Problème de la piscine

Après avoir défini les notions générales d'*action* et d'*ordonnanceur d'actions* dans le TD précédent, nous exploitons maintenant ces notions pour réaliser une application de simulation d'utilisation de ressources partagées, connue sous le nom du *problème de la piscine*.

Q 1 . Lisez l'ensemble du sujet et faites le lien entre le problème de la piscine présenté en fin de sujet et les notions d'*actions* du sujet précédent ainsi que celles de *ressources* et de *pool de ressources* mentionnées dans le sujet.

Pool de ressources

Une ressource (*resource*) est un objet dont la classe implémente l'interface suivante :

```
public interface Resource {
  String description();
}
```

Un gestionnaire de ressources dispose d'un certain nombre initial, paramétrable, de ressources. On parle de *pool* de ressources.
On doit pouvoir demander à un gestionnaire de ressources (`ResourcePool`) de fournir une ressource de son pool (en étant informé si aucune ressource n'est disponible) et on doit pouvoir rendre une ressource au pool pour la libérer et la rendre à nouveau disponible. Cette classe disposera (au moins) des méthodes suivantes :

▷ `provideResource` pour obtenir une des ressources du pool (peu importe laquelle) ; cette méthode a pour résultat la ressource obtenue. Une exception `NoSuchElementException` est levée s'il n'y a pas de ressource disponible;

▷ `recoverResource` pour indiquer qu'une ressource du pool a été libérée et donc « rendue » au pool. Cette ressource est passée en paramètre. Elle pourra donc être à nouveau obtenue auprès du pool.

Une exception `IllegalArgumentException` est levée si la ressource n'est pas une ressource précédemment créée par le gestionnaire (donc si la ressource n'avait pas été obtenue grâce à `provideResource` auprès de ce même pool).

Q 2 . On souhaite disposer de (au moins)[1] deux gestionnaires de ressources différents :

- un « gestionnaire de paniers » (`BasketPool`) qui est un gestionnaire de ressources qui fournit des ressources de type `Basket` (un *panier* pour mettre des vêtements), une classe supposée définie et implémentant l'interface `Resource`.
- un « gestionnaire de cabines » (`CubiclePool`) qui est un gestionnaire de ressources qui fournit des ressources de type `Cubicle` (une *cabine* pour se déshabiller à la piscine), une classe supposée définie et implémentant l'interface `Resource`.

Quels sont les tests communs et propres à chacun des gestionnaires de ressources décrit ci-dessus ?

Q 3 . Faites une proposition sous forme de diagramme UML pour définir ces types de gestionnaires de ressources.

Les constructeurs de ces gestionnaires prennent en paramètre le nombre de ressources gérées par le gestionnaire. Ces ressources **sont créées à la construction du gestionnaire** (ni avant ni après).

Q 4 . Donnez tout le code Java correspondant à votre proposition.

Commencez par donner le code des constructeurs des classes impliquées.

Actions

On souhaite définir une classe `TakeResourceAction` et une classe `FreeResourceAction` qui sont toutes les deux des actions (au sens du TD précédent). Ces actions vont s'adresser à un gestionnaire de ressources communiqué à la construction pour, comme on peut le deviner, respectivement prendre une ressource et libérer un ressource. L'utilisateur de la ressource, de type `ResourceUser` (son code est fourni en annexe), est également communiqué à la construction de l'action. C'est cet utilisateur qui prend et libère la ressource.
Les instances de `TakeResourceAction` ne se terminent (au sens « `isFinished` » du type `Action`) que quand la requête correspondante a abouti, c'est-à-dire que l'utilisateur a effectivement réussi à prendre une ressource. Il peut ne pas y avoir de ressource disponible à la première tentative. Une action peut donc prendre un certain temps, c'est-à-dire nécessiter plusieurs appels à `doStep()` avant d'être terminée.

Q 5 . Est-il nécessaire de tester les classes `TakeResourceAction` et `FreeResourceAction` ? Si oui, quels tests sont pertinents selon vous ?

[1] On doit pouvoir en ajouter d'autres « facilement ».

CC BY NC SA

Q 6 . Modélisez les classes `TakeResourceAction` et `FreeResourceAction` qui permettent à un `ResourceUser` de prendre, respectivement rendre, une ressource fournie par un `ResourcePool`.

Indiquez comment ces classes se raccrochent aux éléments précédents.

Q 7 . Donnez le code associé.

Réalisation d'une simulation : tous à la piscine !
Nous allons utiliser les différentes notions mises en œuvre dans les questions précédentes pour réaliser une simulation. Chaque personne voulant accéder à la piscine doit dérouler le scénario suivant : prendre un panier (*basket*), aller dans une cabine (*cubicle*), se déshabiller (*getting undressed*), libérer la cabine, se baigner (*swim*), retrouver une cabine, s'habiller (*getting dressed*), libérer sa cabine et rendre son panier.
Panier et cabine sont des ressources.
Un nageur (*swimmer*) est caractérisé par (même ordre que dans les appels au constructeur du code ci-après) :

1. son nom,
2. le gestionnaire des paniers à qui s'adresser,
3. le gestionnaire des cabines à qui s'adresser,
4. le temps qui lui est nécessaire pour se déshabiller,
5. la durée pendant laquelle il va se baigner,
6. le temps qui lui est nécessaire pour se rhabiller.

Il suffit, maintenant, qu'un ordonnanceur s'occupe de faire agir les différents nageurs en temps partagé pour produire la simulation du fonctionnement d'une piscine, comme cela est fait dans la classe `SwimmingPool` décrite ci-dessous :

```
public class SwimmingPool {

  public static void main(String[] args) throws ActionFinishedException {
    final BasketPool baskets = new BasketPool(6);
    final CubiclePool cubicles = new CubiclePool(3);
    final FairScheduler s = new FairScheduler();

    s.addAction(new Swimmer("Camille", baskets, cubicles, 6, 4, 8));
    s.addAction(new Swimmer("Lois", baskets, cubicles, 2, 10, 4));
    s.addAction(new Swimmer("Maé", baskets, cubicles, 10, 18, 10));
    s.addAction(new Swimmer("Ange", baskets, cubicles, 3, 7, 5));
    s.addAction(new Swimmer("Louison", baskets, cubicles, 18, 3, 3));
    s.addAction(new Swimmer("Charlie", baskets, cubicles, 3, 6, 10));
    s.addAction(new Swimmer("Alexis", baskets, cubicles, 6, 5, 7));

    int nbSteps = 0;
    while (!s.isFinished()) {
      nbSteps++;
      s.doStep();
    }

    System.out.println("Finished in " + nbSteps + " steps");
  }
}
```

Q 8 . Faites un diagramme de classes UML donnant une vue globale de l'architecture de cette simulation ;

Q 9 . Quelle suite de tests est pertinente pour la classe `Swimmer` ?

Q 10 . Donnez le code Java de la classe `Swimmer`.

Q 11 . Revisitez le code Java de la classe `SwimmingPool` pour adopter une conception dans laquelle il est possible d'enregistrer des nageurs dans la piscine (`registerSwimmer()`) et de lancer la simulation (`run()`) pour obtenir le nombre d'itérations nécessaires pour exécuter le scénario.

Appendice

Class `ResourceUser`

```
public class ResourceUser<R extends Resource> {
  protected R resource;

  public R getResource() {
    return resource;
  }

  public void setResource(R resource) {
    this.resource = resource;
  }

  public void resetResource() {
    this.resource = null;
  }
}
```

Trace d'exécution

Voici un bout de trace d'exécution pour la simulation :

```
Camille's turn
 Camille trying to take resource from pool basket... success
Loïs's turn
 Loïs trying to take resource from pool basket... success
Maé's turn
 Maé trying to take resource from pool basket... success
Ange's turn
 Ange trying to take resource from pool basket... success
Louison's turn
 Louison trying to take resource from pool basket... success
Charlie's turn
 Charlie trying to take resource from pool basket... success
Alexis's turn
 Alexis trying to take resource from pool basket... failed
Camille's turn
 Camille trying to take resource from pool cubicle... success
Loïs's turn
 Loïs trying to take resource from pool cubicle... success
Maé's turn
 Maé trying to take resource from pool cubicle... success
Ange's turn
 Ange trying to take resource from pool cubicle... failed
Louison's turn
 Louison trying to take resource from pool cubicle... failed
Charlie's turn
 Charlie trying to take resource from pool cubicle... failed
Alexis's turn
 Alexis trying to take resource from pool basket... failed
Camille's turn
 undressing (1/6)
Loïs's turn
 undressing (1/2)
Maé's turn
 undressing (1/10)
Ange's turn
 Ange trying to take resource from pool cubicle... failed
Louison's turn
 Louison trying to take resource from pool cubicle... failed
Charlie's turn
 Charlie trying to take resource from pool cubicle... failed
Alexis's turn
 Alexis trying to take resource from pool basket... failed
Camille's turn
```

```
 undressing (2/6)
Loïs's turn
 undressing (2/2)
Maé's turn
 undressing (2/10)
Ange's turn
[...]
Maé's turn
 Maé freeing resource from pool basket
Alexis's turn
 dressing (4/7)
Alexis's turn
 dressing (5/7)
Alexis's turn
 dressing (6/7)
Alexis's turn
 dressing (7/7)
Alexis's turn
 Alexis freeing resource from pool cubicle
Alexis's turn
 Alexis freeing resource from pool basket
Finished in 242 steps
```