Royaume du Maroc

Ministère de l'Education Nationale, du Préscolaire et des Sports

Centre National de l'Évaluation et des Examens

Examen National du Brevet de Technicien Supérieur
Session de Mai 2023
- Sujet -

Filière :	Développement des Systèmes d'Information - DSI -	Durée :	4 heures
Épreuve :	Développement des Applications Informatiques	Coefficient :	45

CONSIGNES

- ✓ **Le sujet comporte 3 dossiers ;**
- ✓ **Chaque dossier (ou sous dossier) doit être traité dans une feuille séparée.**

Barème de notation

Dossier 1 : Gestion des commandes		**22 points**
Sous-dossier 1-1 : Structure de données	11 points	
Sous-dossier 1-2 : Architecture client/serveur	11 points	
Dossier 2 : Validation des commandes		**10 points**
Dossier 3 : Création des commandes		**08 points**
	Total	**40 points**

- ✓ **Il sera pris en considération la qualité de la rédaction lors de la correction.**
- ✓ **Aucun document n'est autorisé.**

ÉTUDE DE CAS : GAL GESTION D'ACHAT EN LIGNE

Le système de gestion d'achat en ligne (**GAL**) est un progiciel qui permet d'assurer la gestion des commandes effectuées par des clients. Le suivi d'une commande est assuré par un responsable d'achat qui peut soit confirmer ou annuler une commande. L'application offre deux espaces :

- Espace client : Une plateforme Web qui permet au client de choisir des articles et passer une commande.
- Espace Responsable d'achat : contient deux applications :
  - ❖ Application Console : permet d'ajouter de nouveaux articles.
  - ❖ Application Desktop : permet de confirmer ou d'annuler une commande.

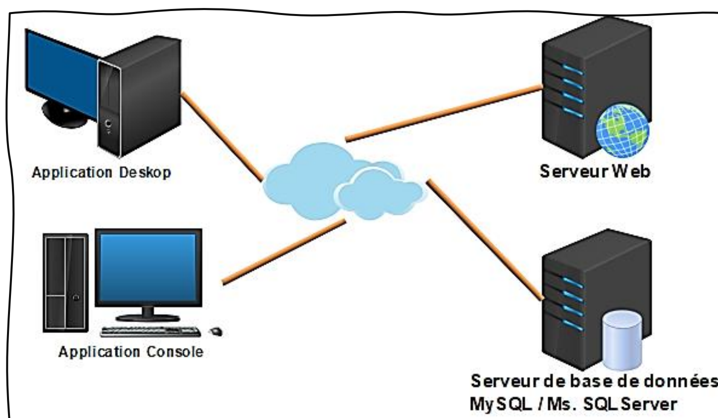

Figure 1 : Schéma structurel de service d'achat en ligne

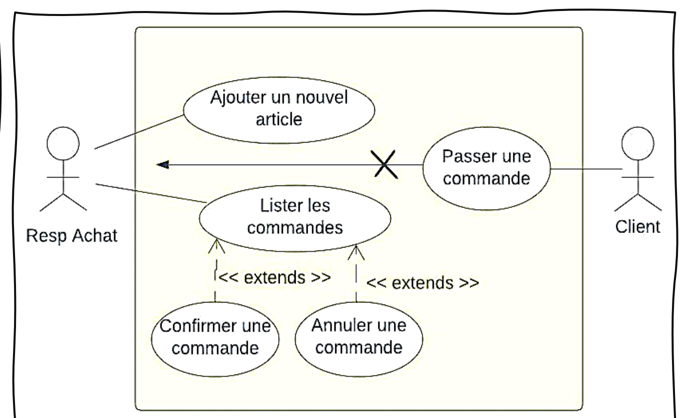

Figure 2 : Interactions entre les utilisateurs et l'application

DOSSIER I : GESTION DES COMMANDES *(22 points)*

❖ Sous Dossier 1-1 : STRUCTURE DE DONNÉES *(11 points)*

Une commande peut contenir plusieurs articles. Un client peut être une personne morale ou une personne physique. Ce processus est modélisé par le diagramme de classes suivant :

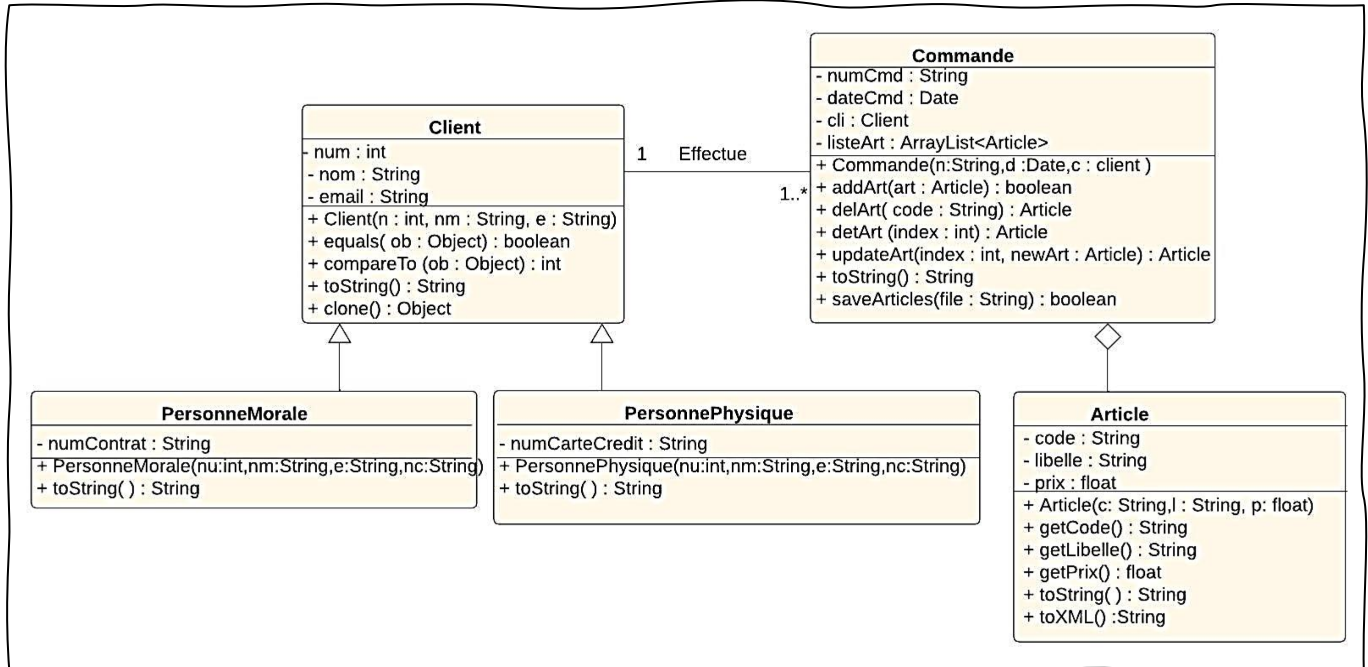

Figure 3 : Diagramme de classes

1. Implémentation de la classe « **Client** » :

```
public class Client implements Comparable, Cloneable {
   private int num;
   private String nom, email;
   public Client(...) {...}
   @Override
   public boolean equals(Object ob) { ... }
   @Override
   public String toString() { ... }
    @Override
   public int compareTo() { ... }

   @Override
   public Object clone() { ... }
}
```

a) Définir un constructeur avec **3** arguments pour initialiser les attributs de cette classe. Ce constructeur lève une exception personnalisée que l'on nomme « **NumInvalideException** » *(à définir)* lorsque le numéro du client est inférieur ou égale à zéro. *(1 pt)*

b) Redéfinir la méthode « **equals** » qui compare deux clients selon leurs numéros. *(0,5 pt)*

c) Redéfinir la méthode « **compareTo** » qui compare deux clients selon leurs numéros. *(0,5 pt)*

d) Redéfinir la méthode « **clone** » qui clone l'objet courant. *(0,5 pt)*

e) Redéfinir la méthode « **toString** », afin de retourner une chaine porteuse d'informations sur un client sous la forme : *(0,5 pt)*

```
Client Numéro : xxxx, nom : xxxx, Email : xxxx
```

2. Implémentation de la classe **« PersonneMorale »** :

```
public class PersonneMorale extends Client {
   private String numContrat;

   public PersonneMorale(...) { ... }

     @Override
   public String toString() { ... }
}
```

a) Définir un constructeur de **4** arguments pour initialiser tous les attributs de cette classe. *(0,5 pt)*

b) Donner la définition de la méthode **« toString »**, afin de retourner une chaine porteuse d'informations sur une personne morale sous-forme : *(1 pt)*

```
Client Numéro : xxxx, nom : xxxx, Email : xxxx, N° de contrat : xxxx
```

3. Implémentation de la classe **« Article »** :

```
public class Article implements Serializable{
   private String code, libelle;
   private float prix;

   public Article(...) { ... }
   public String getCode() { ... }      // Code non demandé
   public String getLibelle(){ ... }    // Code non demandé
   public float getPrix(){ ... }        // Code non demandé

     @Override
   public String toString() { ... }    // Code non demandé
                                       // retourne Code :xxxx, Libellé : xxxx, Prix : xxxx
   public String toXML() { ... }
}
```

a) Définir un constructeur de **3** arguments pour initialiser tous les attributs de cette classe. *(0,5 pt)*

b) Donner la définition de la méthode **« toXML »**, afin de retourner une chaine porteuse d'informations sur un article sous-forme : *(1 pt)*

```
<article>
        <code> xxxx </code>
        <libelle> xxxx </libelle>
        <prix> xxxx </prix>
 </article>
```

4. Implémentation de la classe **« Commande »** :

```
public class Commande{
   private String numCmd;
   private Date dateCmd;
   private Client cli;
   private ArrayList<Article> listeArt;
   public Commande(…) { … }
   public boolean addArt(Article art){… }
   public Article delArt(String code){ … }
   public Article delArt(int index){ … }
   public Article updateArt(int index, Article newArt) { … }
   @Override
   public String toString(){ … }
   public boolean saveArticles(String file){ … }
}
```

Donner le code des méthodes suivantes :

a) Un constructeur avec **3** arguments qui initialise les **3** premiers attributs de la classe **« Commande »** et instancie la collection **« listeArt »** ; *(0,5 pt)*

b) **addArt(…) :** permet d'ajouter à la collection un nouvel article et retourne l'état de l'opération ; *(0,5 pt)*

c) **delArt(String ..) :** permet de supprimer un article de la collection en se basant sur son code et de retourner l'article qui vient d'être supprimé ; *(0,5 pt)*

d) **delArt(int ..) :** permet de supprimer un article de la collection en se basant sur son index et de retourner l'article qui vient d'être supprimé (*la vérification de la validité de l'index est indispensable*) ; *(0,5 pt)*

e) **updateArt(…) :** permet de modifier un article déjà existant (*la vérification de la validité de l'index est indispensable*) et de retourner l'article avant sa modification ; *(0,5 pt)*

f) **toString () :** retourne une chaine de caractères porteuse de toutes les informations d'une commande sous-forme : *(1 pt)*

```
Numéro Commande : xxxx, Date Commande : jj/mm/aaaa
Client Numéro :xxxx, nom : xxxx, Email : xxxx,...
Liste des articles :
1- Code : xxxx, Libellé : xxxx, Prix : xxxx
2- Code : xxxx, Libellé : xxxx, Prix : xxxx
3- ...
```

g) **saveArticles (String f) :** permet de sauvegarder tous les articles dans un fichier texte; le format de sauvegarde est le suivant : *(1,5 pt)*

```
<commande num=" xxx ">
<article>
       <code> xxxx </code>
       <libelle> xxxx </libelle>
       <prix> xxxx </prix>
</article>
<article>
       <code> xxxx </code>
       <libelle> xxxx </libelle>
       <prix> xxxx </prix>
</article>
...
</commande>
```

❖ Sous Dossier 1-2 : ARCHITECTURE CLIENT/SERVEUR *(11 points)*

Afin qu'un responsable d'Achat puisse ajouter de nouveaux articles au système d'information, une architecture client/serveur est mise en place. Son schéma est représenté par la figure suivante :

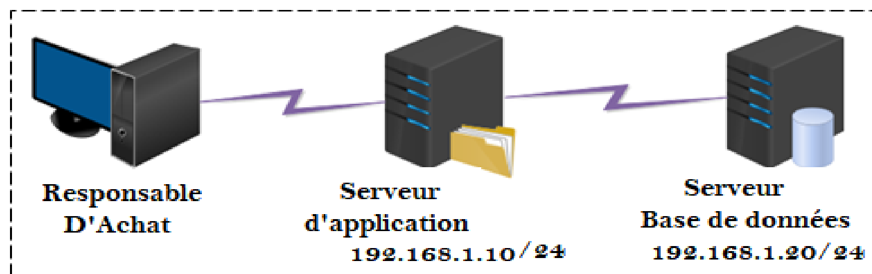

Figure 4 : Architecture client/serveur

Le modèle relationnel et orienté objet d'un article est donné ci-après :

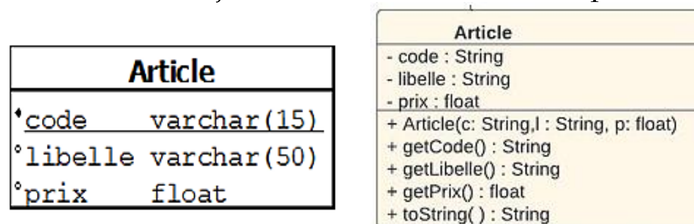

Figure 5 : Modèle relationnel et orienté objet d'un Article

1. Architecture du serveur :

Les classes de cette architecture sont : **Connect** et **Serveur**

1.1. Classe **« Connect » :** permet la connexion à la base de données.

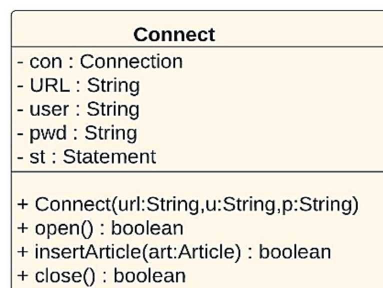

Figure 6 : Classe « Connect »

Implémenter la classe **« Connect »** qui contient : *(2 pts)*

- **5** attributs ;
- Un constructeur avec **3** arguments qui initialise les attributs **url**, **user** et **password** ;
- Une méthode **« open »** qui ouvre une nouvelle connexion avec la base de données et initialise l'objet **Statement**, elle doit retourner l'état de ces opérations ;
- Une méthode d'ajout d'un nouvel article **« insertArticle »**;
- Une méthode de fermeture de connexion.

```
public class Connect {
    private Connection con;
    private String URL,user,pwd ;
    private Statement pst;
    public Connect(String URL, String user, String pwd)   { ... }
    public boolean open()  { ... }
    public boolean insertArticle(Article  art) { ... }
    public boolean close( ) { ... }
}
```

1.2. Classe **« Serveur »** du serveur : *(4 pts)*

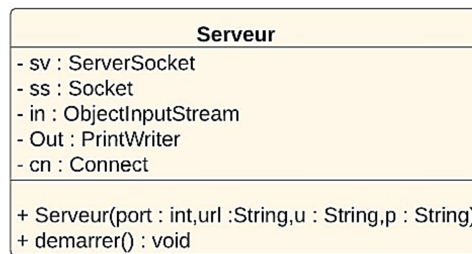

Figure 7 : Classes Serveur

```
public class Serveur {
   private ServerSocket sv;
   private Socket ss;
   private PrintWriter out;
   private ObjectInputStream in;
   private Connect cn ;

   public Serveur(int port, String URL, String user, String pwd) {
          try{
              sv  = ...............................................   [1]
              cn  = ...............................................   [2]
              if(!cn.open()) return ;

           } catch(IOException ex) {  ex.printStackTrace() ; }
       }

   public void demarrer(){
      Article art;
      while(true)  {
        try {
             ss= sv.accept();
             out  =  .......................                                  [3]
             in   =  .......................                                  [4]
             do {
                art =  .................             //lire un article depuis le client    [5]
                if(art != null)
                {
                   cn.insertArticle(art);
                   out.................                                      [6]
                }
                else out.println("end");
             } while(art !=  null);
        } catch(IOException ex) {ex.printStackTrace();  break;}
    }
}
```

Implémenter cette classe en complétant les 2 méthodes suivantes :

- ✓ Un constructeur avec arguments qui instancie les deux objets **ServerSocket** et **Connect** ;
- ✓ La méthode **« demarrer »** qui :
  - Prépare les attributs de type **PrintWriter** et **ObjectInputStream** ;
  - Lit un article envoyé par le client, et :
    - l'ajoute dans la base de données et envoie le message **« Ajout avec succès »** si l'article est non null.
    - envoie le message "**end**" et attend une autre demande de connexion dans le cas contraire.

1.3. Diffusion **Multicast** du serveur *(2 pts)*

On veut que le serveur puisse diffuser un message de notification vers un groupe de machines. Pour ce faire, on utilise le **Multicast UDP**, compléter le script d'envoi suivant :

```
public  void  envoiMulticast(String message)  throws Exception {
           InetAddress  adrGroupe=  InetAddress.…………("224.0.0.1") ; [1]
          int  port = 8888 ;
          MulticastSocket   s = ………………………                   [2]
          s.………(adrGroupe) ;  // ajouter l'adresse au groupe
          DatagramPacket  dp  = ………………………                   [3]
          s.send(dp) ;
          s.………(adrGroupe) ;  // retirer l'adresse du groupe    [4]
   }
```

2. Architecture client :

La classe de cette architecture est :

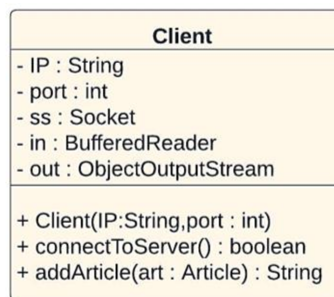

Figure 8 : Classe « client »

```
public class Client{
   private String IP;    private int port ;
   private ObjectOutputStream out;  private BufferedReader in;
   private Socket ss ;
   public Client(...) { ... }
   public boolean connectToServer(){... }
   public String addArticle(Article art){ ... }

}
```

Implémenter cette classe qui contient : *(3 pts)*

- ✓ Un constructeur avec **2** arguments qui initialise les **2** premiers attributs ;
- ✓ La méthode **« connectToServer »** qui instancie le socket **« service »** en préparant les deux objets de communication (**in** et **out**) et retourne l'état de connexion ;
- ✓ La méthode **« addArticle »** qui envoie un article à ajouter et retourne le message de notification envoyé par le serveur.

DOSSIER II : VALIDATION DES COMMANDES *(10 points)*

Afin de valider ou annuler une commande faite par un client, le responsable d'achat utilise une application Desktop développée sous Microsoft Visual Basic. Les données sont stockées dans une base de données SQL-Server. Les caractéristiques de ce serveur sont :

- Nom du serveur : **srv_resp** ;
- Nom de la base de données : **db_commande** ;
- Authentification de Windows.

Le **MLDR** de cette base de données est le suivant :

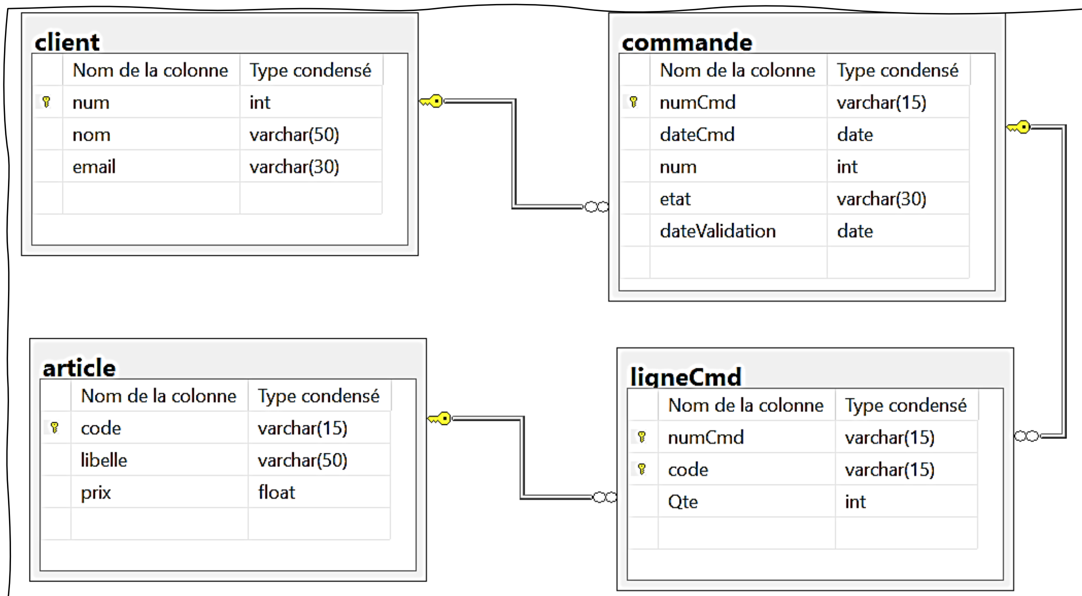

Figure 9 : Modèle logique de données relationnel

Informations complémentaires

- Le champ **«etat»** de la table **«Commande»** peut avoir l'une des valeurs suivantes : *En cours, Validée, Traitée, Annulée.*

> **Consigne** : Veuillez indiquer le mode de connexion utilisé (***mode connecté*** *ou* ***non connecté***).
> Toutes les questions doivent être traitées avec le mode de connexion choisi.

1. Dans le module principal « **Md1.bas** » :

1.1. Déclarer les objets de connexion et les bibliothèques nécessaires pour exploiter ces objets. *(1 pt)*

1.2. Créer la procédure « **Connecter()** » qui établit une connexion avec le serveur de base de données et affiche un message d'erreur en cas d'échec. *(2 pts)*

```
Public Sub Connecter()
    ...
End Sub
```

2. Créer la procédure « **remplir_Clients(…)** » qui prend en argument un contrôle **ComboBox** et le remplit par les noms des clients de la table « **Client** ». *(1,5 pt)*

```
Private Sub remplir_Clients(ByVal cbClients As ComboBox)
    ...
End Sub
```

3. Écrire le code de la procédure **« lister_ArticlesForCmd(…) »** qui affiche les articles d'une commande. La liste doit être affichée dans l'objet **DataGridView** nommé **« DGV_Liste »** comme suit : *(2 pts)*

	Code	Libelle	Prix	Quantité
▶				

```
Private Sub lister_ArticlesForCmd(ByVal numCmd As String)
    ...
End Sub
```

4. Donner le code de la fonction **« LignesCmdExist(…) »** qui prend en argument le numéro d'une commande et retourne : *(1,5 pt)*

- **« true »** s'il existe une ligne de commande correspondant à ce numéro ;
- **« false »** dans le cas contraire.

```
Private Function LignesCmdExist(ByVal numCmd As String) As Boolean
    ...
End Function
```

5. Écrire le code de la procédure **« ValidateCmd(…) »** qui prend en argument le numéro d'une commande et qui modifie son état avec la valeur « *Validée* » et sa date de validation avec la date système de l'application. *(2 pts)*

```
Private Sub ValidateCmd (ByVal numCmd As String)
    ...
End Sub
```

DOSSIER III : CRÉATION DES COMMANDES *(8 points)*

Afin de passer une commande, le client utilise une application Web. Les enregistrements sont stockés dans la base de données « **db_commande** » implémentée sous MYSQL dont une partie de son **MLDR** est la suivante :

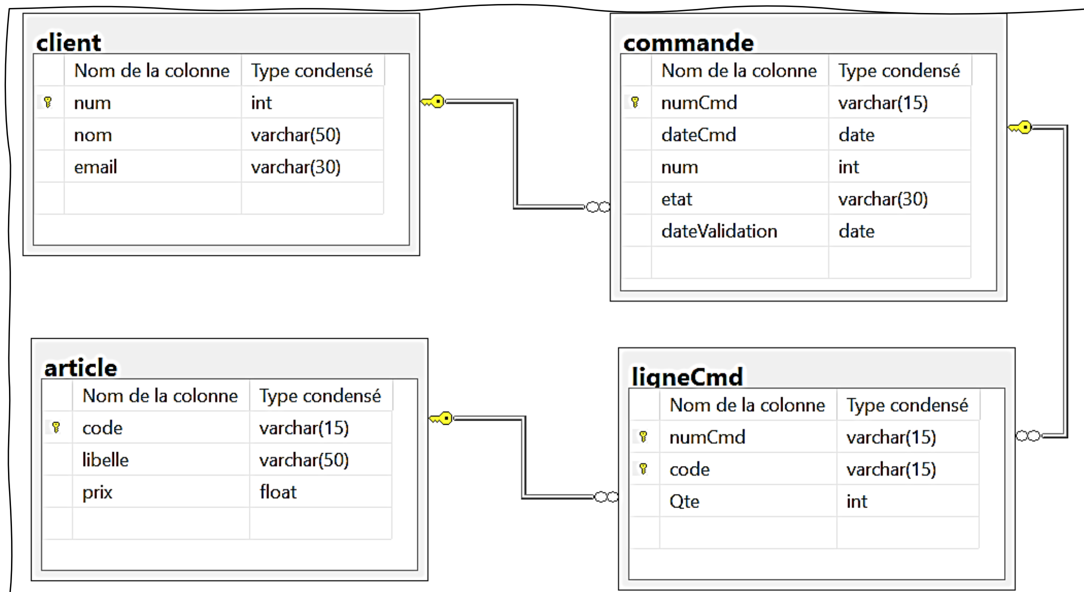

Figure 10 : Partie du Modèle logique de données relationnel

- Nom du serveur : **srv_declaration** ;
- Nom de la base de données : **db_commande** ;
- Nom d'utilisateur : **« user_achat »** et son mot de passe est : **user@achat@**

Informations complémentaires

- Le champ **«etat»** de la table **«Commande»** peut avoir l'une des valeurs suivantes : *En cours, Validée, Traitée, Annulée.*

> **Consigne** : Vous devez utiliser un seul mode de programmation : procédural ou orienté objet.

1. Créer le fichier **« connecter.php »** permettant la connexion à la base de données **« db_commande »** en récupérant son identifiant ; *(1 pt)*

2. La page web permettant de passer une nouvelle commande nommée : **« passerCmd.php »** est la suivante :

Passer une commande

Numéro Client: 22
Numéro Commande: CMD002

Liste des articles

code	Libelle	Prix	Quantité	
art001	PC DELL I7	6500	2	☑
art002	Imprimante EPSON	3000		☐
art003	PC Bureau I5	4000	3	☑

Passer Commande

Figure 11 : Page Web d'ajout d'une nouvelle commande

Le code de cette page est le suivant :

```
<!DOCTYPE html>
<html>
  <head>
      <script type="text/javascript">
        function tester()  {
          tab=document.getElementsByName('cb[]');
          let long=tab.length;
          for(.............................................)
             if(......   .checked)  return ....................
          return ...............................
        }
      </script>
  </head>
  <body>
      <h3> Passer une commande</h3>
      <form method="post" action="addCmd.php" >
      <table>
          <tr><td><label>Numéro Client</label></td>
          <td><input type="text" name="numCl"></td></tr>
          <tr><td><label>Numéro Commande</label></td>
          <td><input type="text" name="numCmd"></td></tr>
      </table>
      <h3>Liste des articles</h3>
      <table border='2'>
         <thead>
            <tr><th>code</th><th>Libellé</th><th>Prix</th><th>Quantité</th>
            <th></th>  </tr>
          </thead>
          <tbody>
             <?php
                 include "listing.php";
             ?>
          </tbody>
        </table>
        <input type="submit" value="Passer Commande" onclick="return tester();">
      </form>
   </body>
</html>
```

2.1. Compléter le code de la fonction « **tester** » qui retourne **true** si au moins un article est sélectionné et **false** dans le cas contraire. *(1 pt)*

2.2. Écrire le code du fichier **« listing.php »** qui liste tous les articles *(2 pts)*

- Les **checkbox** doivent avoir comme attribut **value** le code de l'article et comme **name** : « **chb[ ]** »
- Les champs quantités sont de type « **number** » et doivent avoir comme attribut **name** le code de l'article situé sur la même ligne de la table.

3. Le code du fichier **« addCmd.php »** est le suivant : *(4 pts)*

```
<?php
require "connecter.php";
// Insertion de la nouvelle commande
      ………..[3.1]                                                    (2 pts)

// Insertion des articles choisis
if(isset($_POST['chb']))
{   $art = array();
    $qte= array();
   foreach($_POST['chb'] as $valeur)
   {
      $art[ ]=$valeur;
      $qte[ ]=$_POST[$valeur] ;
   }
   …….     [3.2]                                                     (2 pts)
         // Insérer les données dans la table « ligneCmd »
}
?>
```

Figure 12 : Script « addCmd.php »

La partie **[3-1]** permet d'enregistrer une nouvelle commande avec :

- La date de la commande est celle du système. (**NB** : *utiliser la fonction PHP* **date('Y-m-d')**)
- L'état de la commande est « ***En cours*** ».
- La date de validation doit garder sa valeur par défaut qui est ***NULL***.

La partie **[3-2]** permet d'enregistrer tous les articles choisis pour cette commande.

Donner le code des deux parties **[3.1]** et **[3.2].**

% Bon courage %