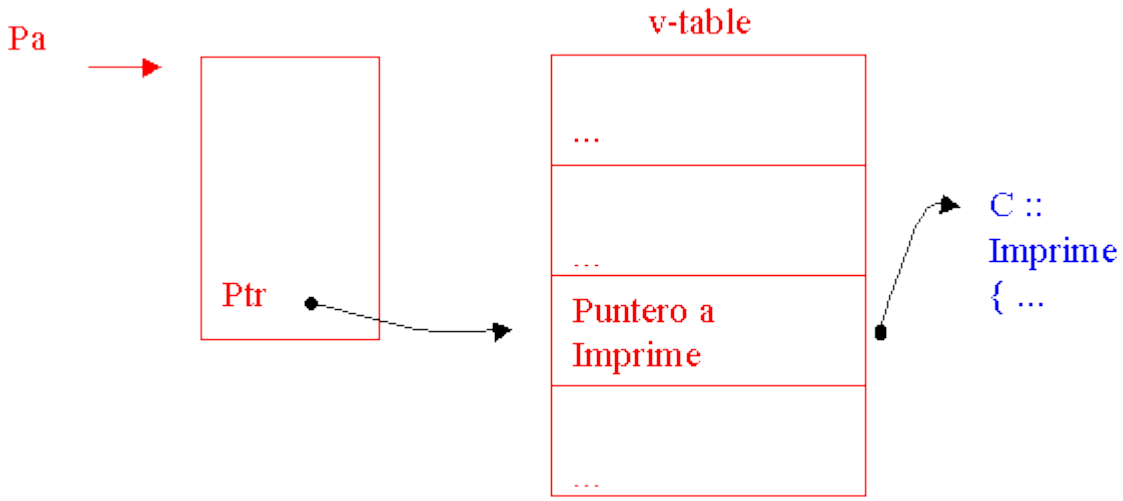
En estos momentos se define un objeto bajo el siguiente esquema:



Si luego se hace:

Delete pa;
pa = new C

Se obtiene:



Funciones Virtuales Puras

Son funciones virtuales que no tienen cuerpo o implementación y que tampoco se pueden ejecutar.

Sintaxis:

virtual tipo nombre (parámetros) = 0;

Ejemplo:

virtual tipo Imprime (void) = 0;

Las funciones virtuales puras nos van a permitir darle la calidad de "virtual" a funciones declaradas en clases derivadas sin tener que definir instrucciones que no vienen al caso en la clase Base. Dicho en otras palabras, las funciones virtuales puras dejan para más adelante (cuando se defina la clase derivada) su implementación:

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

class Persona
{    char Nombre[40];
     char Direccion[40];
     char FechaNac[11];
   public:
     Persona(char *, char *, char *);
     virtual void MuestraDatos(void)=0;
     char *DameNombre(void);
     char *DameDireccion(void);
     char *DameFechaNac(void);
};

Persona::Persona(char *N, char *D, char *F)
{    strcpy(Nombre, N);
     strcpy(Direccion, D);
     strcpy(FechaNac, F);
}

char *Persona::DameNombre(void)
{    return Nombre;
}

char *Persona::DameDireccion(void)
{    return Direccion;
}

char *Persona::DameFechaNac(void)
{    return FechaNac;
}

class Profesor : public Persona
{    char Categoria[30];
     char Dedicacion[5];
   public :
     Profesor(char *, char *, char *, char *, char *);
     void MuestraDatos(void);
     char *DameCategoria(void);
     char *DameDedicacion(void);
};
```

```
Profesor::Profesor(char *N, char *D, char *F, char *C, char *De):
Persona(N, D, F)
{    strcpy(Categoria, C);
     strcpy(Dedicacion, De);
}

void Profesor::MuestraDatos(void)
{    cout << "Datos del Profesor: " << endl;
     cout << "Nombre: " << DameNombre() << endl;
     cout << "Categoria: " << DameCategoria() << endl;
     cout << "Dedicación: " << DameDedicacion() << endl;
}

char *Profesor::DameCategoria(void)
{    return Categoria;
}

char *Profesor::DameDedicacion(void)
{    return Dedicacion;
}

class Alumno: public Persona
{    char Especialidad[40];
     int NumeroDeCreditos;
   public:
     Alumno(char *, char *, char *, char *, int);
     void MuestraDatos(void);
     char *DameEspecialidad(void);
     int DameNumeroDeCreditos(void);
};

Alumno::Alumno(char *N, char *D, char *F, char *E, int C):
Persona(N, D, F)
{    strcpy(Especialidad, E);
     NumeroDeCreditos = C;
}

void Alumno::MuestraDatos(void)
{    cout << "Datos del Alumno: " << endl;
     cout << "Nombre: " << DameNombre() << endl;
     cout << "Especialidad: " << DameEspecialidad() << endl;
     cout << "Número de Créditos: " << DameNumeroDeCreditos() << endl;
}
```

```
char *Alumno::DameEspecialidad(void)
{    return Especialidad;
}

int Alumno::DameNumeroDeCreditos(void)
{    return NumeroDeCreditos;
}

class Empleado: public Persona
{    char Departamento[40];
     char Cargo[40];
   public:
     Empleado(char *, char *, char *, char *, char *);
     void MuestraDatos(void);
     char *DameDepartamento(void);
     char *DameCargo(void);
};

Empleado::Empleado(char *N, char *D, char *F, char *De,
char *C): Persona(N, D, F)
{    strcpy(Departamento, De);
     strcpy(Cargo, C);
}

void Empleado::MuestraDatos(void)
{    cout << "Datos del Empleado: " << endl;
     cout << "Nombre: " << DameNombre() << endl;
     cout << "Departamento: " << DameDepartamento() << endl;
     cout << "Cargo: " << DameCargo() << endl;
}

char *Empleado::DameDepartamento(void)
{    return Departamento;
}

char *Empleado::DameCargo(void)
{    return Cargo;
}

//    Prototipos:
void Imprime(Persona *);

void main(void)
{    Profesor P("Mario Pérez", "Av. ABC 123", "14/03/70" ,
                "Auxiliar","TC") ;
```

```
    Alumno A("Sofía Guzmán", "Av. XYZ","14/09/80",
            "Ing. Informática",120);
    Empleado E("Ana Ruiz", "Jr. PQR 555","07/12/72","RR. HH.",
            "Secretaria");
    Imprime(&P);
    cout << endl;
    Imprime(&A);
    cout << endl;
    Imprime(&E);
}

void Imprime(Persona *Objeto)
{   Objeto->MuestraDatos();
}

/*  Al ejecutar el programa se obtiene:
    Datos del Profesor:
    Nombre: Mario Pérez
    Categoria: Auxiliar
    Dedicación: TC

    Datos del Alumno:
    Nombre: Sofía Guzmán
    Especialidad: Ing. Informática
    Número de Créditos: 120

    Datos del Empleado:
    Nombre: Ana Ruiz
    Departamento: RR. HH.
    Cargo: Secretaria
*/
```

Clases Abstractas

Una clase abstracta es una clase que puede utilizarse sólo como clase base de otras clases. Esto significa que no se podrá definir un objeto de esta clase base, sin embargo, si se podrá definir objetos de sus clases derivadas.

Una clase, para ser definida como abstracta, debe tener al menos una función virtual pura. Una clase abstracta se puede emplear para definir punteros que, a través del programa, apuntarán a objetos de sus clases derivadas, más nunca a un objeto de esa clase.

Aplicación del Polimorfismo

Para que en un programa se produzca el polimorfismo se tiene que seguir los siguientes pasos:

1. Crear una jerarquía de clases en la que operaciones importantes serán declaradas como métodos virtuales, puros o no.
2. Manipular los objetos a través de punteros.

4.2.- Funciones virtuales.

Funciones virtuales: polimorfismo.

En esta ocasión vamos abordar un concepto muy importante de la programación orientada a objetos: el **polimorfismo**. Esta característica de la POO, permite que podamos construirnos métodos para nuestras clase derivadas que parten de una misma clase base, para que adopten comportamientos totalmente distintos. Es un concepto realmente potente y que se lleva a cabo mediante la utilización de **funciones virtuales**. Nosotros ya las hemos utilizado. Si te acuerdas, en el capítulo anterior declarábamos la clase que redefiníamos como virtual. ¡Hemos utilizado el polimorfismo y sin enterarnos!.

Una función virtual es un mecanismo que permite a clases derivadas redefinir a las funciones de las clases base. Por tanto, hasta ahora, nos debemos de quedar con que el polimorfismo es una acción que se puede implementar en clases distintas pero que tienen en común el hecho de heredar de una clase base común. Dicho método, pese a ser común para los objetos derivados, es tratado de forma distinta y, por tanto, dando resultados también diferentes dependiendo de con qué clase lo invoquemos. ¿Cómo conseguirlo?. Para poder implementar el polimorfismo tenemos las funciones virtuales. Las funciones virtuales se definen en la clase base y son las que serán redefinidas, luego, en las clases derivadas.

La declaración de una función virtual se consigue mediante la palabra clave virtual precediendo a la declaración de la función. Por ejemplo:

virtual void PonInformacion(**void**);

En este caso, hemos definido una función virtual llamada *PonInformacion* que pertenece a una clase base y que podrá ser redefinida, completamente, en una clase derivada para que actúe de forma totalmente distinta a cómo lo hace en la clase base. **Es muy importante que la función de la clase base que vamos a redefinir lleve el identificador virtual** pues de lo contrario, la cosa no funcionará como es de esperar. Veremos estos problemas más adelante.

A continuación vamos a poner un ejemplo en el que, utilizando la función de arriba, vamos a ver cómo podemos hacer para que una clase que herede dicha función y produzca resultados totalmente distintos a los que se han definido en la clase base para ella. Así mismo,

utilizamos *new* y *delete* para refrescar la memoria (y nunca mejor dicho :-). Recordad que con *new* y *delete* podemos crear objetos (o variables) dinámicamente y de forma muy sencilla.

```
#include <iostream.h>

class ClaseBase
{
// Esta es la clase base.
// Definimos una función de miembro para la clase base
// que además es virtual e "inline"

public:
vitual void PonInformacion(void) { cout << "\n¡Hola!"; }

};

class ClaseDerivada : public ClaseBase
{

// Hemos definido una clase derivada de la clase ClaseBase.
// Esta clase, va a utilizar el concepto de polimorfismo
// redefiniendo por completo la función  PonInformacion
// que hereda de la clase ClaseBase. También es "inline".

public:
void PonInformacion(void) { cout << "\n¡Adios!"; }

};

int main(void)
{

  ClaseBase *base = new ClaseBase;
  ClaseDerivada *derivada = new ClaseDerivada;

  // Ahora llamamos a los métodos de cada una para observar
  // que el resultado es distinto.

  base->PonInformacion();
  derivada->PonInformacion();

  delete base;
  delete derivada;

  return 0;
```

```
}
```

Bueno, el resultado está claro, ¿no?. Mientras que la clase declarada como *ClaseBase* e instanciada con *base*, pone en pantalla "¡Hola!", la clase derivada, que ha redefinido la función virtual *PonInformacion*, pone "¡Adiós!". Es un ejemplo, en definitiva, muy sencillo en el que se puede ver cómo hacer que una función ya definida en una clase base cambie totalmente según nuestras necesidades: es el polimorfismo.

Otra de las cosas que vimos en el capítulo anterior era el hecho de utilizar dentro de la función de la clase derivada, a la función virtual de la clase base. Si nosotros hubiéramos implementado así a la función virtual de la clase derivada:

```
void PonInformacion(void)
{
// Suponemos que la declaración es inline ya que si no fuera
// inline deberíamos de poner
// void ClaseDerivada::PonInformacion(void)

ClaseBase::PonInformacion();
cout << "\n¡Adios!";
}
```

Cuando llamáramos a la función *PonInformacion*() por medio de la clase derivada, esto es, cuando hiciéramos:

```
derivada->PonInformacion();
```

Lo que nos saldría por pantalla sería:

¡Hola!
¡Adios!

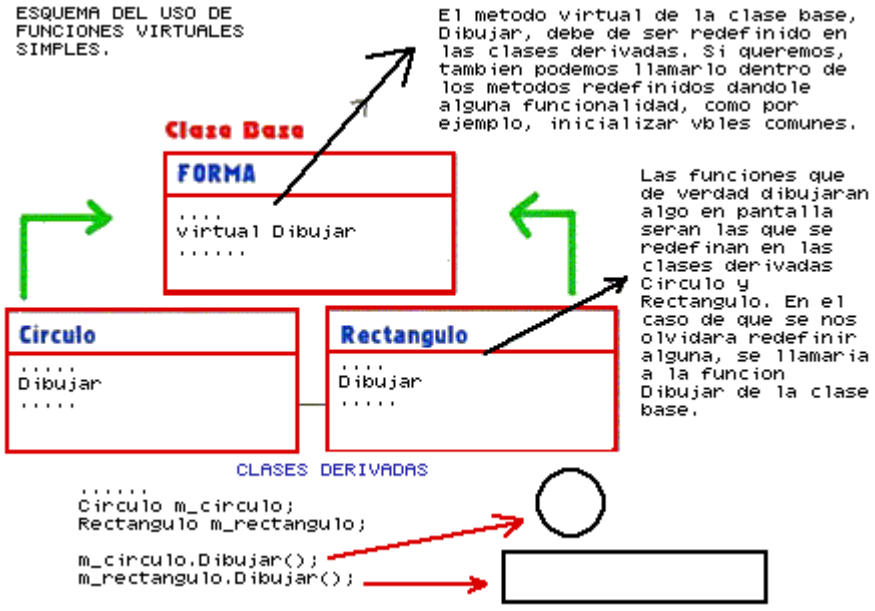
En contraposición a lo que nos sale en el programa original en el que no llamamos a la función de la clase base, es decir, en el ejemplo original saldría nada más ¡Adiós!. Esto es así porque dentro del cuerpo *PonInformacion*() que está implementado en la clase *ClaseDerivada*, llamamos antes de nada a la función de la clase base *ClaseBase* que se trata como una función heredada más.

Aclarando todo un poco

Si recuerdas el capítulo anterior del curso, habrás observado que, en cierta forma, ya hemos utilizado el polimorfismo, es decir, en el anterior capítulo redefiníamos funciones miembro de una clase base utilizando sus características comunes a la clase derivada en la que trabajábamos pero añadiendo una serie de especificaciones extra para que "además se hiciera otra cosa".

Hasta ahora hemos tratado funciones que son virtuales simples. Recordemos que una función virtual es aquella que, habiéndose declarado en una clase base, vuelve a declararse y a implementarse en una clase derivada, es decir, se redefine en la clase derivada. De esta forma, cuando el objeto instanciado a una clase derivada, llama a esa función virtual, lo que se hace es llamar a la función de la clase derivada no a la función de la clase base (a no ser que, dentro de la función de la clase derivada, se llame a la función de la clase base).

El ejemplo clásico de utilización de funciones virtuales está en el de crear una clase base llamada *Forma* con una función de miembro para dibujar y de nombre *Dibujar*. Si nosotros creamos dos clases derivadas de la clase base *Forma* llamadas *Circulo* y *Rectangulo*, respectivamente, y redefinimos la función virtual *Dibujar* en cada una de las clases base, cuando la llamemos desde cada una de las instancias a la función *Dibujar*, el compilador sabrá a qué función llamar.



Cabe decir también, y esto es importante pues puede dar lugar a confusiones, que si tenemos una función en una clase base que no esté marcada como virtual y después creamos en una clase derivada otra función con el mismo nombre los resultados no van a ser los esperados... Lo que hará el compilador será llamar directamente a la función implementada en la clase base y pasar "olímpicamente" de la implementación de la clase derivada. ¿y si, habiendo declarado una función en la clase base como virtual, luego se nos olvida redefinirla en la clase derivada?. Bueno, en este caso, se llamará a la función de la clase base y no pasará nada.

Funciones virtuales puras implican clases abstractas.

Se puede decir que las funciones **virtuales puras** son aquellas que, para implementarse, han de ser redefinidas, es decir, que no sólo tienen sino que ***deben*** ser redefinidas. Aquellas clases que tengan una función virtual pura se denominan **clases abstractas** y tienen la gran particularidad de que de ellas no se puede crear instancias u objetos.

Quedamos, pues, en **que para crear una clase abstracta sólo hace falta definir una función virtual pura**.

¿Y cómo definimos una función virtual pura?. Para definir una función virtual pura, tenemos que asignar a la función un puntero NULL o, lo que es lo mismo, un valor 0. Suponiendo que queremos implementar una función llamada *Forma* como virtual pura, deberíamos de poner así:

virtual void Forma() = 0;

La clase que contenga una sola función virtual pura pasa a ser una clase abstracta independientemente de que el resto de sus funciones no sean virtuales puras. Ya veis que fácil es crear una función virtual pura y que poder tienen al implicar también la creación de una clase abstracta que, recordad, no puede ser nunca instanciada.

Las funciones virtuales puras han de ser definidas cuando queramos crear una verdadera clase raíz de una serie de clases derivadas ciertamente importante. No conviene abusar de esta característica del C++ a no ser que vuestro diseño lo necesite necesariamente y estéis delante de un proyecto con una complejidad notable.

Conclusiones finales

Decir, finalmente, que las funciones virtuales se caracterizan por añadir una mayor cantidad e trabajo computacional y que, por tanto, es recomendable utilizar las prestaciones del polimorfismo, esto es, poner la palabra clave virtual a una función, sólo cuando estemos seguros de que esa función va a ser redefinida. Tampoco es que se consuma mucho tiempo y se sobrecargue todo en exceso pero suele ser una práctica común entre los que comienzan a trabajar más o menos en serio con el C++ utilizar prestaciones de la POO cuando no son necesarias. Si cuando lleves un tiempo codificando descubres que la función declarada como virtual no va a ser redefinida quita la palabra clave virtual de la función de miembro de la clase base.

4.3.- Polimorfismo.

Polimorfismo

Por polimorfismo entendemos aquella cualidad que poseen los objetos para responder de distinto modo ante el mismo mensaje. Pongamos por ejemplo las clases *hombre*, *vaca* y *perro*, si a todos les damos la orden -enviamos el mensaje- **Come**, cada uno de ellos sabe cómo hacerlo y realizará este comportamiento a su modo. Veamos otro ejemplo algo más ilustrativo. Tomemos las clases *barco*, *avión* y *coche*, todas ellas derivadas de la clase padre *vehículo*; si les enviamos el mensaje **Desplázate**, cada una de ellas sabe cómo hacerlo. Realmente, y para ser exactos, los mensaje no se envían a las clases, sino a todos o algunos de los objetos instanciados de las clases. Así, por ejemplo, podemos decirle a los objetos *Juan Sebastián el Cano* y *Kontiqui*, de la clase *barco* que se desplacen, con los que el resto de los objetos de esa clase permanecerán inmóviles. Del mismo modo, si tenemos en pantalla cinco recuadros (marcos) y tres textos, podemos decirle a tres de los recuadros y a dos de los textos que cambien de color y no decírselo a los demás objetos. Todos estos sabrán cómo hacerlo porque hemos redefinido para cada uno de ellos su método **Pintarse** que bien podría estar en la clase padre *Visual* (conjunto de objetos que pueden visualizarse en pantalla).

En programación tradicional, debemos crear un nombre distinto para la acción de pintarse, si se trata de un texto o de un marco; en OOP el mismo nombre nos sirve para todas las clases creadas si así lo queremos, lo que suele ser habitual. El mismo nombre suele usarse para realizar acciones similares en clases diferentes.

Si enviamos el mensaje **Imprímete** a objetos de distintas clases, cada uno se imprimirá como le
Corresponda, ya que todos saben cómo hacerlo. El polimorfismo nos facilita el trabajo, ya que gracias a él, el número de nombres de métodos que tenemos que recordar disminuye ostensiblemente. La mayor ventaja la obtendremos en métodos con igual nombre aplicados a las clases que se encuentran próximas a la raíz del árbol de clases, ya que estos métodos afectarán a todas las clases que de ellas se deriven.

Sobrecarga

La sobrecarga puede ser considerada como un tipo especial de polimorfismo que casi todos los lenguajes de OOP incluyen. Varios métodos (incluidos los "constructores", de los que se hablará más adelante) pueden tener el mismo nombre siempre y cuando el tipo de parámetros que recibe o el número de ellos sea diferente. De este modo, por ejemplo la clase File puede tener tantos método Write() como tipos de datos queramos escribir (no se preocupe si no entiende la nomenclatura, céntrese en la idea):

File::Write(int i); Escribe un integer
File::Write(long l); Escribe un long
File::Write(float f); Escribe un flota
File::Write(string s); Escribe una cadena
File::Write(string s, boolean b); Escribe una cadena pasándola a

Mayúsculas Existe un tipo especial de sobrecarga llamada sobrecarga de operadores que, de los leguajes OOP conocidos, solo incorpora C++. Es por esto que consideramos que el abordar este tema escapa a la finalidad del presente curso.

Conclusión

Los lenguajes de alto nivel se desarrollaron con el objetivo de ser más accesibles y entendibles por la mayoría de programadores, de manera que los programadores pudieran concentrarse más en resolver la tarea o los problemas y no en el lenguaje que la maquina tenía que entender.

C++ surge de fusionar dos ideas: la eficiencia del lenguaje C para poder acceder al hardware al ejecutar tareas que realmente demandaban recursos de memoria; y las ideas de abstracción que representan las el nuevo conceptos de clases y objetos.

El lenguaje C++ presenta grandes herramientas de desarrollo para los programadores como las funciones, bibliotecas, clases y los objetos. De manera que el programador se ocupa de utilizar dichas herramientas para resolver un problema específico.

El lenguaje C++ posee una serie de características que lo hacen distinto del lenguaje C. Aunque es posible verlo como una simple extensión del lenguaje C, en realidad implica un cambio en la forma de pensar por parte del programador.

Referencias

http://azul2.bnct.ipn.mx/academia/problemarios/guia_ets_Fund_prog_orienta_objetos.pdf

https://www.campusmvp.es/recursos/post/los-conceptos-fundamentales-sobre-programacion-orientada-objetos-explicados-de-manera-simple.aspx

https://www.marcoteorico.com/curso/51/fundamentos-de-programacion/390/conceptos-fundamentales-de-la-programacion-orientada-a-objetos

https://www.unirioja.es/cu/jearansa/0910/archivos/EIPR_Tema01.pdf

https://sites.google.com/site/portafoliodjcv270499/conceptos-basicos-de-programacion-orientada-a-objetos