Le patron de conception Chaîne de responsabilité

Source « https://www.math.univ-paris13.fr/~chaussar/ »

Le pattern chaîne de responsabilités permet d'exécuter, à la chaîne, des fonctions comme le Décorateur. Cependant, ici, le but n'est pas d'ajouter des fonctionnalités mais de déléguer des responsabilités : si on ne peut pas traiter la demande, on la passe à son voisin, sinon, on traite la commande et c'est fini (tandis que Décorateur consiste à traiter une commande et passer au voisin).

Nous allons implémenter ce pattern pour gérer une liste de processeurs.

Exercice 1

Le code fourni contient des processeurs lents et des processeurs rapides qui, au travers de la fonction `runTask()` peuvent lancer le calcul de la décomposition en facteurs premiers d'un nombre. Si un calcul était déjà en cours sur le processeur, ce dernier attendra la fin du calcul avant de commencer un nouveau calcul. On peut tester si un processeur est occupé avec la fonction `isBusy`. Un objet de type `ResultPool` est créé dans le `main` et passé aux processeurs pour collecter les résultats de tous les calculs. Lors du calcul de la décomposition, on utilise un `Thread.sleep` pour émuler le traitement, plus ou moins long, de la tâche par le processeur.

Le code fourni doit permettre d'exécuter deux threads en parallèle pour calculer la décomposition en produit de facteurs premiers de 512 et 1000. Mais si vous exécutez le code, seule la première décomposition est exécutée puisqu'il n'y a qu'un seul processeur utilisé.

Exercice 2

Le but de ce TP est de créer plusieurs processeurs (trois rapides, deux lents) et de faire une boucle qui tire vingt nombres au hasard entre 1 et 100 000 afin de calculer leur décomposition en facteurs premiers. Évidemment, si on fait faire tout le travail par un seul et même processeur, on perdra du temps. Le code à développer doit gérer, de façon quasi transparente pour le `main`, la file des processeurs.

Vous devrez créer ces classes :

- La classe `QuickProcessorHandler` permettra de gérer les processeurs rapides (et contiendra donc un processeur `p` de ce type). Elle aura une variable de classe

`nextProcessor` (et un mutateur sur cette variable) permettant de connaître quel `ProcessorHandler` appeler si le processeur `p` est occupé. La fonction `handleRequest(int n)` devra soit calculer la décomposition en facteur premier de `n` (grâce à `p`), ou bien passer le relai à `nextProcessor`.

- La classe `SlowProcessorHandler` fera de même.
- La classe `ProcessorHandler`, qui sera abstraite, sera une classe dont héritent les deux précédentes. Elle contiendra la fonction (abstraite ?) `handlerequest`. De plus, ce sera elle en vérité qui contiendra la variable `nextProcessor` et son mutateur.

Le `main` crée plusieurs `ProcessorHandler`, les lie entre eux, et lance la requête de calcul au premier élément de la file qui transmettra la requête s'il est occupé. Si tous les processeurs sont occupés, on abandonne la requête (en affichant un message). Pensez à afficher un message à l'écran indiquant, pour chaque nombre, quel processeur va le traiter.

Votre fonction `handleRequest` de `ProcessorHandler` ne devrait pas être abstraite afin de justement implémenter un comportement par défaut (que faire par défaut si le processeur est occupé ?)…

Exercice 3

Comment faire pour que, si tous les processeurs sont occupés, l'on boucle de nouveau sur le premier processeur de la file pour tenter de faire traiter la requête.

La fonction `handledRequest()` pourrait retourner un booléen indiquant que le calcul est pris en charge par un processeur. À défaut, attendez un peu avec un `Thread.sleep(100)` avant de réaffecter le calcul à un processeur.

Pensez, quand vous passez la requête d'un processeur à l'autre, à faire un petit pour éviter que le `main` ne passe sont temps à boucler sur les processeurs.