Systèmes d'acquisition - Partie "Java"

S. Reynal

20 septembre 2017

Cette série de trois séances de TP Java a pour objectif de vous initier au développement rigoureux d'applications, et en particulier d'IHM, dans le domaine de l'acquisition et de l'instrumentation. Seront abordés :

- la modularité, autour du paradigme model-view-controller
- la réalisation d'IHM avec, soit Swing, soit le framework JavaFX (nouveauté 2017, cf. document introductif disponible sur mon site)
- les sockets et la communication en java sur un réseau
- la parallélisation avec les Threads

Il y a beaucoup de choses à réaliser : de fait, si vous êtes à cours de temps, vous pouvez sauter certaines parties et choisir celle pour laquelle vous avez le plus de compétences à acquérir (ces TP ne sont pas notés, profitez-en !).

1 Cahier des charges

Il vous est proposé de développer un oscilloscope virtuel en java, qui permet d'afficher des signaux provenant soit d'une carte d'aquisition, soit d'un serveur connecté à des capteurs distants (aka ZigBee par exemple). Ce cahier des charges vous donne quelques contraintes de développement de votre application. Celles-ci étant volontairement succinctes, il s'avère que ce qui est imposé dans la suite doit impérativement être respecté, et qu'a contrario tout ce qui n'est pas imposé constitue votre marge de liberté et de créativité.

Les classes seront organisées en trois packages :

- **model** : classes dédiées à la représentation et aux stockage des signaux ayant précédemment fait l'object d'une acquisition, soit directement depuis une carte NI, soit depuis un fichier stocké sur le disque dur, soit depuis un serveur ;
- **view** : classes dédiées à la visualisation des signaux ; ici on trouvera les éléments habituels d'un panneau avant d'oscilloscope : écran, grilles, axes, boutons de réglage de calibre, etc.

— **controller** : classes dédiées aux évènements de contrôle du panneau avant de l'oscilloscope, c'esà-dire gestion des évènements souris et clavier.

En outre, on créera une classe principale Oscilloscope : elle contient un constructeur qui est chargé d'initialiser toutes les autres classes de l'application (model, view et controller), et une méthode main() qui ne doit faire RIEN D'AUTRE que d'appeler ce constructeur. Attention, si vous utilisez JavaFX, c'est la méthode launch() qui doit se trouver dans main(), et la classe principale doit hériter de `javafx.application.Application`.

2 package "model"

Ce package contient les classes suivantes :

2.1 Signal

— Représente un signal composé d'échantillons de type "double".
— Fields : tableau d'échantillon, fréquence d'échantillonage, nombre de bits de quantification, unité physique.
— Constructeurs : un signal peut être initialisé sous forme de signal prédéfini (prévoir une **enum** contenant les constantes NOISE, SINE, SQUARE) ou à partir d'un fichier sur le disque dur au format matlab binaire
— méthodes : outre les getters et setters habituel, la classe signal permet d'ajouter ou d'enlever des échantillons à la volée.

2.2 SignalTools

— Classe offrant des outils de traitement de signal
— Est initialisé à partir d'un Signal fournit en argument
— méthodes : à votre convenance, mais on peut imaginer : calculs statistiques, transformée de Fourier, etc. Attention, le but est d'implémenter une ou deux méthodes à titre pédagogique, par de réaliser une bibliothèque de qualité industrielle !

3 package "view"

Ce package contient une classe principale "FrontPane" qui représente le panneau avant de l'oscillocope. Elle hérite de JFrame et est un container pour les classes Screen (l'écran) et ControlPane (les boutons) ci-dessous. Attention à bien

spécifier le Layout du container pour que les éléments Screen et ControlPane soient correctement agencés (Screen à gauche et ControlPane à droit par exemple).

Additif 2017 JavaFX : cette partie "view" reste valide dans ses grandes lignes pour une implémentation avec JavaFX. Il s'agit toujours de bien séparer les différents éléments de l'interface graphique, entre Screen, Grid, ControlPane, etc. Toutefois, il n'est plus nécessaire de passer par Graphics2D pour dessiner dans une fenêtre : il est suffisant d'ajouter des primitives graphiques sous forme de nodes "Shape" (lignes, rectangles, texte, ...) à l'arbre des composants de JavaFX. C'est donc beaucoup plus simple et intuitif... On pourra également utiliser avec profit l'objet `javafx.scene.chart.Chart` qui sait parfaitement dessiner des signaux sur un écran...

3.1 Screen

- Représente un écran d'oscilloscope permettant d'afficher jusqu'à quatre signaux avec des couleurs différentes ;
- Hérite de JPanel ;
- Pour dessiner dans un JPanel, on utilisera impérativement la classe awt.Graphics2D (coordonnées de pixels codées en float et non plus en int ; possibilité de zoomer ; formes à dessiner codées sous forme de awt.geom.Shape, etc.) et on surchargera la méthode paintComponent de cette classe ; cette dernière méthode délèguera à la classe Grid le soin de se "dessiner" elle-même
- Une méthode permettra d'ajouter de nouveaux signaux ou d'enlever des anciens
- JavaFX : utiliser l'objet Chart.

3.2 Grid

- Représente une grille graduée ; c'est ce qu'on appelle un helper class (indice : est-elle public ou package du coup ?) car elle sert uniquement à Screen
- Fields : le pas de la grille (vertical et horizontal) ; la couleur de la grille ; la présence ou non de labels et de graduation
- Méthodes : une méthode paintComponent(Graphics2D g) permettra de dessiner la grille (délégation depuis Screen)
- JavaFX : utiliser l'objet Line et une directive de style CSS pour ajouter des flèches, des pointillés, etc.

3.3 ControlPane

— Représente le panneau de contrôle de l'oscilloscope (boutons, potentiomètres, etc)
— Réalisation sous forme de composents Swing (java.swing) : JButton, JTextField, JSpinner, etc
— Le traitement des évènements souris ou clavier reçus par cette classe est délégué aux classes du package "controller" ci-dessous

4 package "controller"

4.1 MouseHandler

— Il s'agit d'une classe gérant les évènements souris (aka awt.event.MouseListener et MouseMotionListener)
— Screen délègue à cette classe le soin de gérer ces évènements

4.2 KeyHandler

Même chose pour les évènements clavier (aka awt.event.KeyListener)

5 Threads

Si vous en êtes là, c'est bien, vous êtes brillants ! Alors venez me voir pour que je vous explique comment paralléliser votre code.

6 sub-package "model.network" et capteurs distants dans le cloud

Cette ultime partie permet de s'initier aux sockets. L'objectif est de simuler la réception par un serveur (la machine sur laquelle vous développez) de données (température, etc.) émises par plusieurs capteurs intelligents connectés au serveur via le réseau (ethernet ou WiFi). Ces données sont un autre moyen d'initialiser un objet Signal, donc.

— Commencer par implémenter la classe "Sensor" qui hérite de la classe "Thread". Cette classe se contente pour l'instant, dés que le thread est démarré, d'afficher régulièrement, chaque seconde, une donnée de mesure factice (un nombre aléatoire) à l'écran (via la console). Le code correspondant se trouvera dans la méthode "run()", et repose sur une boucle infinie

et l'appel de la méthode "sleep()" (attention à bien gérer les exceptions associées). Les threads seront lancés depuis la méthode "main()". L'affichage souhaité pour chaque Sensor est à titre indicatif du type : *"Capteur numéro 4 : température = 28C"*.Dans un second temps, on créera un objet Signal à partir de ces data.

— Implémenter maintenant la classe Serveur. Cette classe hérite de ServerSocket et est chargé d'attendre les connections émanant de clients (les capteurs). Elle repose sur l'appel de la méthode "accept()" de la classe ServerSocket, qui renvoie un Socket représentant la connection active. Une fois la connection activée, récupérer le InputStream associé à la connection via la méthode ad hoc dans Socket, lire une donnée (un double) depuis la connection, fermer la connection, et passer au client suivant. Pour l'instant, le serveur ne peut gérer qu'une connection active avec un client à la fois. On devra utiliser la classe DataInputStream (paquet java.io).

— Le code n'est véritablement fonctionnel que s'il existe des clients ! Ajouter à la classe Sensor dans la méthode run(), la connection au serveur (via la classe Socket). Une fois la connection acceptée par le serveur, récupérer le OutputStream associé (via la méthode getOutputStream de la classe Socket), et écrire un double correspondant à la température.

— Ajouter à la classe Serveur la possibilité de gérer plusieurs connection simultanément, via le multithreading. Pour cela, il faut insérer le code qui attend un client (la méthode accept()), à l'intérieur de la méthode run() d'un thread (ou implémenter la méthode run() de l'interface Runnable, autre approche), puis lancer un nouveau thread à chaque nouveau client. Ensuite, il suffit de laisser les threads tourner en boucle infinie et communiquer avec leur capteur attaché.

Remarque : vous pouvez travailler en équipe, un binôme développant la classe Sensor, un autre la classe Serveur. Ainsi les "faux" capteurs peuvent être disséminés sur plusieurs machines de la salle et émettre leur données pour le reste du groupe !