Polytech Paris-Sud – ET4 Informatique - Concept Objets

sujet de TD et de TP

On va s'intéresser à définir une hiérarchie de classes Java adaptée à l'énoncé ci-dessous. Dans un premier temps, en TD, on ne sera pas obligé de considérer tous les détails de réalisation (constructeurs, affichage) et les méthodes pourront être écrites dans un pseudo-code à la Java sans toutes les contraintes de Java. La priorité est d'identifier les méthodes dont on a besoin dans les différentes classes, de définir celles qui sont abstraites ou qui donnent un comportement par défaut, de fixer les visibilités, et d'ébaucher le corps des principales méthodes. La hiérarchie sera ensuite traduite en Java lors des TPs.

On considère un laboratoire dans lequel des chercheurs utilisent des robots pour déplacer des objets. Les objets sont caractérisés par un identifiant et on donne leur position horizontale dans la pièce (on ne considérera pas la hauteur des objets ou de leur empilement). Certains de ces objets sont inertes, comme les caisses de bois, d'autres sont mobiles, comme les robots. Un robot est capable de soulever et de transporter des objets de tout type, y compris d'autres robots portant eux-mêmes des objets arbitraires, et de les transporter à travers la pièce, sous réserve que la masse totale des objets soulevés soit inférieure à une charge maximale définie pour chaque robot. Un robot pourrait par exemple transporter directement un autre robot et deux caisses, le robot transporté portant lui-même d'autres objets. Il n'y a pas de limitation sur le portage autre que la masse totale des objets transportés et le fait que seuls les robots situés directement sur le sol peuvent charger ou décharger un objets, directement à partir ou sur le sol : un robot transporté ne peut par exemple plus directement charger ou décharger un objet.

Première partie du sujet : Les méthodes auxquelles on s'intéresse sont les suivantes

Méthodes pour tout type d'objet :

Avoir accès à leur identité, masse propre, position, niveau de portage (0 s'il est au sol, 1 + le niveau de son porteur immédiat sinon)
Lister les informations sur un objet
Lister les informations sur tous les objets (robots compris) situés dans une pièce.

Méthodes spécifiques aux robots:

Connaître leur charge maximale transportable et leur masse totale (y compris donc celles des objets portés)
Charger ou décharger un objet arbitraire en faisant les contrôles nécessaires
Déplacer un robot et les objets qu'il transporte en prenant en paramètre un déplacement relatif en x et en y.

1. Décrire un diagramme de classes UML permettant de modéliser le problème. Préciser les attributs avec leur type, les visibilités. Pour chaque association, précisez son nom de rôle et sa cardinalité. Précisez les classes et méthodes abstraites le cas échéant.
2. Écrire une méthode pour calculer la masse totale d'un robot et des objets qu'il transporte
3. Écrire une méthode pour lister tous les objets de la pièce. Pour chacun, on donnera son identifiant, sa position et sa masse (y compris celle des objets transportés pour les robots).
4. Écrire une méthode pour les méthodes de chargement et déchargement d'un objet. Seul un robot posé au sol peut charger ou décharger des objets. Il ne peut charger un objet que si celui-ci est posé au sol et ne peut décharger un objet que s'il le transporte directement. Un robot est muni d'un bras manipulateur qui lui permet de saisir un objet quelle que soit la position actuelle de ce dernier (i.e. on ne représente pas le fait que dans la réalité le robot devrait probablement se déplacer jusqu'à une certaine distance de l'objet à charger). Une fois chargé, l'objet a la même position que son robot porteur (le chargement peut donc induire un déplacement implicite). Quand on décharge un objet il est à la même position que son porteur.

5. Écrire une méthode `deplacer(dx, dy)` pour déplacer un robot posé sur le sol.
6. On considère maintenant un nouveau type d'objets : les sacs de sable. Quand un sac de sable est déplacé, il perd une portion du sable qu'il contient, proportionnellement à la distance sur laquelle il est transporté. Le taux de perte est propre à chaque sac. Modifier la hiérarchie proposée pour intégrer cette nouvelle contrainte. Modifier si besoin la méthode `deplacer`. Le code que vous aviez écrit pour calculer la masse d'un robot est-il toujours correct?

Seconde partie du sujet (*vers le modèle dit MVC = Model/View/Controler*)

Afin d'évaluer différentes stratégies de déplacement des robots, on ajoute des superviseur, chargés de détecter certains événements. On indique à un superviseur la liste des objets qu'il doit contrôler. Cette liste peut varier dans le temps. Un exemple serait de demander que le superviseur affiche un message à chaque fois qu'un objet est chargé à partir du sol ou déposé directement au sol par un robot. Un autre exemple serait d'afficher un message si l'objet supervisé change de position...

7. Modifier la hiérarchie proposée pour intégrer cette nouvelle contrainte. Indiquez le protocole de communication entre les objets et le superviseur. Idéalement on doit pouvoir considérer différents types de supervision (certaines liées à sa position verticale, d'autres à ses déplacements, ou à des modifications de masse, ou n'importe quelle combinaison arbitraire) et un même objet peut être la cible d'un nombre quelconque de supervisions et n'a pas à connaître les évènements qui intéressent ses superviseurs. **On doit pouvoir ajouter un nouveau type de supervision sans être obligé de modifier du code existant.**

Superviseurs à implémenter :

un superviseur pour détecter les chargements du sol / déchargement au sol (et seulement eux)
un superviseur pour détecter les déplacements (explicites ou liés à un chargement)
un superviseur pour détecter les pertes de masse des sacs
un superviseur pour détecter toute modification d'état d'un robot.

Méthodes à implémenter :

ajouter ou retirer un item (Sac, Caisse, Robot, ..) à un superviseur
alerter un superviseur qu'un événement a eu lieu pour un item. C'est le superviseur qui doit décider s'il est intéressé ou non par l'alerte.

On remarque que les 3 premiers superviseurs sont intéressés par certains passages d'un état donné à un autre. le dernier est intéressé par tout changement d'une partie de l'état.

Pour le TP, on pourra notamment de servir des classes et interfaces prédéfinies de l'API Java dont `Set`, `HashSet`, `Map` et `HashMap`.(voir les API dans la doc.)

L'interface générique `Set<E>` et son implémentation `HashSet` font partie de la hiérarchie `Collection` et définissent donc des méthodes telles que `add`, `remove`, `contains`, `size`, etc. Les implémentations de `Set` garantissent qu'un élément n'est présent qu'une fois dans la collection. La classe `HashMap<K, V>` implémente l'interface `Map<K, V>` et permet de représenter des fonctions associant une valeur de type `V` à une clef de type `K`. Les principales fonctions utiles ici sont `put(K key, V value)`, `remove(K key)` et `get(K key)`.

Troisième partie du sujet : modifiez la version précédente en se servant de l'interface `Observer` et de la classe `Observable` de l'API Java (voir les API dans la doc.)

Dans l'API Java : on peut ajouter/retirer un observateur à un objet « observable » via les méthodes `addObserver` et `deleteObserver`. Les observateurs sont prévenus des modifications via un appel à leur méthode `update(Observable o, Object arg)`[1] qu'ils doivent définir pour respecter l'interface `Observer`. L'application prévient (indirectement) les observateurs d'un objet qu'il y a eu modification sur cet objet en appelant la méthode `notifyObservers()` sur l'objet observable. Cet appel est traduit automatiquement en l'appel à la méthode `update` de chacun de ses observateurs mais **uniquement** si la méthode `hasChanged()` appliqué à cet objet renvoie `true`. Ceci permet éventuellement de n'appeler les observateurs qu'après qu'un ensemble de changements a été globalement appliqué à l'objet en question. Pour gérer cet indicateur, on dispose de deux méthodes `setChanged()` et `clearChanged()`. Il est à noter que l'appel à `notifyObservers` appelle automatiquement `clearChanged`. Il de votre ressort d'appeler à bon escient `setChanged` et `notifyObservers` de façon à bien récupérer les alertes de tous les observateurs et à ne pas appeler inutilement ceux-ci plusieurs fois.

On s'attachera à définir les mêmes observateurs que dans la section précédente. Les programmes de test pour les deux sections ne diffèrent que par la manière dont les observateurs sont associés aux objets qu'ils observent. Dans la version API Java, l'ordre dans lequel les observateurs sont appelés n'est a priori pas défini. Les deux programmes de test peuvent donc différer légèrement du point de vue de l'ordre dans lequel les alertes sont signalés - dans la première version, c'est vous qui avez de toutes façons défini cet ordre par votre programmation des méthodes.

1 Dans notre cas, on ignorera `arg` dans le corps de la fonction `update`!