Programmation Orientée Objet
IG2I, LE2 – 2018-2019

TP4 – Ordonnancement

Diego Cattaruzza, Philippe Kubiak, Maxime Ogier

Contexte et objectif du TP

Dans ce TP, nous poursuivons notre travail afin de proposer un outil informatique qui puisse améliorer l'ordonnancement de la production de l'entreprise Masques&Confettis.

Pour rappel, l'entreprise nordiste Masques&Confettis doit faire face à une grosse demande de production de masques colorés, car le carnaval de Dunkerque se déroule prochainement. Cette année, la demande est vraiment exceptionnelle et les gestionnaires de la production ne sont pas certains d'avoir le temps de fabriquer tous les masques. Par ailleurs, dans les contrats conclus avec ses clients, l'entreprise Masques&Confettis s'engage à respecter une date limite de livraison et à payer une forte pénalité en cas de retard (proportionnelle au retard). Il est donc essentiel de respecter au mieux les délais imposés par les clients.

Pour améliorer son planning de production, Masques&Confettis a décidé de faire appel aux meilleurs étudiants de la région Nord-Pas de Calais pour concevoir des outils informatiques qui puissent fournir un planning de production capable de gérer des quantités de demandes importantes.

Dans le TP précédent, nous avons mis en place un modèle objet pour l'atelier de production. L'objectif de ce TP est maintenant de concevoir des algorithmes qui permettent de proposer des solutions de bonne qualité pour l'ordonnancement des tâches de production.

Ce TP permet ainsi d'illustrer les notions suivantes :

- la notion d'interface ;
- la notion de méthode statique ;
- la gestion de collections d'objets en Java ;
- l'interface **List** et ses implémentations (**ArrayList** et **LinkedList**) ;
- l'utilisation de méthodes de tri.

Bonnes pratiques à adopter

Pour réaliser ce TP, il vous est très fortement recommandé d'utiliser la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez. Par ailleurs, il est aussi important que votre propre code soit commenté au format Javadoc (commentaire commençant par `"/**"` au-dessus des définitions des attributs et méthodes). Ceci vous permet (1) de spécifier rigoureusement le comportement d'une méthode au moment où vous l'implémentez, (2) de savoir ce que fait une méthode sans avoir besoin de regarder son code (qui d'ailleurs peut ne pas être accessible), et donc (3) de faciliter le partage et la réutilisabilité du code.

1 Optimisation de la production : minimisation du temps total de production

C'est maintenant qu'il faut surprendre Monsieur Sorcier et lui montrer les capacités des étudiants d'IG2I !

Vous avez dû observer, à la fin du TP précédent, que Monsieur Sorcier ne tient pas compte des caractéristiques des tâches lorsqu'il détermine le planning de production. Or, il parait tout à fait logique de prendre en compte la durée des tâches et leur date limite de livraison. On pourrait ainsi espérer réduire le temps total d'exécution et les pénalités à payer.

Pour ce faire, vous allez procéder en deux temps. Dans cette section, nous nous intéressons à minimiser le temps total d'exécution (ce qui permet à Monsieur Sorcier de terminer les tâches plus vite, et ainsi de payer moins d'heures supplémentaires). Dans un second temps, en bonus, vous pourrez montrer à Monsieur Sorcier que vous pouvez aussi minimiser les coûts de pénalité !

1.1 Mise en place des objets ordonnanceurs

Question 1. Dans un premier temps, afin de pouvoir comparer les performances des différentes méthodes, créez un nouveau paquetage **ordonnancement** dans lequel on aura les objets qui permettent de fabriquer les ordonnancements dans l'atelier. Dans ce nouveau paquetage, créez une classe **Ordonnancement**, ajoutez-y un attribut **ensTaches** de type **List<Tache>** représentant l'ensemble des tâches à ordonnancer, un attribut **nbMachines** de type entier pour spécifier le nombre de machines dans l'atelier, et un attribut de type **Atelier**. Ajoutez un constructeur par données du nombre de machines de l'atelier et de la liste des tâches à ordonnancer.

La classe **Ordonnancement** sera utilisée de la manière suivante : le constructeur permet de donner les caractéristiques du problème à traiter : le nombre de machines dans l'atelier et les tâches à ordonnancer. Nous allons ensuite implémenter un certain nombre de méthodes pour tester différents algorithmes

pour ordonnancer les tâches. Ainsi, afin de pouvoir réaliser plusieurs ordonnancements d'un même ensemble de tâches, il peut être intéressant de pouvoir :

- recréer un atelier (comme cela on n'ordonnance pas les tâches à la suite de celles déjà ordonnancées) ;
- faire une copie des tâches à ordonnancer, et ordonnancer dans l'atelier seulement les copies ; comme cela la date de début des tâches est modifiée seulement sur les copies.

Question 2. Ajoutez dans la classe **Ordonnancement** une méthode **private void resetAtelier()** qui permet de mettre à jour l'attribut **atelier** avec **nbMachines** machines.

Question 3. Ajoutez dans la classe **Ordonnancement** une méthode **private void ordonnancerCopieTaches(List<Tache> taches)** qui réalise une copie de la liste des tâches en paramètres, puis ordonnance ces tâches sur l'atelier. Pour que votre code fonctionne correctement, il faudra que la copie réalisée soit une copie en profondeur (chaque tâche est également copiée). Un constructeur par copie dans la classe **Tache** peut donc être bien utile. Vous pouvez tester votre méthode avec le code suivant :

```
1  List<Tache> taches = new ArrayList<>();
2  taches.add(new Tache(100));
3  taches.add(new Tache(150));
4  Ordonnancement ordo = new Ordonnancement(1, taches);
5  ordo.ordonnancerCopieTaches(taches);
6  System.out.println("Taches initiales :"); // Ne doivent pas avoir
       changé
7  System.out.println(taches.get(0));
8  System.out.println(taches.get(1));
9  System.out.println("Taches ordonnancees :"); // Les dates de début
       sont mises à jour
10 System.out.println(ordo.atelier);
```

Question 4. Ajoutez dans la classe **Ordonnancement** une méthode **public void ordonnancerMonsieurSorcier()** qui remet à jour l'atelier, puis y ordonnance les tâches avec le même modus operandi que celui proposé par Monsieur Sorcier.

Question 5. Ajoutez dans la classe **Ordonnancement** une méthode **public void afficher(boolean verbose)** qui affiche les critères (temps de production, pénalités) de l'atelier. Si le paramètre **verbose** vaut **true**, alors on affiche également l'atelier (ceci peut être utile pour du débogage). Faites un test pour vérifier que tout fonctionne bien si vous essayez d'ordonnancer quelques tâches.

En général, si on veut tester les performances d'un algorithme, il faut se construire un jeu de test avec des instances (ici nombre de machines et liste de tâches) différentes. Sur Moodle, le fichier *TestMethodes.java* contient une classe **TestMethode** avec 6 jeux de test. Pour chaque jeu de test, le nombre de machines et le nombre de tâches sont donnés dans la Table 1.

jeu de test	1	2	3	4	5	6
nb. machines	2	3	4	4	4	3
nb. tâches	6	40	40	60	20	1000

Table 1: Nombre de machines à considérer pour les jeux de test.

Question 6. Le fichier *TestMethodes.java* contient une classe **TestMethodes** à ajouter dans le paquetage **ordonnancement**. Dans cette classe, il y a une méthode statique **comparerMethodesOrdo** qui n'est pas encore implémentée. Implémentez cette méthode : pour le moment, il n'y a que la méthode d'ordonnancement de Monsieur Sorcier. Faites un test avec la méthode **main** qui est fournie dans la classe **TestMethodes** pour vérifier que tout fonctionne correctement.

1.2 Relation d'ordre sur des objets

À présent, nous cherchons à développer de nouvelles méthodes d'ordonnancement afin de minimiser le temps total d'exécution. Pour cela, une première idée simple à mettre en œuvre est de trier les tâches par durée d'exécution croissante avant d'appliquer à nouveau le modus operandi expliqué pas Monsieur Sorcier.

Pour définir l'ordre dans lequel vous souhaitez trier les tâches de la liste, la classe **Tache** doit implémenter l'interface **Comparable<Tache>**. L'entête de votre classe **Tache** doit alors être :

```
1  public class Tache implements Comparable<Tache> {
2      ...
3  }
```

Vous noterez que l'erreur suivante apparaît alors :

> Tache is not abstract and does not override abstract method compareTo(Tache) in Comparable

Vous avez deux possibilités.

1. Déclarer la classe **Tache** comme étant abstraite (avec le mot clé **abstract**). Dans ce cas, vous ne pourriez plus instancier (= créer des objets de) la classe **Tache**. Elle pourrait être utilisée comme une classe mère et une (ou plusieurs) classe filles hériteraient de cette classe (avec l'utilisation du mot clé **extends**, vous vous rappelez des classes Pilotes, Technicien, Camion, Formule1 ?) et implémenteraient les méthodes abstraites.

2. Implémenter les méthodes abstraites définies par l'interface implémentée par **Tache**.

Dans le cas particulier de ce TP, il semble raisonnable de choisir la seconde option. L'interface **Comparable<E>** déclare une seule méthode : **public int compareTo(E o)**. Cette méthode doit comparer l'objet courant (**this**) à l'objet reçu (**o**) et renvoyer un entier dont la valeur exacte est sans importance, mais qui doit être :

- négative si "**this** < **o**" ;
- nulle si "**this** = **o**" ;
- positive si "**this** > **o**".

On définit alors un ordre partiel ou total (il ne peut pas y avoir d'égalité) sur les objets manipulés.

Ci-dessous un exemple d'ordre (par date limite décroissante) vous est proposé :

```
1  @Override
2  public int compareTo(Tache o) {
3     if(o.dateLimite > this.dateLimite) { // 'this' est après 'o'
4        return 1;
5     } else if(o.dateLimite < this.dateLimite) { // 'this' est avant 'o'
6        return -1;
7     } else { // 'this' est égal à 'o'
8        return 0;
9     }
10 }
```

Comme la valeur retournée n'a pas d'importance et que seul le signe compte, cette méthode peut être écrite plus simplement de la manière suivante :

```
1  @Override
2  public int compareTo(Tache o) {
3     return (o.dateLimite - this.dateLimite);
4  }
```

Par ailleurs, chaque objet possède une méthode **public boolean equals(Object o)** qui renvoie **true** si l'objet **o** est égal à l'objet **this**. Cette méthode est implémentée dans la classe **Object** de laquelle dérivent tous les objets, et définit une relation d'équivalence la plus discriminante possible car deux objets sont considérés égaux seulement s'ils ont la même référence. Heureusement, il est possible de redéfinir cette méthode **equals()** dans les objets que nous créons de sorte que cette notion d'égalité soit en accord avec la notion d'égalité "physique" entre deux objets.

De même, la classe **Object** possède une méthode **public int hashCode()** qui renvoie un entier appelé *code de hachage* associé à l'objet. Cette méthode est notamment utilisée lors de la réalisation de tables de hachage, telles que celles utilisées pour l'implémentation des classes **HashSet** et **HashMap**.

Il est très fortement recommandé que l'ordre naturel défini par la méthode **compareTo** soit consistant avec l'égalité, i.e. *(x.compareTo(y)==0) == (x.equals(y))*. Il est de même fortement recommandé que l'égalité soit consistante avec les codes de hachage, i.e. si *x.equals(y)*, alors *x.hashCode() == y.hashCode()*.

Question 7. Modifiez la classe **Tache** de sorte qu'elle implémente l'interface **Comparable<Tache>** et sa méthode **compareTo(Tache o)** de sorte que les tâches soient ordonnées selon le temps d'exécution croissant. Redéfinissez les méthodes **equals()** et **hashCode()** de façon à assurer une consistance sur l'égalité de deux objets. Pour ce faire, vous pouvez utiliser les assistants Netbeans (clic-droit puis *Insert Code ...* puis *equals() and hashCode()...*). Faites bien attention que la méthode **compareTo** renvoie **0** si et seulement si la méthode **equals** renvoie **true**. Testez dans la méthode principale de la classe **Tache** que la méthode **compareTo(Tache o)** fonctionne correctement.

1.3 Tri des éléments d'une collection

La classe **Collections** dispose d'une méthode statique **sort(List<T> list)** qui réalise un tri des éléments de **list** (qui doit implémenter l'interface **List**). Les éléments de **list** sont réorganisés de façon à respecter l'ordre naturel des éléments de la liste (les éléments doivent donc implémenter l'interface **Comparable**). L'efficacité du tri est en $\mathcal{O}(n \log n)$ avec n le nombre d'éléments dans la liste (n'hésitez pas à jeter un œil à la Javadoc pour plus de précisions).

Pour trier une liste selon l'ordre inverse de l'ordre naturel, on peut commencer par faire un tri selon l'ordre naturel, puis on utilise la méthode statique **reverse (List<T> list)** de la classe **Collections**. Cette méthode inverse l'ordre des éléments de la liste (en temps linéaire). Il ne faut pas faire du bricolage : changer la définition de la méthode **compareTo(Tache t)** de la classe **Tache** n'est pas du tout une bonne idée ! On serait alors obligé de modifier cette méthode à chaque fois que l'on souhaite faire un tri selon un nouveau critère.

Par ailleurs, pour mélanger les éléments d'une liste, vous pouvez faire appel à la méthode statique **shuffle(List<T> list)** de la classe **Collections** : elle

"mélange" les élément dans la liste. Cela peut être utile lors de vos tests par exemple.

Question 8. Ajoutez dans la classe **Ordonnancement** deux nouvelles méthodes :

- une première qui trie les tâches par temps d'exécution croissant avant de faire l'affectation aux machines selon le modus operandi de Monsieur Sorcier ;
- une seconde qui trie les tâches par temps d'exécution décroissant avant de faire l'affectation aux machines selon le modus operandi de Monsieur Sorcier.

Modifiez en conséquence la méthode **comparerMethodesOrdo** de la classe **TestMethodes**, de sorte que l'on puisse comparer les trois méthodes d'ordonnancement en terme de temps total d'exécution. Faites les tests sur les 6 jeux de test. Que dire des résultats trouvés ? En regardant plus en détail le jeu de test 1, est-il possible de faire encore mieux (si oui proposez une meilleure solution, si non donnez un argument) ?

1.4 Un peu d'aléatoire

Question 9. Maintenant, c'est à vous ! Proposez d'autres méthodes d'ordonnancement qui permettent d'améliorer encore les résultats précédents. En particulier, on pourra penser à introduire un peu d'aléatoire...

2 Bonus - Minimisation du coût total de pénalité

Trier les tâches pour minimiser le temps total d'exécution de l'ordonnancement vous a permis d'obtenir un planning suffisamment efficace pour surprendre Monsieur Sorcier. Maintenant que vous savez construire de tels ordonnancements, Monsieur Sorcier vous fait remarquer que pour un même ordonnancement il est possible, en échangeant l'ordre d'exécution des tâches sur une machine, de minimiser les coûts de pénalités sans augmenter le temps total d'exécution.

Question 10. Modifiez la méthode **comparerOrdo()** de la classe **Ordonnancement** de telle sorte qu'après avoir ordonnancé les tâches, on affiche les valeurs du temps total d'exécution et du coût total de pénalité, puis on essaye d'améliorer l'ordonnancement de sorte que le temps total d'exécution reste le même et le coût total de pénalité diminue. Vous êtes évidemment fortement encouragés à rajouter de nouvelles méthodes dans les classes du paquetage **atelier** si nécessaire. Si vos résultats sont stupéfiants, courez chez Monsieur Sorcier, peut-être qu'il va vous embaucher !!

References

[1] Delannoy, C., *Programmer en Java, Eyrolles, 2014*