NFA019 : Série 2.1- Morane Gruenpeter

a. La série 2.1 ce focalise sur deux classes métiers et deux classes structures. Les classes métiers, Article et Ligne de commande, sont des composantes de bases dans notre projet de la superette. En les créant et les utilisant on apprend quels sont les paramètres, les constructeurs et les méthodes essentiels de ces classes. En travaillant sur les classes structures, on apprend comment ranger les composantes de base dans des structures un peu plus compliquées. Après on teste les fonctionnalités de toutes les classes dans la classe client.

b.Les classes : Class Article

Constructor Summary
`Article()` Constructeur par defaut
`Article(int c, java.lang.String d, float pu)` Constructeur avec param

Method Summary	
`int`	`getCode()` getter du code article
`java.lang.String`	`getDesignation()` getter de la designation de l'article
`double`	`getPrixUnitaires()` getter du prix unitaires de l'article
`void`	`setCode(int c)` setter du code de l'article
`void`	`setDesignation(java.lang.String d)` setter de la designation de l'article
`void`	`setPrixUnitaire(float pu)` setter du prix unitaire de l'article
`java.lang.String`	`toString()` return String de l'article courant

Class LigneDeCommande

Constructor Summary
`LigneDeCommande()` Constructeur par défaut
`LigneDeCommande(int ca, int q)` Constructeur avec param

Method Summary	
`int`	`getCodeArticle()` getter de codeArticle

int	**getQuantite**() getter de quantité
void	**setCodeArticle**(int ca) setter du code article
void	**setQuantite**(int q) setter de quantité
java.lang.String	**toString**() return String la ligne de commande
String	facturer() return String de la ligne de commande comme elle apparaitra dans une facture (ayant la désignation de l'article, son prix unitaire et le total calculer par rapport à la quantité)

Class TableArticle21

Constructor Summary
TableArticle21() Constructeur par defaut
TableArticle21(Article a, Article b, Article c, Article d) Constructeur avec param

Method Summary	
void	**ajouter**(uUtilsGruenpeterMorane.Article aAjouter) ajout d'un article au tableau d'articles
boolean	**articleExiste**(int code) retourne vrai si l'article avec ce code existe, après avoir effectué une recherche sur les articles du tableau
int	**codeArticleMax**() trouve le codeArticle de valeur la plus élève dans le tableau d'articles
java.util.Vector<Article>	**getTable**() retourne la table courante d'articles
Article	**retourner**(int code) retourne le premier article de ce code après une recherche dans la table des articles.
boolean	**supprimer**(int code) supprime un article de code "code" du tableau des articles
int	**taille**() retourne le nombre d'article dans la table d'articles
java.lang.String	**toString**() return toutes les lignes du tableau d'articles
void	setLesCommandes()

	setteur du vecteur lesCommandes
Vector<LigneDeCommande>	getLesCommandes() getteur du vecteur lesCommandes

Class UneCommande21

Constructor Summary
UneCommande21() Constructeur par défaut
UneCommande21(LigneDeCommande a, LigneDeCommande b, LigneDeCommande c, LigneDeCommande d) Constructeur avec param

Method Summary	
void	**ajouter**(uUtilsGruenpeterMorane.LigneDeCommande aAjouter) ajoute une ligne de commande à la commande
java.lang.String	**facturer**(uUtilsGruenpeterMorane.TableArticle21 tabArt) est divisée en trois chaînes de caractères ; entête, détails et pied. L'entête est identique, les détails est une boucle qui va chercher les informations sur chaque article dans la facture et le pied calcule et concrétise le total de la facture.
java.util.Vector<uUtilsGruenpeterMorane.LigneDeCommande>	**getLesCommandes**() getter du vecteur
uUtilsGruenpeterMorane.LigneDeCommande	**retourner**(int index) retrouve la ligne de commande d'index "index" dans le vecteur lesCommandes (pas de rapport avec le codeArticle).
void	**setLesCommandes**(java.util.Vector<uUtilsGruenpeterMorane.LigneDeCommande> nouveau) setter du vecteur
boolean	**supprimer**(int code) supprime une ligne de commande tel que cette ligne contient l'article de code en paramètre . Vrai si élément est supprimer, faux si n'est pas dans les commandes
int	**taille**() retourne le nombre de ligne

java.lang.String	**toString**() retourne String de toutes les lignes de la commande

Class ClienJava1UneCommande

Method Summary	
static void	**afficheCommande**(uUtilsGruenpeterMorane.UneCommande21 command e) pour afficher la commande en cours seulement les ligne de commandes qui comprenne code article et quantité
static void	**creerLigneDeCommande**(uUtilsGruenpeterMorane.TableArticle21 t able, uUtilsGruenpeterMorane.UneCommande21 commande) fait appel à la méthode saisieLigneDeCommande et ajoute la ligne de commande retournée à la commande en cours, si celle-ci n'est pas nulle.
static void	**facturer**(uUtilsGruenpeterMorane.TableArticle21 table, uUtilsGruenpeterMorane.UneCommande21 commande) en premier temps la methode prend tous les paramètres et les mets dans un tableau et après elle imprime le tableau
static void	**gestionDesArticles**(uUtilsGruenpeterMorane.TableArticle21 art icles) Affichage de tous les articles dans la table des articles
static void	**gestionDesCommandes**(uUtilsGruenpeterMorane.TableArticle21 t, uUtilsGruenpeterMorane.UneCommande21 c) affichage du menuB, plus gestion du choix effectué pour le menuB
static void	**main**(java.lang.String[] args)
static int	**menuA**() Menu principale; la saisie est possible entre 0 et 2 inclus.
static int	**menuB**() Menu de commande; la saisie est possible entre 0 et 3 inclus.
static int	**saisie**(java.lang.String msg, int inf, int sup) détermine si l'entier choisi est bien dans les bornes possible
LigneDeComman de	**sasieLigneDeCommande**(uUtilsGruenpeterMorane.TableArticle21 t) vérification de la présence de l'article saisie dans le tableau d'article s'il existe => création de la ligneDeCommande retournée, sinon

	retourne null.

c. **Le scénario de la classe Client :**

Dans la méthode main on commence par un premier menu (menuA) dans lequel l'utilisateur choisi une option parmi 3, soit un affichage de tous les articles, soit la création d'une nouvelle commande qui met en place un deuxième menu de choix (menuB), soit la sortie du programme.

Dans un second temps, si le choix effectué est celui de la création de commande, s'affiche le menuB de gestion des commandes qui permet un choix parmi 4 ; soit saisir une ligne de commande, soit afficher le commande en cours, soit éditer la facture, soit sortir du menuB et donc reprendre le menuA.

Chaque choix appelle une méthode statique dédiée dans la classe main.

```
MenuA=>
-gestionDesArticles(TableArticle21 articles)
-gestionDesCommandes(TableArticle21 t, UneCommande21 c )

MenuB=>
-creerLigneDeCommande(TableArticle21 table, UneCommande21 commande)
```

qui appelle, `sasieLigneDeCommande(TableArticle21 t)`, qui renvoie une ligne de commande

`-afficheCommande(UneCommande21 commande)`

`-facturer(TableArticle21 table, UneCommande21 commande)` qui appelle la méthode facturer dans la classe UneCommande21 et elle appelle la classe LigneDeCommande à son tour.

d. En cours nous avons traité la structuration des classe métiers et des classes structure en choisissant l'emplacement de chaque méthode. Créer une classe métier qui contient un objet de petite taille et le lié à une classe structure me parait très efficace. Cela fait que les classe métiers n'ont aucune prise sur la structuration de leurs données et facilite la gestion des objets et de leurs données. En plus, la méthode facture que l'on a travaillé en cours, cette méthode est révélatrice d'une nouvelle approche en décomposant la méthode dans les différentes classes. Au départ j'avais édité toute la facture dans la classe main, cette méthode était très lourde et appelait en boucle les différentes méthodes. J'ai compris à quel point la décomposition d'une méthode pouvait permettre une optimisation du code en le rendant plus simple, lisible et efficace.

e. **<u>Conclusion :</u>**

Pour la série 1 j'avais écrit que j'aimerais bien faire des classes Junit pour tester de manière méthodique mes méthodes, car les tests seulement dans la classe Client me paraisse un peu aléatoire. Cette fois ci, j'ai écrit les classes Junit mais je trouvais que pour cette série les tests Junit son moins adaptés et je n'étais pas sûre sur quelles méthodes les appliqués.

En premier temps je pensais faire un tableau pour la facture. Je pensais également que cela pourrait aider la sauvegarde des facture ultérieurement, mais j'ai vu en cours comment la décomposition de la méthode facture est efficace et donc dans mon code actuellement j'utilise cette méthode. J'aimerais bien comprendre plus spécifiquement quelles méthodes sont adaptées à la décomposition et lesquelles ne le sont pas.

Finalement je trouve qu'après les cours mon code est beaucoup plus propre et cela me donne envie de trouver des technique d'écriture plus simple et plus jolie.

NFA019 : Série 2.2- Morane Gruenpeter

a. La série 2.2 est l'élaboration de la série 2.1, en gardent les deux classes métiers, Article et Ligne de commande, on change les deux classes structure, UneCommande22 et TableArticle22. On utilise la structure de Hashtable pour la classe TableArticle22, pour stocker les articles par leur code (on utilise leur code comme clé). On ajoute notamment une nouvelle classe structure, TableDesCommandes qui est une structure de la classe UneCommande22, aussi ranger dans une Hashtable. Cette table des commandes nous permet de contrôler plusieurs commandes, cela implique une classe client plus lourde et des scénarios plus élaborés.

b. les classes :**Class UneCommande22**

Constructor Summary
UneCommande22() Constructeur par défaut
UneCommande22(int num) Constructeur n°2

Method Summary	
void	**ajouter**(LigneDeCommande lig) ajoute une ligne de commande à la commande
String	**facturer**(utilsGruenpeterMorane.TableArticle22 tabArt)
DateUser	**getDateCommande**()
Vector<LigneDeCommande>	**getLesCommandes**() getter du vecteur
int	**getNumeroCommande**()
LigneDeCommande	**retourner**(int index) retrouve la ligne de commande d'index "index"
void	**setDateUser**(DateUser d)
void	**setLesCommandes**(Vector<LigneDeCommande> nouveau) setter du vecteur
void	**setNumeroCommande**(int numero) setter du numeroDeCommande
boolean	**supprimer**(int code) supprime une ligne de commande tel que cette ligne contient l'article de code en paramètre
int	**taille**()
String	**toString**() return String de toutes les lignes de la commande

Class TableArticle22

Method Summary	
void	**ajouter**(utilsGruenpeterMorane.Article aAjouter) ajout d'un article au tableau d'articles après un test si le code de l'article aAjouter n'existe pas dans la table des articles
boolean	**articleExiste**(int code) retourne vrai si existe
String	**cle**() retourne toutes les clés de la structure
int	**codeArticleMax**() trouve le codeArticle de valeur la plus elvee dans le tableau d'articles
Hashtable<Integer,Article>	**getTable**()
Article	**retourner**(int code) trouve un article de code "code"
void	**supprimer**(int code) supprime un article de code "code" du tableau des articles
int	**taille**()
String	**toString**() return toutes les lignes du tableau d'articles !!

Class TableDesCommandes

Constructor Summary
TableDesCommandes() Constructeur par défaut.
TableDesCommandes(Hashtable< Integer, UneCommande22> table) Constructeur avec param , qui retrouve la dernière commande et définie le numéro de la prochaine commande à ce numéro+1.

Method Summary	
void	**ajouter**(utilsGruenpeterMorane.UneCommande22 nouvelle) ajouter une commande à la table des Commandes après vérification que le numéro de la commande est le même que la clé si il existe s'il n'existe pas, attribution de ce numéro à la commande
String	**cle**() retourne toutes les clés de la structure
int	**cleMaximale**()
Hashtable<Integer, UneCommande22>	**getLesCommandes**() getter des commandes
int	**getNumeroCommandeActuelle**()

	getter du numéro de la prochaine commande
UneCommande22	**retourner**(int numCommande) retourne la commande de la clé demandé
void	**setNumeroCommande**(int num) setter du numéro de commande
void	**setTableDesCommandes**(java.util.Hashtable<java.lang.Integer,utilsGruenpeterMorane.UneCommande22> t) setter de la table des commandes
void	**supprimer**(int numCommande) suppression d'une commande de la table des commande
int	**taille**() retourne la taille de la table
String	**toString**() renvoie la chaine de caractères de toutes les commandes dans la table des commandes. après test si la table des commandes est vide, si vide renvoie "Il n'y a aucune commande"

Class ClientJava22

Method Summary	
static void	**afficheCommande**(UneCommande22 uneCommande) Gestion d'une commande- pour afficher la commande en cours
static void	**afficher**(TableArticle22 lesArticles) Gestion des articles-afficher tous les articles
static void	**ajouter**(TableArticle22 lesArticles) Gestion des articles- ajouter un article à la table des articles
static void	**creerLigneDeCommande**(TableArticle22 lesArticles, UneCommande22 UneCommande) Gestion d'une commande- fait appel à la méthode saisieLigneDeCommande (qui retourne une ligne de commande) et ajoute la ligne de commande à la commande en cours, si celle-ci n'est pas nulle.
static void	**facturer**(TableArticle22 lesArticles, UneCommande22 uneCommande) Gestion d'une commande- en premier temps la méthode prend tous les paramètres et les mets dans un tableau et après elle imprime le tableau
static void	**facturer**(TableDesCommandes lesCommandes, TableArticle22 lesArticles) Gestion des commandes- facturation de la commande choisi par l'utilisateur après une vérification que cette le numéro de commande correspond à une commande existante.
static void	**gestionDesArticles**(utilsGruenpeterMorane.TableArticle22 lesArticles) procédure de gestion des articles
static void	**gestionDesCommandes**(TableArticle22 lesArticles, TableDesCommandes lesCommandes) procédure des gestion des commandes

static void	**gestionDuneCommande**(TableArticle22 lesArticles, TableDesCommandes lesCommandes) Gestion des commandes- Gestion D'une Commande => menu de gestion d'une commande et ajout de cette commande à la table des commandes.
static String	**listeDesCommandes**(TableDesCommandes lesCommandes) Gestion des commandes- liste tous les numéro de commandes passées cette méthode est appelée avoir la chaîne de caractères de toutes les commandes qui existes déjà.
static void	**main**(String[] args) Le déroulement des scénarios
static int	**menuGeneral**() Menu principale; la saisie est possible entre 0 et 2 inclus.
static int	**menuGestionDesArticles**() menu de la gestion des articles
static int	**menuGestionDesCommandes**(TableDesCommandes tdc) menu de la gestion des commandes
static int	**menuGestionDuneCommande**() Menu de commande; la saisie est possible entre 0 et 3 inclus.
static int	**saisie**(String msg, int inf, int sup) methode de saisie d'un entier après un message qui le demande, determine si l'entier choisi est bien dans les bornes possible
static LigneDeCommande	**saisieLigneDeCommande**(TableArticle22 lesArticles) Gestion d'une commande- vérification de la présence de l'article saisie dans le tableau d'article s'il existe => création de la ligneDeCommande retournée, sinon retourne null.
static void	**supprimer**(TableArticle22 lesArticles) Gestion des articles- supprimer un article à la table des articles
static void	**supprimer**(TableDesCommandes lesCommandes) Gestion des commandes- supprimer une Commande après visualisation des numéro de commandes existant
static void	**visualisationDUneCommande**(TableDesCommandes lesCommandes) Gestion des commandes- affiche une seule commande après choix et vérification de la commande voulu
static void	**visualisationToutesLesCommandes**(TableDesCommandes lesCommandes) Gestion des commandes- Affichage de toutes les commandes dans la table des commandes

c. **Le scénario de la classe Client :**

Dans la méthode main on commence par un premier menu (menuGeneral) dans lequel l'utilisateur choisi une option parmi 3, soit la gestion des articles, soit la gestion des commandes, soit la sortie du programme.

En passant au deuxième menu l'utilisateur a d'autres choix ; Chaque choix appelle une méthode statique dédiée dans la classe main. Les choix sont représentés dans l'explication ci-dessous :

```
MenuGeneral=>
        -gestionDesArticles(TableArticle22 lesArticles)
        -gestionDesCommandes(TableArticle22 lesArticles, TableDesCommandes
        lesCommandes )
```

```
MenuGestionDesArticles=> (est appelé dans la méthode gestion des articles)
      -ajouter(TableArticles lesArticles).
      -supprimer(TableArticles lesArticles).
      -afficher(TableArticles lesArticles).

MenuGestionDesCommandes=> (est appelé dans la méthode gestion des commandes)
      -gestionDUneCommande(TableArticles      lesArticles,      TableDesCommandes
      lesCommandes).=> menuGestionDUneCommande
      -supprimer(TableDesCommandes lesCommandes) **
      -visualisationDuneCommande(TableDesCommandes lesCommandes)**.
      -visualisationToutesLesCommandes(TableDesCommandes lesCommandes).
      -facturer(TableDesCommandes lesCommandes)**.
**la méthode listeDesCommande(), qui renvoie un String des commandes existantes
MenuGestionDUneCommande (est appelé dans la méthode gestion d'une commande)
      -creerLigneDeCommande(TableArticle21 table, UneCommande21 commande)
      qui appelle, sasieLigneDeCommande(TableArticle21 t), qui renvoie une ligne de
      commande
      -afficheCommande(UneCommande22 commande)
      -facturer(TableArticle22 table, UneCommande22 commande) qui appelle la méthode
      facturer dans la classe UneCommande21 et elle appelle la classe LigneDeCommande à
      son tour.
```

!! à la fin de ce menu, après avoir choisi 0 pour retourner au menu précédent, la commande est ajoutée à la table des commandes.

d. En cours, nous avons traité la nouvelle structure Hashtable<> et l'interface Enumeration<>.
L'hashtable je connaissais déjà mais j'utilisais pour cette sorte de structure une Hashmap<>.
J'ai essayé de trouver la différence sur internet, mais je ne suis pas sûre si elle est significative.
Sinon, l'Enumeration<> est une nouvelle interface que j'ai découvert, auparavant j'utilisais l'Iterator<> et je vois que leur fonctionnalité est remarquablement similaire.

e. **Conclusion :**

Je trouve que la classe client est trop compliqué avec trop de méthodes. J'ai essayé de la ranger d'une telle manière que toutes les méthodes soient regroupé par thème mais je pense que c'est le maximum que la classe client peut contenir.

Par ailleurs, je me suis posée plusieurs questions pendant le travail pour un protocole de travail régulier. Est-ce que les tests ce font dans les classes structure, avant un appel de remove() ou put() ? Est-Ce que la gestion des articles et des commandes doivent faire aussi des tests avant de faire appel aux classes de structure ? Est-ce que la classe métier, LigneDeCommande, doit garder l'information concernant le prix total de la ligne de commande ? en appelant la méthode toString() d'une structure, Est-ce que l'on doit mettre le titre de ce String (par exemple « *****le Catalogue de tous les articles » dans la tête de la méthode toString() ou dans la méthode gestion de l'appel de ce String ?
Autre souci, en supprimant une commande je ne peux pas attribuer ce numéro déjà utilisé mais supprimer à une nouvelle commande. Je ne suis pas sûre si cela devrait être ainsi.

NFA019 : Série 2.3- Morane Gruenpeter

a. La série 2.3 consiste à organiser la série 2.2, en gardent les classes métiers, classes structures. On ajoute trois classes de gestion qui contrôlent les procédures de l'utilisateur. On a sorti toutes les méthodes qui existaient dans la classe ClientJava22 vers les classes de gestion, maintenant la classe ClientJava23 est presque vide.

b. les classes (les nouvelles classes qui sont dans le package serie23) :

Class GestionTableDesArticles

Method Summary	
`static void`	**ajouter**`(serie22.TableArticle22 lesArticles)` procédure d'ajout d'un article dans la table des articles après test de vérification si le code n'existe pas déjà.
`static void`	**editer**`(serie22.TableArticle22 lesArticles)` affichage tous les articles
`static int`	**menuChoix**`()` retourne l'entier choisi par l'utilisateur qui permet la continuation de la procédure.
`static void`	**menuGeneral**`(serie22.TableArticle22 lesArticles, serie22.TableDesCommandes lesCommandes)` procédure de gestion des articles. La première méthode d'entrée dans la gestion des articles
`static void`	**supprimer**`(serie22.TableArticle22 lesArticles, serie22.TableDesCommandes lesCommandes)` supprimer un article de la table des articles

Class GestionTableDesCommandes

Method Summary	
`static void`	**ajouter**`(serie22.TableArticle22 lesArticles)` procédure d'ajout d'une commande dans la table des commandes en dirigeant l'utilisateur vers la gestion d'une commande.
`static void`	**facturer**`(serie22.TableDesCommandes lesCommandes, serie22.TableArticle22 lesArticles)` facturation de la commande choisi par l'utilisateur après une vérification que cette le numéro de commande correspond à une commande existante.
`static java.lang.String`	**listeDesCommandes**`(serie22.TableDesCommandes lesCommandes)` liste tous les numéro de commandes passées

static int	**menuChoix**(serie22.TableDesCommandes lesCommandes) affichage des possibilités de choix et le retour du choix (int)
static void	**menuGeneral**(serie22.TableDesCommandes lesCommandes, serie22.TableArticle22 lesArticles) la première méthode d'entrée dans la gestion des commandes
static void	**supprimeCascade**(int code, serie22.TableDesCommandes lesCommandes) méthode de suppression des commandes d'un article supprimé en cascade.
static void	**supprimer**(serie22.TableDesCommandes lesCommandes) supprimer une Commande après visualisation des numéro de commandes existant
static void	**visualisationDUneCommande**(serie22.TableDesCommandes lesCommandes) affiche une seule commande après choix et vérification de la commande voulu (remplace editer())
static void	**visualisationToutesLesCommandes**(serie22.TableDesCommandes lesCommandes) Affichage de toutes les commandes dans la table des commandes (remplace editer())

Class GestionDUneCommande

Method Summary	
static void	**afficheCommande**(serie22.UneCommande22 uneCommande) affichage de la commande en cours
static void	**ajouter**(serie22.TableArticle22 lesArticles, serie22.UneCommande22 UneCommande) fait appel à la méthode saisieLigneDeCommande (qui retourne une ligne de commande) et ajoute la ligne de commande à la commande en cours, si celle-ci n'est pas nulle.
static void	**editer**(serie22.TableArticle22 lesArticles, serie22.UneCommande22 uneCommande) imprime la facture de la commande en cours
static int	**menuChoix**() affichage des possibilités de choix et le retour du choix (int)
static void	**menuGeneral**(serie22.TableDesCommandes lesCommandes, serie22.TableArticle22 lesArticles) la première méthode d'entrée dans la gestion d'une commande
static serie22.LigneDeCommande	**saisieLigneDeCommande**(serie22.TableArticle22 lesArticles) vérification de la présence de l'article saisie dans le

	tableau d'article s'il existe => création de la ligneDeCommande retournée, sinon retourne nulle.
static void	**supprimer**(serie22.UneCommande22 commandeEnCours, serie22.TableArticle22 lesArticles) supprimer une LigneDeCommande par numéro d'article après vérification que cet article existe dans la commande.

Class Tools

Method Summary	
static int	**saisie**(java.lang.String msg, int inf, int sup) méthode de saisie d'un entier après un message qui le demande, détermine si l'entier choisi est bien dans les bornes possible
static boolean	**valider**(int codeElement, java.util.Hashtable tableDesElements) méthode de validation si la clé existe dans une hashtable donnée
static boolean	**validerLigneDeCommande**(int codeArticle, serie22.UneCommande22 commande) méthode de validation si la code article existe dans la commande

Class ClientJava23

Method Summary	
static void	**main**(java.lang.String[] args)
static int	**menuGeneral**() Menu principale; la saisie est possible entre 0 et 2 inclus.

c. **Le scénario de la classe Client :**

Dans la méthode main on commence par un premier menu (menuGeneral()) dans lequel l'utilisateur choisi une option parmi 3, soit la gestion des articles, soit la gestion des commandes, soit la sortie du programme. En choisissant, on passe dans la classe de gestion associé au choix et le reste de la procédure est effectué en boucle dans la classe de gestion.

```
MenuGeneral=>
      -GestionTableDesArticles(TableArticle22    lesArticles,    TableDesCommandes
      lesCommandes)
      -GestionTableDesCommandes(TableArticle22    lesArticles,    TableDesCommandes
      lesCommandes )
```

!! J'ai ajouté la classe Tools (boite à outils) dans laquelle j'ai trois méthodes de saisie et validation qui sont appelées par les trois gestionnaires.

d. En cours, nous avons traité la création des classes de gestion et la transition entre la classe ClientJava et le traitement des procédures par les classes de gestions. La création de la méthode purge() (que j'appelle supprimeCasscade()) qui efface en cascade toutes les ligne de commande d'un article qui a été supprimé. Celle-ci on avait pas fait en classe, personnellement j'ai préféré d'utiliser l'Iterator et pas le Enumeration, parce que le Enumeration n'a pas de méthode remove().

e. **Conclusion :**

Je trouve que la classe client est parfaite et j'adore la manière de déléguer les tâches à des classe de gestion.

Cette fois ci j'ai essayé d'organiser mes imports correctement, en créant un projet central avec un package par série j'ai bien réussi à connecter les différente classes, ceci un petit plus pour l'organisation et en voyant une classe BoiteAOutils chez un collègue, j'ai décidé d'ajouter la classe Tools pour ne pas recopier dans chaque gestionnaire la méthode saisie() et valider().

Par ailleurs, j'ai trouvée des réponses pour quelques questions que je me suis posée pendant la série21

Est-ce que les tests ce font dans les classes structure, avant un appel de remove() ou put() ?

Les tests ce font pendant la gestion des éléments, avant chaque appel des classes métier, comme cela les classes métiers ne comporte pas de tests redondant et les gestionnaire contrôle tous les procédures.

Est-Ce que la gestion des articles et des commandes doivent faire aussi des tests avant de faire appel aux classes de structure ?

Maintenant que ces méthodes sont intégrées dans les gestionnaire elle sont les seules à faire des tests, les structure ne doivent pas en générale faire des tests.

Est-ce que la classe métier, LigneDeCommande, doit garder l'information concernant le prix total de la ligne de commande ?

Je ne suis pas sûre, mais je pense que non, parce que si on fait une mise à jour d'une ligne de commande et on voudrait changer la quantité de départ, ou changer le prix de l'article, ça serait encombrant et ça pourrait générer des problèmes de cohérence.

en appelant la méthode toString() d'une structure, Est-ce que l'on doit mettre le titre de ce String (par exemple « *****le Catalogue de tous les articles » dans la tête de la méthode toString() ou dans la méthode gestion de l'appel de ce String ?

Je ne suis pas sûre, mais je pense qu'il faut mettre le titre dans la structure elle-même car on peut envisager des gestionnaires plus générique, comme ça le gestionnaire retourne un String que l'on lui donne.