UE Conception Orientée Objet

Devoir Surveillé
3h
Indiquez votre numéro de groupe sur la copie.
Copie des diapositives de cours annotée autorisée.
Fiches « *design pattern* » du cours autorisées.
Dictionnaires de langue autorisés.
Autres documents interdits.

- *Les exercices sont indépendants.*
- *Sauf mention explicite, la javadoc et les tests des classes à écrire ne sont pas demandés.*
- *Prenez le temps de lire calmement l'énoncé avant de commencer à y répondre.*

Exercice 1 : Commandes
Dans le contexte d'un logiciel de gestion des commandes d'un restaurant, les éléments qu'il est possible de commander sont de type `Orderable`.
Tous les éléments `Orderable` ont une description (une chaîne de caractères) qui correspond à leur nom dans le menu et un prix (de type `float`). Ces informations sont fournies à la création.
Parmi les éléments `Orderable`, on distingue les éléments de la carte et les menus.

Les éléments de la carte se répartissent en trois catégories :

1. les *boissons* ;
2. les *plats* ;
3. les *desserts*.

Tous les éléments de carte existent en version XL. Par rapport à la version « normale », les portions sont augmentées de 50% pour un prix augmenté de 25%. La description est quant à elle la description « normale » simplement complétée de la mention `"XL"` ajoutée à la fin.

Les menus sont composés d'une boisson, d'un plat et d'un dessert à choisir parmi les éléments de la carte. Le prix d'un menu est obtenu en diminuant de 15% le prix de la somme ses trois éléments. La description d'un menu est la concaténation des trois descriptions préfixée de `"Menu : "`. La version XL des menus n'existe pas, on peut cependant construire un menu à partir d'éléments XL ou non.

Q 1 . Donnez un **ou plusieurs** diagrammes UML de classes détaillés qui présentent clairement votre proposition de conception pour les éléments `Orderable`.

Dans votre diagramme, vous ferez apparaître clairement les attributs, constructeurs et méthodes (et leurs signatures), ainsi que les surcharges.

Faites des diagramme clairs et lisibles. N'hésitez pas à prendre votre copie au format paysage.

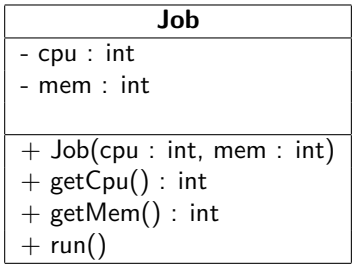
Q 2 . Quel(s) *design pattern(s)* avez-vous mis en œuvre dans votre solution ?

Justifiez **clairement** dans chaque cas, en quelques phrases claires, pourquoi l'utilisation de ce *design pattern* est pertinent dans ce cas. De plus, vous identifierez dans votre solution les éléments de conception qui caractérisent la mise en place de chaque design pattern.

Exercice 2 :
Dans cet exercice, nous considérons une ferme de machines (*cluster*) qui peut mettre à disposition des ressources de calcul pour un ensemble d'utilisateurs. Les ressources de calcul peuvent prendre deux formes différentes, CPU et mémoire, et sont consommées par des tâches (*job*) qui s'exécutent pour une durée de temps inconnue *a priori*. L'objectif est de proposer un ordonnanceur qui place un ensemble de tâches sur les nœuds de manière à maximiser l'utilisation des ressources afin de minimiser le temps d'exécution global.

La tâche de calcul

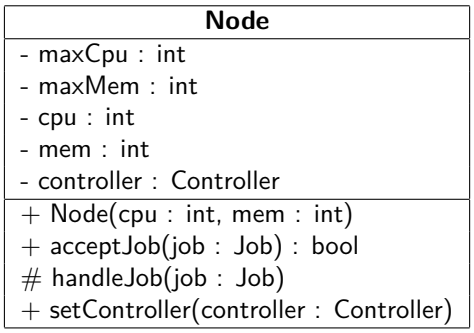
Une tâche de calcul (*Job*) est décrite par un ensemble de ressources à consommer sur un nœud. Nous supposons ici que ces ressources seront consommées de manière uniforme, tout au long de l'exécution de la tâche. Une tâche est donc modélisée de la manière suivante :



La méthode `run()` déclenche l'exécution du job, sur un nœud de calcul et prend donc un certain temps à s'exécuter.

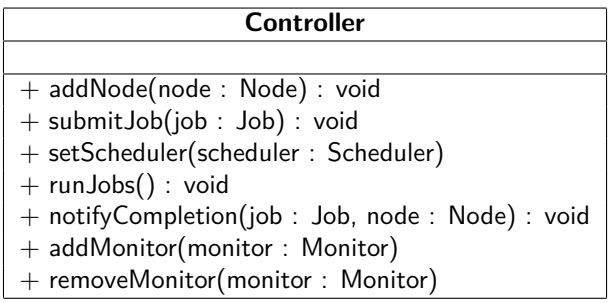
La ferme de calcul

Une ferme de calcul se compose d'un ensemble de nœuds (*node*) qui ont chacun une capacité courante et une capacité maximale pour chacune des ressources de calcul disponibles (cpu, mémoire). Ces ressources sont modélisées sous la forme d'un entier qui représente la capacité disponible pour cette ressource. Il prend donc la valeur `0` quand la ressource est indisponible et la valeur de la capacité maximale (celle fournie à la construction du nœud) quand elle est totalement libre. Chaque nœud est donc modélisé comme suit :



La méthode `acceptJob()` déclenche une exception `NotEnoughResourceException` si le nœud ne dispose pas des ressources disponibles suffisantes pour exécuter la tâche fournie. Sinon le nœud prend en charge le job fourni, en invoquant sa méthode `handleJob`, et ses capacités en ressources sont alors réduites en fonction des besoins de la tâche traitée jusqu'à la fin du traitement de la tâche (c'est-à-dire la fin de l'exécution de sa méthode `run()`).

Un contrôleur (*controller*) permet d'organiser la répartition des différentes tâches sur les nœuds de la ferme. Pour cela il faut préalablement ajouter les nœuds gérés et soumettre les tâches à prendre en compte. L'interface du contrôleur peut être résumée ainsi :



La méthode `addNode()` permet d'associer un nœud à la ferme de calcul et le contrôleur est alors enregistré auprès du nœud (méthode `setController`). Dès lors que la méthode `runJobs()` est invoquée, plus aucun nœud ne peut être ajouté à la ferme.

La méthode `submitJob()` permet de soumettre une nouvelle tâche à la ferme. Il n'est pas possible de le faire tant qu'aucun nœud n'a été associé à la ferme de calcul. Si aucun nœud ne dispose des ressources

suffisantes pour exécuter une tâche, alors le contrôleur rejette la soumission d'une tâche en déclenchant une exception.

Dès lors que la méthode `runJobs()` est invoquée, plus aucun nœud ne peut être ajouté à la ferme et plus aucune tâche ne peut être soumise. Lors de l'exécution de cette méthode le contrôleur s'efforce de placer les tâches sur les nœuds. Le placement des tâches sur les nœuds se fait en appliquant un algorithme d'ordonnancement (configuré via la méthode `setScheduler()`). Ces algorithmes de type `Scheduler`[1] se basent sur leur méthode

```
Set<Job> scheduleJobs(job :  Set<Job>, nodes :  Set<Node>)
```

qui essaie de placer un ensemble de tâches sur un ensemble de nœuds. Pour tenter d'affecter une tâche à un nœud, cet algorithme appelle la méthode `acceptJob()` de ce nœud. La méthode `scheduleJobs()` retourne l'ensemble des tâches qui **n'ont pas pu être placées**, faute de disponibilité des ressources en fonction des tâches déjà affectées. L'ensemble retourné est vide si toutes les tâches ont pû être placées sur les nœuds de la ferme de calcul.
Il existe différents algorithmes d'ordonnancement :

- `BinpackScheduler` consomme un maximum de ressources d'un nœud avant de passer au nœud suivant. Pour chacune des tâches, on parcourt les tâches dans le même ordre et on place la tâche sur le premier nœud qui l'accepte. Si aucun ne l'accepte, elle est ajoutée à l'ensemble des tâches non placées.

- `SpreadScheduler` répartit les tâches en maximisant le nombre de nœuds utilisés.
  Une première tâche est placée sur un nœud qui l'accepte. Pour les suivantes on cherche à chaque à placer la tâche sur un nœud qui l'accepte. Un nœud déjà servi n'est à nouveau sollicité que lorsque tous les autres nœuds ont été servis à leur tour ou ont refusé la tâche. Les tâches refusées par tous les nœuds font partie de l'ensemble résultat.

- `RandomScheduler` place les tâches au hasard sur les nœuds.
  Pour chacune des tâches à placer, un nœud est pioché au hasard et on essaie d'y placer la tâche. Si le nœud n'accepte pas la tâche, celle-ci est ajoutée à l'ensemble des tâches non placées.

Dès que l'exécution d'une tâche est terminée, le nœud associé à la tâche récupère ses ressources et il en informe son contrôleur par un appel à sa méthode `notifyCompletion()`. D'une part, cela permet au contrôleur de relancer l'algorithme d'ordonnancement pour chercher à placer les tâches qui n'avaient pas été placées précédemment. D'autre part, le contrôleur notifie alors tous les éventuels différents moniteurs (de type `Monitor`) qui sont abonnés au contôleur (méthode `addMonitor()`) en invoquant leur méthode `jobFinished()`. Les moniteurs sont de différentes natures : ils peuvent journaliser la progression dans un fichier ou l'afficher dans une interface graphique ou ...

Quand toutes les tâches sont exécutées, le contrôleur notifie tous les moniteurs abonnées, via leur méthode `allJobsFinished()`. Il remet également à zéro la liste des tâches et est à nouveau disponible pour accepter de nouvelles soumissions.

Q 1 . Donnez un **ou plusieurs** diagrammes UML de classes détaillés qui présentent clairement votre proposition de conception de la ferme de calcul qui intègre la gestion des différents types d'algorithmes d'ordonnancement et des moniteurs.

Dans votre diagramme, vous ferez apparaître clairement les attributs, constructeurs et méthodes (et leurs signatures), ainsi que les surcharges.

Faites des diagramme clairs et lisibles. N'hésitez pas à prendre votre copie au format paysage, ou mieux à faire votre ou vos diagrammes sur les deux pages intérieures de votre copie.

Q 2 . Quel(s) *design pattern(s)* avez-vous mis en œuvre dans votre solution ?

Justifiez **clairement** dans chaque cas en quelques phrases claires pourquoi l'utilisation de ce design pattern est pertinent dans ce cas. De plus, vous identifierez dans votre solution les éléments de conception qui caractérisent la mise en place de chaque design pattern.

[1] Il n'y a aucune relation avec les *schedulers* du TD sur les actions.

Q 3 . Donnez le code de la ou des méthodes de test pour vérifier le comportement de la méthode `acceptJob()` de la classe `Node`.

Vous précéderez le code de chacune des méthodes de test d'un commentaire **précis** qui précisera la situation testée par la méthode.

Q 4 . Donnez le code de la ou des méthodes de test pour vérifier que la méthode `submitJob()` déclenche une exception si aucun nœud ne peut exécuter la tâche soumise.

Q 5 . Donnez le code de la ou des méthodes de test qui permet de vérifier que les moniteurs abonnés sont bien notifiés lors de l'exécution de la méthode `notifyCompletion()` de la classe `Controller`.

Q 6 . Donnez le code de la méthode `scheduleJobs` des `BinpackScheduler`.