Unidad 4 Modularización

Introducción a la Programación (IP) Tecnicatura en Desarrollo Web

31 agosto, 2023

1 Introducción

Modularizar es una estrategia de resolución de problemas y de ingeniería de software que consiste en dividir el problema original en un conjunto de subproblemas . Consiste en aplicar una estrategia heurística fundamental: descomposición y recombinación . Esta heurística consiste en poder descomponer un todo en sus partes y poder recombinar las partes en un todo.

En programación y en el diseño de algorítmos, un buen diseño estructurado persigue elaborar algorítmos que cumplan la propiedad de modularidad , esto es dado un problema que se pretende resolver mediante la utilización de una computadora, se busca dividir dicho programa en módulos independientes.

El diseño estructurado incluye la descomposición, para lo cual se requiere un adecuado análisis de dicho problema, siendo necesario definir primeramente el problema. Merece la pena el esfuerzo de dividir un problema grande en subproblemas más pequeños.

Ahora la cuestión es ¿cómo realizar la descomposición?; una manera es realizando un estudio descendente (top-down) que nos lleve desde la concepción del problema (algorítmo, programa…) global hasta identificar sus partes (módulos).

Esta técnica se repite una y otra vez refinando el problema hasta obtener subproblemas suficientemente pequeños, que puedan ser resueltos por módulos que cumplan, en la medida de lo posible, las características deseables en un módulo en el ámbito de la programación. En cada paso del refinamiento, una o varias instrucciones del programa dado, se descomponen en instrucciones más detalladas.

¿Cuándo parar el refinamiento?. Un refinamiento excesivo podría dar lugar a un número tan grande de módulos que haría poco práctica la descomposición. Se tendrán en cuenta este criterio para dejar de descomponer: Cuando el MODULO definido realice una única tarea, lo suficientemente simple y entendible, y no existan subtareas que requieran descomposición.

El uso apropiado de la modularización tiene importantes ventajas:

1. Cada módulo debe hacer una única cosa (y de manera genérica).
2. Cada módulo oculta algo (encapsulamiento).
3. Facilita el desarrollo del software, dado que los subproblemas con más fáciles de resolver
4. Facilita la reutilización del software, dado que un módulo se puede usar en muchos programas.

5. Facilita el mantenimiento. Se puede profundizar en las pruebas de cada módulo más de lo que se hace un programa mayor.

Los módulos se pueden ver como cajas que tiene datos de entrada, que realizan alguna acción y devuelven o no algún valor al respecto.

Para entender claramente en que consiste la modularización, mostraremos un ejemplo introductorio.

2 Definición de Funciones

Una de las herramientas más importantes en cualquier lenguaje de programación son los módulos o funciones. Un modulo o función es un conjunto de instrucciones que a lo largo del programa van a ser ejecutadas las veces que se requiera. Es por ello, que este conjunto de instrucciones se agrupan en el módulo o función. Los módulos o funciones pueden ser llamados y ejecutados desde cualquier punto del programa.

Los nombres de los módulos o funciones siguen las mismas reglas que otras etiquetas de PHP. Un nombre de función válido comienza con una letra o guión bajo, seguido de cualquier número de letras, números, o guiones bajos.

Como expresión regular se expresaría así: [a-zA-Z_][a-zA-Z0-9_]*.

Ejemplo:

```
(** funcion que saluda a una persona
* cuyo nombre se recibe como parámetro
*
*)
MÓDULO saludar(TEXTO nombre) RETORNA ∅
  ESCRIBIR("Hola "+ nombre +"!")
FIN MÓDULO saludar

ALGORITMO principal RETORNA ∅
  (* saludar a Juan y Maria*)
  saludar("Juan");
  saludar("Maria");
FIN ALGORITMO principal
```

```
<?php

/** funcion que saluda a una persona
* cuyo nombre se recibe como parámetro
* @param STRING $nombre
*/
function saludar($nombre) {
    echo "¡Hola".$nombre."!";
}

// Principal
// Llamamos a la función
saludar("Juan");
saludar("Maria");
?>
```

La primera línea del pseudocódigo nos indica que este módulo realiza una acción pero no debemos esperar que devuelva nada, esto se detalla en la parte RETORNA Ø.

A continuación se expresa la sintáxis general para declarar una función en PHP:

```
function nombre(parametro1,parametro2,..., parametroN){
  instruccion1
  instruccion2
  ...
  ...
  instruccionN
}
```

Las variables encerradas en los paréntesis del encabezado del módulo ($parametro1, $parametro2, . . . , $parametroN) se las llama parámetros formales.

Una función recibe parámetros actuales (valores externos) en sus parámetros formales o de entrada, de los cuales va a depender el resultado de dicha función. Es decir, según el parámetro o parámetros con los que se invoque a la función, ´esta devolverá un resultado u otro.

Para invocar a (hacer que se ejecute) la función usaremos esta sintaxis: nombre($par1, $par2, $par3, …, $parN); donde $par1, $par2, $par3, … , $parN son los parámetros actuales (información) que le pasamos a la función. Una función puede necesitar ningún, uno o varios parámetros para ejecutarse, lo que va a depender de la funcionalidad que implemente la función.

3 Retornando resultado

Los módulos o funciones pueden retornar un valor como resultado de su ejecución, para esto se requiere incluir la primitiva RETORNAR (en pseudocódigo) ó la sentencia **return** (en PHP) seguida del valor retornado. Esta sentencia si existe debe ser la última sentencia del módulo o función. Es posible retornar una variable, un número, una cadena de texto, etc.

Por ejemplo en PHP: **return** ”No dispone de permisos”; /* significa que la función devuelve esta cadena de texto.*/

Otro ejemplo en PHP: **return** $calculo; /* indica que la función devuelve el contenido que se encuentre almacenado en la variable $calculo.*/

Otro ejemplo en PHP: **return** ”Lo sentimos ”.$usuario.” pero no dispone de permisos. Para solicitar información puede escribir a ”.$emailAdministrador; /* haría que la función devuelva una cadena de texto donde intervienen diversas variables.*/

4 Trazas

Las trazas de estos algorítmos deben cumplir lo siguiente.

1. Se deben crear tantas tablas de trazas como módulos sean invocados
2. Cada tabla ademas de las variables propias del método, incluirá parámetros y el valor retornado
3. El método invocador, o main, incluirá además una columna etiquetada como ’Salida por Pantalla’
4. Si una misma función es invocada más de una vez deberemos crear nuevas instancias de la misma tabla de la función

Ejemplo: Haremos un ejemplo de una traza del algoritmo saludosVarios considerando que el usuario ingresará un primer nombre “Juan” y un segundo nombre “Pedro”.

```
(** escribe un saludo de la persona
    ingresada como parametro *)
MÓDULO saludo(TEXTO nombre) RETORNA ∅
    ESCRIBIR(”Hola, buenos días!” + nombre)
    ESCRIBIR(”¿Cómo le va?”)
FIN MÓDULO saludo ´

ALGORÍTMO saludosVarios() RETORNA ∅
    (* saluda a dos hermanos *)
    TEXTO nombre1, nombre2
    ESCRIBIR(”Ingrese el nombre de la primera persona”)
    LEER(nombre1)
    ESCRIBIR(”Ingrese el nombre de la segunda persona”)
    LEER(nombre2)
    saludo(nombre1)
    saludo(nombre2)
FIN ALGORÍTMO saludosVarios
```

saludosVarios()

nombre1	**nombre2**	**Salida por pantalla**
Juan	Pedro	Ingrese el nombre de la primera persona
		Ingrese el nombre de la segunda persona
		Hola, buenos días! Juan
		¿Cómo le va?
		Hola, buenos días! Pedro

saludo(“Juan”)

nombre
Juan

saludo(“Pedro”)

nombre
Pedro

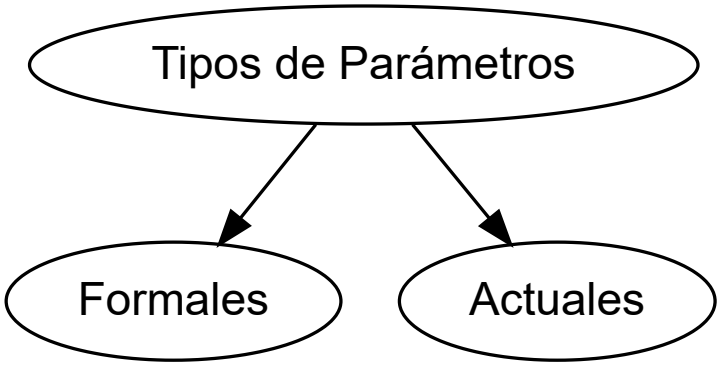
nombre1	nombre2	Salida por pantalla
		¿Cómo le va?

En la última sección incluiremos un ejemplo de un módulo o función que retorna un valor, con el objetivo de mostrar como representar el valor retornado.

5 Más sobre Parámetros

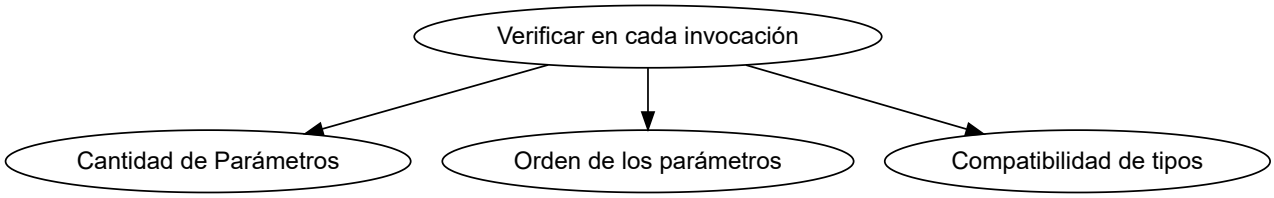
Habiamos expresado anteriormente que las variables encerradas en los paréntesis del encabezado del módulo se las llama parámetros.

Una función recibe parámetros actuales (valores externos) en sus parámetros formales o de entrada, de los cuales va a depender el resultado de dicha función. Es decir, según el parámetro o parámetros con los que se invoque a la función, ´esta devolverá un resultado u otro.



Es importante notar que cuando invocamos a un módulo no es necesario utilizar una variable con el mismo nombre del parámetro de definición del módulo. Si es necesario que los valores sean de tipos compatible. La variable que se utiliza en la invocación como parámetro (hs, mins y segs) se denomina parámetro actual. En cambio la variable que se utiliza como parte de la de definición del módulo (horas, minuots y segundos) se denomina parámetro formal.

No es necesario que el parámetro actual tenga el mismo nombre que el parámetro formal, simplemente que se requiere que sean de tipos compatibles. La cantidad de parámetros puede ser mayor a uno, con lo cual, al invocar al módulo debemos pasar la misma cantidad de parámetros en el orden esperado, y los tipos de los parámetros deben coincidir.



Cuando la cantidad es superior a uno, se separan los mismos utilizando comas. Los algorítmos que creemos pueden utilizar modularización y tienen un solo punto de acceso. El punto de acceso es el algorítmo principal.

¡Importantísimo!

```
Siempre debemos verificar 3 cosas en la correspondencia entre parámetros actuales y formales:
  + cantidad de parámetros
  + conformidad de tipos de los parámetros
  + orden de los parámetros
```

6 Funciones Predifinidas

En PHP, además de las funciones creadas por el usuario, existen múltiples funciones para realizar distintas tareas. Todas ellas están ya predefinidas, por lo que lo único que tenemos que hacer es llamarlas, pasarles los parámetros necesarios, y ellas realizan la tarea.

Para poder usarlas debemos conocer, el nombre de la función, los parámetros que debemos pasar, y por supuesto saber que tarea realizan.

Ya hemos visto algunas de estas funciones por ejemplo: trim($param) y fgets(STDIN), que nos permiten leer los valores ingresados por el usuario. También hemos utilizado las funciones pow($base, $exponente) para obtener $\$base^{\$exponente}$ y sqrt($base) para obtener la raíz cuadrada de un valor $base, es decir $\sqrt{\$base}$. Por supuesto, hay muchas más funciones predefinidas que vamos a ir viendo con el transcurso de la materia.

La totalidad de las funciones predefinidas que pueden usarse en PHP la podemos ver desde la página http://php.net/manual/es/funcref.php (http://php.net/manual/es/funcref.php) Desde ahi se encuentran los distintos enlaces para ver las funciones predefinidas, agrupadas por categorías.

Entre la categoría de funciones predefinidas podemos consultar las funciones matemáticas http://php.net/manual/es/book.math.php (http://php.net/manual/es/book.math.php).

7 Documentación

Cada módulo o función implementado por un programador realiza una tarea específica. Cuando la cantidad de funciones disponibles para ser utilizadas es grande, puede ser difícil saber exactamente qué hace una función. Es por eso que es extremadamente importante documentar cada módulo o función indicando cuál es la tarea que realiza, cuáles son los parámetros que recibe y qué es lo que devuelve.

La documentación de una función se coloca antes del encabezado de la función, en un párrafo encerrado entre /** y */. Así, para la función vista en el ejemplo anterior:

```
<?php
 /*** Imprime por pantalla un saludo,
 * dirigido a la persona que se indica por parámetro
 * @param STRING $alguien
 */
 function saludo( $alguien ){
 echo "\n Hola " . $ alguien . " !" ;
 echo "\n Estoy programando en PHP." ;
 }

 echo " Ingrese su nombre:" ;
 $nombre = trim( fgets( STDIN ) ) ;
 saludo( $nombre ) ;
 ?>
```

Cuando una función está correctamente documentada, es posible acceder a su documentación desde el entorno de programación que estamos utilizando

Fig. 1. comentarios desde el entorno de programación Eclipse

8 Imprimir versus devolver

A continuación se define una función print asegundos (horas, minutos, segundos) con tres parámetros (horas, minutos y segundos) que imprime por pantalla la transformación a segundos de una medida de tiempo expresada en horas, minutos y segundos. Para ver si realmente funciona, podemos invocar a la función de la siguiente manera: print asegundos (1, 10, 10), el resultado obtenido de la invocación: Son 4210 segundos

```
(** Transforma en segundos una medida de tiempo
   expresada en horas, minutos y segundos *)
MODULO aSegundos(ENTERO horas, mins, segs) RETORN
A ∅
  ENTERO segsal
  segsal <- 3600 * horas + 60* mins + segs
  ESCRIBIR(" son: " + segsal + " segundos ")
FIN MODULO aSegundos

ALGORITMO principal RETORNA ∅
   (* *)
   aSegundos(1,10,10)
   aSegundos(2,10,20)
FIN ALGORITMO principal
```

```
<?php
 /**
 * Transforma en segundos una medida de tiempo
 * expresada en horas, minutos y segundos
 * @param INT $horas
 * @param INT $minutos
 * @param INT $segundos
 */

 function aSegundos($horas, $mins, $segs){
    // regla de transformación
    $segsel = 3600 * $horas + 60* $mins + $segs;
    echo "Son: " + $segsel + " segundos ";
 }

// principal
aSegundos(1, 10, 10);
aSegundos(2, 10, 20);
```

Contar con funciones es de gran utilidad, ya que nos permite ir armando una biblioteca de instrucciones con problemas que vamos resolviendo, y que se pueden reutilizar para resolver nuevos problemas. Sin embargo, más útil que tener una biblioteca donde los resultados se imprimen por pantalla, es contar con una biblioteca donde los resultados se devuelven, para que la gente que usa esas funciones manipule los resultados a su voluntad: los imprima, los use para realizar cálculos más complejos, etc.

```
(** Transforma en segundos una medida de tiempo
   expresada en horas, minutos y segundos *)
MODULO aSegundos(ENTERO horas, mins, segs) RETORN
A ENTERO
  ENTERO segsal
  segsal <- 3600 * horas + 60* mins + segs
  RETORNAR segsal
FIN MODULO aSegundos

ALGORITMO principal RETORNA ∅
 (* ....*)
 ENTERO cantSeg, cantSeg2, sumSeg
 cantSeg <- aSegundos(1,10,10)
 ESCRIBIR("La cant. de segundos es: "+ $cantSeg)
 cantSeg2 <- aSegundos(2,10,20)
 ESCRIBIR("La cant. de segundos es: "+ $cantSeg2)
 sumSeg <- cantSeg + cantSeg2
 ESCRIBIR("La suma de segundos es: "+sumSeg)
FIN ALGORITMO principal
```

```
<?php
 /**
 * Transforma en segundos una medida de tiempo
 * expresada en horas, minutos y segundos
 * @param INT $horas
 * @param INT $minutos
 * @param INT $segundos
 * @return INT
 */

function aSegundos($horas, $mins, $segs){
// regla de transformación
$segsel = 3600*$horas + 60*$mins + $segs;
return $segsel;
}

// principal
// variables INT $cantSeg, $cantSeg2, $sumSeg
$cantSeg = aSegundos(1, 10, 10)
echo "La cant. de segundos es: " . $cantSeg;
$cantSeg2 = aSegundos(2, 10, 20);
echo "La cant. de segundos es: " . $cantSeg2;
$sumSeg = $cantSeg + $cantSeg2
echo "La suma de segundos es: " . $sumSeg;
```

De esta forma, es posible realizar distintas operaciones con el valor obtenido de la función y no solo obtener una visualización de un resultado.

9 Cómo usar una función

Una función es útil porque nos permite repetir la misma instrucción con diferentes valores en sus parámetros actuales (si es que los tiene), todas las veces que las necesitemos en un programa.

Para utilizar las funciones definidas anteriormente, escribiremos un programa que pida dos duraciones, y en los dos casos las transforme a segundos y las muestra por pantalla.

1. **Análisis**: El programa debe pedir dos duraciones expresadas en horas, minutos y segundos, y las tiene que mostrar en pantalla expresadas en segundos.
2. **Especificación**:

- Entradas: dos duraciones leídas de teclado y expresadas en horas, minutos y segundos.
- Salidas: Mostrar por pantalla cada una de las duraciones ingresadas, convertidas a segundos. Con los valores de las entradas (h, m, s) se calcula 3600 * h + 60 * m + s, y luego se visualiza el resultado por pantalla.

3. **Diseño**:

- Se tienen que leer dos duraciones (expresadas en horas, minutos y segundos) y para cada una de ellas invocar a la función. El pseudocódigo final quedaría como se muestra a continuación.

4. **Implementación**: A partir del diseño, se escribe el programa PHP que se muestra a continuación:
5. **Prueba**: Probamos el programa con las ternas (0,1,0) y (0,0,1):

```
(** Transforma en segundos una medida de tiempo
   expresada en horas, minutos y segundos *)
MODULO aSegs(ENTERO horas, mins, segs) RETORNA EN
TERO
  ENTERO segsal
  segsal <- 3600 * horas + 60* mins + segs
  RETORNAR segsal
FIN MODULO aSegs

ALGORITMO principal() RETORNA ∅
  (* ....*)
  ENTERO hs, min, seg, cantseg
  ESCRIBIR("¿Cuántas horas?")
  LEER(hs)
  ESCRIBIR("¿Cuántos minutos?")
  LEER(min)
  ESCRIBIR("¿Cuántos segundos?")
  LEER(seg)
  cantSeg <- aSegs(hs, min, seg)
  ESCRIBIR("La cantidad de segundos es: "
            + cantSeg)

  ESCRIBIR("¿Cuántas horas?")
  LEER(hs)
  ESCRIBIR("¿Cuántos minutos?")
  LEER(min)
  ESCRIBIR("¿Cuántos segundos?")
  LEER(seg)
  cantSeg <- aSegs(hs, min, seg)
  ESCRIBIR("La cant. de segundos es: "
             + cantSeg)
FIN ALGORITMO principal
```

```
<?php
 /**
  * Transforma en segundos una medida de tiempo
  * expresada en horas, minutos y segundos
  * @param INT $horas
  * @param INT $mins
  * @param INT $segs
  * @return INT
  */

 function aSegs($horas, $mins, $segs){
    // regla de transformación
    $segsel = 3600 * $horas + 60* $mins + $segs;
    return $segsel;
 }

// principal
// INT $hs, $min, $seg, $cantseg
echo "¿Cuántas horas? \n";
$hs = trim(fgest(STDIN));
echo "¿Cuántos minutos?\n";
$min = trim(fgest(STDIN));
echo "¿Cuántos segundos?\n";
$seg = trim(fgest(STDIN));
$cantSeg = aSegs($hs, $min, $seg)
echo "La cant. de segundos es: \n". $cantSeg

echo "¿Cuántas horas?\n";
$hs = trim(fgest(STDIN));
echo "¿Cuántos minutos?\n";
$min = trim(fgest(STDIN));
echo "¿Cuántos segundos?\n";
$seg = trim(fgest(STDIN));
$cantSeg = aSegs($hs, $min, $seg)
echo "La cant. de segundos es:\n". $cantSeg
?>
```

Haremos una traza del programa PHP para ejemplificar trazas con módulos que retornan un valor. Supondremos que las entradas para horas, minutos y segundos son: 01:10:11 y 02:03:15.

principal()

$hs	$min	$seg	$cantSeg	Salida
~~1~~				¿Cuántas horas?
	~~10~~			¿Cuántas minutos?
		~~11~~		¿Cuántas segundos?
			~~4211~~	La cant. de seg. es:4211
2				¿Cuántas horas?
	3			¿Cuántas minutos?
		15		¿Cuántas segundos?
			7395	La cant. de seg. es:7395

aSegs(1,10,11)

$horas	$mins	$seg	valor retornado
1	10	11	4211

aSegs(2,3,15)

$horas	$mins	$seg	valor retornado
2	3	15	7395

¡Importante!

> Si bien la traza mostrada para el algoritmo principal muestra espacios intermedios en cada columna, no es necesario dejar espacios intermedios, ya que cada columna representa los distintos valores que asume una variable en memoria. Lo más correcto es una tabla como la siguiente:

$hs	$min	$seg	$cantSeg	Salida
~~1~~	~~10~~	~~11~~	~~4211~~	¿Cuántas horas?
2	3	15	7395	¿Cuántas minutos?
				¿Cuántas segundos?
				La cant. de seg. es:4211
				¿Cuántas horas?
				¿Cuántas minutos?
				¿Cuántas segundos?
				La cant. de seg. es:7395

10 Importancia de la Modularización

Ahora que hemos visto distintos conceptos sobre la modularización podemos decir que modular favorece la:

1. Construcción de algorítmos y programas * En vez de construir un programa muy grande mejor escribir varios programas pequeños. * Permite que los equipos de programadores trabajen en módulos independientes

(modularidad en mantenibilidad ISO 9126, ISO/IEC 25010).

2. Depuración de algorítmos y programas
    - Depurar un programa muy grande es mucho más difícil que depurar varios programas pequeños
    - La modularización aisla los errores.
3. Lectura de algorítmos y programas
    - Aumenta la legibilidad y comprensión de un programa (usabilidad, inteligibilidad) ISO 9126, ISO/IEC 25010).
    - Un módulo debe ser inteligible a partir de su nombre, los comentarios escritos en su cabecera y los nombres de los módulos que los llaman.
4. Modicación de algorítmos y programas
    - Un pequeño cambio en los requerimientos de un programa debería implicar sólo un pequeño cambio en pocos módulos (capacidad de modificarlo ISO 9126, ISO/IEC 25010)
    - La modularización aisla las modificaciones.
5. Eliminación de redundancia de código
    - Localizar operaciones que ocurren en diferentes lugares de un programa
    - El código de una operación aparecerá una sola vez

11 Otro ejemplo

A continuación un ejemplo de una función PHP para calcular el perímetro de un rectángulo:

```
<?php

  /** Función que calcula el area de un rectangulo
   * @param FLOAT $lado1
   * @param FLOAT $lado2
   * @return FLOAT
  */
  function perimetroRectangulo($lado1, $lado2) {
     // FLOAT $perimetro
     $perimetro = 2 * $lado1 + 2 * $lado2;
     return $perimetro;
  }

  // principal
  // FLOAT $ladoA, $ladoB, $resultado

  echo "Ingrese el valor de un lado \n";
  $ladoA = trim(fgets(STDIN));
  echo "Ingrese el valor del otro lado \n";
  $ladoB = trim(fgets(STDIN));
  $resultado = perimetroRectangulo($ladoA, $ladoB);
  echo "El perímetro del rectángulo es : " .  $resultado ;


?>
```