Projet de synthèse L3 IFA/CDA
Deuxième partie
28 mars – 7 avril 2023
ROBI-client et ROBI-serveur
Yvon Autret

Version du 27 mars 2023

Dans la première partie du projet, on a vu comment interpréter du code ROBI. Dans cette seconde partie, on va réaliser une IHM (Interface Homme Machine) distante. Le programme ROBI pourra être saisi sur une machine, celle qui gère l'IHM, et exécuté sur une autre, celle qui exécute le code ROBI.

Cette seconde partie du projet peut être basée sur les exercices 3 ou 4 de la première partie du projet.

On aura deux programmes qui vont communiquer par socket :

- Robi-serveur (exécution du code ROBI)
- Robi-client (saisie du code ROBI, envoi du code saisi à Robo-serveur sur un socket, affichage du résultat, debug)

Fonctions à mettre en œuvre :

- envoi d'un programme ROBI à Robi-serveur (chaîne de caractères)
- envoi du mode d'exécution à Robi-serveur (en bloc ou en pas à pas)
- envoi d'une demande d'exécution à Robi-serveur (en bloc ou instruction par instruction)

Notation : une note correcte peut être obtenue pour cette seconde partie si les points 1 à 3 ont été faits. Les points 4 et 5 sont indépendants, on peut traiter l'un ou l'autre en premier.

1. Conception de Robi-serveur

- Partir de la solution de l'exercice 3 ou de l'exercice 4
- Intégrer l'interpréteur de code ROBI dans un serveur socket
  - le serveur socket est connu par son adresse IP (éventuellement localhost) et écoute sur un port (par exemple 8000)
  - le serveur socket reçoit les commandes à exécuter sous la forme de lignes de texte, les exécute et retourne le résultat
- Robi-serveur ne gère qu'un seul client à la fois

Remarques

- Un élément du résultat de l'exécution de code ROBI est une image. Pour renvoyer l'image résultat, on peut utiliser la méthode "createScreenCapture" de la classe "java.awt.Robot" pour faire une copie d'écran. L'image obtenue peut être convertie en texte base64 puis envoyée sur le socket et affichée par Robi-client.
- Un autre élément important de la réponse si on est en mode pas à pas, c'est l'instruction ROBI qui vient d'être exécutée. La réponse envoyée à Robi-client doit contenir le texte du code exécuté et le résultat de l'exécution sous forme d'image.

2. Conception de Robi-client

- Utiliser Java FX ou Swing pour l'IHM
- Prévoir une zone pour la saisie du code ROBI, le texte doit pouvoir être saisi manuellement ou chargé à partir d'un fichier
- Prévoir une zone pour l'affichage du résultat de l'exécution (image produite et texte du code ROBI exécuté)
- Un premier bouton permet d'envoyer le code ROBI à Robi-serveur.
- Un second bouton permet de sélectionner le mode d'exécution (en bloc ou en pas à pas)
- Un troisième bouton permet de lancer l'exécution

3. Echanges entre Robi-client et Robi-serveur

- Robi-client et Robi-serveur doivent communiquer en s'échangeant des objets. Robi-client va créer un objet contenant un nom de commande et le cas échéant le code ROBI à exécuter. Cet objet sera converti en texte JSON et envoyé sur le socket sous la forme d'une ligne de texte.
- De la même façon, quand Robi-serveur aura traité la commande, un objet contenant le texte du code ROBI exécuté et le résultat de l'exécutiion sous forme d'image, sera envoyé en réponse à Robi-client.
- Définir la nature des objets qui vont être envoyés. On pourra s'inspirer des classes DataCS et DataSC données ci-dessous.
  - L'attribut "cmd" de DataCS est un code qui indique la commande à exécuter (envoi de code ROBI, changement du mode d'exécution, demande d'exécution, …).
  - L'attribut "txt" de DataCS contient null s'il n'y a pas de code ROBI à envoyer, ou sinon le code ROBI à envoyer.
  - L'attribut "im" de DataSC contient une image codée en texte base64.

Indications

- Pour convertir un objet Java en texte JSON et inversement, **voir sur Moodle exemple_json.zip**

```
// ces classes sont données à titre d'exemple
// elle peuvent être modifiées

public class DataCS { // CS : client vers serveur
   String cmd = null;
   String txt = null;       // utilisé s'il y a du
code ROBI à envoyer

   // ajouter les accesseurs (méthodes get et set)
   // sous Eclipse clic droit puis
   // "Source" et "Generate Getters and Setters"
}

public class DataSC { // SC : serveur vers client
   String cmd = null;
   String txt = null;       // texte du code ROBI
exécuté
   String im = null;   // résultat de l'exécution sous
forme d'image

   // ajouter les accesseurs (méthodes get et set)
   // sous Eclipse clic droit puis
   // "Source" et "Generate Getters and Setters"
}
```

4. Affichage de l'environnement et des SNode

Quand on envoie du code ROBI à Robi-serveur, on appelle aussi souvent que nécessaire la méthode "compute" de "Interpreter" avec comme paramètre un environnement et un objet SNode.

On demande de renvoyer dans la réponse l'environnement et l'objet SNode pour les afficher dans Robi-client (effectuer les modifications nécessaires dans Robi-serveur et Robi-client). Ceci suppose qu'il faut ajouter dans Robi-client une zone pour afficher l'environnement et une zone pour afficher le SNode.

Remarque : les objets Environment et SNode risquent de ne pas pouvoir être convertis en JSON parce qu'ils peuvent contenir des liens circulaires. On peut bloquer la conversion d'un attribut en JSON en ajoutant juste avant l'attribut concerné l'annotation `@JsonIgnore`. Cela risque ne pas être suffisant et on pourra envisager de reconstruire les objets concernés pour qu'ils puissent être convertis en JSON, envoyés à Robi-client, puis lus sous formes d'objets par Robi-client, et affichés sous forme d'arbres.

5. Renvoi de commandes graphiques à la place d'une image

Envoyer directement une image comme résultat d'exécution de code ROBI n'est pas une bonne solution parce qu'il faut gérer un environnement graphique sur le serveur.

On demande de modifier Robi-serveur pour renvoyer comme résultat d'exécution de code ROBI, non pas une image, mais des objets qui permettront de produire l'image. Pour une exécution de code ROBI, on pourra envoyer comme réponse une série d'objets.

On peut renvoyer des chaînes. Par exemple la chaîne "rect 10 10 100 100" peut correspondre à l'affichage d'un rectangle.

L'inconvénient c'est qu'il va falloir décoder la chaîne. Une meilleure solution consisterait à définir une série d'objets graphiques pouvant être envoyés à Robi-client.

Par exemple, on peut définir la classe suivante :

```
public class Graph {
    String cmd;
    int x;
    int y;
    int dx;
    int dy;
}
```

Pour produire un rectangle, cmd serait initialisé à "drawRect". Les paramètres du rectangle seraient définis par x, y, dx et dy.

On peut aussi définir Graph de manière plus universelle de la façon suivante :

```
public class Graph {
    String cmd = null;              // nom de la commande awt graphics
    int [ ] entiers = null;         // paramètres de type int
    String [ ] chaines = null;      // paramètres de type String
    int [ ] couleurs = null;        // couleurs RGB (3 entiers)
}
```

On regoupe les paramètres de même type dans des tableaux.

Pour afficher un rectangle, cmd serait initialisé à "drawRect". Les paramètres du rectangle seraient placés dans "entiers", la couleur dans "couleurs" et l'attribut "chaines" ne serait pas utilisé.

Pour afficher du texte, cmd serait initialisé à "drawString". La position d'affichage serait définie dans "entiers", le texte à écrire dans "chaines" et la couleur dans "couleurs".

Pour utiliser cette classe, voir sur Moodle l'**exemple Swing client_robi_swing.zip**. Le bouton "Rect 2" construit des objets Graph servant à afficher des rectangles et des textes. On utilise la réflexion Java (package java.lang.reflect) pour exécuter automatiquement la méthode awt correspondante.