Unitat Formativa 5. POO. Llibreries de classes fonamentals

1. Gestió d'errors: Control de Excepcions

Un dels problemes habituals al desenvolupar codi és el tractament que cal donar als errors, especialment els que provocaran l'aturada del programa al no poder realitzar una operació o els que no permeten accedir a un recurs del sistema. La **gestió d'excepcions** permet capturar els errors i donar una alternativa enlloc de deixar que el programa s'aturi presentant a l'usuari un error intern.

Exemples d'errors:

- Intentar llegir un fitxer que no existeix o del que no tenim permisos.
- Intentar accedir a posicions de arrays o a objectes que no existeixen.
- Enviar/rebre dades per una xarxa quan es produeix un pèrdua de connectivitat.

PHP permet gestionar els errors amb les clases **Exception** i **Error**, ambdues implementen la interfície **Throwable.** Per tant podem retornar, indistintament, amb la sentència **throw** tant objectes de classe Exception com de classe Error.

https://www.php.net/manual/es/class.throwable.php
https://www.php.net/manual/es/class.exception.php
https://www.php.net/manual/es/class.error.php

La gestió d'excepcions porta associat un conjunt de paraules clau per a la seva gestió: **try, catch, finally i throw:** https://www.php.net/manual/es/language.exceptions.php

Mecanismes de control i captura d'excepcions facilitats pel llenguatge.

L'objectiu d'aquest mecanisme es poder protegir blocs de codi previsibles de patir fallides i preparar una "sortida" per a aquests errors, encara que només sigui informant amb un missatge o amb un registre a un fitxer de log de la aplicació. Sense aquest control el programa s'aturaria amb el perjudici que podem provocar en un entorn de producció, i també podria provocar un forat de seguretat.

El codi a protegir davant d'una previsible fallida restarà tancat dins d'un bloc marcat per **try { }**. Amb **catch** podrem capturar la possible excepció i tractar-la. Hi podem tenir moltes sentències catch per a un mateix bloc marcat amb try, una per cada tipus d'excepció que volem capturar.

Les sentències incloses al bloc marcat per **finally** s'executen després de haver-se executat correctament el codi, o bé després del catch, permeten gestionar alternatives en funció de si el codi ha presentat una fallida o no, o bé alliberant recursos o connexions.

Exemple:

```
// Dissenyem una funció que controla els paràmetres d'entrada per evitar errors de funcionament
function div(int $number, int $div): float {
   if ($div === 0) {              //Comprova que no ha rebut un 0, únic cas que no pot executar
      throw new Exception("Division by Zero"); // llença la excepció informant del motiu
   }
   return $number / $div;     // Si ha superat la comprovació executa la operació i retorna resultat
}
// Al provar la funció, hem de protegir el codi susceptible de presentar excepcions dins d'un bloc try
try {
   echo div(56, 0);              //Si els paràmetres son correctes presenta resultat
} catch (Exception $ex) {       //Si la funció retorna una excepció la capturem i la presentem
   echo $ex->getMessage();
}
```

A *la llibreria Standard de PHP* -> **SPL** trobem un conjunt de excepcions que hereten de **Exception** https://www.php.net/manual/es/spl.exceptions.php.

Com exemple, treballem amb **OutOfBoundsException**

```
// Dissenyem una funció que eleva al cub el valor de la posició indicada del array rebut
function cubeOfPos(array $array, int $pos): int {
   if ($pos < count($array)) {          // Comprova que la posició demanada no sobrepassa el array
      return ($array[$pos] * $array[$pos] * $array[$pos]);
   } else {
      throw new OutOfBoundsException("Index out of limit of elements");
   }
}

$array = [1, 2, 3, 4];
// De nou protegirem el codi susceptible de presentar excepcions dins d'un bloc try
try {
   echo "Cube = " . cubeOfPos($array, 8);
} catch (OutOfBoundsException $ex) {
   echo $ex->getMessage();
}
// I podem protegir diferents funcions, preparant un catch exclusiu per a cada excepció
try {
   echo div(56, 3);
   echo "<br><br>";
   echo "Cube = " . cubeOfPos($array, 8);
} catch (OutOfBoundsException $ex) {
   echo $ex->getMessage();
} catch (Exception $ex) {
   echo $ex->getMessage();
}
```

També podem estendre de Exception per crear excepcions personalitzades al nostre domini.
https://www.php.net/manual/es/language.exceptions.extending.php

Exemple:

```
// Dissenyem una excepció particular per als errors d'un servei
class ServiceException extends Exception{
   //put your code here
}

// Dissenyem una funció que llença una aquest tipus d'excepció
function serviceExample() {
        ... // Imaginem que realitza una sèrie de càlculs...
   throw new ServiceException("DomainServiceException");
}

$result= "";
// cridem a la funció preparats per a la excepció i es mostra com amb finally podríem
//realitzar una o altra tasca segons el servei hagi tingut èxit o no
try{
   serviceExample();
   echo "Domain Service runs ok";
   $result = "done";
} catch (ServiceException $ex) {
   echo $ex->getMessage();
   $result = "error";
} finally{
   if ($result == "done"){
      echo "<br>Operation Successfully Executed";
   }else if ($result == "error"){
      echo "<br>Not possibly Execution of Operation";
   }
}
```

Tradicionalment la classe Error ha estat independent de Exception, amb un ús més intern, però amb la classe **ErrorException** podem fer un tractament conjunt. Sense aquesta classe si hem capturar un Error no ho podrem fer amb un ***catch (Exception $ex)*** sinó amb un ***catch (Error $er).***

https://www.php.net/manual/es/class.errorexception.php
https://www.php.net/manual/es/language.errors.php7.php

Els errors es poden gestionar amb funcions https://www.php.net/manual/es/book.errorfunc.php. Però com observem als exemples de la documentació de ErrorException, aquesta classe ens permet encapsular aquests errors en un objecte que pot transportar la informació del error entre els diferents components del nostre programa.

Tractament front delegació.

Quan hem de gestionar una excepció tenim 2 opcions:

- ✓ Que sigui tractada pel propi mètode que ha detectat la incidència.
- ✓ Que sigui tractada pel codi que ha cridat al mètode.

La delegació del tractament de la excepció es realitza quan llencem la excepció amb un **throw**. Un mètode que està dissenyat per a una operació concreta no té sentit assignar-li "responsabilitats" que no li corresponen, com per exemple saber si quan detecta un error aquest ha de ser comunicat per consola, dins d'una pàgina web, crear un registre a un fitxer de log, crear un registre a una base de dades... o una combinació de les accions anteriors.

Així podríem enllaçar una cadena de excepcions fins que arribi al component del programa que ha de prendre aquestes decisions, normalment estem parlant d'algun tipus de controlador de cas d'ús.

Es recomana capturar les excepcions quan:

- Podem resoldre l'error i continuar amb l'execució del programa.
- Volem registrar l'error sofert pel programa, continuant o no amb la seva execució.
- Volem rellançar l'error amb un tipus d'excepció distint al capturat (pot ser necessari).

Evitar aplicacions errònies del control d'excepcions:

- Capturar una errada però no fer res, ocultant l'error i que no sigui visible per l'error hi és. Això va en contra de la filosofia de la gestió dels errors, els tractem (encara que sigui mostrant un missatge aclaridor o registrant-lo a un fitxer) o el rellancem per delegar el seu tractament

```
try {
        // Codi que genera una excepció
} catch ( Exception $excep) {
    //buit, no fem res, pretenem que no es vegi que s'ha produït un error
}
```

- Llençar excepcions genèriques que no permeten distingir en quines condicions especifiques s'ha produït la fallida, i per tant no podrem gestionar-les adequadament

Bones practiques amb el control d'excepcions:

- Fer servir les excepcions existents,
- Investigar les API's amb les que es treballa i no desenvolupar excepcions que ja existeixen
- Crear només aquelles que aporten un significat específic al tractament dels errors.

2. Gestió avançada de dades

2.1. Expressions regulars. Recerca de patrons en cadenes de text

És una situació molt comú que les nostres aplicacions hagin d'analitzar una cadena de caràcters per localitzar un determinat valor o determinar si compleix amb un determinat patró de contingut, com per exemple el que han d'acomplir les dates o les adreces de correu electrònic.

Les expressions regulars són patrons de text que permeten fer operacions de localització, neteja o substitució sobre cadenes de caràcters.

Com es construeix una expressió regular:

PHP fa servir una llibreria PCRE (compatible amb Perl), amb una sintaxi i unes funcions preestablertes que ens ajudarà amb l'ús dels patrons: https://www.php.net/manual/es/ref.pcre.php

Una expressió regular es crearà dins d'uns limitadors. Habitualment el delimitador més utilitzat es la barra **/** o el caràcter **#**. Per escapar valors dins de l'expressió farem servir la barra invertida \ i així escapen del control.

Agrupadors: delimitadors de sub-expressions o parts de l'expressió que podríem definir.

[]	permeten crear rangs de valors.
()	permeten crear sub-expressions, expressions regulars contingudes dins d'altres que poden ser captures individualment.

Quantificadors: defineixen el nombre d'aparicions d'un caràcter.

*****	el caràcter pot aparèixer zero o més vegades.
+	el caràcter pot aparèixer una o més vegades.
?	el caràcter pot aparèixer zero o una vegada.
{n}	el caràcter apareix exactament n vegades.
{n,}	el caràcter apareix n o més vegades.
{n,m}	el caràcter pot aparèixer entre n i m vegades.

Modificadors: Permeten canviar com s'executa l'expressió regular. S'afegeixen després del delimitador de tancament.

I	No distingir entre majúscules i minúscules.
Altres modificadors veure més	

Seqüències de caràcters: utilitzen la barra invertida \ (doble \\ quan construïm el patró)

s	Match amb qualsevol espai en blanc
S	Match amb qualsevol que no sigui espai en blanc
d	Match amb qualsevol dígit
D	Match amb qualsevol caràcter que no sigui un dígit
w	Match amb qualsevol caràcter alfanumèric (lletra, dígit, guió baix)
W	Match amb qualsevol caràcter que NO sigui alfanumèric (lletra, dígit, guió baix)

Metacaràcters fora dels claudàtors: permeten establir la forma de buscar el patró

.	Match amb qualsevol caràcter
^	Match al començament del string
$	Match al final del string
A	Inici d'un string.
z	Final d'un string.
|	Match amb qualsevol de les expressions que separa (és un **or**).

Metacaràcters dins dels claudàtors: ajuden a definir el rang de valors

****	caràcter de escapament general
^	Indica els caràcters que no hi poden formar part del text
-	Indica un rang de caràcters

Exemples de patrons:

[a-z]	De la a la z minúscula
[a-z][A-Z]	Per exemple aA
[a-z][A-Z][0-9]	Per exemple aA8
[^aeiou]	Excepte una de les vocals
\\w : [a-zA-Z_0-9]	Un caràcter alfanumèric qualsevol
\\d.	Un numero i qualsevol caràcter
\\d.[a-z]	Un numero, un caràcter qualsevol i una lletra minúscula
\\d\.[a-z]	Un numero, un punt i una lletra minúscula
\\d{3}	Un numero de 3 dígits
\\d{n,}	un numero almenys n vegades
\\d{n,m}	Un numero entre n i m vegades
[aeiou]?	Res o una sola vocal
x[aeiou]?	x sola, o seguida d'una vocal
x[aeiou]*	x sola, o x seguida de varies vocals
x[aeiou]+	x como mínim amb una vocal

Funcions associades a les Expressions Regulars

preg_match https://www.php.net/manual/es/function.preg-match.php

Permet conèixer si la cadena acompleix amb el patró establert. Indicant amb el tercer paràmetre de tipus array, opcional, que el patró conte subpatrons i volem obtenir el seu contingut per tractar-lo per separat. Retorna un **1** si es troba el patró, un **0** si no coincideix i **false** si troba un error.

Es similar a **preg_match_all**, excepte que aquest últim acostuma a utilitzar-se quan necessitem passar per paràmetre l'array per obtenir els valors de les sub-expressions.

preg_replace https://www.php.net/manual/es/function.preg-replace.php

Realitza la funció "buscar i reemplazar" dins del string o array de strings rebut per paràmetre.

A diferència de **preg_filter** no retorna un null o un array buit sinó ha realitzat cap substitució, sinó que retorna el string o array rebut per paràmetre sense canvis.

preg_split https://www.php.net/manual/es/function.preg-split.php

Permet dividir un string rebut per paràmetres a partir d'una expressió regular o patró també rebut per paràmetre. Retorna un array amb les divisions obtingudes o false si no s'ha trobat el patró.

Es similar a **explode** però treballant amb expressions regulars enlloc de cadenes fixes.

preg_grep https://www.php.net/manual/es/function.preg-grep.php

Versió més avançada de la funcionalitat anterior, on enlloc de rebre un string a dividir en substrings, rep un array per escollir i retornar totes les posicions del mateix que acompleixen amb el patró o expressió regular rebuda per realitzar la selecció.

La funció retorna un altre array indexat a partir dels índexs del array rebut, però que només conté posicions del array principal que acompleixen amb el patró.

2.2. Classes que gestionen Dates

La manipulació de dades de tipus ***datetime*** és una tasca comú a qualsevol llenguatge de programació, per tant disposen de diverses classes que ajuden al seu tractament. A PHP les funcions per tractar aquestes dades les trobem a: https://www.php.net/manual/es/ref.datetime.php

On trobem una explicació bàsica sobre el seu funcionament, i a cada funció trobarem una explicació detallada i exemples.

Per treballar sota l'àmbit de la POO s'afegeix un conjunt de classes i interfícies específiques que amplien la seva utilització. Algunes que podem destacar son:

https://www.php.net/manual/es/class.datetime.php
https://www.php.net/manual/es/class.datetimeinterface.php
https://www.php.net/manual/es/class.dateinterval.php
https://www.php.net/manual/es/class.dateperiod.php

Aquestes classes i interfícies ens permeten la seva inclusió a les classes del nostre projecte, reaprofitant un ampli ventall de solucions que ja no haurem de programar. Per exemple podem afegir objectes de classe **DateTime** com atributs a les nostres classes.

Classe DateTime

DateTime disposa d'un constructor increïblement versàtil, amb una gran varietat de *formats relatius* per crear dates, on podem arribar a definir una data en un llenguatge pràcticament formal:

```
$relativeDate = new DateTime('first day of next month');
```

Documentació dels formats relatius: https://www.php.net/manual/es/datetime.formats.relative.php

De forma habitual utilitzarem formats més típics com:

```
$fixedDate = new DateTime('23-10-2022 13:58:23');
$currentDate = new DateTime();          //per defecte obtindrem la data i hora actual
$currentDate->modify ( '+7 day ');      //però es pot modificar fàcilment
```

A través del seu mètode **format** podem representar una data amb una amplia varietat de formats, des del més simple al més complert que es pugui necessitar:

```
echo $fixedDate->format('m-d-y');
echo $fixedDate->format('H:i:s, l j \d\e F \d\e\l Y(L) (z \d\i\a \d\e \l\a W \s\e\t\m\a\n\a)');
```

Possibilitats del paràmetre **format**: https://www.php.net/manual/es/function.date.php

Classes relacionades amb el tractament de les dades que ofereix DateTime:

Interfície DateTimeInterface

Implementada per DateTime i DateTimeImmutable, permet que els objectes de ambdues classes puguin ser utilitzats indiferentment, tractant-los com si fossin del mateix tipus. Ofereix un conjunt de constants predefinides i mètodes com **diff** o **format** que utilitzarem sovint per operar amb aquests tipus de dades. Observem que alguns mètodes de DateTime reben paràmetres d'aquest tipus.

Classe DateInterval

Representa i conté un interval de dates amb el que podrem operar. Els mètodes **add** i **sub** de la classe **DateTime** el fan servir com paràmetre d'entrada per operar amb la data emmagatzemada, i el mètode **diff** retorna un objecte d'aquesta classe per donar el interval entre dues dates.

```
$datesDiff = $currentDate->diff($relativeDate);
var_dump($datesDiff);
$fixedDate->add($datesDiff);
var_dump($fixedDate);
```

Podem crear intervals específics a través del constructor, amb una cadena que acomplexi amb el següent format: https://www.php.net/manual/es/dateinterval.construct.php

```
$interval = new DateInterval('P1Y2M3D');   //1 any, 2 mesos i 3 dies
$fixedDate->sub($interval);
var_dump($fixedDate);
```

Classe DatePeriod

Representa a un període de dates i permet iterar entre les mateixes amb intervals establerts.

```
$interval = new DateInterval('P3D');
$daterange = new DatePeriod($currentDate, $interval ,$relativeDate);

foreach($daterange as $date){
   echo $date->format( "Y-m-d" ) . "<br>";
}
```

També podem crear el període a partir d'una data i un número concret de intervals:

```
$daterange = new DatePeriod($relativeDate, $interval , 2);
foreach($daterange as $date){
   echo $date->format("Y-m-d") . "<br>";
}
```

3. Gestió d'operacions bàsiques d'entrada i sortida de la informació

Tota aplicació acostuma a precisar d'una comunicació amb sistemes externs, coneguts genèricament com infraestructures externes: sistemes de fitxers, bases de dades, sistemes de missatgeria, de gestió de logs, servidors web, terminals... La gestió d'aquestes comunicacions son tasques amb les que gairebé tot sistema ha de conviure.

La complexitat de la comunicació amb infraestructures externes precisa de coneixements sobre arquitectures de disseny del software, però bàsicament es tracta de definir interfícies que realitzin la tasca de fer de ports de entrada/sortida de la nostra aplicació, i a continuació desenvolupar els adaptadors específics amb cada infraestructura concreta amb la que establir la comunicació.

En aquest apartat ens centrarem únicament en la infraestructura d'un sistema de fitxers per entendre com s'estableix la interconnexió.

Gestió de fitxers

Conceptes relacionats amb les operacions amb fitxers:

- Els errors amb l'entrada i sortida són susceptibles de provocar excepcions.
- **Serialització**: sistema automatitzat per importar i exportar el contingut d'un objecte.

SplFileObject classe que permet accedir al sistema de fitxers per gestionar-los (obté les seves dades, característiques com mida, pathname, tipus, permisos, si es un director, etc. El constructor pot rebre un o dos paràmetres que indiquen la ruta (absoluta o relativa) del directori/fitxer i un string que indicarà el mode d'accés (escriptura, lectura, etc.).

Exemple bàsic: guardem el contingut d'un array a un fitxer i a continuació el llegim per mostrar que podem exportar i importar dades sense pèrdua d'informació.

```
$arrayData = ["primera línia", "segona línia", "tercera linia", "darrera línia"];      //dades exemple
var_dump($arrayData);                                                                  //es mostren
$pathname = 'd:\\fitxers\\php\\lines.txt';                                  //pathname del fitxer
$access = "w";                                                              //tipus d'accés
$file = new SplFileObject($pathname, $ access);                             //creació de l'objecte

if ($access == "w") {
   foreach ($arrayData as $line) {                  //demanem les posicions del array una per una
      $line[strlen($line)] = "\n";                  //afegim un salt de línia abans de guardar-les
      $file->fwrite($line);                         //les guardem al fitxer una per una
   }
}
```

```
$access = "r";
$dataArray = [];                        //array buit que omplirem amb les dades del fitxer

$file = new SplFileObject ($pathname, $access);

if ($access == "r") {
   while ($file->eof() != 1) {          //mentre no arribem al final del fitxer
      $line = $file->fgets();           //llegim el fitxer línia a línia
      if (strlen(trim($line)) > 0) {    //si la línia té contingut (no està buida)
         $dataArray[] = trim($line);    //carreguem el seu contingut a l'array
      }
   }
}

var_dump($dataArray);                   //mostrem el contingut, que ha de ser idèntic al primer array
```

4. Serialització/Deserialització d'objectes

Anomenen serialització a la conversió del contingut dels objectes a un flux de dades. Aquest flux potser de caràcters o de bytes, i pot acabar convertint-se en un string, un fitxer o bé ser transmès cap a un port de E/S.

El tractament de la serialització acostuma a incloure també el procés invers de deserialització, que obtindria l'objecte a partir d'un flux de dades que podem rebre en forma de string, fitxer, etc.

Tant si parlem de strings com de fitxers, el seu contingut hauria de guardar un determinat format, hem d'establir una forma d'indicar el nom dels atributs i el seu valor i tipus per poder tenir una representació vàlida de la informació.

Aquests són els formats més utilitzats:

- **JSON** (***JavaScript Object Notation***) és un format de text obert i lleuger utilitzat per a la transferència de dades entre sistemes que no volem acoblar. Un sistema estaria acoblat a un altre si ha de conèixer detalls interns del seu disseny, com per exemple les classes d'objecte que el conformen.

  JSON es molt utilitzat a la comunitat de desenvolupament web i podem trobar molta informació de com construir missatges o respostes seguint les especificacions d'aquest format:

  https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/JSON
  https://www.json.org/json-es.html
  https://opensource.adobe.com/Spry/samples/data_region/JSONDataSetSample.html

- **XML** (***EXtensible Markup Language***) és un metallenguatge de marques extensible i autodescriptiu, amb unes característiques molt més complexes que JSON. Encara que és utilitzat al desenvolupament web, les seves possibilitats van més enllà i es un potent transmissor de informació, i és compatible per a la transferència d'informació amb bases de dades.

  https://es.wikipedia.org/wiki/Extensible_Markup_Language
  https://developer.mozilla.org/es/docs/Web/XML/XML_introduction
  https://opensource.adobe.com/Spry/samples/data_region/NestedXMLDataSample.html

  Formats com **JSON, XML,**... ajuden al desacoblament entre sistemes, ja que únicament son formats per a la transmissió d'informació o dades, són el resultat estructurat del processament d'una petició, però no cal conèixer cap detall de com l'altre sistema ha realitzat el processament.

Tractament de documents JSON

Consideracions sobre el procés de serialització d'un objecte a un document Json:

- Les funcions que proporciona PHP per serialitzar objectes només son efectives amb objectes d'una classe genèrica anomenada **stdClass**, on els seus atributs són públics per defecte.

- La conversió dels nostres objectes específics a json no es pot fer directament amb aquestes funcions sense perdre coherència. Hem de realitzar un procés per a la seva serialització.

Per començar hem de fer que la classe a serialitzar implementi la interfície **JsonSerializable** i el seu mètode **jsonSerialize**. Per exemple, volem serialitzar les dades dels nostres objectes de classe Client:

```
class Client extends Person implements Stakeholder, JsonSerializable {

//A la definició dels atributs, constructor i mètodes afegim la implementació del mètode jsonSerialize
public function jsonSerialize() {
      return
            [
               'name' => $this->getName(),
               'ident' => $this->getIdent(),
               'email' => $this->getEmail(),
               'phone' => $this->getPhone(),
               'address' => $this->getAddress(),
               'birthday' => $this->getBirthday(),
               'password' => $this->getPassword(),
               'clientcode' => $this->getClientCode(),
               'numequipments' => $this->getEquipments()
      ];
   }              //aquest mètode converteix l'objecte a un array associatiu
}
```

Al implementar la interfície **JsonSerializable** ja podem fer servir les funcions genèriques com **json_encode** i obtenir el document json que volem:

```
$c = new Client("namec", "identc", "emailc", "phonec", "addressc", "birthdatec", "ceman", 10011, 3);
var_dump($c);                          //comprovem el seu estat
$jsonC = json_encode($c);              //al implementar JsonSerialize retornen el array adient
var_dump($jsonC);                      //comprovem el valor del string obtingut
```

Un altre cas d'ús podria ser el d'haver de serialitzar una col·lecció o array d'objectes de classe Client, comprovarem si la preparació realitzada es efectiva:

```
$cc = new Client("namecc", "identcc", "emailcc", "phonecc", "addresscc", "birthdatecc", "cceman",
10012, 2);
```

```
$ccc = new Client("nameccc", "identccc", "emailccc", "phoneccc", "addressccc", "birthdateccc",
"cccceman", 10013, 1);

$clients = array($c, $cc, $ccc);        //omplim el array amb els 3 objectes instanciats
var_dump($clients);                     //comprovem el seu contingut

$jsonClients = json_encode($clients);   //serialitzem
echo $jsonClients;                      // i comprovem el string obtingut amb éxit
```

Ara comprovem el procés invers, la **deserialització**: obtenir un objecte concret a partir d'un document json rebut. Continuem amb Client i la funció **json_decode**:

```
$clientObj = json_decode($jsonC);       //amb json_decode obtenim un objecte de stdClass
var_dump($clientObj);                   //comprovem el seu contingut
```

Fins aquí hem provat com podem realitzar la serialització i deserialitació, ara resta organitzar i distribuir les responsabilitats de les tasques en classes.

Comencem per la classe que tindrà la responsabilitat de realitzar el servei:

```
class JsonClientSerializer {

   public function serialize(Client $c): string {
      return json_encode($c);
   }

   public function unserialize ( string $stringJson ) : Client {
      $obj = json_decode($stringJson);
      return new Client($obj->name, $obj->ident, $obj->email,  $obj->phone, $obt->address,
          $obj->birthday,   $obj->password, (int)($obj->clientcode), (int)($obj->numequipments));
   }
}
```

Codi per comprovar la serialització/deserialització d'un Client:

```
$c = new Client("namec", "identc", "emailc", "phonec", "addressc", "birthdatec", "ceman", 10011, 3);

$clientJsonSerializer = new JsonClientSerializer();
$jsonClient = $clientJsonSerializer->serialize($c);

var_dump ( $jsonClient );                                   //es mostra l'objecte serialitzat a JSON
var_dump ( $clientJsonSerializer->unserialize( $jsonClient ) );      //objecte deserialitzat
```

Tractament de documents XML

A la serialització "objecte <-> document XML" ens trobem problemàtiques similars a les trobades amb la codificació JSON. Podem trobar funcions com **simplexml_load_string** i classes com **simpleXMLElement** que també precisen d'objectes de la classe genèrica **stdClass.**

Aprofitarem el desenvolupament realitzat per al tractament de documents JSON, els objectes a serialitzar ja implementen la interfície **JsonSerializable** que retorna un array associatiu amb les seves dades i coneixem les funcions **json_encode** i **json_decode,** i afegirem les conversions necessàries:

```
$c = new Client("namec", "identc", "emailc", "phonec", "addressc", "birthdatec", "ceman", 10011, 3);
$jsonC = json_encode($c);
$clientObj = json_decode($jsonC);
```

Una vegada obtingut l'objecte genèric podem provar el codi que necessitarem implementar a les nostres classes específiques per a la serialització/deserialització.

Dissenyarem una funció recursiva que ens permetrà recórrer el objecte i tots els objectes interns que pugui incloure i que també han de quedar serialitzats a XML:

```
function xmlSerialize(stdClass $obj): string {
   $xml="";
   foreach ( $obj as $k => $v ) {
      if ( is_object ( $v ) ) {
         $xml .= '<' . $k . '>' . xmlSerialize($v) . '</' . $k . '>';
      }else {
         $xml .= '<' . $k . '>' . $v . '</' . $k . '>';
      }
   }
   return $xml;
}
```

Preparem la capçalera del document i provem a crear-lo amb la funció dissenyada:

```
$stringXML = '<?xml version="1.0" encoding="UTF-8"?>';
$stringXML .= '<client>' . xmlserialize($clientObj) . '</client>';

echo '<br>XML obtingut: <br>';
var_dump($stringXML);
```

Ara que ja tenim el document XML a partir d'un objecte, provarem sistemes per al procés invers, el de deserialització.

Primer amb la funció **simplexml_load_string**:

```
$xml = simplexml_load_string ( $stringXML );
var_dump ( $xml );
```

Després provem la classe **simpleXMLElement,** que realitza el mateix procés de conversió a objecte genèric:

```
$objectXML = new simpleXMLElement ( $stringXML );
var_dump ( $objectXML );
```

I comprovem que a partir d'aquest objecte podem obtenir fàcilment la conversió a document XML:

```
$xmlData = $objectXML->asXML();
var_dump( $xmlData );
```

Una vegada entès el procés i els passos necessari podem començar a incloure aquests conceptes al disseny de les classes especialitzades en la serialització dels nostres objectes de negoci.

Primer extraurem a una classe base abstracta el codi que podrà reutilitzar qualsevol classe específica. Analitzem les parts comunes i les deixem definides per ser reaprofitades.

Classe base:

```
abstract class XmlSerializerBase {

   protected string $xmlHeader = '<?xml version="1.0" encoding="UTF-8"?>';

         //Adaptació de la funció recursiva
   protected function objectSerialize ( stdClass $obj ): string {
      $xml = "";
      foreach ($obj as $k => $v) {
         if (is_object($v)) {
            $xml .= '<' . $k . '>' . objectSerialize($v) . '</' . $k . '>';
         } else {
            $xml .= '<' . $k . '>' . $v . '</' . $k . '>';
         }
      }
      return $xml;
   }
}
```

Classe específica que realitzarà el servei de serialització/deserialització XML:

```
class XMLClientSerializer extends XMLSerializerBase {

   public function serialize ( Client $c ) : string {
      $jsonC = json_encode ( $c );
      $clientObj = json_decode ( $jsonC );
      return '<?xml version="1.0" encoding="UTF-8"?> <client>' . $this->objectSerialize($clientObj) .
'</client>';
   }

   public function unserialize ( string $stringXML ) : Client {
      $obj = new simpleXMLElement ( $stringXML );
      return new Client ( $obj->name, $obj->ident, $obj->email, $obj->phone, $obj->address,
          $obj->birthday, $obj->password, (int)($obj->clientcode), (int)($obj->numequipments ) );
   }
}
```

A **XMLClientSerializer** hem aplicat les conversions necessàries, ocultant aquesta complexitat als usuaris que les hagin de fer servir, on amb només 2 crides als mètodes **serialize** i **unserialize,** ja obtenen els serveis que necessiten.

Codi per comprovar la serialització/deserialització d'un Client amb la classe dissenyada:

```
$c = new Client("namec", "identc", "emailc", "phonec", "addressc", "birthdatec", "ceman", 10011, 3);

$clientXmlSerializer = new XMLClientSerializer();
$stringXML = $clientXmlSerializer->serialize($c);

 var_dump ( $stringXML );                                    //es mostra l'objecte serialitzat a XML
 var_dump ( $clientXmlSerializer->unserialize ( $stringXML ) );          //objecte deserialitzat
```

5. Interfícies amb l'usuari

Una aplicació pot tenir diverses interfícies amb els usuaris: consola, pàgina web, app mòbil... i no hem de pensar únicament en usuaris que siguin persones, poden ser sistemes, o programes que realitzen consultes per analitzar les dades que genera el nostre negoci de forma automàtica.

Per tant el disseny de les interfícies es una cosa particular a cada possible forma de comunicació amb uns determinats clients. Aquestes interfícies no han de "contaminar" el codi del nucli de la nostra aplicació, que ha de limitar-se a oferir uns serveis o un càlculs, independentment de la forma en que serà presentada o lliurada al client.

Hem treballat aquests aspectes des del començament, dissenyant unes capes que ens separen el codi destinat a la recepció de peticions i presentacions de resultats (*Vista*), del nostre nucli de funcionalitats (*Model*), mitjançant uns controladors (*Controlador*) que s'encarreguen de gestionar la comunicació entre les capes esmentades.

Sobre aquesta arquitectura bàsica, hem afegit la conversió de les dades guardades als objectes de negoci del model a uns formats que siguin reconeguts per la resta de sistemes, como són les codificacions **json** o **xml**.

A partir d'aquest disseny, els nostres controladors del backend poden enviar i rebre les dades dels nostres objectes, en format json/xml, convertint-se en el punt d'entrada/sortida de la nostra aplicació, el que anomenem ports d'entrada/sortida a la nostra aplicació.

I tal com hem anat desenvolupat fins ara, farem servir un controlador específic per a cada servei o cas d'ús que proporciona la nostra aplicació als diferents tipus de clients als que hem d'atendre, i sota aquest plantejament continuarem desenvolupant el nostre portal web.