E5: PRODUCTION ET FOURNITURE DE SERVICES INFORMATIQUES

Durée : 4 heures **Coefficient : 5**

CAS WALNUT

Ce sujet ***est composé de quatre dossiers indépendants.***
Tous les types de calculatrice sont INTERDITS pour cette épreuve.

Barème :

Dossier 1 : Distribution de noix	**30 points**
Dossier 2 : Commercialisation de plants de noyers	**16 points**
Dossier 3 : Suivi des commandes	**42 points**
Dossier 4 : Mise en production du service	**12 points**
Total :	**100 points**

Présentation du contexte

WALNUT est une coopérative agricole spécialisée dans la collecte, la transformation et le conditionnement de la noix qualifiée "noix de Grenoble". Les membres de la coopérative sont des producteurs situés dans la vallée de l'Isère (affluent du Rhône).

Dossier 1	**Distribution de noix**

Documents à utiliser : annexes 1A, 1B et 1C.

WALNUT est engagée dans une démarche qualité et souhaite améliorer la traçabilité de ses produits. Elle envisage donc de réorganiser la partie de son système d'information relative à la gestion des approvisionnements et des ventes.

Les producteurs

Les membres de la coopérative sont des producteurs de noix, appelés nuciculteurs. Chaque année, la coopérative accepte, dans une certaine proportion, la production de producteurs non adhérents.

Les producteurs qui sont engagés dans la lutte pour le respect de l'environnement obtiennent des certifications garantissant leur engagement dans ce domaine (par

exemple les labels Agriculture Biologique et GLOBALGAP). Pour les producteurs adhérents, on conserve une trace des différentes certifications qu'ils possèdent (certification et date d'obtention).

La coopérative souhaite conserver dans son système d'information certaines caractéristiques des producteurs avec lesquels elle travaille : nom et adresse de la société, nom et prénom du responsable de production. Pour les producteurs adhérents, il faut également conserver la date de leur adhésion.

Les vergers

Un producteur peut exploiter plusieurs vergers. Parmi les caractéristiques d'un verger, on trouve la variété de la noix, la superficie plantée et le nombre d'arbres à l'hectare. Certains vergers permettent au producteur d'obtenir les labels AOC (appellation d'origine contrôlée), avec l'appui du Comité Interprofessionnel de la Noix de Grenoble.

Pour avoir droit à l'appellation d'origine contrôlée (AOC), les productions de noix doivent remplir deux conditions :

- provenir de vergers situés dans les communes référencées par la réglementation. Ces communes sont toutes situées le long de la vallée de l'Isère,
- correspondre aux variétés de noix Franquette, Mayette ou Parisienne.

La production

Chaque année la coopérative réceptionne les récoltes de ses producteurs. Cette activité de l'entreprise, comme toutes les autres, doit être inscrite dans une démarche qualité assurant la traçabilité des produits finis.

Afin d'assurer cette traçabilité, la coopérative identifie chaque livraison de noix réalisée par un producteur. Une livraison est caractérisée par le verger dont proviennent les noix, sa date, le type de produit (entière fraîche ou entière sèche) et la quantité livrée.

Après réception, les noix sont triées en fonction de leur taille (voir *annexe 1A*). La coopérative utilise pour cela une calibreuse destinée à répartir la livraison en différents lots de production. Un lot de production est donc constitué de l'ensemble des noix d'une livraison conformes à un calibre particulier. L'*annexe 1B* présente un exemple de livraison répartie en différents lots de production.

Les campagnes de ventes

Les noix sont vendues à des clients qui sont pour l'essentiel des grossistes, des entreprises de la grande distribution ou des professionnels des métiers de la bouche (pâtissiers, restaurateurs).

Une commande est caractérisée par un numéro de commande, une date de commande et un client. Chaque commande porte sur un seul lot de production déterminé en fonction de la demande du client (variété de noix, type de produit et calibre), ce qui permet d'assurer la traçabilité des produits vendus.

Les informations concernant le client (nom, adresse, nom du responsable des achats) doivent être enregistrées.

Pour chaque commande, il faut également mémoriser les types de conditionnement et les quantités commandés. Les différents types de conditionnement proposés par WALNUT sont : le sachet (de 250 g, 500 g et 1 kg), le filet (de 1 kg, 5 kg, 10 kg et 25 kg) et le carton de 10 kg.

Une commande peut par exemple comporter 800 sachets de 1 kg et 300 filets de 10 kg pour un poids total de 3 800 kg.

Travail à faire	
1.1	Un de vos collègues a débuté l'analyse (Annexe 1C). Vous êtes chargé de produire un schéma relationnel correspondant à cette partie de l'analyse. Vous utiliserez le formalisme de votre choix parmi (Annexe 1D) parmi : ● Diagramme de schéma relationnel (connexions FK - > PK) ● Description textuelle
1.2	À partir des informations fournies dans ce dossier et des *annexes 1A et 1B*, compléter l'analyse du domaine, concernant les approvisionnements et les ventes de noix. Vous prolongerez l'analyse directement sur l'annexe 1C, en utilisant le formalisme UML de diagramme de classes entité du domaine.

Dossier 2	Commercialisation de plants de noyers

Document à utiliser : aucun.

À la demande de ses adhérents, WALNUT développe une nouvelle activité de culture de plants de noyers compatibles avec la charte qualité associée au label AOC. Elle cultive et commercialise des plants de noyers de différentes variétés. Pour cela elle utilise une base de données, dont voici un extrait :

```
TypeUsageArbre(id, libellé)

      id : clé primaire

      Limité à  : (1, 'Fruitier'), (2, 'Huile'), (3, 'Greffon')
      Il s'agit du type d'usage principal d'une variété d'arbres

EspèceArbre(id, libellé)

      id : clé primaire

      Par exemple : (1, 'noyer commun'), (2, 'noyer cendré'),
                    (3, 'noyer noir')

VariétéArbre(id, idEspèce, idTypeUsage, libellé)

      id : clé primaire
      idEspèce : clé étrangère en référence à id de EspèceArbre
      idTypeUsage : clé étrangère en référence à id de TypeUsageArbre

      Par exemple : (1, 1, 1,'franquette'), (2, 1, 1,  'parisienne'),
                    (3, 1, 1, 'mayette'), (4, 3, 3, 'soligo')

Dimension(id, min, max)

      id : clé primaire

      min, max :  plage de hauteur approximative de l'arbre, en mètre
      Par exemple : (1, 0.15, 0.30), (2, 0.30, 0.50), (3, 0.4, 0.6),
                    (4, 2.5, 3.0)

PrixVenteArbre(idVariété, âge, idDimension, prix)

      idVariété, âge, idDimension : clé primaire
      idVariété : clé étrangère en référence à id de VariétéArbre
      idDimension : clé étrangère en référence à id de Dimension

      âge : valeur entière en année
      prix : prix de vente hors taxes à l'unité, en euros

      Par exemple : (1, 2, 2, 9),  (1, 2, 5, 12.5)
```

Travail à faire	
2.1	En prenant en compte les exemples de données ci-dessus, expliquer l'intention de cette requête : `INSERT INTO PrixVenteArbre VALUES(1, 1, 1, 8)`

2.2	Expliquer le résultat que permet d'obtenir la requête ci-dessous puis en proposer une version simplifiée.
	`SELECT EspèceArbre.libellé AS espèce,`
	`VariétéArbre.libellé AS variété`
	`FROM VariétéArbre, TypeUsageArbre, EspèceArbre`
	`WHERE VariétéArbre.idTypeUsage = TypeUsageArbre.id`
	`AND VariétéArbre.idEspèce = EspèceArbre.id`
	`ORDER BY EspèceArbre.libellé`

2.3	Écrire la requête SQL permettant à un utilisateur de connaître le nombre de variétés d'arbres proposées par type d'usage (id, libellé).
2.4	Écrire la requête SQL permettant de connaître les variétés d'arbres (libellé) ne faisant l'objet d'aucun prix de vente.

Au début de cette année, la coopérative WALNUT désire augmenter de 3% le prix de vente de tous ses noyers âgés d'au moins 3 ans, toute variété confondue.

Travail à faire	
2.5	Écrire la requête SQL effectuant cette augmentation.

Dossier 3	Suivi des commandes

Documents à utiliser : annexes 2B, 2C, 2D et 2E.

Les distributeurs de noix sont des partenaires privilégiés de WALNUT. Ils représentent en effet 80% des débouchés de la production de WALNUT.

WALNUT réalise 10% de son chiffre d'affaires avec l'un de ces distributeurs (SUPERMARKET). Ce distributeur vient d'actualiser son système d'information et impose à ses fournisseurs une interconnexion des systèmes d'information au moyen d'une architecture orientée "*Web Service*" (architecture WS).

Chez WALNUT les commandes sont gérées par téléphone tant pour la prise de commande que pour le suivi. Afin d'améliorer le suivi des commandes et de répondre à la demande de SUPERMARKET, WALNUT décide donc de mettre à la disposition de ses clients un service *web* qui leur permettra d'obtenir l'état de leurs commandes en cours, en format XML et JSON.

Principe de fonctionnement

- Le distributeur devra avoir obtenu auprès de WALNUT un identifiant de connexion au service (exemple carr15432 pour le distributeur *carreclerc*).
- À l'aide de cet identifiant de connexion, une application cliente pourra interroger le service *web* à l'aide d'une requête HTTP GET :

http://ws.walnut.fr/services/etatCommandes/distributeur/carr15432

- Il obtiendra en retour la liste de ses commandes en souffrance (non expédiées) au format XML (*annexe 2B*).

Travail à faire	
3.1	Indiquer les avantages apportés aux distributeurs par le futur service par rapport au système actuel.
3.2	Donner trois arguments montrant l'intérêt de la mise en place de cette solution pour WALNUT.

Vous participez au développement d'une application métier.

Un ensemble de classes présentées en *annexes 2C et 2D* est en cours de développement.

- La classe *PersistanceSQL* permet d'accéder à la base de données. Elle permet de charger et/ou d'enregistrer les données dans la base.
- La classe *GestionCommandes* permet d'orchestrer les traitements liés au service Web.
- La dépendance *Utilise* représente simplement le fait que la classe *GestionCommandes* utilise la classe *Distributeur* puisqu'elle possède une méthode retournant un objet de la classe *Distributeur*.

Vous disposez également de la classe *Collection* présentée en *annexe 2E*.

Chargement des données depuis la base

Le fonctionnement de la méthode *GetDistributeur*() de la classe *GestionCommandes* est le suivant :

- Elle utilise la classe *PersistanceSQL* pour charger les données concernant un distributeur depuis la base (l'identifiant de ce distributeur étant passé en paramètre).
- Ce chargement provoque de manière automatique le chargement des données concernant l'ensemble des commandes de ce distributeur, ainsi que les produits associés.
- Elle retourne l'objet *Distributeur* ainsi créé, cet objet possédant la collection des commandes du distributeur.

Travail à faire	
3.3	Écrire la méthode *GetDistributeur()* de la classe *GestionCommandes*.
3.4	Écrire la méthode *EnCours()* de la classe *Commande*.
3.5	Écrire la méthode *GetCommandesEnCours()* de la classe *Distributeur*.
3.6	Écrire la méthode *XmlCommande()* de la classe *Commande*.
3.7	Réécrire (déplacer) la méthode **de classe** *XmlNonLivrées()* de la classe *GestionCommandes* en tant que méthode **d'instance** de la classe *Distributeur*

Une classe *TestCommande* a été créée dans le but de tester votre travail, sur la base d'une base de données de test.

Sa méthode *TestNombreCommandesEnAttenteCarreClerc()* ébauchée ci-dessous doit vérifier que le nombre de commandes en souffrance du distributeur *Carreclerc* d'identifiant "carr15432", est bien conforme à l'annexe 2B. Les données se trouvent dans la base de données de nom "baseWalnutTest" sur le serveur d'adresse IP 192.168.10.240 écoutant le port 2207.

Classe TestCommande

```
Public
      @Test
      Procédure TestNombreCommandesEnAttenteCarreClerc ( )
            persist : PersistanceSQL
            persist =  new PersistanceSQL( /* arguments à compléter */)
            assertNotNull(persist);

            // à compléter…
      fin procédure
```

Fin classe

Travail à faire	
3.8	Compléter la méthode *TestNombreCommandesEnAttenteCarreClerc()* de la classe *TestCommande*. (voir Annexe 2F)

Dossier 4	Mise en production du service

Document à utiliser : aucun.

L'application de gestion du service *web* de suivi des commandes est terminée ; elle est actuellement installée sur votre poste en local (poste dédié au développement).

Elle paraît dans cette configuration conforme aux exigences définies au départ par WALNUT. Vous devez maintenant la déployer sur le serveur *web* (serveur HTTP) de la coopérative pour tester la mise en production.

Vous devez transférer le service *web* sur le serveur HTTP à partir de votre poste. Or le poste de développement ne dispose actuellement que d'un éditeur de texte avec reconnaissance syntaxique et d'un serveur *web* de test accessible seulement en local (*localhost*).

Travail à faire	
4.1	Proposer une solution pour transférer facilement vos scripts sur le serveur.

Walnut va utiliser un prestataire de service pour cette application de suivi de commandes et utilisera le protocole TLS/SSL pour sécuriser celui-ci. Ce prestataire est connu et répertorié dans les navigateurs en tant qu'autorité de certification.

Travail à faire	
4.2	Présenter le rôle d'une autorité de certification dans l'échange TLS/SSL.
4.3	Expliquer le rôle du certificat d'une autorité de certification présent dans la configuration d'un navigateur.

Annexe 1A - Les calibres de noix

Identifiant	libellé
1	inférieur à 24 mm
2	24 à 28 mm
3	28 à 30 mm
4	30 à 32 mm
5	32 à 34 mm
6	supérieur à 34 mm

Annexe 1B - Exemple de livraison

Code Livraison : L0512
18/04/2010

Date de livraison :

Producteur :

Les noix de Polienas
52, route de Grenoble
38 210 POLIENAS

Verger :

V058 - Verger du moulin

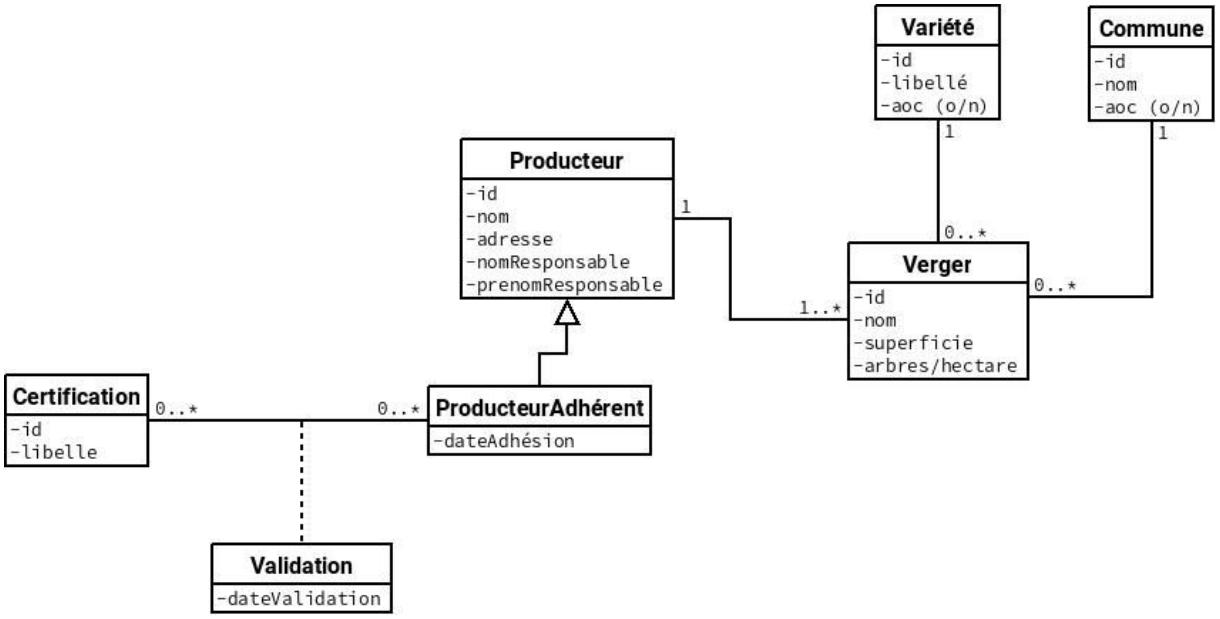
Variété : Franquette
Type de produit : Entière fraîche

Quantité : 12 520 Kg

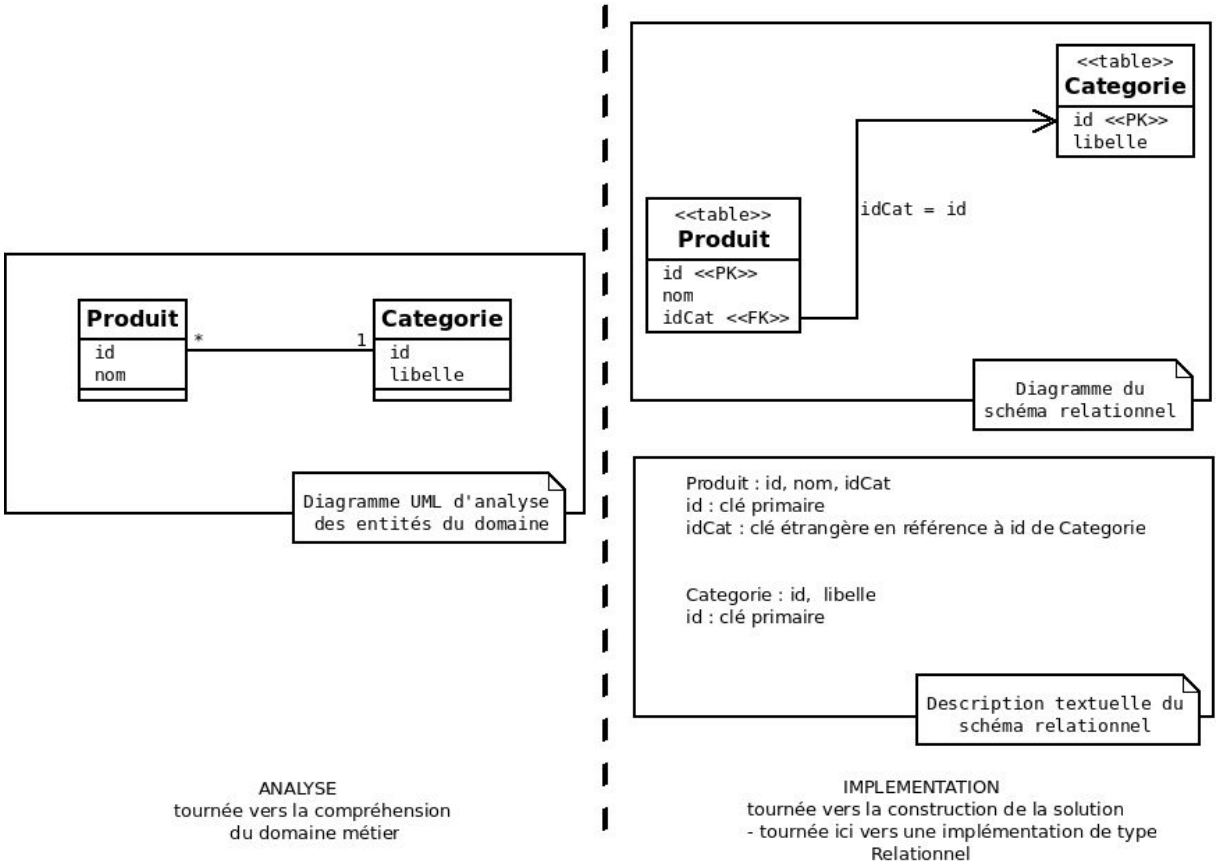
Lots de production après calibrage :

Numéro de lot	Calibre	Quantité
1	24 à 28 mm	4 200 Kg
2	28 à 30 mm	3 820 Kg
3	30 à 32 mm	1 580 Kg
4	32 à 34 mm	2 920 Kg

Annexe 1C - Diagramme de classes partiel



Annexe 1D - Exemple de formalismes

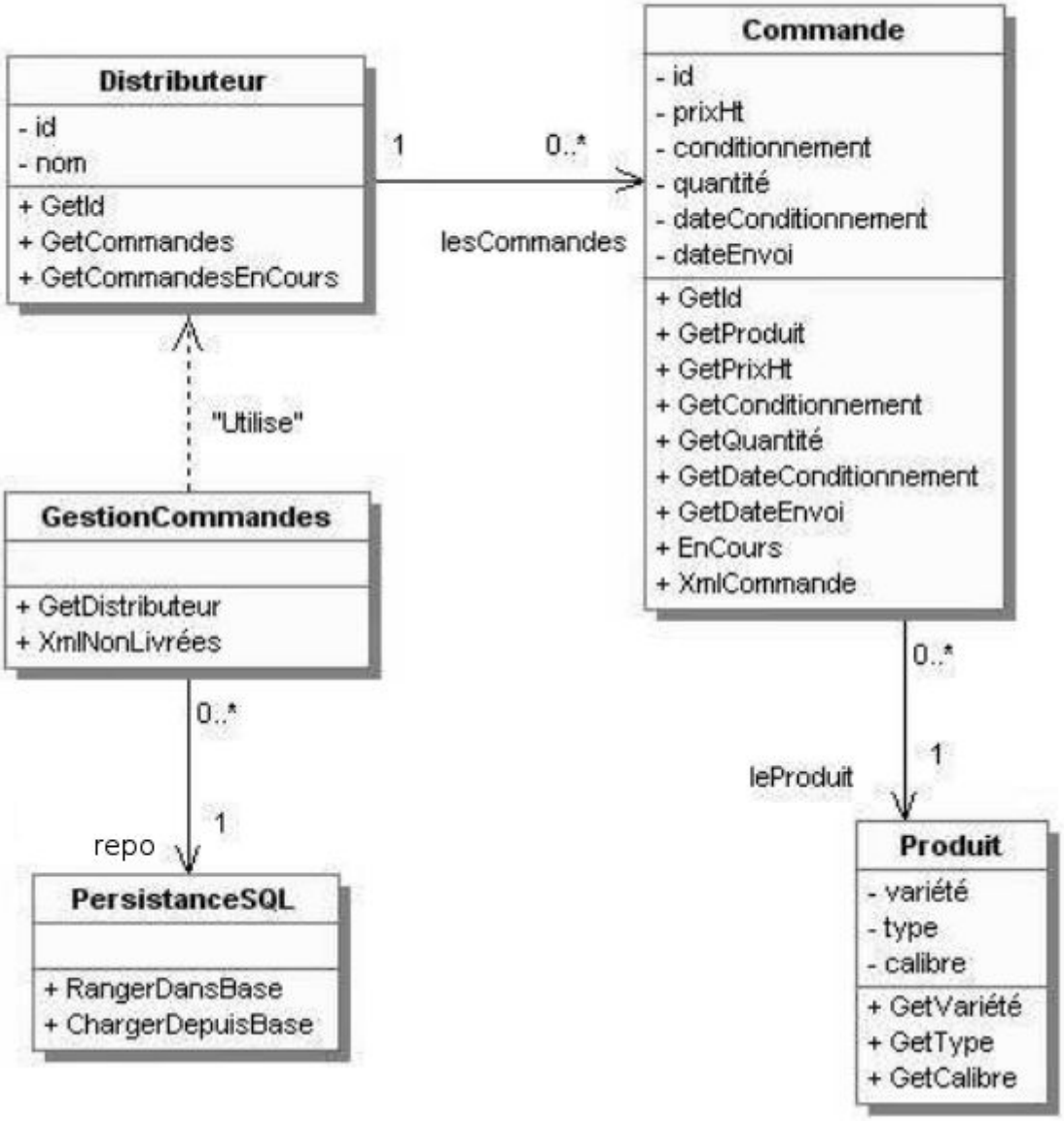


Annexe 2B - Exemple d'utilisation du service *web*

Exemple : le distributeur *carreclerc* désire obtenir l'état de ses commandes restant à livrer. Une commande qui reste à livrer est une commande dont la date d'envoi n'est pas renseignée (sa valeur est NULL). Il sollicite le service *web* "etatCommandes" et obtient le fichier XML ci-dessous.

```
<?xml version="1.0" encoding="UTF-8"?>
<commandes idDistributeur="carr15432"
xmlns:xlink="http://www.w3.org/1999/xlink">
      <commande id="00213">
            <produit variete="Mayette" type="Fraiche Entière" calibre="2" />
            <conditionnement type="filet 1kg" />
            <quantite>50</quantite>
            <date_conditionnement>12-05-09</date_conditionnement>
            <date_envoi>null</date_envoi>
      </commande>
      <commande id="00215">
            <produit  variete="Parisienne"  type="Cerneaux"  calibre="1"  />
            <conditionnement type="filet 5kg" />
            <quantite>100</quantite>
            <date_conditionnement>08-05-09</date_conditionnement>
            <date_envoi>null</date_envoi>
      </commande>
</commandes>
```

Annexe 2C - Diagramme de classes partiel



Par souci de simplification, on considère ici qu'il n'y a qu'une seul conditionnement possible par commande.

Annexe 2D - Classes métier

Classe PersistanceSQL

Public
    // Constructeur
    PersistanceSQL (ipBase : chaîne, port : entier, nomBaseDonnee : chaîne)
    // Construit un objet *PersistanceSql.* Cet objet permettra de charger les données depuis une base
    // de données ou de les sauvegarder dans la base.

    procédure RangerDansBase (unObjet : Objet)
    // Stocke les données de l'objet dans la base de données.

    fonction ChargerDepuisBase (id : chaîne, nomClasse : chaîne) : Objet de la classe *NomClasse*.
    // Retourne l'objet de la classe *NomClasse* dont l'identifiant est "id". Cet objet est chargé
    // depuis la base de données, ainsi que l'ensemble de ses objets liés (voir l'exemple d'utilisation
    // ci-dessous). Retourne NULL si aucun objet de cette classe ne possède cet identifiant.
Fin classe

Exemple d'utilisation :

// *persist* est une instance de PersistanceSQL
Distributeur leDistributeur = persist.ChargerDepuisBase ("2", "Distributeur")
// *leDistributeur* référence une instance de la classe Distributeur dont l'identifiant est 2.

Toutes les commandes du distributeur sont automatiquement chargées dans la collection ***leDistributeur.lesCommandes***. Chaque produit commandé est également chargé, et se trouve donc référencé par le champ *leProduit* de l'objet Commande correspondante.

Classe Distributeur

Privé
    id : chaîne
    nom : chaîne
    lesCommandes : Collection de <Commande> // Toutes les commandes du distributeur.

Public
    // Constructeur
    Distributeur (unId : chaîne , unNom : chaîne)
    // Construit un objet Distributeur. Á ce stade, il n'est pas stocké dans la base de données.

    fonction GetId () : chaîne
    // Retourne l'identifiant du distributeur.

    fonction GetCommandes () : Collection de <Commande>
    // Retourne l'ensemble des commandes passées par ce distributeur.

    fonction GetCommandesEnCours () : Collection de <Commande>
    // Retourne une collection constituée des commandes en cours (non expédiées) du distributeur.

Fin classe

Classe Commande

Privé

id : entier	// Identifiant de la commande.
leProduit : Produit	// Le produit commandé (catégorie de noix).
prixHt : réel	// Prix unitaire du produit négocié avec le client.
conditionnement : chaîne	// Type de conditionnement.
quantité : entier	// Quantité de produits conditionnés commandée.
dateConditionnement : Date	// Date de conditionnement de la commande.
dateEnvoi : Date	// Date d'envoi de la commande.

Public

// Constructeur
Commande (…)
// Construit un objet Commande, la liste des paramètres est sans importance.
// Le champ dateEnvoi est initialisé à NULL.

fonction GetId () : entier
// Retourne l'identifiant de la commande.

fonction GetProduit () : Produit
// Retourne le produit commandé.

fonction GetPrixHt () : réel
// Retourne le prix unitaire négocié avec le client.

fonction GetConditionnement () : chaîne
// Retourne le type de conditionnement des produits de cette commande.

fonction GetQuantité () : entier
// Retourne la quantité commandée.

fonction GetDateConditionnement () : Date
// Retourne la date de conditionnement de la commande.

fonction GetDateEnvoi () : Date
// Retourne la date d'envoi de la commande.

fonction EnCours () : booléen
// Renvoie vrai si la commande n'est pas encore expédiée, faux sinon.
// Une commande n'est pas expédiée si sa date d'envoi contient NULL.

fonction XmlCommande () : chaîne
// Retourne la chaîne correspondant au code XML représentant la commande (voir *annexe **2B***).
// Cette fonction est appelée par la méthode *XmlNonLivrées()* de la classe
// *GestionCommandes* décrite ci-après.

Fin classe

Classe Produit

```
Privé
    variété : chaîne      // Variété de noix, exemple : "Mayette".
    type : chaîne         // Type de noix, exemple : "fraîche entière".
    calibre : entier      // Calibre des noix, exemple : 2.

Public
    // Constructeur
    Produit ( … )     // Construit un objet Produit, la liste des paramètres est sans importance.

    fonction GetVariété () : chaîne
    //  Renvoie la variété du produit.

    fonction GetType () : chaîne
    // Renvoie le type du produit.

    fonction GetCalibre () : entier
    // Renvoie le calibre du produit.
```

Fin classe

Classe GestionCommandes

```
Privé
    repo : PersistanceSQL
    // Attribut qui permet de rendre les objets métiers accessibles.

Public
    //Constructeur
    GestionCommandes (repo: PersistanceSQL)
    // Construit un objet GestionCommandes avec un gestionnaire de persistance associé.
```

fonction GetDistributeur (idDistributeur : chaîne) : Distributeur

```
    // Retourne l'objet Distributeur qui possède l'identifiant idDistributeur passé en paramètre,
    // retourne null si aucun Distributeur ne possède cet identifiant.

    static fonction XmlNonLivrées (unDistributeur :  Distributeur) : chaîne
    // Retourne une chaîne de caractères qui représente le document XML de la liste des commandes
    // non livrées du distributeur passé en paramètre comme le montre l'exemple de l'annexe 2B.
    {
        xml : chaîne
        xml  =  ‘ <?xml version="1.0" encoding="UTF-8"?> ’
        xml =  xml + ‘ <commandes idDistributeur=" ‘ + unDistributeur.GetId() + ‘ " ‘
        xml =  xml + ‘ xmlns:xlink="http://www.w3.org/1999/xlink"> ‘
        enCours : Collection de <Commande>
        enCours = unDistributeur.GetCommandesEnCours()
        commande : Commande
        Pour chaque commande dans enCours faire
            xml = xml + commande.XmlCommande()
        FinPour
        xml = xml + "</commandes>"
        retourner xml
    }
```

Fin classe

Annexe 2E - Classe Collection

Classe Collection de <TypeElément>
// TypeElément peut être un type simple ou une classe.
Public
    //Constructeur
    Collection de <TypeElément> () // Construit une collection d'objets "TypeElement".
    …
    fonction nbEléments () : entier // Retourne le nombre d'éléments de la collection.
    fonction getElément (unIndex : entier) : TypeElément // Retourne l'objet d'index *unIndex*.
    procédure ajouter (unObjet : TypeElément) // Ajoute la référence *unObjet* à la collection.

Fin Classe Collection

Exemple d'utilisation de la collection :

```
lesClients : new Collection de <Client>
unClient : Client
[...]
lesClients.ajouter(unClient)
[...]

// Parcours de la collection :
Pour chaque client dans lesClients faire
    Si client.getVille() == "Paris" alors
        Afficher (client.geNom() + " habite Paris")
    FinSi
FinPour
```

Annexe 2F - Méthodes de tests

Voici quelques unes des fonctions de test unitaire :

- `assertNul(refObject)` // vérifie que refObject ne référence aucun objet en mémoire
- `assertNotNul(refObject)` // vérifie que refObject référence bien un objet en mémoire
- `assertTrue( <expr. booléenne>)` // vérifie que l'expression booléenne rend true
- `assertEquals( a, b )` // vérifie que a et b ont même état