Programmation Orientée Objet
IG2I, LE2 – 2023-2024

TP3 – Ordonnancement

Diego Cattaruzza, Maxime Ogier, Pablo Torrealba

Contexte et objectif du TP

Le carnaval de Dunkerque va bientôt commencer. L'entreprise nordiste Masques&Confettis doit faire face à une grosse demande de production de masques colorés. 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 décide donc de faire appel aux meilleurs étudiants de la région Hauts-de-France pour concevoir des outils informatiques qui puissent fournir un planning de production capable de gérer des demandes importantes.

L'objectif de ce TP est donc de concevoir un outil pour aider Masques&Confettis dans son travail. Nous nous concentrerons dans un premier temps sur le modèle de données à mettre en œuvre.

Ce TP permet ainsi d'illustrer les notions suivantes :

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

On s'attachera en particulier dans ce TP à commenter proprement le code et à garantir que le code fonctionne et respecte les spécifications données.

Bonnes pratiques à adopter

- Testez votre code au fur et à mesure de votre avancement.
- Respectez les conventions de nommage Java.
- Faites des indentations de manière correcte.
- Donnez des noms compréhensibles et pertinents à vos variables, attributs, classes, méthodes.
- Respectez autant que possible le principe d'encapsulation.
- Utilisez des structures de données adaptées.
- Faites des méthodes courtes avec peu de niveaux d'indentation.
- Ne faites pas de duplication de code, mais privilégiez le code réutilisable.
- Utilisez les outils de votre IDE pour faire du *refactoring* (clic-droit puis '*Refactor*' sur IntelliJ IDEA).
- Utilisez la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez.
- Quand c'est nécessaire, commentez votre propre code de façon pertinente au format Javadoc (commentaire commençant par `"/**"` au-dessus des définitions des attributs et méthodes).

1 Modélisation de l'atelier de production de Masques&Confettis

Le responsable de l'atelier de production, Monsieur Sorcier, vous explique que son travail consiste à planifier la production des masques au sein de l'atelier. Cela consiste à décider avec quelles ressources et dans quel ordre on effectue les tâches pour minimiser les délais ou les coûts de production tout en respectant les contraintes imposées par le système de production.

Monsieur Sorcier explique qu'il dispose dans son atelier de n machines identiques. Une tâche T_i est une entité élémentaire de travail qui nécessite un temps de production p_i sur une machine. Pour fixer les idées, une tâche correspond à un ensemble de masques identiques à livrer à un même client. Par ailleurs, une fois affectée à une machine, chaque tâche T_i est localisée dans le temps par une date de début t_i et une date de fin c_i ($c_i = t_i + p_i$). Notez que quand nous parlons ici de dates, nous ne sommes pas intéressés par connaître le mois et le jour, mais la durée écoulé depuis le début de la production des tâches. Ainsi, nous supposons que l'instant initial est 0, et nous utilisons la minute comme unité temporelle. Donc une date $t = 120$ signifie 120 minutes après l'instant initial.

Par ailleurs, chaque tâche est caractérisée par une date limite de livraison d_i, et un coût unitaire de pénalité r_i (pour chaque minute de retard) en cas de retard de livraison. Une fois affectée à une machine, le coût total de pénalité f_i d'une tâche sera donc de $f_i = r_i \times \max\{c_i - d_i; 0\}$.

Construire un ordonnancement consiste à affecter les tâches aux machines de manière à établir la séquence de réalisation des tâches sur chaque machine (il faut donc aussi décider la date de début de chaque tâche). Un exemple est donné dans la Figure 1. Dans cet exemple on a 13 tâches. La tâche T_4 est affectée à la machine M_1, elle commence à $t = 3$ et termine à $t = 8$. À la date $t = 8$, la machine M_1 est disponible et la tâche T_8 peut alors commencer.

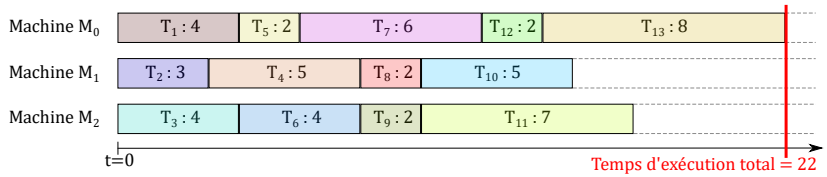

Figure 1: Exemple d'un ordonnancement sur 3 machines. Chaque tâche est notée $T_i : p_i$.

Une tâche peut être démarrée sur n'importe quelle machine du moment que cette dernière est disponible. Ainsi la date de début d'une nouvelle tâche sur cette machine est donnée par la date de fin de la tâche précédente sur cette même machine. Il n'y a aucun intérêt à laisser des temps morts. De plus, Monsieur Sorcier explique qu'une tâche ne peut pas être interrompue pour être continuée plus tard.

Dans ce TP, nous allons mettre en place la modèle objet afin de pouvoir ordonnancer un ensemble de tâches sur les machines d'un atelier de production. Le diagramme de classe UML pour cette application est donné dans la Figure 2.

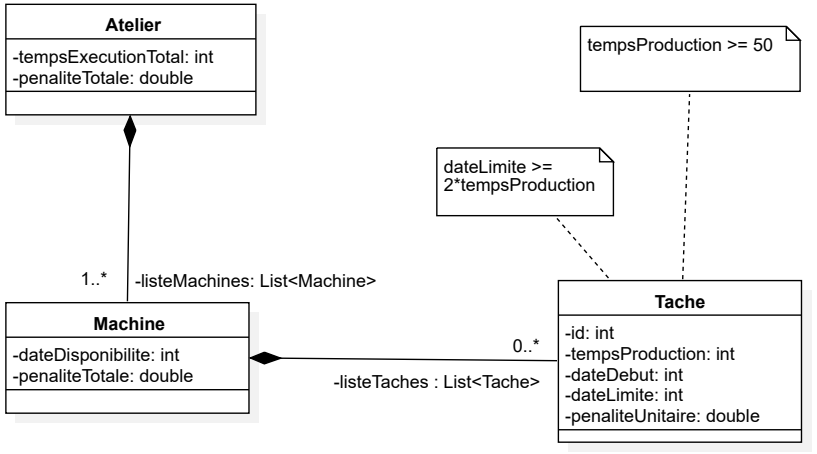

Figure 2: Diagramme de classe UML.

1.1 Création de la classe Tache

Nous nous intéressons dans un premier temps à la représentation d'une tâche de production.

Pour cela, Monsieur Sorcier explique que dans l'atelier de Masques&Confettis l'exécution d'une tâche dure au minimum 50 minutes. Il arrive que certaines tâches n'aient pas de date limite de livraison : on considère alors que la date limite de livraison est infinie et que le coût unitaire de pénalité est nul. Par ailleurs, Monsieur Sorcier a réussi à négocier avec le service commercial que la date limite de livraison d'une tâche soit au moins égale à deux fois le temps de production de la tâche (donc $d_i \geq 2p_i$). Monsieur Sorcier indique que le temps de production ne peut pas être modifié et, comme c'est un bon négociateur, il a également obtenu du service commercial que les dates limite de livraison et les coûts unitaires de pénalité ne puissent pas être modifiés. De plus, chaque tâche doit être caractérisée par un identifiant unique.

Question 1. Après avoir ouvert IntelliJ IDEA, créez un nouveau projet **MasquesConfettis**, un paquetage **atelier** (qui contiendra les classes pour la modélisation de l'atelier de production), et écrivez une classe **Tache** qui

représente une tâche à faire par Masques&Confettis, dont vous définirez les attributs. Commentez chacun des attributs sous format Javadoc.

Question 2. Ajoutez un constructeur par défaut, un constructeur par donnée du temps de production, et un constructeur par données du temps de production, de la date limite de livraison et du coût unitaire de pénalité en cas de retard. N'oubliez pas que les assistants de IntelliJ IDEA sont là pour vous aider (clic-droit puis *Generate...*).

Question 3. Complétez la classe **Tache** avec les accesseurs et mutateurs nécessaires et une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur la tâche (en fait il s'agit d'une redéfinition de la méthode **toString()** de la classe **Object**). N'oubliez pas que les assistants de IntelliJ IDEA sont là pour vous aider (clic-droit puis *Generate...*).

Question 4. Écrivez une méthode principale (**public static void main(String[] args)**) afin de tester l'utilisation de la classe **Tache**. Assurez-vous de bien tester tous les cas possibles et surveillez que les préconisations sur le temps de production et la date limite soient toujours bien respectées. N'oubliez pas que les assistants de IntelliJ IDEA sont là pour vous aider (tapez "*psvm*" pour écrire la signature de la méthode principale, ou encore "*sout*" pour afficher une ligne de texte sur la console.

1.2 Création de la classe Machine

1.2.1 Notion d'interface

Nous avons vu lors du TP précédent qu'une classe abstraite permettait de définir dans une classe de base des fonctionnalités communes à toutes ses descendantes, tout en leur imposant d'implémenter certaines méthodes (celles définies comme abstraites dans la classe abstraite).

Si l'on considère une classe abstraite qui n'a aucun attribut (sauf des constantes déclarées **static** et **final**), qui n'a aucun constructeur, et qui ne possède que des méthodes abstraites, on aboutit alors à la notion d'interface. La définition d'une interface se présente comme celle d'une classe. On y utilise simplement le mot-clé **interface** à la place de **class**, comme ci-dessous :

```
1  public interface I {
2     void f(int n); // En-tête d'une méthode f
3     double g(); // En-tête d'une méthode g
4  }
```

Par essence, les méthodes d'une interface sont abstraites et publiques, néanmoins il n'est pas nécessaire de le mentionner avec les mots clés **abstract** et **public**. Par ailleurs, notez qu'une interface ne possède pas de constructeur et n'a que des méthodes abstraites, donc on ne peut pas l'instancier (c'est-à-dire on ne peut pas écrire **new I()** avec l'exemple ci-dessus).

Lorsqu'on définit une classe, on peut préciser dans sa signature qu'elle implémente une interface donnée en utilisant le mot clé **implements** comme dans :

```
1  public class A implements I {
2     // A doit (re)définir les méthodes f et g prévues dans
        l'interface I
3     @Override
4     public void f (int a) {
5        // Implémentation de la méthode f
6     }
7
8     @Override
9     public double g () {
10       // Implémentation de la méthode g
11    }
12 }
```

Avec le langage Java, une classe peut hériter d'une seule classe mais une classe peut implémenter plusieurs interfaces (on sépare les noms des interfaces par des virgules). C'est une manière de gérer l'héritage multiple avec le langage Java.

1.2.2 Polymorphisme

Le polymorphisme en programmation orientée objet est un concept qui permet de manipuler un objet sans en connaître nécessairement le type exact. C'est ainsi que dans le TP précédent on manipulait dans une écurie de Formule 1 des voitures et des personnes sans savoir exactement si les voitures étaient des Formule 1 ou des camions, et si les personnes étaient des pilotes ou des techniciens. Pour ce faire, on utilisait le fait qu'un objet est reconnu comme une instance de sa classe et de toute la hiérarchie de ses classes parentes. Par exemple, un objet de type **Pilote** était aussi de type **Personne** et de type **Object**.

Par ailleurs, si une classe définit une méthode, on est certain que toutes ses classes filles possèdent également cette même méthode, c'est le principe même de l'héritage. Mais en plus de cela, les classes filles ont la possibilité de redéfinir (i.e. de modifier) les méthodes héritées de leur classe mère. C'est typiquement ce que l'on fait en général avec la méthode **toString()** de la classe **Object**. C'est cette possibilité de redéfinition de méthode héritées qui est au cœur du concept de polymorphisme. Ainsi, on peut traiter de la même manière des objets de types différents, à condition qu'ils héritent d'une même classe parente. On peut ainsi appeler la méthode **toString()** sur n'importe quel objet, sans connaître le comportement effectif (la redéfinition) de cette méthode.

Notez que le concept de polymorphisme décrit précédemment dans le cadre de l'héritage fonctionne également dans le cas des classes abstraites avec des méthodes abstraites. Ainsi, dans le TP précédent, on pouvait faire appel à la méthode abstraite **estCompatible(Voiture v)** dans la classe **Personne** sans en connaître l'implémentation exacte de cette méthode dans les classes filles.

Le concept de polymorphisme s'applique aussi dans le cas des interfaces :

- un objet est reconnu comme une instance de chacune des interfaces qu'il implémente (ainsi dans l'exemple un objet de type **A** est aussi de type **I**) ;
- si une classe (non abstraite) implémente une interface, on est certain que les méthodes définies par l'interface sont implémentées, et peuvent donc être appelées sans connaître le type exact de l'objet.

1.2.3 Collections d'objets

Pour manipuler des collections d'objets en Java, le paquetage **java.util** fournit un ensemble de classes qui permettent de manipuler des vecteurs dynamiques, des ensembles, des listes chaînées, des files et des tables associatives. Les classes relatives aux vecteurs, aux listes, aux ensembles et aux files implémentent une même interface (**Collection**) qu'elles complètent de fonctionnalités propres.

En fait, les collections sont manipulées par le biais de classes génériques implémentant l'interface **Collection<E>**, où **E** représente le type des éléments de la collection. Tous les éléments d'une collection sont donc du même type **E**. Notez que grâce au polymorphisme, des éléments de la collection peuvent être d'un type dérivé de **E**, c'est-à-dire d'une classe fille de **E**.

1.2.4 Parcours des éléments d'une collection

La boucle *for...each* permet de parcourir de manière simple une collection d'objets de type **E** nommée **col** en procédant ainsi :

```
1  for (E elem : col) {
2     // Utilisation de elem
3  }
```

La variable **elem** prend successivement la valeur de chacune des références des éléments de la collection. **ATTENTION**, ce schéma n'est pas exploitable si l'on doit modifier la collection, en utilisant des méthodes telles que **remove()** ou **add()** !

Il existe aussi des objets spécifiques, appelés "itérateurs" qui permettent de parcourir un par un les éléments d'une collection. Ils ressemblent à des pointeurs (tels que ceux de C ou C++) sans en avoir exactement les mêmes propriétés. Chaque classe qui implémente l'interface **Collection** dispose d'une méthode nommée **iterator** fournissant un itérateur monodirectionnel, i.e. un objet d'une classe implémentant l'interface **Iterator<E>**. On pourra ainsi parcourir une collection d'objets de type **E** nommée **col** de la manière suivante :

```
1  Iterator<E> iter = col.iterator();
2  while (iter.hasNext()) {
3     E elem = iter.next();
4     // Utilisation de elem
5  }
```

En fait, la méthode **iterator()** renvoie un objet désignant le premier élément de la collection s'il existe. La méthode **next()** fournit l'élément courant désigné par **iter** et avance l'itérateur à la position suivante. Si l'on souhaite parcourir plusieurs fois une même collection, il suffit de réinitialiser l'itérateur en appelant à nouveau la méthode **iterator()**.

L'interface **Iterator** prévoit la méthode **remove()** qui supprime de la collection le dernier objet renvoyé par **next()**.

1.2.5 Liste d'objets

Une liste d'objets correspond à une collection ordonnée d'objets. Ceci signifie que :

- chaque objet possède une position dans la liste, et on peut donc accéder aux objets par leur position : il y a un premier élément, un deuxième, un troisième, etc. ;
- plusieurs objets de même valeur peuvent apparaître dans la liste.

Notez bien que le fait que la liste soit ordonnée ne signifie pas que les éléments sont triés selon un certain critère : la notion d'ordre est différent de la notion de tri. Par ailleurs, comme une liste est ordonnée, les méthodes d'ajout et de suppression requièrent comme paramètre la position à laquelle doit s'effectuer l'opération. Enfin, notez que comme il n'y a pas de garantie que les éléments soient triés, la recherche d'un élément par sa valeur (savoir si un élément est présent) se fait avec une complexité temporelle en $O(n)$ (n étant la taille de la liste). Il est en effet nécessaire de parcourir tous les éléments de la liste.

Les listes d'objets en Java sont des objets qui implémentent l'interface **List** (qui elle-même hérite de l'interface **Collection**). **List** est une interface et vous ne pouvez donc pas l'instancier. Dans le langage Java, il existe principalement deux classes qui implémentent l'interface **List** : **ArrayList** et **LinkedList**. Pour connaître les méthodes offertes par ces objets, reportez-vous à la Javadoc. Lorsque que l'on souhaite utiliser une structure de données de type liste, il faut donc choisir parmi **ArrayList** et **LinkedList** celle qui convient le mieux.

ArrayList est implémentée comme un tableau dynamique (i.e. un tableau de taille variable). Ceci offre donc des fonctionnalités d'accès rapide comparables à celles d'un tableau d'objets (en $\mathcal{O}(1)$). En outre, cette classe offre plus de souplesse que les tableaux d'objets dans la mesure où sa taille (son nombre d'éléments) peut varier au fil de l'exécution (comme celle de n'importe quelle collection). Mais pour que l'accès direct à un élément de rang donné soit possible, il est nécessaire que les emplacements des objets (plutôt de leur référence) soient contigus en mémoire (à la manière de ceux d'un tableau). Aussi cette classe souffrira d'une lacune inhérente à sa nature : l'ajout ou la suppression d'un objet à une position donnée ne pourra plus se faire en $\mathcal{O}(1)$ comme dans le cas d'une liste chaînée, mais seulement en moyenne en $\mathcal{O}(n)$. En définitive, les **ArrayList** seront bien adaptées à l'accès direct à condition que les ajouts et les suppressions restent limitées.

LinkedList est implémentée comme une liste doublement chaînée. Le grand avantage d'une telle structure est de permettre des ajouts ou des suppressions lors du parcours de la liste avec une efficacité en $\mathcal{O}(1)$ (ceci grâce à un simple jeu de modification de références). En revanche, l'accès à un élément en fonction de sa position dans la liste sera peu efficace puisqu'il nécessitera obligatoirement de parcourir une partie de la liste. L'efficacité sera donc en moyenne en $\mathcal{O}(n)$. En définitive, les **LinkedList** seront bien adaptées à l'ajout et à la suppression à condition que les accès directs à un élément restent limités.

De manière générale, il n'y a pas d'implémentation de l'interface **List** qui soit toujours meilleure qu'une autre. Le choix doit être fait par rapport aux caractéristiques particulières de votre problème. Cette observation ne s'applique pas seulement aux implémentations de l'interface **List**, mais aussi aux autres interfaces pour les ensembles (**Set**) et les tables associatives (**Map**).

1.2.6 Retour chez Masques&Confettis

Masques&Confettis dispose d'un nombre limité de machines pour exécuter les tâches. Monsieur Sorcier qui a ordonnancé pendant vingt ans les tâches avec son papier et son stylo vous explique que lors de l'affectation des tâches à une des machines, il est très important de pouvoir connaître :

- la date de disponibilité actuelle de la machine, i.e. l'instant à partir duquel on peut démarrer une nouvelle tâche sur la machine ;
- le cumul des coûts de pénalité liés à cette machine.

Monsieur Sorcier se retrouve chaque vendredi soir à devoir résoudre un problème très difficile : comment affecter les tâches aux machines pour minimiser le temps d'exécution total de l'ordonnancement (i.e. la date à laquelle toutes les tâches sont terminées), et puis si possible à minimiser les coûts de pénalité en cas de retard.

Nous essayerons d'aider Monsieur Sorcier plus tard. Pour le moment Monsieur Sorcier vous explique que pour réaliser son ordonnancement, il affecte à chacune des machines une *liste de tâches*, puis ajoute successivement des tâches sur chacune de ces listes. Ensuite, sur chaque machine, les tâches sont exécutées dans l'ordre dans lequel elles apparaissent dans la liste.

Question 5. Dans le paquetage **atelier**, ajoutez une classe **Machine** avec ses attributs.

Question 6. Ajoutez un constructeur par défaut à la classe **Machine**. Ajoutez les accesseurs et les mutateurs nécessaires.

Question 7. Ajoutez une méthode **public boolean addTache(Tache t)** qui ajoute la tâche **t** à la liste des tâches exécutées par la machine, et renvoie **true** si l'ajout a bien été effectué, **false** sinon. Vous pouvez faire appel à la méthode **add(Object o)** de l'interface **List** qui permet d'ajouter un objet à la liste (l'ajout se fait en fin). Pour rappel, Monsieur Sorcier propose que lorsqu'une tâche est ajoutée à la liste, elle soit exécutée dès que la machine est disponible, pour ne pas avoir de temps morts. Votre méthode **addTache(Tache t)** doit aussi mettre correctement à jour les données liées à la machine et à la tâche : la date de début de la tâche, la date de disponibilité de la machine, la pénalité totale de la machine.

Question 8. Ajoutez une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur la machine (vous ferez en sorte que ceci soit lisible facilement) et écrivez une méthode principale afin de tester l'utilisation de la classe **Machine**. Quel est le temps total de production et la pénalité que Masques&Confettis doit payer si elle reçoit les tâches suivantes (définies par temps de production, date limite de livraison et coût unitaire de pénalité) et les exécute dans l'ordre sur une seule machine ? Quelle est la date de début pour la tâche **t5** ? Quel est le coût de pénalité pour la tâche **t5** ?

```
1  Tache t1 = new Tache(150, 300, 2.5);
2  Tache t2 = new Tache(140, 400, 1.5);
3  Tache t3 = new Tache(50, 200, 2.5);
4  Tache t4 = new Tache(85, 200, 1.0);
5  Tache t5 = new Tache(75, 160, 0.5);
6  Tache t6 = new Tache(80, 500, 1.5);
```

1.3 Création de la classe Atelier

Dans la première partie de ce sujet vous avez fait la connaissance de Monsieur Sorcier. Dans vos conversations, vous lui avez montré votre intelligence et votre perspicacité. Il décide donc de vous ouvrir les portes de l'atelier de Masques&Confettis.

L'atelier est principalement composé d'un nombre limité de machines sur lesquelles les tâches peuvent être exécutées. Par ailleurs, Monsieur Sorcier vous explique que pour l'atelier il a besoin de connaître :

- le temps total d'exécution, i.e. la date à laquelle toutes les tâches sont terminées (car on suppose que l'on commence à l'instant 0) ; il s'agit donc de la plus grande date de disponibilité des machines, comme présenté dans la Figure 1 ;
- le coût total de pénalité, i.e. la somme des coûts de pénalité des machines.

Question 9. Dans le paquetage **atelier**, ajoutez une classe **Atelier** avec ses attributs. L'ensemble des machines est mémorisé dans une structure de données de type liste. Quel type de liste choisissez-vous ? Pour prendre la décision correcte, considérez que vous devrez faire appel autant de fois que le nombre des tâches, à la méthode **get()** de l'interface liste (cela pour sélectionner la machine sur laquelle exécuter la tâche). Documentez-vous pour savoir si la méthode **get()** est plus rapide dans l'implémentation **ArrayList** où **LinkedList**. Des sources importantes d'informations sont la Javadoc et le site web `http://docs.oracle.com/`. Commentez le code sous format Javadoc.

Question 10. Écrivez un constructeur qui prend en paramètre le nombre de machines présentes dans l'atelier (on suppose qu'il y a au moins une machine, sinon ce n'est pas un véritable atelier). Ajoutez les accesseurs et mutateurs nécessaires, ainsi qu'une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur l'atelier. Commentez votre code et écrivez une méthode principale afin de tester l'utilisation de la classe **Atelier**.

Question 11. Ajoutez une méthode **public void miseAJourCriteres()** qui calcule les critères d'ordonnancement (temps total d'exécution et coût total de pénalité) et met à jour les attributs correspondants.

Monsieur Sorcier vous explique que le vendredi soir, avant de partir pour le week-end, il conçoit le planning pour la semaine suivante avec ce modus operandi :

- le responsable des ventes lui fournit une liste avec l'ensemble des tâches à effectuer dans la semaine ;
- Monsieur Sorcier prend les tâches une par une dans l'ordre dans lequel elles sont listées ;
- la première tâche de la liste est affectée à la première machine ;
- pour chacune des autres tâches, il l'affecte à la machine qui se libère en premier (celle dont la date de disponibilité est la plus petite).

Question 12. Ajoutez à votre classe **Atelier** les méthodes nécessaires pour implémenter le modus operandi proposé par Monsieur Sorcier :

- une méthode **private Machine getMachine(int posMachine)** qui renvoie la machine en position **posMachine** de votre liste des machines (faites appel à la méthode **get()** de l'interface **List**) ;
- une méthode **private boolean addTache(Tache t, int posMachine)** qui ajoute la tâche **t** à la machine en position **posMachine** dans votre liste des machines (faites appel à la méthode **addTache()** de la classe **Machine**) ; cette méthode renvoie **true** si l'ajout a été correctement effectué, **false** sinon ;
- une méthode **private int getPremiereMachineLibre()** qui renvoie un entier indiquant la position, dans la liste des machines, de la machine qui est disponible en premier ;
- une méthode **public void ordonnancerTaches(List<Tache> taches)** qui ordonnance les tâches de la liste **taches** selon le modus operandi de Monsieur Sorcier, puis met à jour les critères d'ordonnancement (temps total d'exécution et coût total de pénalité).

Commentez ces méthodes au format Javadoc.

Question 13. Testez la méthode d'ordonnancement pour un atelier à deux machines sur l'ensemble des tâches fournies à la Question 8. Quel est alors le temps total d'exécution ? Quel est le coût total de pénalité ?

Question 14. Si l'on s'intéresse à minimiser le temps total d'exécution des tâches sur les machines, que pensez-vous du modus operandi proposé par Monsieur Sorcier ? Sur le test effectué à la question précédente, pouvez-vous proposer un ordonnancement des tâches qui permette de diminuer le temps total d'exécution ? Quelle valeur obtenez-vous ? Proposez un algorithme pour ordonnancer les tâches afin de minimiser le temps total d'exécution.