Elementos de Programación y Estructura de Datos
Versión 1.0
Universidad Mayor de San Simón

Maria Leticia Blanco Coca

15 de agosto de 2012

Tabla de Contenido

Capítulo 1

Recursividad

1.1. Concepto

La recursividad es un elemento de programación que provee un mecanismo poderoso para la fase de diseño. La recursividad es una alternativa a la iteración y se puede definir en términos generales de la siguiente forma:

"Alguna cosa se define como recursiva si está definida en términos de sí misma"

La recursividad se puede presentar en muchos objetos, por ejemplo la famosa muñeca rusa (ver Figura 1.1) tradicional denominada *matrioska*, creada en 1890, cuya originalidad consiste en que se encuentran huecas por dentro, de tal manera que en su interior albergan una nueva muñeca, y ésta a su vez a otra, y ésta a su vez otra, en un número variable que puede ir desde cinco hasta el número que se desee.

Otro ejemplo es la figura 1.2 que ilustra una imagen recursiva formada por un triángulo. Cada triángulo está compuesto de otros más pequeños, compuestos a su vez de la misma estructura recursiva.

1.2. Definición de proceso recursivo

Un proceso es recursivo, si está definido en términos de sí mismo.

Uno de los ejemplos más clásicos de la recursividad es la función factorial, cuya definición es:

n! = n * (n - 1)!

Figura 1.1: Modelo Muñeca matrioska

$0! = 1$
$\forall n \in N$

La función factorial está definida por el símbolo **!**, que puede ser observado en ambos lados de la definición

n**!** = n * (n - 1)**!**

Es decir, para poder calcular el *factorial* de n, debemos calcular el *factorial* de n - 1, se necesita de la misma función; por lo que, se dice que ésta definición del factorial es recursiva.

Sin embargo esta definición, involucra varios aspectos importantes que se deben tomar en cuenta cuando se define procesos recursivos, continuamos con el ejemplo anterior y en esta deficnión se puede identificar tres elementos importantes en la definición: la parte recursiva, la parte básica y las condiciones de contexto.

1.2.1. Parte Recursiva

La parte recursiva de la definición es aquella en la que se advierte la definición del proceso en términos de si mismo:

PR: n**!** = n * (n - 1)**!**

Esta parte es la que cualifica a un proceso como recursivo.

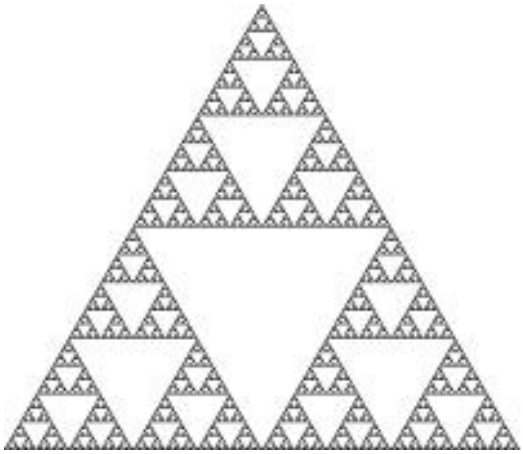

Figura 1.2: Modelo Triángulo

1.2.2. Parte Básica

La parte básica de un proceso, se llama a aquella que no está definida en términos de sí mismo, y que muestra la obtención de un resultado directo:

PB: $0! = 1$

Esta definición es muy importante, en la recursividad ya que asegura que en el algún momento es posible obtener una respuesta.

1.2.3. Condiciones de Contexto

En computación, las condiciones de contexto son un concepto implícito a la hora de definir o concebir un proceso automatizado, en caso de la recursión este aspecto también es importante ya que determina los valores válidos sobre los cuales se ha definido el proceso:

CC: $\forall n \in N$

esta línea en la definición, pone candados fuertes al uso correcto del factorial. La definición no es útil para números negativos ni para números reales. En las secciones adelaate veremos la importancia de las condiciones de contexto cuando se trata de asegurar la correctitud de la definición.

1.3. Correctitud de recursión

Un proceso es correcto si:

- Resuelve el problema para el cual fue diseñado,
- Para cada entrada válida, genera la salida(resultado/efecto) deseada
- Termina en un tiempo finito

Para, en fase de diseño asegurar la escritura de un proceso recursivo correcto, se debe cumplir las siguientes reglas:

1.3.1. Regla del caso básico

Los procesos se ejecutan en distintos casos (escenarios), esta regla indica que debe haber un caso especial para el cual el proceso no haga uso de la definición recursiva, es decir, no utilizar la PARTE RECURSIVA. Sino utilizar la PARTE BÁSICA. El caso particular para el cual fue definido la parte básica debe cumplir las condiciones de contexto estipuladas para la definición, si es así este caso se denomina *caso básico*.

La parte básica de la definición del factorial, se ha definido para un caso particular 0. Este caso cumple la regla del caso básico, si y solamente si, el 0 pertenece y cumple las condiciones de contexto de la definición.

Es importante entonces, a la hora de definir, no sólamente tener una parte básica, sino asegurar que el caso para el cual se ha definido esta parte, cumpla la regla del caso básico.

En la definición de factorial:

PR: n! = n * (n - 1)!
PB: 0! = 1
CC: $\forall n \in N$

El 0 es un caso básico ya que $0 \in N$

Formalizando, sea

$PB : proc(cb) = res$ y
$CC : \forall n \in$ TIPO,
se dice que *cb* cumple la regla del caso básico, si y sólo sí,
$cb \in$ TIPO.

Esta regla, coadyuva a asegurar la terminación del proceso en un tiempo finito.

1.3.2. Regla de progreso

La regla del progreso, indica que toda definición recursiva debe en su llamada recursiva progresar hacia el caso básico. La PARTE RECURSIVA de un proceso, hace uso del mismo proceso, y funciona porque trabaja sobre distintos casos, usualmente estos casos son las entradas a los procesos.

Entonces se dice que los casos de las sucesivas llamadas recursivas deben progresar al *caso básico*, para asegurar la finalización del proceso y así obtener un resultado (si fuera el caso).

Consideremos nuevamente la definición del factorial, en su PARTE RECURSIVA y sus CONDICIONES DE CONTEXTO:

> PR: n! = n * (n - 1)!
> CC: $\forall n \in N$
>
> Es evidente que la PARTE RECURSIVA se ejecutará cuando el valor de $n \neq 0$, pero por las CONDICIONES DE CONTEXTO, la única posibilidad de n *es que* $\in N$, por lo que se puede deducir que:
>
> $n > 0$
>
> Por lo que la llamada recursiva al factorial sería
>
> $(n-1)!$
>
> Esto quiere decir, que el caso para esta llamada será $n-1$, pues bien, como $n \neq 0$ *y* $n \in N$, entonces $(n-1) \geq 0$ lo que significa que $n-1$ esta más próximo a 0 que n. Por lo tanto *si* n *se reduce en* 1 estaremos *progresando* a 0.
>
> En este caso la definición de la PARTE RECURSIVA, cumple la regla del progreso.

Formalizando, sea

> $PR : proc(c1) = proc(c2)$ y
> cb es *caso básico*;
> Se dice que PR cumple con la regla del progreso, si y sólo sí,
> $c1 \overset{progresa}{\longrightarrow} c2 \overset{progresa}{\longrightarrow} \ldots \overset{progresa}{\longrightarrow} cb$

Esta regla aporta a que el proceso para cada entrada válida, genera la salida(resultado/efecto) deseada y que termine en un tiempo finito.

1.3.3. Regla del diseño

Un proceso recursivo, debe ser correcto en todas sus partes. Aparentemente no es difícil verificar la correctitud de la PARTE BÁSICA de la definición, pero en la PARTE RECURSIVA es un más difícil verificar, por lo que la forma de hacerlo es asegurando el diseño del proceso, y se dice que el proceso recursivo es correcto sí y solamente sí la llamada recursiva es correcta.

En el caso del factorial:

$n!$ es correcto, si y sólamente sí $(n-1)!$ es correcto.

Formalizando, sea:

$PR : proc(c1) = proc(c2)$ y
Se dice que $proc(c1)$ es correcto si y sólamente sí $proc(c2)$ es correcto

Esta regla aporta a que el proceso resuelva el problema para el cual fue diseñado.

1.3.4. Regla del interés compuesto

Esta regla trata de que en lo posible se diseñe procesos eficientes y dice: efecto de las sucesivas llamadas recursivas al proceso no se debería ejecutar más de una vez el proceso para un mismo caso. Este problema usualmente se encuentra cuando el proceso se define en base a dos llamadas recursivas, y es probable que por ambas se repita un caso.

El factorial no rompe esta regla, pero un otro ejemplo clásico, es encontrar el n-simo número de la serie de Fibonacci. La serie de Fibonacci es una secuencia de números naturales cuyos dos primeros números son datos y a partir del tercero se autogeneran sobre la base de los dos anteriores, la serie tiene la siguiente forma:

$0, 1, 1, 2, 3, 5, 8, \ldots$

En este ejemplo los números base son: 0 y 1 que vendrían a ser el 1er. y 2do. números de la serie. Por lo que si se quiere calcular el 8vo. número de la serie se necesita tener el 7mo. y 6to. número previamente. Por regla general, para calcular el n-simo número se requiere del (n-1)-simo y del (n-2)-simo término de la serie, salvo para el 1ro. que es 0 y para el 2do. que es 1; por lo que es correcto realizar la siguiente definición:

PR: $fib(n) = fib(n-1) + fib(n-2)$
PB: $fib(1) = 0$
PB: $fib(2) = 1$
CC: $\forall n \in N$

Este es un ejemplo de un proceso recursivo con más de una parte básica.

En este caso para calcular el 4to. número Fibonacci se debe calcular el 3er. Fibonacci y el 2do. Fibonacci. Pero para calcular el 3er. Fibonacci se necesita calcular el 2do. y el 1er. En este momento ya se ha calculado dos veces el 2do. Fibonacci, tratando de encontrar el resultado del 4to Fibonacci. Por lo que, la definición del proceso incumple la regla del interés compuesto.

1.4. Diseño de procesos recursivos

La recursión es un elemento poderoso de programación simple, pero complejo. Para poder diseñar procesos recursivos, se utiliza dos principios matemáticos, que desde distintos puntos llevan a obtener las partes de un proceso recursivo de forma sistemática y apoyan el cumplimiento de la regla del diseño.

1.4.1. Deducción

Consiste en analizar el problema desde sus casos más sencillos y conocidos; para continuar con casos sucesivos e intentar deducir una regla de generalización.

Por ejemplo se quiere encontrar la cantidad de asteriscos que se requieren para dibujar un triángulo rectángulo isóceles de base n, de acuerdo a la Figura 1.3. Debemos primero dar un nombre a nuestro problema *numAst(n)* y encontrar los casos más simples y sobre la base de este conocimiento tratar de encontrar casos más difíciles o mayores, entonces analicemos:

$numAst(0)$ *es* 0 que es el caso más simple lo único que sabemos es el dato de la base 0
$numAst(1)$ *es* 1, pero para calcular este resultado ya sabemos más cosas: el resultado numAst(0) y 1 que es la base como dato de entrada...
$numAst(2)$ *es* 3, para calcular este resultado ya sabemos más cosas: el resultado numAst(1) y 2 que es la base como dato de

Figura 1.3: Numeros triangulares de Pitágoras

entrada...
$numAst(3)$ *es* 6, pero para calcular este resultado ya sabemos más cosas: el resultado numAst(2) y 3 que es la base como dato de entrada...

Ahora tratemos de encontrar una relación entre el resultado y los que se conoce para poder deducir una regla general:

$numAst(0) = 0$, este no cambia es el caso más sencillo
$numAst(1) = 1+numAst(0)$ reemplazando se tiene $numAst(1) = 1 + 0 \Rightarrow 1$
$numAst(2) = 2+numAst(1)$ reemplazando se tiene $numAst(2) = 2 + 1 \Rightarrow 3$
$numAst(3) = 3+numAst(2)$ reemplazando se tiene $numAst(3) = 3 + 3 \Rightarrow 6$
$numAst(4) = 4+numAst(3)$ reemplazando se tiene $numAst(3) = 4 + 6 \Rightarrow 10$
. . . generalizando . . .
$numAst(n) = n + numAst(n - 1)$

Hasta aquí, se ha definido un dominio de valores válidos para el proceso e indirectamente se ha asegurado que se cumpla la regla del progreso, ya que se ha deducido a partir del caso más simple - que vendría a ser el *caso básico* - a un caso general n que son valores mayores 0.

Al finalizar la definición queda:

PR: $numAst(n) = n + numAst(n-1)$
PB: $numAst(0) = 0$
CC: $\forall n \in N$

$$numAst(n) = \begin{cases} 0 & \text{si } n = 0 \\ n + numAst(n-1) & \text{otro caso} \end{cases}$$

1.4.2. Inducción

La inducción es otra de las herramientas que se utilizan para diseñar un proceso recursivo, a diferencia de la deducción se parte de una hipotesis y sobre esta se genera una generalización para un caso mayor:

Por ejemplo, se desea encontrar la cantidad de dígitos que tiene un número entero positivo. En este caso, aplicar deducción es un tanto complicado, por lo que asumiremos que casos menores ya se conocen.

El problema se puede denominar $cantDig(n)$ - cantidad de dígitos del número n, procedamos con el análisis:

En este problema no interesa el valor del número, sino más bien su estructura; por lo que se puede decir que el número estructuralmente es una secuencia de dígitos, $n = ddddddd \ldots dd$, y que:
$cantDig(n) = cantDig(ddd \ldots dd))$
si a la estructura de n le quitamos un dígito, lo que queda - $n1$ - sigue siendo un número y también tiene cantidad de dígitos:
$n = \underbrace{ddd \ldots dd}_{n1} d$
ahora supongamos que el resultado de calcular $cantDig(n1)$ *es* m y además es correcto. Con este conocimiento, cuántos dígitos tendrá n?, La respuesta es fácil - $m + 1$, por lo que:
$cantDig(n) = cantDig(n1) + 1$
Lo siguiente que hay que hacer es, definir la forma de encontrar $n1$ dado que se tiene n, o que es lo mismo, cuál es la manera más

fácil de quitarle un dígito a n?, dividiendo entre 10 y el número sin un dígito viene a ser el *cociente* de la operación, entonces:
$n1 = n/10$, reemplazando
$cantDig(n) = cantDig(n/10) + 1$
Hasta aquí, se ha encontrado la PARTE RECURSIVA de la definición, lo que nos falta es la PARTE BÁSICA.
Cuándo dejaremos de dividir a n?, cuando n sea un número unidígito, en este caso la cantidad de dígitos de un número unidígito es 1.
$cantDig(n) = 1, \forall n \ < \ 10$
La definición completa seria:
$PR: \ cantDig(n) = cantDig(n/10) + 1$
$PB: \ cantDig(n) = 1, \forall n \ < \ 10$
$CC: \ \forall n \in Z^+$

$$cantDig(n) = \begin{cases} 1 & \text{si } n \ < \ 10 \\ cantDig(n/10) + 1 & \text{otro caso} \end{cases}$$

1.5. Ejecución de recursión

Por que la recursión funciona?, en términos de ejecución cada llamada tiene su propio espacio y el puntero de ejecución respeta el orden de ejecución de forma estricta. Para poder mostrar una ejecución, escribiremos el código en Java del proceso factorial, mostrado en la sección 1.2, para ello se debe denominar la función, en nuestro caso llamaremos $calcularFactorial(n)$ y su definición:

$$calcularFactorial(n) = \begin{cases} 1 & \text{si } n \ = \ 0 \\ n * calcularFactorial(n-1) & \text{otro caso} \end{cases}$$

El programa es:

Ejem05_Factorial **Clase *Factorial***

```
/**
 * Proyecto Ejem05_Factorial
 * Clase que muestra la implementacion del metodo
 * recursivo calcularFactorial
 *
 * @author MLBC
```

```
 * @version 01.09
 */
public class Factorial
{
    /**
     * Metodo que calcula el factorial de un numero
     *
     * @param  n    parametro de la funcion factorial
     *              que debe ser un entero positivo
     *              incluido el 0
     * @return     el factorial
     */
    public int calcularFactorial(int n)
    {
        int factorial;
        if(n == 0)
            factorial = 1;
        else
            factorial = n * calcularFactorial(n-1);
        return factorial;
    }
}
```

La ejecución del método $calcularFactorial(4)$ se realiza como se muestra en la Figura 1.4

Una vez que se tiene el código, se observa que el método $calcularFactorial(n)$, utiliza dos variables: un parámetro - n - y una variable local - $factorial$.

Considerando los detalles de programación y la relación tiempo y espacio en la Figura 1.4 se tiene que para toda llamada a $calcularFactorial(n)$ se definen los valores para n y $factorial$ para luego preguntar sobre el valor de n y dependiendo de la respuesta se procede a definir el $factorial$ con 1 ó con la siguiente expresión $n * calcularFactorial(n-1)$. Cada espacio representa una ejecución particular de $calcularFactorial(n)$ y muestra cual de las dos posibles opciones de definir $factorial$ se ha tomado. En base a esta definición se van generando nuevas llamadas al método, las mismas que se denominan *llamadas recursivas*. En un instante de tiempo se observa que se tienen al

mismo tiempo varios *espacios locales*[1] abiertos esperando terminar su ejecución. Pero también se observa que el *puntero de ejecución*[2] se encuentra en exactamente un espacio, esto debido a que estamos procesando en un entorno no distribuido[3].

La única vez que un *espacio local* desaparece, es cuando termina de ejecutar el método y retorna el resultado y el *puntero de ejecución* al lugar donde fue originada la llamada.

Al finalizar, todos los *espacios locales* se habrán cerrado y el *puntero de ejecución* retornará a la primera llamada.

Notar que:

- Tanto el parámetro n y la variable local $factorial$, asumen distintos valores pero en distintos espacios lo cual, permite utilizar los mismos nombres en las diferentes llamadas.
- Que cada llamada (en el ejemplo del factorial) no tiene los mismos casos, por lo tanto son diferentes.

[1] Espacio de direcciones de memoria asociado y dedicado a un proceso

[2] Señala el lugar en el que se encuentra una ejecución y permite conservar el orden de ejecución de procesos/instrucciones.

[3] Se dice entorno no distribuido, cuando la carga de ejecución descansa sobre un procesador

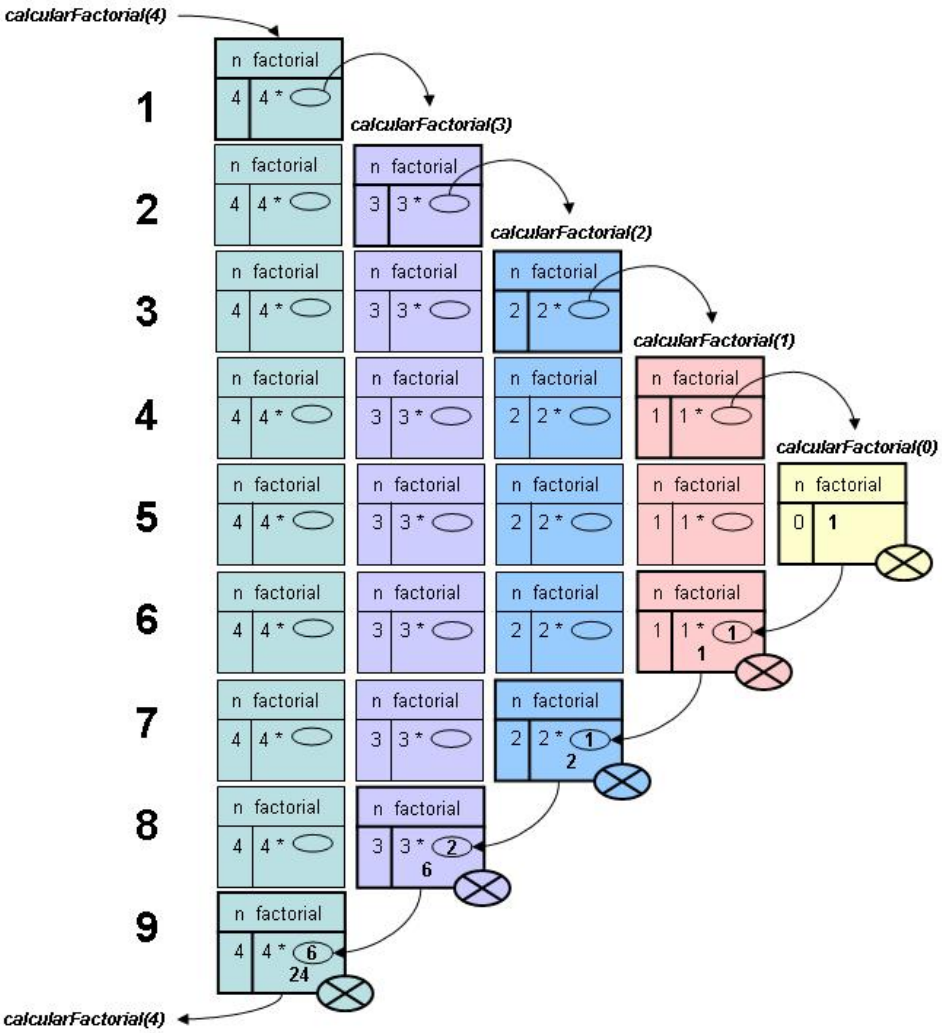

Figura 1.4: Ejecución de *calcularFactorial*

1.6. Recursión directa

Se denomina recursión directa, cuando la definición del proceso recursivo hace una llamada al mismo proceso en su parte recursiva. Hasta ahora todos los ejemplos que se han planteado son directos.

1.7. Recursión indirecta

Considere el problema de decidir si un número natural es par o no, la condición es que no se puede usar la operación de división, ni de módulo.

El razonamiento es muy sencillo, si el número es 0, entonces es par; caso contrario, si es mayor que 0, el número es par sólo si el anterior es impar.

Formalicemos:

$$esPar(n) = \begin{cases} true & \text{si } n = 0 \\ esImpar(n-1) & \text{otro caso} \end{cases}$$

Aqui no existe recursividad ya que la parte que no tiene solución directa no está definida en términos de sí misma. Pero aún nos falta definir $esImpar(n)$. Pues bien, definamos $esImpar(n)$: se sabe que el 0 no es impar, pero si el número fuera mayor que 0, éste es impar, sí y sólo sí el anterior es par.

Formalicemos:

$$esImpar(n) = \begin{cases} false & \text{si } n = 0 \\ esPar(n-1) & \text{otro caso} \end{cases}$$

$esImpar(n)$ no es recursivo tampoco. Pero juntando las dos definiciones, podemos constatar que para decidir $esPar(n)$ se requiere de $esImpar(n)$ y para decidir $esImpar(n)$ se requiere de $esPar(n)$, de todas maneras si tenemos que usar $esPar(n)$ para el caso que no es básico terminaremos usando nuevamente $esPar(n)$ (ver Figura 1.5). Este es un ejemplo de *recursión indirecta*, que se presenta en $esPar(n)$ y también en $esImpar(n)$.

La recursión indirecta, se da cuando un proceso está definido en términos de un segundo proceso y éste segundo proceso está definido en términos del primero. Este concepto puede darse entre varios procesos y generar llamadas cíclicas, por ejemplo: el *proceso*1 llama al *proceso*2, el *proceso*2 usa el *proceso*3, el *proceso*3 llama al *proceso*4 y el *proceso*4 usa el *proceso*1. En este caso todos los procesos son recursivos indirectos.

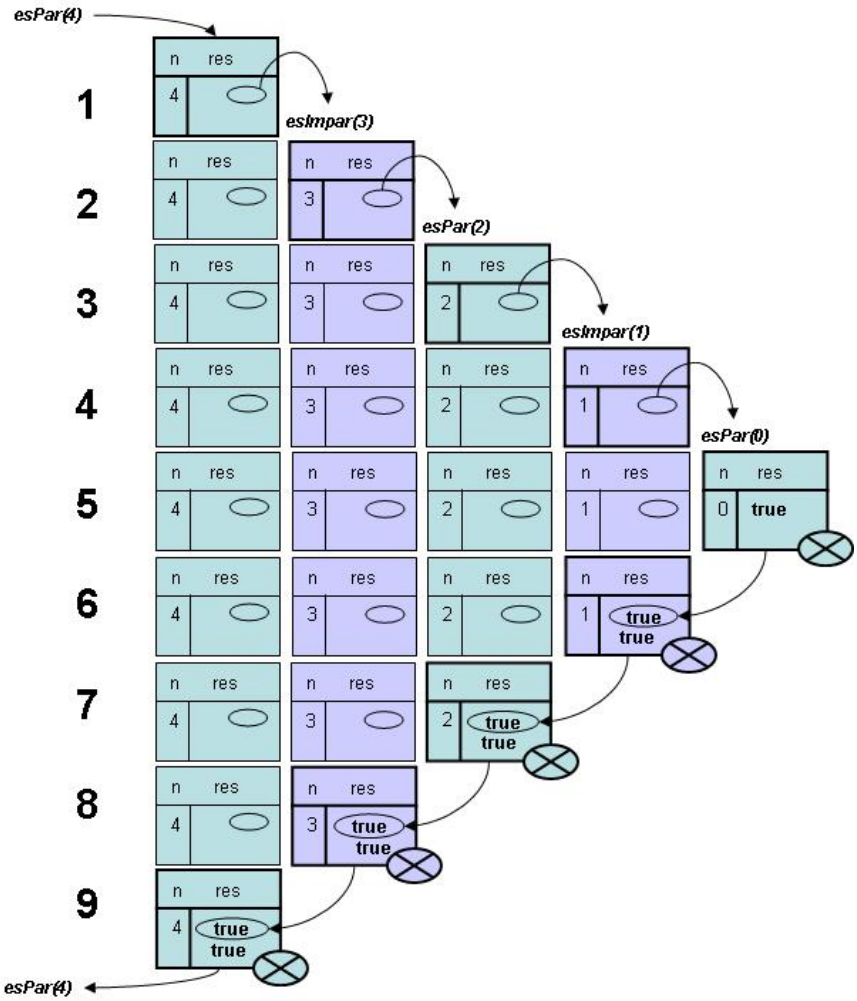

Figura 1.5: Ejecución de $esPar$

1.8. Ejercicios

1. Diseñe procesos recursivos, utilizando deducción/inducción, para los siguientes problemas:

   a) Se desea calcular el a^b.

   b) Dada una secuencia de caracteres, encuentre la cantidad de vocales que tiene.

Por ejemplo, sea la cadena *contenedora*, el resultado es *5*.

c) Se tiene una cadena, la cual se debe rotar hacia la izquierda tantos caracteres como consonantes tenga al principio (de izquierda a derecha).

Por ejemplo, sea la cadena *crespo*, el resultado es *espocr*, note que se ha rotado las consonates que la cadena original tiene al principio al final de la cadena resultante.

d) Recorte una cadena en "posibles" sílabas, considerando que cada que aparece una vocal en la cadena posiblemente hasta ese caracter hay una sílaba.

Por ejemplo, dados los siguientes casos:
Caso1: *cada*, el resultado es {*ca, da*}
Caso2: *oreja*, el resultado es {*o, re, ja*}
Caso3: *programacion*, el resultado es {*pro, gra, ma, ci, o, n*}
Caso4: *crespo*, el resultado es {*cre, spo*}

2. Dados los siguientes métodos, identifique la(s) PARTE(S) RECURSIVA(S) y BÁSICA(S), e indique si cumplen las reglas: del caso básico, del progreso, de diseño y de interés compuesto. Realice la ejecución e indique que hacen.

a) Para valores $x, y \in Z$, para una ejecución considere los siguientes valores: $x = 5$ e $y = 7$

```
public int dudoso1(int x, int y)
{
   int res = 0;
   if(x < 0 || y <0)
        res = x - y;
   else
        res = dudoso1(x-1, y) + dudoso1(x, y-1);
   return res;
}
```

b) Para valores $a, b \in Z^+$.

```
public int dudoso3(int a, int b)
{
```

```
    int res = 0;
    if(a < b)
         res = a;
    else
        if(b  ==  a)
           res = dudoso3(a, b);
        else
           res = a * dudoso3(a - b, b);
    return res;
}
```

c) Para valores $a, b \in Z^+$. Para una ejecución considere los valores $a = 8$ y $b = 4$

```
public int dudoso2(int a, int b)
{
    int res = 0;
    if(b == 1)
         res = a;
    else
        if(b%2 == 0)
           res = dudoso2(a, b/2) * dudoso2(a, b/2);
        else
           res = a * dudoso2(a, b/2) * dudoso2(a, b/2);
    return res;
}
```

Capítulo 2

Programación Orientada a Objetos

La programación orientada a objeto - POO, provee de distintos elementos que permiten abstraer problemas. Las propiedades más importantes están dadas por: el ***encapsulamiento***, el ***polimorfismo*** y la ***herencia***.

Este capítulo brevemente cita los conceptos básicos de la POO y su terminología.

2.1. Concepto de Clase y Objeto

Para poder entender el paradigma de POO es necesario definir conceptos que son fundamentales, como son ***la clase*** y ***el objeto***, ambos fuertemente relacionados.

2.1.1. Clase

La clase define *tipos de objetos*, cuya misión es determinar dominios de datos. Las clases rescatan las características escenciales comúnes a un grupo de objetos, estas características son propiedades y comportamientos. Se denominan *escenciales* porque distinguen a un grupo de objetos de otro grupo, y porque - en términos de programación - son las características más relevantes para representar y que responden a las condiciones del problema que

se está queriendo resolver.

Por ejemplo, si se quiere representar a rectángulos, probablemente, la imaginación lleve a pensar en la forma (figura) de un rectángulo, pero si se quiere representar en forma escrita, entonces se debe pensar qué características son escenciales. Una forma de hacerlo es: preguntarse Qué tiene todo rectángulo? Todo rectángulo, tiene base y altura Qué puede hacer todo rectángulo? Dibujarse, moverse, pintarse.
Nuestro razonamiento puede ser bastante convincente, pero se tiene un problema de consistencia ya que la única forma de dibujar un rectángulo es teniendo conocimiento de un punto en el espacio que ubique al dibujo; bueno podemos pensar.... eso no es problema se puede tener como datos adicionales de este comportamiento las coordenadas donde se dibujará.
Pero, tambiés es posible mover un rectángulo sobre el eje x en 10 unidades. En este caso aunque se reciba las 10 unidades de desplazamiento ya no hay forma de moverlo consistentemente, pues no se tienen las coordenadas anteriores. Recordemos que se habia planteado que las coordenadas para dibujar eran datos adicionales del comportamiento *dibujar* sin embargo no es verdad que TODO rectángulo conoce sus coordenadas de ubicación. Por lo tanto, el modelo que se ha planteado para representar a todo rectángulo con esas propiedades y comportamiento no es consistente. Por lo que se debe pensar en completar el modelo y además de las preguntas anteriores es importante, indagar más acerca del comportamiento requerido, y sobre la base de esto preguntarse si las propiedades que tiene la clase son suficientes para cumplir con los comportamientos que se pide.
Para finalizar, la clase tendrá: *base*,*altura*, *coordenada x* y *coordenada y*, como propiedades, y por otro lado sabrá *dibujarse*, *moverse* y *pintarse* como comportamientos.

Es importante considerar que la clase debe representar de forma abstracta - sin pensar en un objeto en particular - a todos los objetos de esa naturaleza de forma escencial y consistente.

El nombre de clase viene de un proceso natural de clasificación, que permite encontrar grupos de tipos diferentes de objetos en un contexto.

2.1.2. Objeto

El objeto viene a ser un ejemplar particular de una clase. El objeto es individual, que tiene las propiedades descritas por una clase y sabe hacer los comportamientos estipulados en la clase. La diferencia entre una clase y un objeto, es que el objeto define estados en las propiedades enumeradas por la clase, cada objeto tiene identidad y comportamiento, los mismos que dependiendo de sus estados generan diferentes salidas a otro objeto de su

propia clase.

Si se considera el ejemplo de los rectángulos. Se puede tener dos ejemplares que respondan a la representación de Rectángulo que se obtuvo en la anterior sección. Uno de ellos se llamará *rect1* y el otro *rect2*.
rect1 y *rect2* vienen a ser la identidad de los objetos, a partir de este momento ambos son diferentes rectángulos.
Completando, definamos el estado del *rect1* en sus propiedades *base*,*altura*, *coordenada x* y *coordenada y*:

base = 10
altura = 5
coordenada x = 5
coordenada y = 10

Y para el *rect2*, se definen los siguientes estados:

base = 100
altura = 25
coordenada x = 200
coordenada y = 100

A ambos rectágulos se les pide moverse en 5 unidades sobre el eje x (lo saben hacer), pero al terminar de hacerlo cada uno de ellos se habrá desplazado en distintos puntos del eje de coordenadas. Por qué es esto?, porque sus estados son diferentes, y eso los hace individuos. Usan distinto modo de moverse? No, lo que hacen es utilizar distintos datos para realizar su trabajo.

2.2. Concepto de Atributo y Método

Las características escenciales que se describen en una clase, pueden ser propiedades y comportamientos, que juntos definen tipos de objetos o unidades de datos complejas pero consistentes.

2.2.1. Atributo

Es una propiedad que se atribuye a un objeto, los atributos corresponden a las propiedades y usualmente son datos de información que contienen estados que son individuales a un objeto.
Usualmente en las clases los atributos se listan identificándolos a cada uno de ellos con nombres distintos y significativos, además de estipular qué *tipo* de contenido tendrá. Los atributos cumplen todas las normas de contenedoras de datos, respecto a que tienen nombre, tipo y contenido.
Un objeto es el que decide qué estado (contenido) tendrá cada uno de sus atributos. Otra forma de ver la separación es la siguiente:
En la clase se define el tipo y nombre de un atributo y el objeto es el que define el contenido de un atributo (usualmente).

Continuando con el ejemplo, para definir técnicamente la propiedad base se requiere decidir qué tipo de contenido tendrá esta propiedad y cual será la nominación que le daremos, por lo que en la clase se declara el atributo de la siguiente manera:

```
int base;
```

En consecuencia, cuando un objeto desee definir el estado de su atributo base, ya no puede establecer cualquier valor a menos que sea un entero, entonces es correcto que un objeto al instanciarse ejecute una asignación por ejemplo:

```
base = 10;
```

que es consistente con el tipo definido para el atributo.

2.2.2. Método

Es el término que se utiliza para los comportamientos de los objetos. Se dice que para cada comportamiento identificado en una clase le corresponde un método. Usualmente los métodos utilizan las propiedades para definir su funcionamiento, esto es lo que hace que dependiendo de los datos que se tengan, los métodos reaccionen distinto.

Los métodos son las unidades que contienen conjuntos de instrucciones que realizan una tarea determinada.

Al ejemplo del Rectángulo, le añadimos un comportamiento de *calcular el área* y si consideramos los dos objetos que tienen sus atributos ya definidos (con contenido), es bastante natural que si al *rect1* le pedimos que realice *calcular el área* este objeto responda 50, en cambio, el objeto *rect2* responderá 2500. Sin embargo la forma en que calculan es la misma:

area = base * altura;

2.3. Encapsulamiento

El encapsulamiento es un mecanismo que permite controlar el acceso a las características de un objeto. Recordar que las características son atributos y métodos, ambos pueden ser accedidos y "usados" por otros objetos. El "uso" de un atributo, es permisible en las siguientes operaciones: acceder al contenido y cambiar el contenido. En cambio el "uso" de un método se restringe a la ejecución del mismo, no así a la modificación de su definición. Es importante hacer hincapié que en programación imperativa se tiene tres estadíos a la hora de trabajar con contenedoras y/o procesos la primera es declarar, luego definir y usar.

En términos de implementación un atributo se declara, define y "usa":

```
int base;                  declaración
base = 10;                 definición / uso interno
area = base * altura;      uso interno
rect1.base = 100;          definición / uso externo
```

En términos de implementación un método se declara, define y "usa":

```
void mover(int desp);          declaración
```

```
void mover(int desp)
{
    centro.desplazarX(desp);   definición
}
```

```
rect1.mover(50);               uso externo
mover(50);                     uso interno
```

En estos ejemplos, se puede observar que cuando un método se "usa", no significa un cambio de su definición. De hecho en ejecución, la definición de un método es encapsulada.

El encapsulamiento, entonces permite restringir el acceso directo por terceros a las características de un objeto. En la clase el mecanismo se implementa haciendo uso de modificadores de acceso: **private** para restringir acceso y **public** para autorizar acceso. Usualmente los atributos son de acceso **private**, y lo que es público - generalmente métodos - se constituyen en la interfaz del objeto, mediante la cual se comunica con otros.

Para evitar el "uso" directo de un atributo desde el exterior, se puede encapsular con la palabra **private**. Esto es deseable ya que interesa mantener

la consistencia interna de los objetos.

En términos de implementación un atributo privado tendría esta secuencia:

```
private int base;          declaración
base = 10;                 definición / uso interno
area = base * altura;      uso interno
rect1.base = 100;          YA NO ES POSIBLE!!
```

2.4. Interacción

La unidad básica para programar en POO es el objeto. Sin embargo, un sólo objeto no resuelve problemas, entonces pueden existir varios objetos y de distintos tipos que colaboran en la solución del problema. Esta colaboración se denomina interacción, que permite a objetos tener distintos tipos de relacionamientos, una de ellas es la composición, que relaciona objetos en una forma *todo - parte*; otra es la agregación que permite a un objeto agregar otro objeto para ampliar sus prestaciones de servicio y la otra es simplemente la colaboración en la que un objeto probablemente necesita de la ayuda de otro para poder realizar bien su trabajo.

La interacción modela los relacionamientos entre objetos desde una facción *Cliente*, *Servidor*. Uno de los objetos es el que genera la interacción y a este usualmente se le dice que es el cliente.

En el caso de la composición, el relacionamiento es *todo - parte*, en este caso se dice que el *todo* es el *cliente* y la *parte* es el *servidor*. Ya que para que el todo funcione necesita de su parte; y probablemente no es posible que exista el todo sin una de sus partes.

En el caso de agregación, se tiene un relacionamiento de forma *todo - parte*, sin embargo, la diferencia es que la parte no es vital para que el todo exista, sino más bien como su nombre dice es un agregado más. En este caso nuevamente el *todo* juega el rol de *cliente* y la *parte* juega el rol de *servidor*.

En el caso de una colaboración casual, se dice que el que requiere de la ayuda del otro objeto es el *cliente* y el que ayuda es el *servidor*.

El relacionamiento Auto - Motor, tiene una facción de composición, porque el Auto no se podrá mover si no tiene un motor.

El relacionamiento PC - Impresora, tiene una facción de agregación, ya

que la PC funcionará aún si no tiene impresora. Si tu decides enviar a imprimir desde una PC que no tiene conectada ninguna impresora el sistema seguira funcionando y te dirá que no tiene el recurso agregado para brindar el servicio, pero, sigue funcionando.

El relacionamiento Auto - Chofer, es simplemente colaboración, ya que el auto necesita de la ayuda de un chofer para poderse mover. Y por supuesto el chofer deberia poder manejar para cumplir con el servicio básico de mover una auto.

Si consideramos refinar un poco más el ejemplo del Rectángulo, es correcto y lógico decir que la coordenada x e y, juntas representan otro tipo de objeto que es el Punto, por lo que en consecuencia todo Rectángulo tiene: base, altura y centro que a su vez es un Punto. En este momento se dice que el Punto es parte del Rectángulo y al revés que el Rectángulo tiene un Punto. Este sería un ejemplo de composición

2.5. Ejemplo práctico

En esta sección se muestra el modelo y código del ejemplo que se ha tomado para explicar el capítulo.

2.5.1. El modelo

La Figura 2.1, muestra el modelo que relaciona las clases *Rectangulo* y *Punto*. En el modelo la interacción es una flecha dirigida que vá desde el *cliente* al *servidor*.

2.5.2. El código

Ejem01_01 **Clase *Rectangulo***

```
/**
 * Proyecto Ejem01_01
 * Clase que representa a objetos de la clase Rectangulo
 *
 * @author MLBC
```

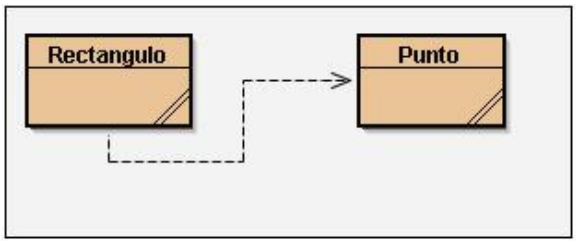

Figura 2.1: Modelo Ejem01_01

```
 * @version 01.09
 */
public class Rectangulo
{
    // atributos
    private int     base, altura;
    private Punto   centro;

    /**
     * Constructor para objectos de la clase Rectangulo
     */
    public Rectangulo(int nBase, int nAltura, Punto nCentro)
    {
        // definiendo los estados del objeto que se esta creando
        base        = nBase;
        altura      = nAltura;
        centro      = nCentro;
    }
    /**
     * Metodo que permite mover un rectangulo sobre el eje x
     * @param  desp un entero que representa el desaplazamiento sobre
     *              el eje x
     * @return
```

```
     */
    public void mover(int desp)
    {
        centro.desplazarX(desp);
    }
    /**
     * Metodo que permite calcular el area de un Rectangulo
     * @param  ninguno
     * @return el area de un rectangulo
     */
    public int area()
    {
        return base * altura;
    }
}
```

Ejem01_01 **Clase *Punto***

```
/**
 * Proyecto Ejem01_01
 * Clase que describe a un Punto en lun espacio bidimensional
 *
 * @author MLBC
 * @version 01.09
 */
public class Punto
{
    // atributos
    private int coordX, coordY;

    /**
     * Constructor para objetos de la clase  Punto
     */
    public Punto(int nCoordX, int nCoordY)
    {
```

```
        coordX  = nCoordX;
        coordY  = nCoordY;
    }
    /**
     * Metodo que permite desaplazar el punto en su coordenada X
     *
     * @param  desp   el desplazamiento de X
     * @return        nada
     */
    public void desplazarX(int desp)
    {
        coordX  = coordX + desp;
    }
    /**
     * Metodo que permite desaplazar el punto en su coordenada Y
     *
     * @param  desp   el desplazamiento de Y
     * @return        nada
     */
    public void desplazarY(int desp)
    {
        coordY  = coordY + desp;
    }
}
```

2.6. Ejercicios

1. Defina el modelo para representar a los siguientes conceptos:

   a) Un libro dentro una biblioteca.

   b) Una computadora en una tienda de venta de computadoras.

   c) Un método como un conjunto de instrucciones.

   d) Una factura emitida.

2. Ordene las siguientes palabras en conceptos mayores y relacionelos:

a) Codigo SIS
b) Nombre
c) Carrera
d) Estudiante
e) Plan de estudios
f) Materia
g) Horas teóricas
h) Horas prácticas
i) Prerequisito
j) Semestre
k) Año de ingreso
l) Fecha de nacimiento
m) Día
n) Mes
$\tilde{n}$) Año
o) Nombre de materia

Capítulo 3

Herencia

La herencia es una de las características más importantes de la POO, ya que permite definir clases en distintos niveles de abstracción; el reuso de código y la extensibilidad.

En este capítulo abordaremos la herencia, considerando la forma en que varias clases de la misma naturaleza pueden relacionarse, manteniendo aún sus diferencias.

3.1. Relación de generalización

Permite relacionar dos clases a través de la generalización y especialización, también conocida como la relación *es_un*. Por un lado están las re-presentaciones de nivel general y por otro las específicas, las generales usualmente se conocen como: superclases, clases base y las específicas se conocen como: subclase o clases extendidas. En el mundo se pueden encontrar muchas de estas relaciones: un auto es un medio de transporte; una bicicleta es un medio de transporte. La clase general es medio de transporte y las clases específicas: auto y bicicleta. También es correcto decir que un auto es un tipo de un medio de transporte y bicicleta otro tipo de medio de transporte. Se denomina la relación *"es_un"* por que leyendo desde la específica a la general existe la dependencia *es un* de forma obligatoria, pero al revés, de la general a la específica se relaciona con *"puede_ser"*. Por lo que en el ejemplo anterior, se encuentran mínimamente cuatro oraciones elementales:

Un Auto *es un* Medio de Transporte

Una Bicicleta *es un* Medio de Transporte
Un Medio de Transporte *puede ser* un Auto
Un Medio de Transporte *puede ser* una Bicicleta

Esta relación es útil cuando se quiere "factorizar" características comunes de varios grupos de objetos en una representación para finalidades de reuso, o cuando se identifica una subclasificación en la representación de una clase.

En programación Java la herencia se denota por la palabra **extends** que relaciona la clase extendida (subclase) con la clase base (superclase).

En el ejemplo del medio de transporte, auto y bicicleta; la formalización en Java es:

```
class MedioDeTransporte
{
.....
}
class Auto extends MedioDeTransporte
{
.....
}
class Bicicleta extends MedioDeTransporte
{
.....
}
```

3.1.1. Asignaciones

La relación *es_un*, brinda posibilidades de asignación de objetos de distinta clase en contenedoras de generalización. Hasta ahora las asignaciones eran posibles, si y solo si la contenedora (lado izquierdo de la instrucción) y el valor (lado derecho) son del mismo tipo. Con la relación de generalización, la asignación - valor a contenedora - es posible si el tipo del valor es del mismo tipo o subtipo de la contenedora. Esto querria decir que la contenedora es de

un super tipo.

En el ejemplo de los medios de transporte, y dadas las siguientes declaraciones:

```
MedioDeTransporte medTrans;
Auto aut1;
Bicicleta bic1;
```

Las siguientes asignaciones son correctas:

```
medTrans = new Auto();
aut1 = new Auto();
bic1 = new Bicicleta();
medTrans = bic1;
```

Pero las que siguen no lo son:

```
aut1 = new MedioDeTransporte();
bic1 = medTrans;
```

3.1.2. Conversiones

Cuando se tiene un objeto de una superclase, es evidente que su instancia puede ser cualquiera de sus subclases. Hay que tener mucho cuidado cuando se asigna un objeto de subclase a una contenedora tipificada por una superclase, la asignación de ninguna manera cambia el tipo de la contenedora, lo que hace es establecer un contenido consistente. Para hacer referencia al contenido, se hace uso del identificador, en caso de hacerlo con un objeto de superclase, automáticamente no devolverá un objeto de esa clase, veamos la siguiente secuencia:

```
1 MedioDeTransporte med1 = new Auto();   Correcto
2 Auto aut1 = med1;                      Incorrecto
```

La segunda asignación es incorrecta, por que cuando se referencia a *med1* lo que hace es devolvernos un objeto de la clase *MedioDeTransporte* que no puede asignarse a una contenedora que espera un objeto de la clase *Auto*. Aqui es natural preguntarse por que no es correcta, si en la primera instruc-

ción en *med1* se ha asignado un objeto de la clase *Auto*, entonces el contenido de *med1* es un objeto de la clase *Auto*. La razón es que devuelve un objeto de la clase *MedioDeTransporte*, por que esa es la tipificación de *med1*, y como no es verdad que todo medio de transporte *es un* auto entonces la asignación es incorrecta.

Dado que sabemos que la instancia de *med1* es una objeto de la clase *Auto*, entonces es posible rescatar el mismo en su naturaleza original (como un Auto), la forma de hacerlo es mediante la *conversión* o *casting*. Observemos las siguientes instrucciones:

```
1 MedioDeTransporte med1 = new Auto();   Correcto
2 Auto aut1 = (Auto)med1;                Correcto
```

Lo que se ha hecho en la segunda instrucción es una conversión, la misma que se denota de la siguiente manera:

(tipo1) E

Lo que se está haciendo es aplicando una conversión de tipo al resultado de *E*. Donde *E* es una expresión cuyo tipo es *tipo2*, y al evaluarla devolverá un resultado de *tipo2*, con la conversión ese resultado se convertirá a *tipo1*, por supuesto siempre y cuando sea posible. Lo mismo se aplica cuando se hace uso de clases, recordemos que la clase identifica *tipos de objetos*.

Las conversiones, no son un hecho, se realizan cuando las condiciones se dan, por ejemplo veamos la siguiente secuencia:

```
1 MedioDeTransporte med1 = new Auto();   Correcto
2 Auto aut1 = (Auto)med1;                Correcto
3 Bicicleta bic1 = (Bicicleta)med1;      Incorrecto
```

La tercera instrucción es incorrecta, ya que lo que contiene *med1* no es un objeto de la clase Bicicleta, y la conversión no es posible.

Para que la conversión se realice debe ser posible y además existir relación generalización/especialización entre los tipos.

La conversión no se aplica a cualquier contexto. Veamos el siguiente escenario:

Sea *cont1* de la clase *A*
Sea *cont2* de la clase *B*
Sea *cont3* de la clase *C*
Sea *cont4* de la clase *D*
A *es un* B
C *es un* B

1	cont1 = new A();	Correcto
2	cont2 = new C();	Correcto
3	cont3 = (C) cont2;	Correcto
4	cont4 = (D) cont2;	Incorrecto
5	cont1 = (A) cont2;	Incorrecto

La intrucción 4, es incorrecta porque no existe ninguna relación de tipos entre *B* y *D*. La instrucción 5 es incorrecta por las mismas razones que *Bicicleta bic1 = (Bicicleta)med1;*, a pesar de que existe relación de tipos entre *A* y *B*.

3.1.3. Operador **instanceof**

Para asegurar la conversión y evitar errores en tiempo de ejecución, se tiene el operador booleano **instanceof** cuya misión es verificar el tipo de la instancia que contiene una contenedora:

1	MedioDeTransporte med1 = new Auto();	Correcto
2	Auto aut1 = (Auto)med1;	Correcto
3	if(med1 **instanceof** Bicicleta)	
	Bicicleta bic1 = (Bicicleta)med1;	Correcto

Ahora la instrucción 3, antes de realizar la conversión, verifica si es posible. Esto asegura que no se genere un error en tiempo de ejecución. El operador **instanceof** tiene dos parámetros: un objeto *obj1* y una clase *claseA*, y lo que hace es verificar si la instancia del objeto *obj1* es de la clase *claseA*. Su notación es:

obj1 **instanceof** *claseA*

3.2. La herencia en la relación *es un*

Tal vez una pregunta obligatoria es: Por qué la relación de generalización/especialización se denomina herencia? Intentaremos explicar en esta sección el origen de la nominación.

Parece natural que si un Auto *es un* Medio de Transporte, todas las características identificadas para un medio de transporte las tiene también una auto. Es decir, los atributos y métodos que tiene la clase *MedioDeTransporte* se heredan/transfieren a la clase *Auto* de forma obligatoria (eso es herencia). Veamos el siguiente modelo (en parte):

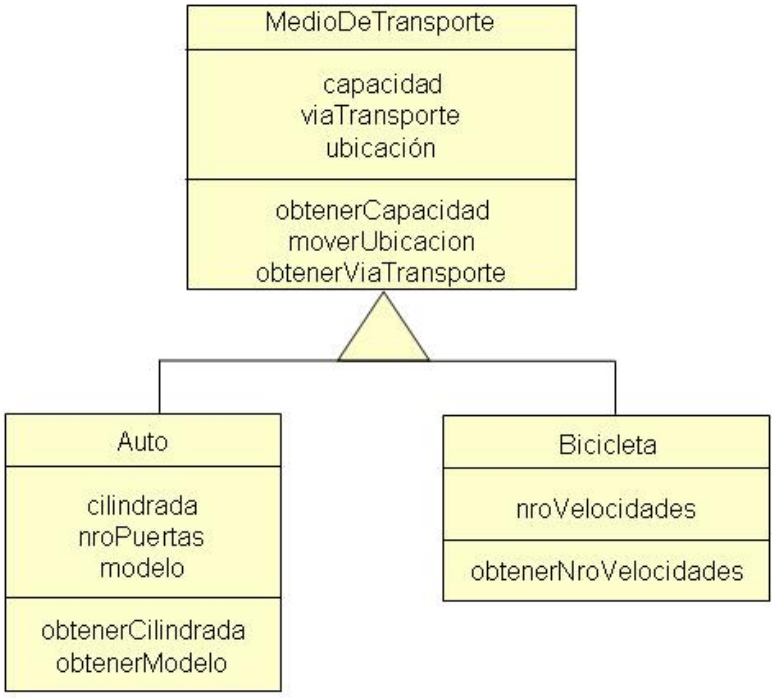

Figura 3.1: Modelo Herencia

Encontremos oraciones elementales directas de este modelo:

Un Auto *es un* MedioDeTransporte
Una Bicicleta *es un* MedioDeTransporte
Un MedioDeTransporte *puede ser* un Auto
Un MedioDeTransporte *puede ser* una Bicicleta
Todo MedioDeTransporte *tiene* capacidad
Todo MedioDeTransporte *tiene* viaTransporte
Todo MedioDeTransporte *tiene* ubicacion
Todod MedioDeTransporte *sabe* obtenerCapacidad
Todod MedioDeTransporte *sabe* moverUbicacion

Todod MedioDeTransporte *sabe* obtenerViaTransporte
Toda Bicicleta *tiene* nroVelocidades
Toda Bicicleta *sabe* obtenerNroVelocidades
Todo Auto *tiene* cilindrada
Todo Auto *tiene* nroPuertas
Todo Auto *tiene* modelo
Todo Auto *sabe* obtenerCilindrada
Todo Auto *sabe* obtenerModelo

Como efecto de la herencia, encontremos oraciones elementales indirectas de este modelo:

Toda Bicicleta *tiene* capacidad
Toda Bicicleta *tiene* viaTransporte
Toda Bicicleta *tiene* ubicacion
Toda Bicicleta *sabe* obtenerCapacidad
Toda Bicicleta *sabe* moverUbicacion
Toda Bicicleta *sabe* obtenerViaTransporte
Todo Auto *tiene* capacidad
Todo Auto *tiene* viaTransporte
Todo Auto *tiene* ubicacion
Todo Auto *sabe* obtenerCapacidad
Todo Auto *sabe* moverUbicacion
Todo Auto *sabe* obtenerViaTransporte

Lo cual es completamente lógico, y se dice que las subclases han heredado las características de su superclase.

3.3. Sobreescritura

La sobreescritura permite a clases extendidas, redefinir el cuerpo de un método heredado. En la siguiente sección se verá un ejemplo de sobreescritura.

3.4. Operador **super**

El operador **super** permite hacer referencia al ámbito de una super clase, desde una subclase.

Usualmente se utiliza cuando hay colisión de nombres entre la super y sub clase o cuando existe sobreescritura.

Por ejemplo si se tiene el siguiente pedazo de código:

```
public class A
{
    private int     atr1;
    private double  atr2;
    public A(int a1, double a2)
    {
        atr1 = a1;
        atr2 = a2;
    }
    public void mostrar()
    {
        System.out.println(atr1 + "\t" + atr2);
    }
}
public class B extends A
{
    private double atr1;
    private double atr3;
    public B(int a1, double a2, double a3)
    {
        super(a1, a2);
        atr1 = a3 * a1;
        atr3 = a3;
    }
 /**
 * Metodo SOBREESCRITO, se esta poniendo
 * otro conjunto de instrucciones al metodo
 * mostrar.
 **/
    public void mostrar()
    {
        super.mostrar();
        System.out.println(atr1 + "\t" + atr3);
    }
```

```
}
```

En esta implementación, se tiene que el *atr1* en la clase B tiene 2 declaraciones y definiciones: una la heredada que es un **int** y la otra propia que es un **double**. El *atr1* heredado está oculto para la clase B por dos razones: es privado en la clase base y esta "ensombrecido" por el *atr1* directo que tiene B.

Por otro lado, la clase B ha *sobreescrito* (redefinido) el método *mostrar*, pero requiere aún el comportamiento heredado. Para hacer uso del comportamiento de la *super clase*, se debe cambiar de ámbito, la forma de hacerlo es con el operador **super** y la instrucción sería: *super.mostrar()*. Cada vez que un objeto de la clase B haga uso de su método *mostrar*, se ejecutará el método definido en la clase B, pero a través de éste método se utilizará el método *mostrar* de la clase A también.

3.5. Modificador de acceso **protected**

Controlar el accesos a las características de un objeto es importante, para asegurar la consistencia de la información que mantienen los mismos. Cuando se habla de herencia, todas las características de la clase base se heredan a las clases extendidas; esto no significa necesariamente que las clases extendidas pueden tener accceso irrestricto a las caraceterísticas heredadas, en el caso de que sean de caracter **private**, no podrán acceder desde las subclases. Una alternativa, si es completamnete necesario es flexibilizar el acceso a un acceso **public**, pero es una medida extrema, ya que no es deseable que cualquier objeto tenga acceso a estas características.

Un control de acceso intermedio, es el **protected** que permite a los objetos de las clases extendidas acceder a recursos protegidos en la clase base.

Veamos el anterior ejemplo y flexibilicemos el acceso a los atributos de la clase base:

```
1  public class A
2  {
3          protected int atr1;
4          protected double atr2;
5          public void mostrar()
6          {
7                  System.out.println(atr1 + ""+ atr2);
8          }
9  }
10 public class B extends A
11 {
12         private double atr1;
13         private double atr3;
14         public B(int a1, double a2, double a3)
15         {
16                 super.atr1 = a1;
17                 atr2 = a2;
18                 atr1 = a3 * a1;
19                 atr3 = a3;
20         }
21         public void mostrar()
22         {
23                 super.mostrar();
24                 System.out.println(atr1 + "" + atr3);
25         }
26 }
```

En la linea 16 y 17 del código propuesto, se está accediendo a los atributos de la clase *A*, debido a que tienen un modificador de acceso **protected**. Hay que notar que en la linea 16 para usar el atributo *atr1* de la clase base se ha requerido el uso del operador **super**, para diferenciar entre el atributo *atr1* de la clase *A* del atributo *atr1* de la clase *B*. En este caso, el método constructor de la clase *B* define los estados de sus atributos de forma directa, incluyendo a los heredados.

3.6. Clases abstractas

Las clases a tiempo de definirlas, no necesariamente deben (o pueden) definir el cuerpo de todos los métodos que representan a los objetos de esta clase. En este caso, se dice que el método representa un comportamiento abstracto, por lo que el método se convierte en un método abstracto, y sólamente se lo declara, considerando las condiciones de contexto - precondiciones y poscondiciones.

Cuando una clase tiene un método abstracto, entonces la clase se denomina abstracta. En consecuencia, de esta clase no se puede instanciar objetos, ya que el objeto estaría incompleto. A su vez esta consecuencia, hace que las clases abstractas sean clases *diferidas*. Esto significa que difiere la definición de los métodos abstractos a otras clases que tendrían que ser sus subclases. Por lo que no tiene sentido tener una clase abstracta, sin pensar en que esté involucrada en una jerarquía de clases, jugando el rol de *superclase*.

Otra situación en la que se puede decidir tener una clase abstracta, es cuando dependiendo del dominio del problema no se desea tener objetos de una clase determinada, entonces un mecanismo para restringir la instanciación es definiendo la clase en cuestión como abstracta.

Consideremos un ejemplo en el que se quiere modelar mascotas, se sabe que toda mascota tiene: nombre, raza y sabe comer, decir algo y moverse. Sin embargo, nos vemos en imposibilidad de decir cómo y qué comen, cómo y qué dicen y cómo y con que se mueven TODAS LAS MASCOTAS, en este caso se dice que si bien es cierto que todas las mascotas saben hacer cosas, no se puede definir con precisión el CÓMO todas lo hacen, es demasiado ambigüo, por lo que la clase *Mascota* sólamente declara estos métodos pero no los define, más bien deja que la definción del método la realicen las clases extendidas, que serán más especificas.

3.7. Herencia múltiple

La herencia múltiple, es otro de los conceptos de la POO, que permite a una subclase extenderse de más de una clase base. Esta propiedad, no es soportada en Java debido a algunos problemas, el principal se conoce como "colisión" de nombres y se da cuando la subclase no sabe el atributo y/o

método de que superclase considerará.

Si observamos la Figura 3.2 se tiene un ejemplo de herencia múltiple y el problema de "colisión" de nombres que puede generar, si no se espcifica explícitamente qué y de dónde se heredarán características.

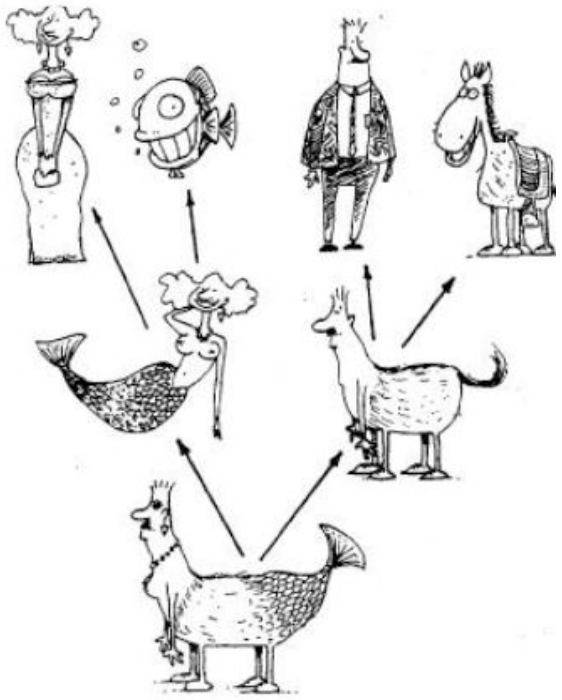

Figura 3.2: Herencia múltiple

La Figura 3.3 muestra otro caso de herencia múltiple.

En este caso se tiene algunas preguntas que hay que responderse: Cuál es la vía de un *Anfibio*?, Cómo se mueve un *Anfibio*?. En términos de herencia la clase *Anfibio* hereda de la clase *Vehiculo* el atributo *viaTransporte*, y de las clases *VehiculoTerrestre* y *VehiculoAcuatico* el método *moverse*. Es muy lógico pensar, que un anfibio tiene dos vias de transporte: terrestre y acuático, pero sólamente tiene un atributo para explicar sus dos condiciones. Los vehículos terrestres, se mueven por tierra y los acuáticos por agua, entonces todo Anfibio tiene las dos vías? no, porque sólo una de ellas debería heredar, entonces la clase *Anfibio* tiene que decidir: de dónde heredar o en otro caso decidir su propio atributo *viaTransporte* que tendrá dos valores.

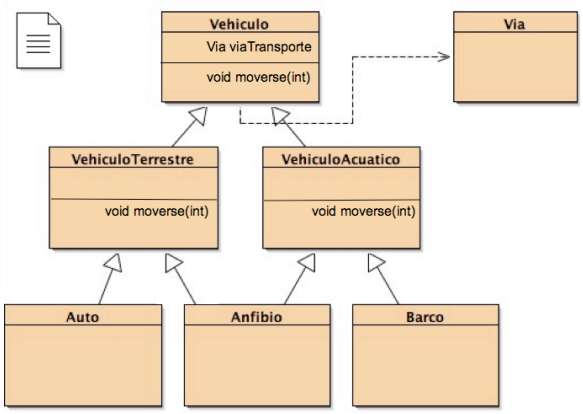

Figura 3.3: Modelo de herencia múltiple

En el caso del método *moverse* está en la misma disyuntiva, dependiendo en que vía de transporte se encuentre se moverá como un vehículo terrestre o como uno acuático. Entonces el *Anfibio* de nuevo se ve en la necesidad de redefinir el método *moverse* considerando sus estados propios. Por lo que el modelo quedaría como la ilustración de la Figura 3.4.

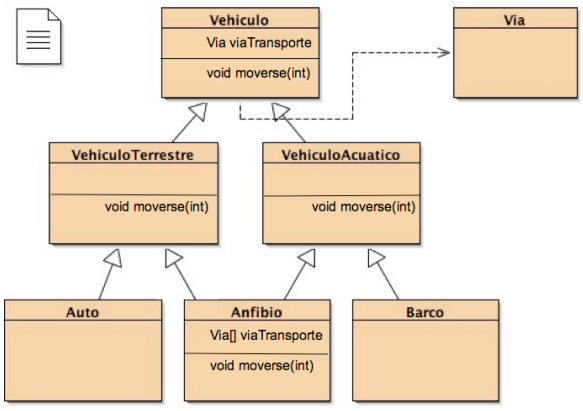

Figura 3.4: Modelo herencia múltiple: decisiones de la subclase

En el mundo real, los objetos heredan de una clase la mayoría de las características escenciales y usualmente suelen estar bajo otras clasificaciones cuando quieren aumentar, sobre todo, su comportamiento de generalización, es decir, que los puedan identificar como objetos de otras clasificaciones de generalización.

3.8. Interfaces

Las interfaces son un mecanismo que inicialmnete Java ha provisto para evitar el conflicto de "colisión", y que permite de alguna manera permite a las clases ser subtipificadas por mas de un modelo.

Las interfaces, en términos POO, vienen a ser los puntos a través de los cuales los objetos colaboran/inetractuan con otros, esto mediante los métodos que se han definido de caracter **public**.

Es posible también definir *tipos de interfaces* comunes a grupos de objetos, que enumeren los comportamientos que se desea que sean los puntos de acceso. En Java a esto se le llama una **interface**.

Una **interface**, declara una lista de métodos, es decir no los define. Usualmente, enlista los comportamientos que son comunes a ms de un tipo de objetos, y al tener los métodos sólo declarados, la responsabilidad de definición de los métodos recae en las clases que deciden comportarse de acuerdo al modelo de la **interface**.

Nos aventuramos a decir que la interface identifica comportamiento común a distintos tipos de objetos, en cambio la clase identifica las propiedades y comportamientos de un tipo de objeto.

Otra característica de las interfaces es que, todos los comportamiento enlistados son de caracter **public**, eso es lo que hace que el modelo represente a una interfaz.

Las interfaces, no factorizan código, y esa es una de las diferencias con la relación de herencia entre clases. Sin embargo, cumple el rol de generalización en una jerarquía de clases. Las interfaces representan abstracciones de generalización diferida.

No se pueden instanciar objetos de una interface, ya que no tiene cuerpo - al igual que de las clases abstractas. Pero es muy util en la tipificación de contenedoras de generalización.

Las interfaces proveen de herencia múltiple entre ellas y anulan los problemas de "colisión" de nombres ya que de todas maneras obliga a las clases que se extienden de ella a implementar los métodos descritos en la interface.

Un otro ejemplo de herencia múltiple que puede ser mejorado con concepto de **interface**, es el que se ilustra en la Figura 3.5, en este caso se tiene un contexto bien conocido en la universidad que muestra la relación que existe entre *Estudiante, Docente* y *Auxiliar*.

De este modelo se puede deducir, que todo *Auxiliar* además de lo que tiene todo *Estudiante* tiene *salario, material de ense nanza* y sabe *ense nar*

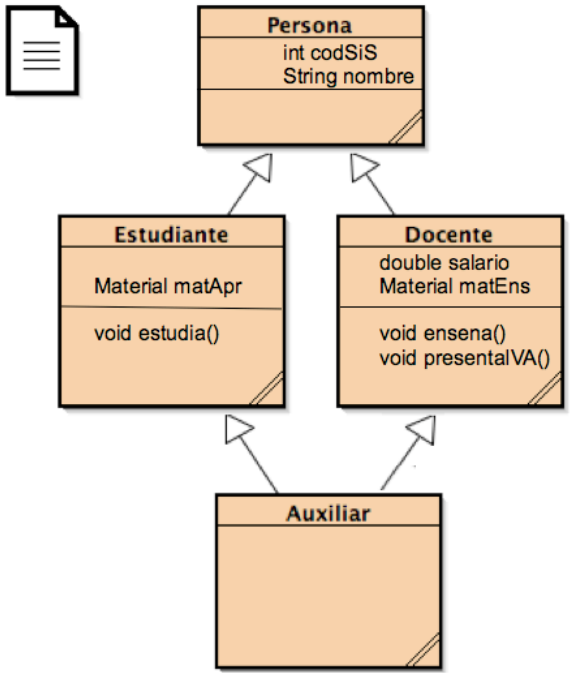

Figura 3.5: Modelo con herencia multiple

y presentar IVA. Esto no es verdad, los auxiliares no tienen sueldos, sino más bien honorarios, los mismos que no generan una realción obrero - patronal y no tienen que presentar descargo por impuestos al valor agregado (IVA), sin embrago si son parte del proceso de ense nanza y saben ense nar. Por la herencia, no se puede no heredar, en caso extremo se debería redefinir las características de todo *Auxiliar*.

Se propone un modelo alternativo (ver Figura 3.6, en el que solamente se tiene el comportamiento común a ambos tipos de objetos: *Auxiliar* y *Docente*. El comportamiento común se modela en una **interface** la misma que es extendida por *Auxiliar* y *Docente*, con el compromiso que cada una de estas clases definirá la implementación del comportamiento de acuerdo a sus estados.

Algunas de las oraciones elementales de la lectura de este modelo son:

Un Estudiante *es una* Persona
Un Docente *es una* Persona
Un Auxiliar *es un* Estudiante
Un Auxiliar *se comporta como un* Ensenante

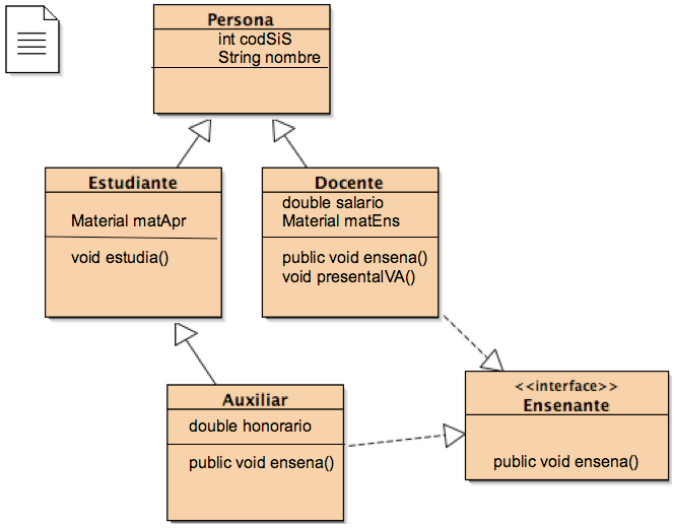

Figura 3.6: Modelo que incluye definición de interface

Un Docente *se comporta como un* Ensenante
Una Persona *puede ser* un Estudiante
Una Persona *puede ser* un Docente
Una Persona *puede ser* un Auxiliar
Un Estudiante *puede ser* un Auxiliar
Un Ensenante *puede ser* un Auxiliar
Un Ensenante *puede ser* un Docente

En este caso, la **interface** *Ensenante* se constituye en un modelo de generalización, por lo que donde se espere un *Ensenante* puede asignarse un *Docente* ó un *Auxiliar*.

La forma de relacionar en términos de Java una interfaz con una clase es a través de la palabra reservada **implements** y la referencia es siempre de la subclase a la superclase. El esqueleto de la implementación del ejemplo se muestra en el siguiente código:

```
1  public class Persona
2  {
3          ......
4  }
5  public class Estudiante extends Persona
6  {
7          ......
8  }
9  public interface Ensenante
10 {
11         public void ensena();
12 }
13 public class Docente extends Persona implements Ensenante
14 {
15         ......
16         public void ensena()
17         {
18           ......
19         }
20 }
21 public class Auxiliar extends Estudiante implements Ensenante
22 {
23         ......
24         public void ensena()
25         {
26           ......
27         }
28 }
```

Las interfaces se guardan en archivos que llevan el mismo nombre de la **interface** respectiva y con extensión .java.

3.9. Ejercicios

1. Dadas la siguientes descripciones, define cuáles son candidatas a clase y cuáles candidatas a objetos, por qué? Define el modelo que mostraría el contexto y sus relaciones.

a) En una villa se tienen varios pobladores que viven en armonía, ya que suelen compartir el lugar de trabajo, y los lugares de distracción. Juan Pablo es uno de los habitantes de esta villa que le gusta ir mucho al Parque "El aguilita" ya que tiene muchos juegos sobre todo infantiles.

b) Una empresa de telecomunicaciones se caracteriza por brindar diferentes servicios, tales como la telefonía celular, entre otros. Una de sus unidades se encarga de atender a los clientes que tiene la empresa y ayuda a resolver problemas que los mismos tengan con algún servicio que ofrecen. La velocidad de atención de los operadores de atención al cliente es crucial ya que esto puede generar una llamada de atención por la superintendencia de telecomunicaciones. Se dió un caso en el que Maria (una operadora) se tardó en atender una solicitud de reclamo mas de 20 días por lo cual Honorato Orfebre, uno de los mejores clientes elevó una queja.

c) Un sistema de identificación de personas en la oficina de lucha contra el crimen desea tener formas más rápidas de construir un dibujo que muestre los rasgos de un sospechoso. Se sabe que los dibujos de las caras se dividen en tres regiones: la parte alta, la parte media y la baja. En la parte alta se encuentran elementos como el cabello, la frente, los ojos y las cejas. En la parte media se describe la nariz y en la parte baja la boca y barbilla. De cada elemento se puede tener a su vez ciertos modelos establecidos como: boca grande y carnosa o boca chica y carnosa.

2. Desarrolle los programas que permitan resolver los siguientes problemas. Para ello se requiere el modelo, el programa y definir casos de prueba.

   a) En una villa se tienen varios pobladores que viven en armonía, ya que suelen compartir el lugar de trabajo, y los lugares de distracción. Juan Pablo es uno de los habitantes de esta villa que le gusta ir mucho al Parque "El aguilita" ya que tiene muchos juegos sobre todo infantiles. Otra punto de distracción son los cines, la mayoría de ellos cuentan con distintas salas en las cuales generalmente se dan distintas películas en distintos horarios.

   La forma en que la villa genera recursos es quedándose con el 3 % de las recaudaciones diarias de los lugares de diversión, los mismos

que son cobrados cada día.

Los parques recaudan de distintas fuentes. Determinan un costo a la entrada al parque diferenciada, una para mayores y otra para ni nos.

Cada juego dentro de un parque, también tiene un costo de uso el mismo que no tiene diferenciación alguna. Cada juego debe registrar el nombre, la edad mínima y la edad máxima de los que pueden acceder al juego y la cantidad de personas que ha atendido.

Los cines recaudan sus fondos por cobro de entradas a las salas que tiene, las mismas que son diferenciadas para ni nos y mayores. Las salas de los cines por dia registran la película que proyectarán, los horarios de proyección (hh:mm), la capacidad de la sala, el número de entradas vendidas y el número de sala.

Las películas necesitan el título de la misma, una lista de actores principales, la duración en minutos y la categoría (terror, comedia, etc.).

De cada poblador es importante tener el nombre y su edad.

En este contexto se pide:

1) Calcular el monto que le corresponde a la villa.
2) Un reporte de los parques que cuentan con juegos aptos para personas menores de X a nos.
3) Un reporte de las salas que están proyectando la película X, además de sus horarios.
4) Las salas que estan proyectando películas de categoria Y.
5) La recaudación de los parques por sólo cobro de uso de juegos.

Cuando un juego es para toda edad, la forma de indicar esto es en edad mínima 0 y en edad máxima 0. Esto quiere decir que el 0 identifica la ilimitación en edad, de este modo se puede configurar un juego en distintas rangos de edad sin límite.

b) Se desea hacer un sistema que permita construir componentes electrónicos en base a partes. Las condiciones es que las partes pueden se catalogadas en forma general como: procesadores, dispositivos I, dispositivos O, unidades de almacenamiento, tarjetas adicionales. Para construir cada componente se requiere especificar la cantidad que soporta de cada tipo de parte. Una condición

es que jamás puede tener dos partes que son la misma. Es decir, puede tener dos procesadores, pero no pueden ser el mismo. Cada una de las partes se identifica unívocamente por un código.

Realice:

1) El modelo que resuelva el problema y sus relaciones
2) El programa que resuelva el problema considerando todas los requerimientos que se detallan más adelante

Los requerimientos que se tiene son los siguientes:

1) El programa debe permitir crear una placa base para cada componente, que permita indicar el número de partes que se quiere de cada tipo.
2) El programa debe permitir, insertar una parte en el componente, considerando las condiciones de unicidad.
3) El programa debe controlar que jamás se introduzca más partes de las que se permite.
4) El programa debe controlar que de un tipo de parte no se introduzca más elementos que los permitidos.
5) El programa debe emitir un reporte, cuando el componente en su placa base esta completo.
6) El programa debe mostrar el (los) componentes creados e indicar si esta completo o si tiene espacio para recibir mas partes. Además debe indicar qué tipo de parte se espera.
7) Se quiere saber cuántos partes puede contener un componente.

De los procesadores se tiene la velocidad de procesamiento, el modelo y la marca, además de un código de identificación. De los dispositivos I, un código de identificación y la descripción, lo mismo para dispositivos O. De las unidades de almacenamiento, se requiere su capacidad de almacenamiento direccional, capacidad de registro, su condición (volátil/no volátil), marca y código de identificación. De las tarjetas adicionales, se necesita marca, modelo, función (red, sonido, gráfica, etc.) y código de identificación.

3. Al anterior problema extendamos con los siguientes requerimientos:

   a) Se debe permitir el reemplazo de una parte en el componente. En este requerimiento hay que cuidar que el tipo de parte a reemplazar sea consistente con el tipo de parte que se está queriendo

reemplazar. Para efectuar el reemplazo, se requiere la parte que reemplazará y el codigo de la parte a ser reemplazada.

b) Se quiere saber cuánto es el costo de un componente X, de acuerdo a la configuración de partes. Cada parte tiene un precio.

c) Se debe permitir el retiro de una parte de un componente. Para ello sólo se necesita el código de identificación de la parte a retirar.