# Programmation Orientée Objet

TP Collections – BatchMan, le Retour

Ensimag 2ème année

Un ordonnanceur de job (*batch scheduler* ou gestionnaire de ressources, comme OAR [`oar.imag.fr`] utilisé sur des architectures de type grille de calcul) est un système qui contrôle l'exécution sur des ressources partagées de jobs soumis par différents utilisateurs. Dans ce TP, nous allons créer notre propre ordonnanceur de jobs : BatchMan [1].

Voici maintenant la description du principe d'*ordonnancement par score* qui sera utilisé dans notre système :

- Chaque utilisateur (identifié par une chaine de caractères) soumet des jobs à exécuter. Un utilisateur possède un score porté par une classe `Score` (supposée fournie) qui implémente l'interface `Comparable<Score>` ;
- Chaque job est identifié par un objet de type `JobId`, et associé à un fichier exécutable et une durée maximale d'exécution (durée maximale entre le début de job et sa terminaison, si nécessaire tué par le système) ;
- L'ordonnancement choisit toujours l'utilisateur de score minimal ayant des jobs soumis non démarrés ; il prend le plus ancien de ses jobs, puis augmente le score de l'utilisateur et démarre l'exécution du job ;
- Quand on doit départager deux utilisateurs qui ont le même score, on choisira celui ou celle dont le nom arrive en premier dans l'ordre lexicographique, parce que c'est comme ça ;
- Bien entendu, l'ordonnanceur devra toujours privilégier un utilisateur qui a des jobs à exécuter par rapport à un utilisateur qui n'en a pas, indépendamment de leurs scores respectifs ;
- A partir de l'identificateur d'un job, un utilisateur peut aller voir son état (en attente, en cours, terminé), ou modifier certains attributs.

Un cas pratique est de l'ordre de $10^5$ utilisateurs et $10^6$ jobs [2]. Le temps de réponse à une requête (soumission d'un job, recherche du job à ordonnancer, accès aux attributs d'un job) est considéré critique.

Votre mission pour ce TP est de proposer et de coder une architecture qui respecte les spécifications données ci-dessus. Vous allez devoir entre autre faire les bons choix de collections Java et bien réfléchir aux relations entre les classes. Le choix des collections devra être justifié par le coût des différentes opérations. De vos choix judicieux dépend le bon déroulement des tests fournis (voir plus loin). Robin compte sur vous pour lui fournir un BatchMan digne de ce nom. Alors, à vos claviers et à vos (chauves-)souris !

1. D'aucuns pourraient s'interroger sur la raison pour laquelle ce nom a été choisi. Je mets ceux-là au défi de trouver un autre calembour plus malin que celui-là avec le nom « batch » à l'intérieur. Merci d'envoyer vos propositions à votre enseignant de POO.
2. ce qui incita un responsable politique célèbre à déclarer en 2018 que pour trouver un job, suffisait de traverser la rue.

# 1 Quelques précisions pratiques

Comme point de départ de votre travail, nous vous fournissons gracieusement quelques classes et squelettes de classes, sous la forme d'une archive téléchargeable depuis le site du cours. Toutes les classes que nous vous fournissons peuvent bien sûr être complétées et modifiées.

L'objectif final de votre TP sera d'obtenir une classe `scheduler.Scheduler` complète. En particulier, vous devrez travailler sur l'implantation des trois méthodes suivantes :

- `JobId addJob(String execFile, int dureeMax, String userId) ;` // retourne l'identifiant du job ajouté
- `Job getJob(JobId id) ;` // retourne un objet avec les infos sur le job
- `Job extractNextJobToSchedule() ;` // retourne le prochain job à exécuter, le supprime de l'ensemble des jobs en attente et met à jour le score de l'utilisateur.

## 1.1 Tester votre implémentation

Nous fournissons également trois classes de test utilisant `junit` :

- `SimpleTest.java` contient un petit nombre de tests fonctionnels de départ, que nous vous conseillons de valider avant tout chose. L'objectif plus ou moins caché de ces tests est de vérifier que vous avez bien compris les spécifications du TP. En particulier, lisez bien les commentaires de ce fichier : à défaut d'apprendre des trucs, ça vous fera peut-être rigoler ;
- `SchedulerTest.java` contient des tests fonctionnels de plus grande échelle, qui sont censés vérifier que votre ordonnanceur fonctionne comme attendu ;
- `StressTest.java` contient un test de performance. Il ajoute un grand nombre de jobs dans l'ordonnanceur, puis essaie de les extraire tous. Si vous avez utilisé des collections judicieuses, ce test ne devrait pas prendre énormément de temps. S'il s'exécute en plus de temps qu'il ne faut à l'Homme Chauve-Souris pour sauter dans la BatMobile et démarrer le moteur (au mépris des règles les plus élémentaires du code de la route et des limites antipollutions), alors c'est que vous avez un problème de choix de collections.

Le principe est d'effectuer systématiquement ces tests au « furet à mesure » (!) du développement. A la fin tout doit passer, et vous pourrez ensuite optimiser les performances.

## 1.2 Compilation et exécution du TP

### 1.2.1 IDE Intellij

Dans l'IDE Intellij IDEA, ça devrait passer facilement. Si JUnit n'est pas trouvé, suivez la résolution des problèmes proposés par l'IDE genre `@Test -> add JUnit to classpath`.

### 1.2.2 Compiler et exécuter directement depuis le terminal

En cas de soucis ou si vous n'utilisez pas d'IDE, nous fournissons aussi un moyen de compiler et lancer les tests directement en ligne de commandes.

Vous trouverez dans l'archive de départ un fichier `Makefile` permettant de compiler les fichiers source (commande `make`) et lancer les tests (commande `make tests`). Une archive `.jar` de `junit` est fournie en local pour éviter les soucis d'installation.

La jolie figure suivante vous donne un aperçu de résultat.


FIGURE 1 – Au début ce sera moins vert et vos tests seront tous `failed`, mais à la fin ce sera bô !