Unitat Formativa 2. Programació modular

1.1. Concepte

La programació modular és una evolució de la programació estructurada que consisteix en dividir un programa complex en blocs independents, de major simplicitat i de funcionalitat molt específica, completament funcionals de forma desacoblada d'un programa en concret. Això permetrà que aquests blocs es puguin tornar a utilitzar amb altres programes.

Amb aquest canvi de filosofia cal replantejar la forma de programar i entrar en aspectes nous com el **pas de paràmetres i resultats** entre funcions. Ja que per que un bloc sigui reutilitzable ja no podrem assumir que els valor sempre s'introduiran per teclat i es presentaran els resultats per pantalla.

1.2. Avantatges i inconvenients

La programació modular presenta principalment avantatges, l'únic inconvenient és un major esforç per part del programador per estudiar l'estructura dels programa i la seva progressiva descomposició en blocs. Pot succeir que el programa sigui una mica més lent amb alguns casos per la gestió de les comunicacions entre els diferents blocs, però en contrast amb els avantatges son inconvenients menyspreables.

Amb la programació modular podem gaudir dels beneficis de:

- **Disminució de la complexitat del problema**: Cadascun dels blocs en que descompondrem el problema principal per força ha de presentar una dificultat inferior i per tant es pot analitzar per separat i podem obtenir solucions parcials amb les que començar a treballar mentre intentem resoldre aquells blocs que presentin major dificultat.

  Aquest punt també permet comprendre millor el funcionament del programa ja que es pot centrar l'atenció amb blocs d'una mida molt inferior (nombre de línees).

- **Divisió de tasques a realitzar entre un grup de programadors**: Un dels objectius d'aquesta filosofia és la independència funcional dels mòduls, això permet que puguin ser desenvolupats en paral·lel per un conjunt de persones. Per aconseguir aquesta independència modular cal tractar cada mòdul com una "**caixa negra**": un bloc de programa que rep unes dades (o no, depèn de la funció a realitzar), realitza unes operacions (encara que ni tan sols imaginem com les realitza exactament) i torna uns resultats (o no, depèn de la funció a realitzar).

  Amb aquest punt podem reduir el temps de realització del programa i repartir els recursos humans, assignant un major nombre de programadors als mòduls més complexes o bé assignant varis mòduls a un sol programador.

- **Reutilització**: com a resultat indirecte de la descomposició d'un problema en diversos blocs obtenim un codi més especialitzat a cada bloc. Aquesta filosofia de disseny els blocs, l'aplicarem per descompondre tasques que presentin una gran dificultat, pensant que més endavant poden ser reutilitzats i que cal generalitzar al màxim la seva funcionalitat, adquisició i devolució de valors. Tornem al concepte de "caixa negra".

  Amb aquest punt augmenten la productivitat del codi, ja que si els blocs son realment independents i amb una funcionalitat específica cada vegada que els necessitem per a un altre programa els podrem tornar a utilitzar i estalviarem l'esforç d'un nou desenvolupament.

- **Millora de la depuració i el manteniment**: avantatge addicional derivat dels punts anteriors, ja que a l'hora de depurar el funcionament del programa es pot depurar per mòduls, que sempre serà molt més senzill. A més si cal realitzar modificacions, tant al moment de la depuració com durant el posterior manteniment, aquestes estaran fàcilment localitzades a parts molt concretes del programa, concretament a un mòdul o un petit conjunt de mòduls.

1.3. Anàlisis descendent (Top-Down)

El anàlisis descendent és la tècnica que permet modularitzar els programes complexes, aplicant el concepte "divideix i triomfaràs". Es tracta d'analitzar "què" ha de realitzar el programa i dividir-lo en blocs completament independents entre sí. Per exemple: bloc d'adquisició de dades, bloc de comprovació i validació de les dades obtingudes, bloc del tractament de les dades per obtenir uns resultats i bloc de presentació de resultats.

La clau es començar amb un anàlisi superficial però suficient per establir el "què" necessitem i no el "com" ho aconseguirem, per obtenir blocs completament funcionals i independents entre ells. Continuant l'anàlisi descendent agafarem cadascun dels blocs i el descompondrem en altres blocs una mica més específics i també independents entre ells, i així fins que ja no tingui sentit continuar.

Per exemple, si agafem el bloc d'adquisició de dades, podríem treure altres blocs com el de selecció del tipus de l'origen de dades, connexió amb l'origen de dades, adquisició de les dades, ubicació de les dades obtingudes i desconnexió amb l'origen de dades. Si agafem el bloc de adquisició de dades ja podem començar a implementar els mòduls necessaris per obtenir les dades de la font (que per exemple, podria ser una base de dades, un conjunt de fitxers, etc.).

1.4. Modularització de programes

Com a pràctica inicial per començar a treballar amb mòduls i aprofitar el seus avantatges, podem modularitzar els programes més complexes que ja hem realitat a la unitat formativa anterior.

Tornarem a analitzar el problema enfocant-lo amb la perspectiva de la programació modular, i començarem a crear els nostres propis mòduls i trobar-nos amb els típics problemes i errors que es produeixen sempre que afrontem una nova filosofia de treball.

Aspectes que contribueixen a la millora de la qualitat dels programes

- ***Integritat:*** que els càlculs siguin los més exactes possible.
- ***Claredat:*** lectura fàcil del programa, utilitzant tabulacions o espais en blanc per a marcar les diferents sagnies. El codi ha d'estar documentat i ser per sí mateix explícit.
- ***Senzillesa:*** buscar la solució més simple per a l'elaboració del programa.
- ***Eficiència:*** rapidesa d'execució del programa i optimitzar la utilització de recursos.
- ***Modularitat:*** divisió del programa en petits mòduls per a augmentar la claredat i facilitar les modificacions futures.
- ***Generalització:*** utilització de paràmetres generals a través de variable i no de valors fixes, que permetin que el programa es pugui adaptar fàcilment a canvis.

1.5. Disseny i crides a funcions

Per aconseguir una divisió del programa en blocs de funcionalitat definida i independents introduirem el concepte de **funció**. Una funció és un conjunt de sentències que realitzen una tasca concreta, poden rebre una sèrie de valors als que anomenem **paràmetres,** i també poden tornar valors que seran el resultat de les operacions realitzades per la funció.

Els paràmetres rebuts són variables, i poden ser rebuts **per valor** o **per referència:**

- Quan una funció rep un paràmetre per valor, rep realment una còpia del valor de la variable (les modificacions que es puguin fer sobre la còpia **no** canviaran el valor de la variable original).

```
function maxValue ($value1, $value2):int  {
    if ( $value1 > $value2 ) {
            return $value1;
    }
    else {
            return $value2;
    }
}
```

- Quan ho rep per referència, la funció rep realment la adreça de la variable (el que permet arribar fins a la variable original i modificar el seu valor). Especificarem aquesta forma de treballar amb el caràcter & just davant del nom de la variable al declarar la funció.

```
function exchange (&$value1, &$value2)  {
    $aux = $value2;
    $value2 = $value1;
    $value1 = $aux;
}
```

Dins d'una funció també podem definir variables, que es consideren **locals** o **d'àmbit local** (només son reconegudes dins de la funció on es declaren, com al exemple anterior la variable $aux), això determina que el seu **abast** sigui únicament la pròpia funció.

També tenim la opció d'establir valors per defecte pels nostres arguments, **però sempre s'estabiran per als darrers argument de la llista**, els que establim com opcionals, ja que sinó rebem l'argument treballarem amb aquest valor per defecte.

```
function resta ($value1, $value2, $positiva=false):int  {
    if ($positiva == true){
            if ( $value1 < $value2 ) {
                    return ($value2- $value1);
            }
    }
    return ($value1 - $value2);
}
```

Podem establir que els paràmetres de les funcions siguin d'un determinat tipus per estalviar-nos errades sense sentit:

```
function resta (int $value1, int $value2, bool $positiva=false):int  {
...
}
```

Però caldrà especificar que volem que es compleixen aquestes condicions de tipat de forma efectiva. Per imposar la **tipificació estricta** a les funcions hem d'afegir al començament una línia:

```
<?php
declare(strict_types=1);

function sum(int $a, int $b):int {
   return $a + $b;
}
?>
```

El mateix entorn de programació subministra un extens conjunt de funcions (les llibreries estàndard), però els programadors també podem crear-nos les nostres pròpies funcions i guardar-les a les nostres llibreries específiques. Es poden cridar a funcions de la llibreria estàndard o definides pel programador des de qualsevol altre funció.

Declaració d'una funció

Ja hem vist als exemples anteriors com es pot realitzar la declaració d'una funció, però ara incidirem en les diferents parts que inclou.

Comencem definint el nom que li donarem a la funció, i entre parèntesi la llista de paràmetres rebuts amb el seu tipus (ja siguin rebuts per valor o referència) separats per comes, podem terminar aquesta part amb 2 punts i el tipus de variable retornat (establir el tipus de retorn es opcional però molt recomanable).

Continuem amb la definició de les variables locals necessàries, per després codificar les estructures de codi que resolen el problema, i acabem amb el retorn d'un resultat (si es necessita un retorn).

Exemple:

```
function_isprimer (int $num):bool  {        //nom de la funció i llista d'arguments
   $div=2;                                  //definició de variables locals

   while ( $div <= $num/2 ) {               //condicions per sortir del bucle
                                            //optimitzant el seu funcionament
          if  ( $num % $div == 0 ) {
                return false;               //retorn per a indicar que no lo es
          }
          $div++;
   }
   return true;                             //retorn per indicar que ho es
}
```

NOTA: els noms dels paràmetres que rep la funció no ha de coincidir obligatòriament amb els noms de les variables que passarem al cridar a la funció.

El valor de retorn de la funció s'envia a qui l'ha cridat amb la sentencia **return**. Si la funció no torna cap valor, no cal escriure **return**;

Cridar a una funció

Per cridar a una funció només cal escriure el seu nom i passar-li les variables necessàries separades per comes. Com que no s'estableix el tipus de dada a passar com paràmetre caldrà tenir cura que es passen els tipus adients per no provocar errades de programa. Exemple de crida a funcions:

```
<?php
        $value=122441;
        $prime = isprimer ($value);
        echo "Es el numero $value un valor primer?. Resposta =" . $prime;
?>
```

Concepte de llibreries

L'exemple de crida a funcions anterior serveix quan la funció està definida al mateix fitxer on se la crida, però el més habitual es que no sempre sigui així. Les funcions restaran definides en altres fitxers, formant conjunts agrupats per un tipus de funcionalitat o tractament de dades, a aquests conjunts els hi anomenem llibreries.

Les funcions s'agrupen per funcionalitat específica dins de llibreries. Una llibreria no és més que un fitxer, que pot ser inclòs al nostre programa per fer servir les funcions que agrupa.

Així ens trobarem amb un conjunt de llibreries estàndard que venen subministrades per el fabricant de l'entorn de programació i que són comuns a tots el entorns de programació. A més els fabricants per captar als programadors poden afegir llibreries específiques amb funcionalitats noves i atraients (connexions a diferents bases de dades, pàgines web, etc.).

I nosaltres mateixos poden agrupar les nostres funcions creant llibreries específiques, així podríem agrupar les funcions matemàtiques a una, les funcions amb vector numèrics a altra i les funcions que tracten cadenes de caràcters a una tercera.

Utilització de llibreries

Tenim diferents possibilitats per incloure una llibreria o fitxer extern amb funcions:

```
<?php
        include 'llibreria.php';
        require 'llibreria.php';
        include_once 'llibreria.php';
        require_once 'llibreria.php';
?>
```

Diferencies:

- **Include** es fa servir quan necessitem incloure la llibreria però si es produeix una errada al carregar-la el programa continua la seva execució i únicament notifica la errada. Cada recàrrega de la pàgina on tenim el include implica tornar a carregar la llibreria.

- **Include_once** es una alternativa on si ja hem inclòs la llibreria una vegada no la torna a carregar.

- **Require** es una alternativa a include si la llibreria es imprescindible per subministrar el servei a l'usuari i en cas de no poder carregar-la no volem que el programa continuï executant-se.

- **Require_once** es una alternativa a require per carregar una única vegada la llibreria.

Operacions fonamentals amb vectors numèrics que podem realitzar amb funcions

Ordenar el contingut d'un vector numèric

Ordenar dades és una de les aplicacions més comuns. La manera més senzilla és diu Mètode de la Bombolla o Ordenació per enfonsament, que implica que els valors majors és guardaran a les darreres posicions del vector (quedant ordenat de menor a major).

Aquest mètode ha de recorre el vector complert comparant el valors de posicions consecutives d'elements, si estan en ordre decreixent els seus valors s'intercanvien. A cada passada és realitzen tantes comparacions com elements del vector menys una, i cal recorre el vector tantes vegades com elements del vector menys un per a assegurar que el vector està totalment ordenat.

Exemple:

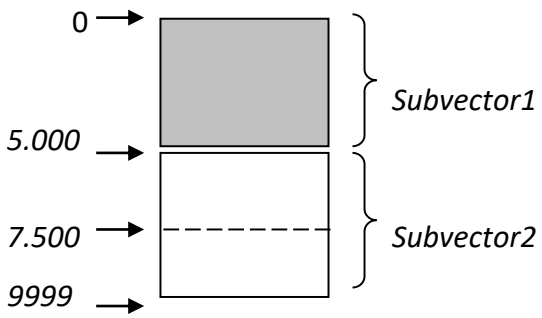
```
for ( $pas = 0;  $pas < (count($vector) – 1);  $pas++ )          // recorreguts pel vector
    for ( $posic = 0;  $posic < (count($vector) – 1); $ posic++ )  // comparacions a realitzar
        if  ( $vector[$posic]  >  $vector [$posic+1])  {
            $aux = $vector [$posic];
            $vector [$posic]  = $vector [$posic+1];
            $vector [$posic + 1]  = $aux;
        }
```

Recerca de valors dins d'un vector

Tenim la ***recerca lineal***, que consisteix a comparar el valor a buscar amb tots els elements del vector fins trobar-ho. Només s'utilitza amb vectors petits o bé que no estiguin ordenats, per la seva gran ineficàcia. Com a mitjana necessita buscar per el 50% del vector per a trobar el valor buscat.

Si és vol eficàcia, i el vector està ordenat, utilitzarem la ***recerca binària***: Aquest mètode després de cada comparació descarta la meitat dels elements del vector, per a continuar amb la recerca amb l'altre meitat. Per aconseguir això localitza l'element mitjà del vector ordenat i ho compara amb el valor buscat, si no es troba i el valor de l'element és menor que el buscat és continua la recerca amb la 2ª meitat del vector (subvector). Repetint la operació fins trobar el valor buscat a la meitat del subvector, o bé si el subvector únicament conté 2 elements només cal comparar-los directament.

if ($valor_bus > $vector [$max/2])



Exemple de la seva eficàcia: amb un vector de 1024 elements, es troba el valor buscat (si hi és dins del vector) en 10 comparacions (com a màxim), però amb un vector de 1000 milions d'elements només necessitaria 30 (com a màxim).

2. Introducció al concepte de recursivitat

Concepte: La recursivitat és una forma de resoldre un problema a base de intentar donar sempre una mateixa solució o resoldre el problema sempre de la mateixa forma. Així quan el problema es molt complex el que s'intenta es reduir la seva complexitat paulatinament fins a arribar al que es coneix com "cas base" que és el que sabem resoldre amb el nostre sistema.

Aquest concepte s'aplica a la programació a través del disseny recursiu o la definició de funcions recursives.

Funcions recursives: Son funcions que es criden a sí mateixes per a resoldre un problema. Aquestes funcions només saben resoldre un problema simple (el ***cas base***), i per resoldre casos més complexes la funció divideix el problema en 2 parts: una que la funció sap resoldre i l'altra on la funció es crida a sí mateixa (crida recursiva) amb un cas una mica més simple. A través del **return** els resultats parcials es van combinant amb les parts del problema que la funció anterior pot resoldre per a acabar obtenint el resultat final.

La diferencia entre **recursivitat** (que una funció es cridi a sí mateixa) i la **iteració** (bucle on es crida repetidament a la mateixa funció), es que la 1ª utilitza una estructura de selecció per cridar a les funcions i la 2ª utilitza una estructura de repetició. La recursivitat te efectes negatius per al rendiment del sistema, ja que consumeix una gran quantitat de memòria per a la generació de les seves pròpies còpies o "clons"; la iteració no realitza còpies o clons de la funció que treballen simultàniament al sistema, sinó que crea i destrueix a la funció en cada volta dins de la iteració.

Anàlisi de programes recursius i casos "acadèmics" on es pot aplicar la recursivitat.

Exemple 1: Càlcul del valor d'una determinada posició de la sèrie de Fibonacci, per exemple de la sèrie: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34... a la posició 9 trobem el valor 21.

```
<?php

function fibonacci ( int  $n ) : int {
  if ( $n == 0  or  $n == 1 )              // cas base, l'únic que sap resoldre directament
     return  $n;
  else return fibonacci ($n-1) + fibonacci ($n-2);   //es crida a sí mateixa amb versions més simples
}


$posicio = 9;
$result = fibonacci ( $posicio );
echo "El valor de la posició " . $posicio . " de la sèrie de Fibonacci es: " . $result );

?>
```

Exemple 2: Càlcul del valor del factorial d'un número:

```
<?php

function factorial (int $number) : int {
   $resultat;
   if ( $number <= 1 )                    // cas base, l'únic que sap resoldre directament
      $resultat = 1;
   else  $resultat = ($number * factorial ($number-1));  //es crida a sí mateixa amb un cas més simple
   return $resultat;
}

$num = 6;
$result = factorial ($num);
echo "El valor del factorial de " . $num . " es: " . $result );

?>
```

Al primer exemple la funció crea constantment 2 noves còpies de sí mateixa amb lo qual la seva expansió te una progressió exponencial i el seu ús de recursos per a valor gran també.

Al segon exemple cada funció es torna a cridar a si mateixa una sola vegada.

Aquests són només 2 exemples molt "acadèmics" de com treballen aquest tipus de funcions, i per poder practicar amb aquesta forma de dissenyar programes, però amb l'àmbit purament informàtic podem trobar altres aplicacions molt més interessants. Per exemple: com ho podríem fer recórrer tota la estructura de directoris que gestiona el sistema operatiu dins del disc dur? Ja que l'arrel tots tenim ben clar on queda, però una vegada comencem a entrar als seus directoris ens trobem que a dins hi ha més directoris, i la situació es va repetint fins a arribar al darrer directori on ja només hi trobem fitxers.

En aquest cas la programació recursiva no seria una bona i senzilla solució per a un problema que es planteja realment feixuc?

Prova de funcions recursives.

A la prova i depuració ens podem arribar a perdre, ja que seguir les passes d'una funció que es crida a sí mateixa ens pot tornar bojos per que sempre hi som davant del mateix codi font, i l'única eina per saber en quin punt del problema hi som es revisar els paràmetres rebuts i tenir molt clar el problema que hem de resoldre per conèixer exactament a quina fase de la seva solució ens hi trobem.