Université du Québec à Chicoutimi
Département d'informatique et de mathématique
8INF957 – Programmation objet avancée : TP1

Professeur : Hamid Mcheick	Groupe:individuel (1 étudiant au maximum)
Session : Hiver2021	Date de distribution : 11 janvier 2021
Pondération : 15 points	Date de remise : 08 février 2021

Objectifs

Le but de ce projet est de familiariser les étudiants avec les concepts suivants :

- Principes SOLID
- Paradigme objet : abstraction, héritage, modularité
- Paquetage d'introspection et de réflexion (« reflection package »)
- Chargement dynamique des classes
- Instancier des classes
- Invoquer des fonctions dynamiquement
- Programmation de socket
- Sérialisation
- Divers mécanismes avancés de Java : généricité, lambda, entres-sorties nio, etc.

Vous devez réaliser un SEUL choix parmi les deux donnés ici-bas:

Choix 1 :

Considérons les principes SOLID :

- Expliquez en détail les cinq principes avancés de logiciels SOLID
- Énumérez deux avantages de chaque principe
- Donnez une implémentation complète de chaque principe.

Considérons la réflexivité :

- Énumérez deux avantages de la réflexivité
- Donnez un exemple complet (code) de la réflexion

Langages de programmation utilisés : Java, Python, Go, C/C++, JavaScript, etc.

Choix 2 :

Structure générale du programme

Il s'agit de développer un programme client-serveur illustré dans la figure 1. La partie client ira lire dans un fichier de données un ensemble de commandes que le client devra faire exécuter par le serveur. Le client et le serveur communiquent à l'aide de sockets, RMI, etc. La séquence de commandes commence par :

- Des demandes de compilation d'un fichier source (« UneClass.java ») présentes sur le serveur
- Le chargement des classes ainsi compilées

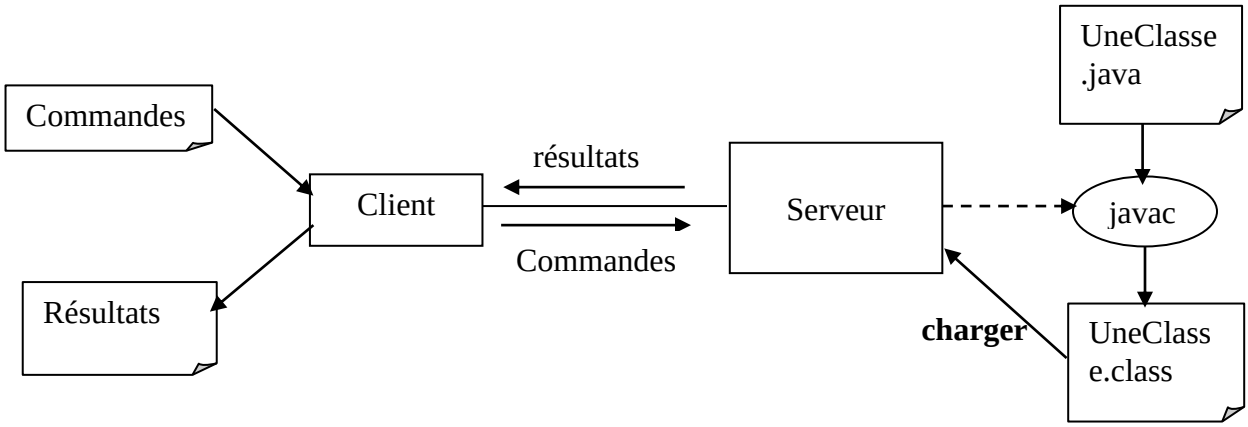

Figure 1. Structure du programme.

Par la suite, le programme client pourra demander au serveur de créer des objets de ces classes là, pour ensuite exécuter des opérations dessus. Le programme client imprime les résultats de l'exécution des commandes envoyées au serveur dans un fichier de sortie local (client).

Données du programme

Le programme client saisit, à partir d'un fichier, les commandes à envoyer au serveur. Les commandes possibles sont :

1) Compilation d'un fichier source du côté du serveur : le format est **compilation#***chemin_relatif_du_fichier_source_1*, *chemin_relatif_du_fichier_source_2,…#chemin_relatif_des_fichiers_class* Par exemple : `compilation#./src/ca/uqac/8inf853/Cours.java, ./src/ca/uqac/8inf853/Etudiant.java#./classes`

2) Chargement d'une classe. La donnée correspondante est le nom (qualifié) de la classe. Le format est **chargement**#*nom-qualifié-de-classe*. Par exemple,
`chargement#ca.uqac.8inf853.Cours`
3) Création d'une instance d'une classe. Les données sont : a) nom de la classe, et b) identificateur. Le format est
creation#*nom_classe*#*identificateur*.
Par exemple
`creation#ca.uqac.8inf853.Cours#c1234`
ou
`creation#ca.uqac.8inf853.Etudiant#marc`
4) Lecture d'un attribut d'un objet. Les données sont : a) identification de l'objet, et b) nom de l'attribut. Le format est **lecture**#*identificateur*#*nom_attribut*. Par exemple
`lecture#c1234#titre`
ou
`lecture#marc#prenom`
Si l'attribut en question n'est pas public, le serveur devra essayer d'appeler une fonction portant le nom get*nom_attribut_premiere_lettre_majuscule*. Par exemple, si `prenom` ci-haut est **`private`**, le serveur essayera d'appeler la fonction `getPrenom()` sur l'objet appelé « marc »
5) Écriture d'un attribut d'un objet. Les données sont : a) identificateur de l'objet, b) nom de l'attribut, et c) valeur à écrire. Le format est **ecriture**#*identificateur*#*nom_attribut*#*valeur*. Par exemple
`ecriture#c1234#titre#Architecture des applications`
`ecriture#marc#age#23`
Si l'attribut en question n'est pas public, le serveur devra essayer d'appeler la fonction set*nom_attribut_premiere_lettre_majuscule*(*type_attribut*) en lui passant l'argument comme paramètre. Par exemple, si `age` est **`private`**, le serveur essaiera d'appeler `setAge(int)` sur l'objet appelé « marc »
6) Appel d'une fonction. On doit fournir, a) identificateur de l'objet, b) nom de la fonction, et c) liste des paramètres. Le format est
fonction#*identificateur*#*nom_fonction*#*type1*:*val1,type2:val2,…*. Par exemple
`fonction#c1234#setNomProfesseur#java.lang.String:Labonté`
Ou bien
`fonction#marc#setMoyenne#float:3.74`

Lorsque l'argument de la fonction est un autre des objets crées par ce programme, alors l'argument est présenté comme suit : *type :***ID(***identificateur***)**. Par exemple,
`fonction#c1234#ajouteEtudiant#ca.uqac.8inf853.Etudiant:ID(marc)`

Classes à implémenter

Vous devez implémenter au moins les trois classes suivantes :

- Classe **Commande** : cette classe implémente l'interface **Serializable**. Elle est utilisée pour emmagasiner la description d'une commande, selon le format spécifié ci-haut. Ce sont des

instances de cette classe qu'on sérialisera et qu'on enverra à travers les sockets ou RMI. Prévoir un champ pour emmagasiner le résultat

- Classe **ApplicationClient** : cette classe gère la partie client. Elle devra implémenter les méthodes suivantes (au moins) :

```
public class ApplicationClient {

    /**
    * prend le fichier contenant la liste des commandes, et le charge dans une
    * variable du type Commande qui est retournée
    */
    public Commande saisisCommande(BufferedReader fichier) {…}

    /**
    * initialise : ouvre les différents fichiers de lecture et écriture
    */
    public void initialise(String fichCommandes, String fichSortie) {…}

    /**
    * prend une Commande dûment formatée, et la fait exécuter par le serveur. Le résultat de
    * l'exécution est retournée. Si la commande ne retourne pas de résultat, on retourne null.
    * Chaque appel doit ouvrir une connexion, exécuter, et fermer la connexion. Si vous le
    * souhaitez, vous pourriez écrire six fonctions spécialisées, une par type de commande
    * décrit plus haut, qui seront appelées par  traiteCommande(Commande uneCommande)
    */
    public Object traiteCommande(Commande uneCommande) {…}


    /**
    * cette méthode vous sera fournie plus tard. Elle indiquera la séquence d'étapes à exécuter
    * pour le test. Elle fera des appels successifs à saisisCommande(BufferedReader fichier) et
    * traiteCommande(Commande uneCommande).
    */
    public void scenario() {…}

    /**
    * programme principal. Prend 4 arguments: 1) "hostname" du serveur, 2) numéro de port,
    * 3) nom fichier commandes, et 4) nom fichier sortie. Cette méthode doit créer une
    * instance de la classe ApplicationClient, l'initialiser, puis exécuter le scénario
    */
    public static void main(String[] args) {…}
}
```

- La classe ApplicationServeur devra implanter au moins les méthodes suivantes:

```
public class ApplicationServeur {
    /**
    * prend le numéro de port, crée un SocketServer sur le port
```

```
*/
public ApplicationServeur (int port) {…}

/**
* Se met en attente de connexions des clients. Suite aux connexions, elle lit
* ce qui est envoyé à travers la Socket, recrée l'objet Commande envoyé par
* le client, et appellera traiterCommande(Commande uneCommande)
*/
public void aVosOrdres() {…}

/**
* prend uneCommande dument formattée, et la traite. Dépendant du type de commande,
* elle appelle la méthode spécialisée
*/
public void traiteCommande(Commande uneCommande) {…}

/**
* traiterLecture : traite la lecture d'un attribut. Renvoies le résultat par le
* socket
*/
public void traiterLecture(Object pointeurObjet, String attribut) {…}

/**
* traiterEcriture : traite l'écriture d'un attribut. Confirmes au client que l'écriture
* s'est faite correctement.
*/
public void traiterEcriture(Object pointeurObjet, String attribut, Object valeur) {…}

/**
* traiterCreation : traite la création d'un objet. Confirme au client que la création
* s'est faite correctement.
*/
public void traiterCreation(Class classeDeLobjet, String identificateur) {…}

/**
* traiterChargement : traite le chargement d'une classe. Confirmes au client que la création
* s'est faite correctement.
*/
public void traiterChargement(String nomQualifie) {…}

/**
* traiterCompilation : traite la compilation d'un fichier source java. Confirme au client
* que la compilation s'est faite correctement. Le fichier source est donné par son chemin
* relatif par rapport au chemin des fichiers sources.
*/
public void traiterCompilation(String cheminRelatifFichierSource) {…}
```

```
    /**
    * traiterAppel : traite l'appel d'une méthode, en prenant comme argument l'objet
    * sur lequel on effectue l'appel, le nom de la fonction à appeler, un tableau de nom de
    * types des arguments, et un tableau d'arguments pour la fonction. Le résultat de la
    * fonction est renvoyé par le serveur au client (ou le message que tout s'est bien
    * passé)
    /**
    public void traiterAppel(Object pointeurObjet, String nomFonction, String[] types,
                                                                    Object[] valeurs) {…}

    /**
    * programme principal. Prend 4 arguments: 1) numéro de port, 2) répertoire source, 3)
    * répertoire classes, et 4) nom du fichier de traces (sortie)
    * Cette méthode doit créer une instance de la classe ApplicationServeur, l'initialiser
    * puis appeler aVosOrdres sur cet objet
    */
    public static void main(String[] args) {…}
}
```

Livrables

- Copie électronique du tout, sur USB durant le cours : code source documenté, pour les classes, pour les fichiers batch (compilation/déploiement, exécution client, exécution serveur), et des fichiers d'entrées et de sortie. Vous allez compiler votre code et l'exécuter à partir des batch files (ou Eclipse, Netbeans, …) sur une machine du labo, sur ta machine ou sur ma machine.
- Quelques pages d'explication sur la façon dont vous avez implémenté les différentes classes et méthodes.

Le barème est le suivant :

- Tout fonctionne (compilation, déploiement, exécution) : 90 points sur 100
- Documentation (vos explications comment vous réalisez cette application + documentation interne au code surtout des classes et de méthodes) 10 points sur 100

Données de test

- Fichier Etudiant.java
- Fichier Cours.java
- Fichier commandes.txt
- Fichier scenario.txt (contient le code de la méthode scenario)