Ecole Nationale des Sciences Appliquées - Al Hoceima
Département Mathématiques et Informatique

Travaux pratiques N°6 : Exercices de synthèse

Module : Programmation Orientée Objet en Java
Première Année Ingénierie des données

Année Universitaire 2022/2023

Exercice 1 :

Première Partie

Dans ce sujet on s'intéresse au cryptage *(i.e. transformer un texte clair en un texte codé)* des chaînes de caractères avec des algorithmes simples et naïfs. Nous ne considérons que les chaînes de caractères représentant un texte en anglais qui ne contient que des lettres majuscules et des espaces. Soit T une chaîne de caractères respectant les conditions du problème c.à.d. elle ne contient que des lettres majuscules et des espaces. (*Exemple T= " HELLO MY FRIEND ").*

Dans tout le problème, n'écrivez pas le code des accesseurs et les mutateurs (*getters/setters*), on suppose que chaque attribut *x* que vous écrivez possède une méthode *getX* et *setX*.

On considère les deux méthodes naïves de cryptage décrites ci-dessous :

Méthode de cryptage 1 :

- Etape 1 : Inverser la chaîne de caractères
- Etape 2 : Remplacer chaque lettre de T par un entier de la façon suivante 'A'→20, 'B'→21, ..., 'Z'→45.

Exemple : Pour crypter la chaîne de caractères "B Z" voici les étapes à suivre :

- Etape 1 : "B Z" devient "Z B"
- Etape 2 : La chaîne "Z B" est constituée des caractères 'Z', ' ', 'B'. On remplace par les entiers correspondants : 'Z'→ 45 ; ' '→' ' et 'B'→21.
- Finalement la chaîne de caractère "B Z" devient après cryptage sous format codé égale à "45 21"

Méthode de cryptage 2 :

Le principe de la deuxième méthode est d'inverser la chaîne puis faire un certain décalage constant et circulaire de l'alphabet. Par exemple si *décalage = 2*, alors *A devient C, B devient D,…, Y devient A,* et *Z devient B*. Plus concrètement voici les étapes de cette deuxième méthode :

- Etape 1 : Inverser la chaîne de caractères

- Etape 2 : Soit X une lettre quelconque de T, on associe à X un entier (*qu'on note Code(X))* de la façon suivante Code('A')→0, Code('B')→1, Code('C')→2, ..., Code('Z')→25. Remplacer chaque lettre X de T par le caractère qui correspond à l'entier *((Code(X) + décalage) modulo 26).*

Exemple : On suppose que le décalage constant choisi est égale à 3 (*décalage = 3*). Pour chiffrer la chaîne de caractères "B Z" voici les étapes à suivre :

- Etape 1 : "B Z" devient "Z B"
  Etape 2 : On remplace les lettres par leurs équivalents numériques : 'B' →1 et 'Z'→25
  Pour 'B' : (Code('B') + décalage) modulo 26 = (1+3) % 26 = 4% 26 = 4. Donc 'B' sera remplacé par 'E' (car Code('E') égal 4)
  Pour 'Z' : (Code('Z') + décalage) modulo 26 = (25+3) % 26 = 28% 26 = 2. Donc 'Z' sera remplacé par 'C' (car Code('C') égal 2). Donc finalement la chaîne de caractère "B Z" devient après cryptage égale à "C E"

Questions :

1- Ecrire la classe abstraire **AbstractCryptage** ayant un attribut nommé *strategie* de type *String* et les méthodes suivantes :
- Un seul constructeur avec un paramètre permettant d'initialiser son unique attribut.
- Implémentation d'une méthode *reverseString* qui permet d'inverser une chaîne de caractères et retourne le résultat.
- Méthode **abstraite** *transform* qui prend en paramètre une lettre majuscule et retourne son équivalent entier.

2- Ecrire une interface nommée **ICryptage** ayant une seule méthode abstraite nommée *encrypteText* qui prend une chaîne de caractères en paramètre et retourne sa version cryptée.

3- Ecrire la classe concrète **CryptageNaifMethode1** qui fournit une implémentation de l'interface **ICryptage** en utilisant la **méthode de cryptage 1**. Cette classe hérite également de la classe abstraire **AbstractCryptage** et possède un constructeur sans argument permettant d'initialiser l'attribut *strategie* avec la chaîne "Sratégie1" en appelant le constructeur de sa classe de base.

4- Ecrire la classe concrète **CryptageNaifMethode2** qui fournit une implémentation de l'interface **ICryptage** en utilisant **la méthode de cryptage 2**. Cette classe hérite de la classe abstraire **AbstractCryptage** et elle possède un attribut qui présente le décalage ainsi que deux constructeurs un sans argument et un autre ayant un seul argument, ils permettant de définir le décalage et d'initialiser l'attribut *strategie* avec la chaîne "Sratégie2" en appelant le constructeur de sa classe de base. Le constructeur sans argument initialise le décalage avec une valeur par défaut égale à 3.

5- Ecrire le code de la classe nommée **CryptageFactory** qui ne peut être instanciée qu'une seule fois et qui permet de construire les objets des classes de cryptage précédentes, ceci avec sa méthode *getCryptage* ayant un paramètre permettant de déterminer le type de cryptage souhaité. Ci-dessous un exemple de code utilisant cette classe :

```
ICryptage cryptage1 =
CryptageFactory.getInstance().getCryptage(CryptageFactory.METHODE_1);

ICryptage cryptage2 =
CryptageFactory.getInstance().getCryptage(CryptageFactory.METHODE_2);

// Cryptage avec la méthode 1

System.out.println(cryptage1.encrypteText("B Z"));

// Cryptage avec la méthode 2

System.out.println(cryptage2.encrypteText("B Z"));
```

6- Ecrire un programme console qui effectue les opérations suivantes :

- Lire une chaîne de caractères T au clavier
- Transformer tous les lettres de T en majuscule.
- Filtrer les caractères de T qui ne sont pas des espaces ou des lettres de l'alphabet.
- Demander à l'utilisateur de choisir une méthode de cryptage
- Crypter T (déjà filtrée) par la méthode choisie, puis afficher le résultat sur l'écran.

Deuxième Partie

On suppose que nous possédons un schéma en base de données contenant les tables ci-dessous :

- La table ***MotsTable*** (***id, texte, signification***) qui constitue un dictionnaire de la langue anglaise, elle contient des mots avec leur signification.
- La table ***CharFrequence*** ayant trois colonnes : un identifiant *idChar*, la colonne *lettre* de type char, la colonne *frequence* de type réel. Cette table contient les données du tableau 1.

Fréquences d'apparition des lettres			
Lettre	Fréquence	Lettre	Fréquence
A	*8.08 %*	*N*	*7.38 %*
B	*1.67 %*	*O*	*7.47 %*
C	*3.18 %*	*P*	*1.91 %*
D	*3.99 %*	*Q*	*0.09 %*
E	*12.56 %*	*R*	*6.42 %*
F	*2.17 %*	*S*	*6.59 %*
G	*1.80 %*	*T*	*9.15 %*
H	*5.27 %*	*U*	*2.79 %*
I	*7.24 %*	*V*	*1.00 %*
J	*0.14 %*	*W*	*1.89 %*
K	*0.63 %*	*X*	*0.21 %*
L	*4.04 %*	*Y*	*1.65 %*
M	*2.60 %*	*Z*	*0.07*

Tableau 1

On suppose qu'on dispose d'une classe *DBConnexion* ayant une méthode *getConnection() throws DataBaseException* permettant de créer une connexion à la base de données et la retourner sous forme d'un objet Connection et en cas d'échec elle déclenche une exception *DataBaseException* qui hérite de Exception. *(Il n'est pas demandé d'écrire les classes DBConnexion et DataBaseException)*

Questions :

7- Ecrire une classe **DecalageIntrouvableException** qui définit une exception **explicite** ayant un constructeur permettant de définir un message d'erreur.

8- Ecrire une classe nommée **Mot** ayant les attributs *id*, *texte* et *signification* avec un constructeur permettant d'initialiser ces attributs. En considérant que deux mots sont égaux si et seulement si ils ont la même signification doter cette classe d'une méthode *equals.*

9- Ecrire une classe nommée **CharFrequence** ayant les attributs *idChar*, *lettre* et *frequence* avec un constructeur permettant d'initialiser ces attributs.

10- Ecrire une classe **Dictionnaire** ayant un attribut de type collection permettant de stocker une liste d'objets de type **Mot,** ainsi qu'une méthode permettant d'ajouter des mots à la collection.

11- Ecrire une classe **JdbcDictionnaireDaoImpl** ayant une méthode *public Dictionnaire getDictionnaireFromDataBase ()* qui permet de charger le dictionnaire de la base de données et le retourner sous forme d'un objet de type **Dictionnaire**.

Troisième Partie

Dans cette partie, l'objectif est de réaliser un simple programme permettant de casser la méthode de cryptage 2 par la technique décrite ci-dessous.

Dans une langue quelconque les lettres ont une fréquence d'apparition standard dans un texte. Par exemple la lettre A apparait dans un texte en anglais avec une fréquence de 8.08% et B avec une fréquence de 1.67 % (cf. tableau 1).

Soit **T** un texte supposé assez grand et **T'** sa version cryptée avec la méthode 2 (*bien entendu **T'** ne contient que des lettres majuscules et espaces*), nous voulons retrouver ***T*** en partant de **T'**, pour ce faire on commence par inverser ***T'*** on obtient ***T''*** puis on prend la première lettre de ***T''*** et on calcule sa fréquence d'apparition dans ***T''*** soit *f* la valeur de cette fréquence (*i.e.* ***f*** *=nombre de fois la lettre est apparue dans **T''** divisé sur le nombre total de caractères de **T''***). On sélectionne du tableau 1 (table ***CharFrequence)*** la lettre ayant la fréquence la plus proche de ***f*** *(s'il y en a plusieurs lettres avec des fréquences équidistantes de **f** on sélectionne la plus petite dans l'alphabet)* cette lettre est considéré comme la lettre d'origine avant cryptage, on déduit ainsi le décalage et on décode les autres caractères. On nomme ***T'''*** la chaîne obtenue après décodage. Maintenant nous voulons consolider le résultat obtenu en cherchant dans le dictionnaire les mots de ***T'''*** (*les mots sont obtenu en segmentant le texte par rapport au séparateur espace*), si plus que 70% des mots de ***T'''*** existent dans le dictionnaire (table ***MotsTable)*** alors on retourne un objet qui encapsule

T''' et le pourcentage des mots de ***T'''*** trouvés dans le dictionnaire. Sinon on déclenche l'exception **DecalageIntrouvableException**.

12- Ecrire la classe **EncrypteAttack** qui possède une méthode nommée *decrypteString* qui implémente algorithme ci-dessous qui permet de décoder une chaîne de caractères cryptée par la méthode 2. Cette classe utilise les classes déjà implémentées dans les parties précédentes.

13- Nous souhaitions faire une abstraction du type de stockage utilisé, qui peut être de type base de données ou de type XML. Le code de la classe **EncrypteAttack** ne doit pas dépendre de ce type de stockage utilisé. Nous proposons d'utiliser le pattern **Abstract Factory** pour répondre à cette problématique. On suppose que nous disposons déjà des classes :

- **JdbcFrequenceCharDaoImpl** ayant une méthode *public List<CharFrequence> getCharFrequencesFromDataBase ()* qui permet de charger de la base de données les lettres et leurs fréquences d'apparition sous forme d'une collection d'objets. Cette classe a la même conception que **JdbcDictionnaireDaoImpl**.
- **XmlFrequenceCharDaoImpl** ayant les mêmes fonctionnalités que **JdbcFrequenceCharDaoImpl** mais qui utilisent l'XML comme support de stockage.
- **XmlDictionnaireDaoImpl** ayant les mêmes fonctionnalités que **JdbcDictionnaireDaoImpl** mais qui utilisent l'XML comme support de stockage.
- **ConfReader** ayant la méthode *static String readConf()* permettant de lire un fichier de configuration qui fixe le type de stockage à utiliser par l'application, cette méthode retourne le type de stockage à utiliser sous forme d'une chaîne de caractère égale à "XML" ou "BD".

Donner les modifications nécessaires à faire dans les classes précédemment implémentées pour s'adapter à la nouvelle conception et donner le code de toutes les classes et interfaces permettant d'implémenter la solution.

Exercice 2 : Programmation réseau et Threads

L'objectif de cet exercice est l'écriture d'un système de chat multi-utilisateurs qui se présente sous forme d'un salon de discussion *chat room* et qui fonctionne sur un réseau local. Ce système de chat est composé de deux programmes : **Programme Client** et **Programme Serveur**.

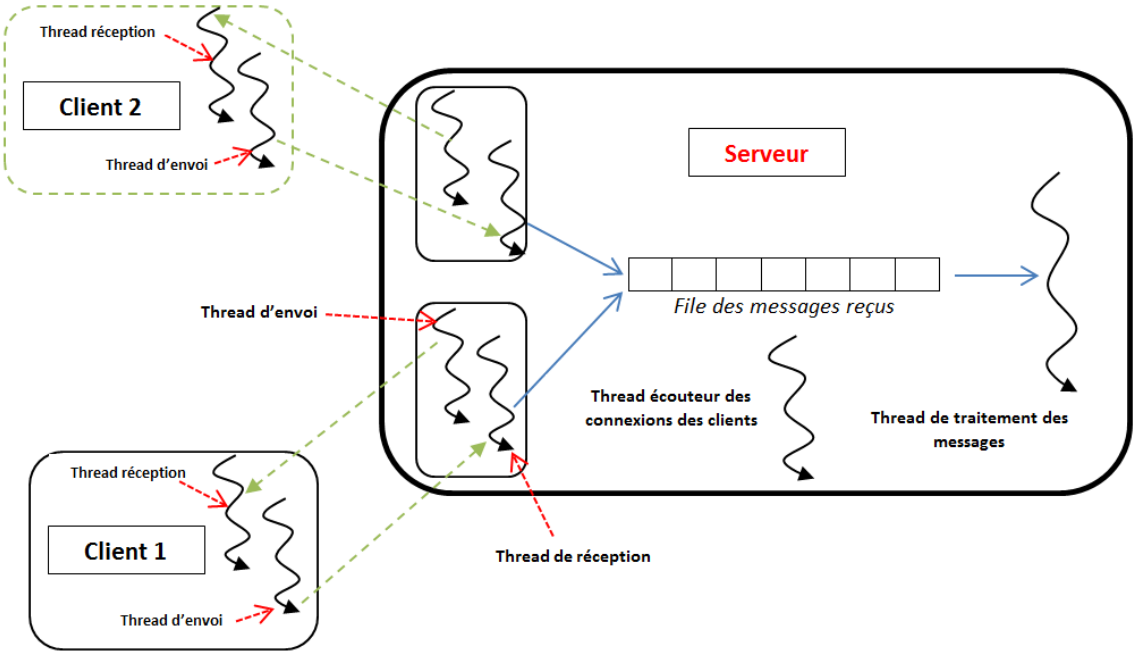

Figure 1

Le programme client permet d'envoyer les messages au serveur qui s'occupe de les dispatchers sur tous les utilisateurs connectés au serveur. Les opérations d'envoi et de réception des messages au niveau du client ne doivent pas bloquer l'exécution de ses autres fonctionnalités, ainsi, il faut confier ces tâches à deux threads séparés (thread pour l'envoi et un autre pour la réception des messages). Le client permet également d'envoyer des fichiers sans bloquer les autres tâches en cours d'exécution.

Le serveur permet de centraliser la communication entre les nœuds qui utilisent le programme Client. Il s'occupe de gérer les connexions des clients et leurs messages, à chaque fois que le serveur reçoit un message d'un client il l'ajoute à une file d'attente des messages reçus, un autre thread s'occupe de dispatcher les messages de cette file sur tous les utilisateurs connectés sur le serveur.

La figure 1 donne une vue générale sur la conception de cette application, vous êtes donc invité à analyser et compléter cette conception puis l'implémenter en Java.

Exercice 3: Application de gestion des étudiants avec interface graphique et base de données

Il s'agit de développer une application à interface graphique ayant pour objectif la gestion des étudiants. Les fonctionnalités à implémenter sont :

- Rechercher un étudiant
- Modifier les informations d'un étudiant
- Afficher la liste des étudiants d'une classe

Les interfaces doivent être développées conformément aux figures ci-dessous.

L’architecture de l’appli cation doit respecter le modèle en couche (Couche IHM / Couche BLL / Couche DAO)

L’application utilise Hibernate pour gérer l’accès aux données.

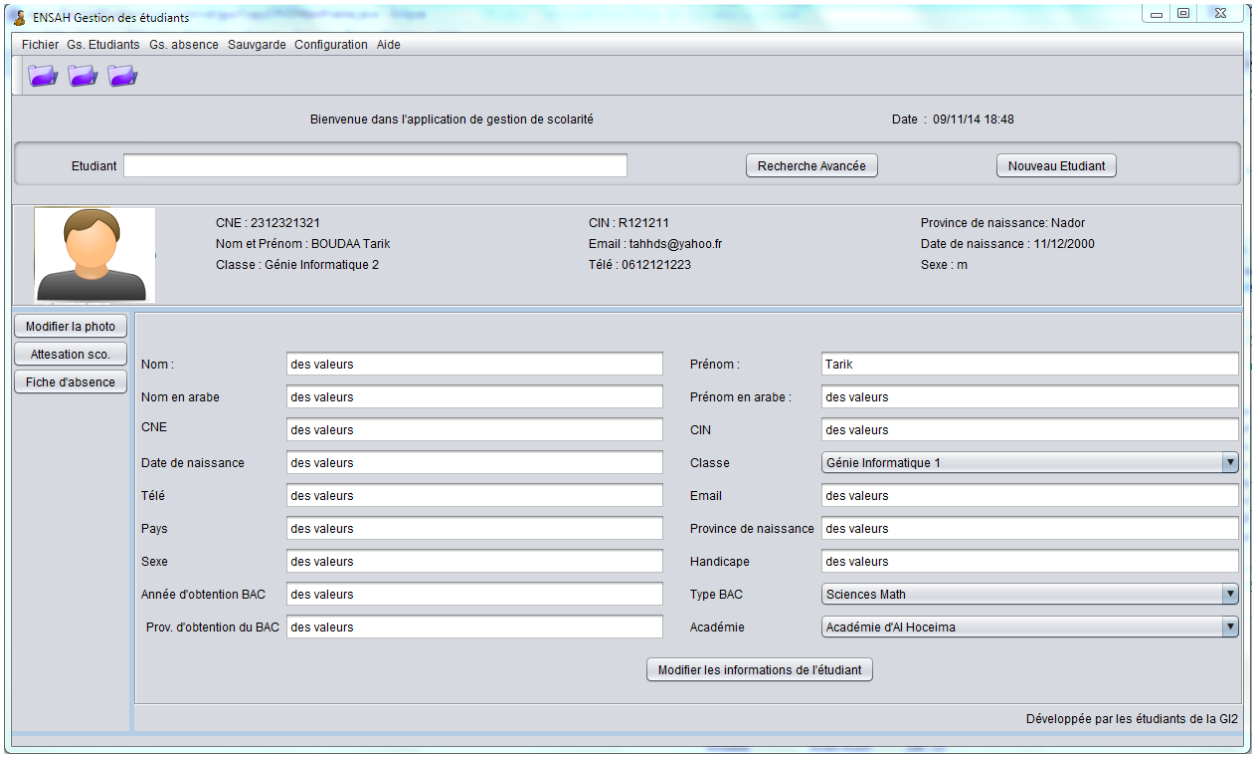

Figure 2 : Interface de recherche et de modification d'un étudiant

L’interface décrite par la figure 1 s’affiche soit via le menu « Gs. Etudiants » ou le raccourci (alt+G) ou depuis la barre d’outils.

Pour la recherche simple il suffit d’entrer le nom de l’étudiant puis cliquer sur le bouton « Entrer »

La recherche multicritères s’effectue à l’aide d’une boite de dialogue qui s’affiche via le bouton « recherche avancée ».

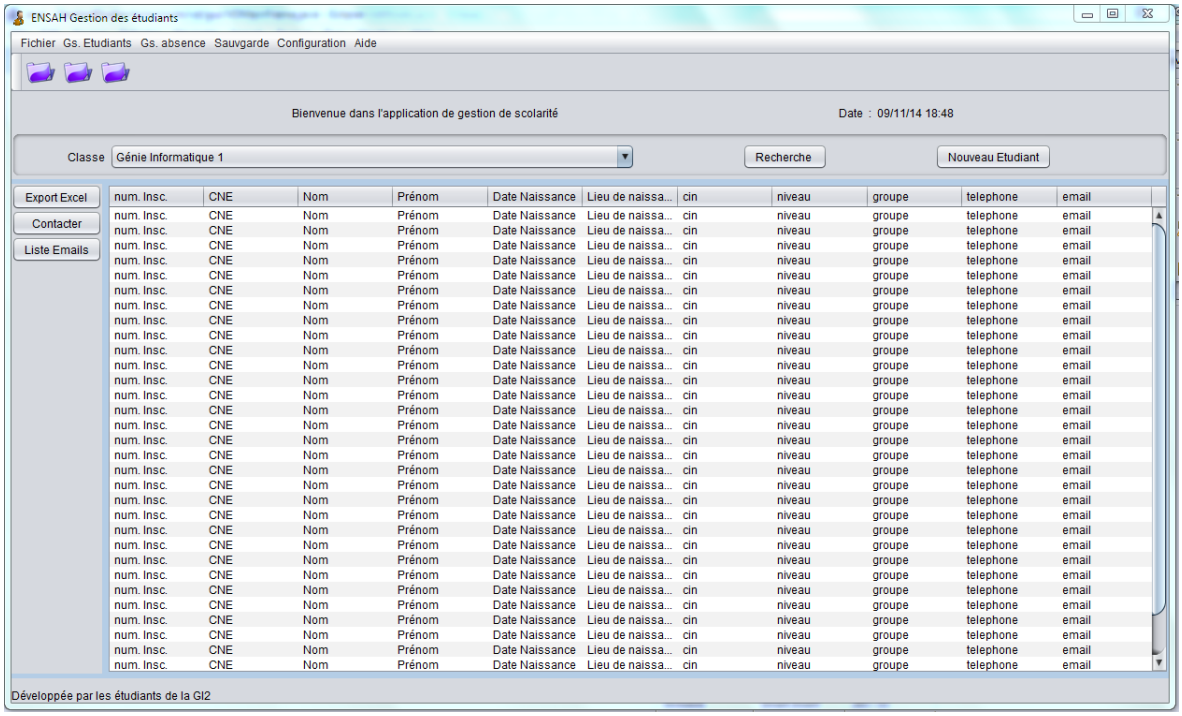

Figure 3 : Interface permettant l'affichage d'une classe

L'interface décrite par la figure 2 s'affiche soit via le menu « Gs. Etudiants » ou le raccourci (alt+L)