

UE Conception Orientée Objet

Problème de la piscine

Dans ce problème nous définissons les notions générales d’*action* et d’*ordonnanceur d’actions* que nous exploiterons ensuite pour réaliser une application de simulation d’un exemple de protocole d’utilisation de ressources partagées connu sous le nom du *problème de la piscine*.

Actions et Ordonnanceur d’actions Nous définissons une *Action* comme un objet qui progresse de son état initial (*non commencée*) jusqu’à son état final (*terminée*) par application successive de sa méthode `faire()`.

Certaines actions passent de l’état *non commencée* à l’état *terminée* par une seule exécution de `faire()`, d’autres peuvent nécessiter un certain nombre d’exécutions de leur méthode `faire()`.

On peut considérer d’une certaine manière que le nombre fois où il faut appeler la méthode `faire` correspond au «temps que prend l’action à s’exécuter».

Q 1 . Donnez un code JAVA pour l’abstraction *Action*.

Q 2 . Donnez le code JAVA d’une classe *Attente*, dont les instances sont des actions qui ne sont terminées qu’au bout d’un nombre paramétrable d’invocations à leur méthode `faire()`. Une instance de *Attente* correspond donc à une action qui prend un «certain temps à s’exécuter».

Un *Ordonnanceur* possède un ensemble d’actions et on peut s’adresser à lui pour faire progresser une ou plusieurs de ces actions. En ce sens un ordonnanceur est donc une action dont la méthode `faire` consiste à faire progresser la «prochaine» action prévue. Un *Ordonnanceur* peut donc être vu comme une action composée¹. Un ordonnanceur est terminé quand toutes ses («sous-»)actions sont terminées.

Que fait un ordonnanceur quand on invoque sa méthode `faire()`? On peut imaginer en fait plusieurs modes de fonctionnement :

- *séquentiel* : dans ce cas, l’ensemble des sous-actions est complètement ordonné et l’ordonnanceur invoque la méthode `faire()` de la première de ses sous-actions non terminées.

Un tel ordonnanceur sera appelé *Scenario*.

- *en temps partagé* : ici, la progression des actions se fait en «*parallèle*» ; l’ordonnanceur dispose d’une action courante *a* (non terminée) dont il invoque la méthode `faire()`. Cette action est ensuite placée en fin de la liste des actions pour un prochain appel de la méthode `faire` de l’ordonnanceur lors duquel la première action non terminée qui suit *a* dans la liste devient l’action courante. On gère la liste des actions par une liste en anneau.

Un tel ordonnanceur sera appelé *Scheduler*.

Dans tous les cas, il faut prévoir la possibilité d’ajouter une nouvelle sous-action aux ordonnanceurs. Celle-ci est ajoutée en fin de liste (afin d’être cohérent avec la notion d’ordonnanceur *Scenario*).

Q 3 . Donnez un diagramme UML détaillé regroupant les différents types : *Action*, *Attente*, *Ordonnanceur*, *Scenario* et *Scheduler*.

Q 4 . Donnez le code JAVA des types *Ordonnanceur*, *Scenario* et *Scheduler*.

Ressources et Gestionnaire de ressources Une ressource sera un objet dont la classe implémente l’interface

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

On suppose définie la classe *RessourceUser* dont le code est fournie en annexe.

Un gestionnaire de ressources dispose d’un certain nombre initial paramétrable de ressources. Ces ressources sont créées à la construction du gestionnaire, on parle de *pool de ressources*.

On doit pouvoir demander à un gestionnaire de ressources de fournir une ressource de son pool (en étant informé si aucune ressource n’est disponible) et on doit pouvoir l’informer qu’une ressource est libérée.

Q 5 . Donnez le code d’une classe paramétrée (générique) *GestionnaireRessources* pour représenter ces gestionnaires.

Cette classe disposera (au moins) des méthodes :

- ▷ `prendreRessource` pour prendre une des ressources du pool; cette méthode a pour résultat la ressource prise. Une exception `NoSuchElementException` est levée si `ln’y` a pas de ressource disponible.

- ▷ `libererRessource` pour indiquer qu’une ressource du pool a été libérée, cette ressource est passée en paramètre. Une exception `RessourceInvalideException` est levée si la ressource n’est pas une ressource géré par le gestionnaire.

Q 6 . Donnez le code de la classe *RessourceInvalideException*.

Q 7 . Donnez le code d’une classe *GestionnaireDePaniers* qui fournit des ressources de type «*Panier*», une classe supposée définie implémentant l’interface *Ressource*.

Actions et Ressources On souhaite définir une classe *PrendreRessource* et une classe *LibererRessource* qui sont toutes les deux des *Action*. 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 une ressource. L’utilisateur de la ressource, de type *RessourceUser*, est également communiqué à la construction de l’action. C’est cet utilisateur qui prend et libère la ressource (celle qui aura été précédemment prise).

Les actions instances de *PrendreRessource* ne se terminent 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 *PrendreRessource* peut donc prendre un «certain temps», c’est-à-dire nécessiter plusieurs appels à `faire()` avant d’être terminée.

Q 8 . En factorisant au mieux, modéliser des classes *PrendreRessource* et *LibererRessource* qui permettent à un *RessourceUser* de prendre, respectivement libérer, une ressource fournie par un *GestionnaireRessources*.

Indiquez comment elles se «raccrochent» aux éléments précédents.

Q 9 . Donnez les codes associés.

Réalisation d’un simulation : tous à la piscine! Nous allons utiliser les différentes notions mises en oeuvre dans les questions précédentes pour réaliser une simulation.

Chaque personne voulant accéder à la piscine doit dérouler le scénario suivant : prendre un panier, aller dans une cabine, se déshabiller, libérer la cabine, se baigner, retrouver une cabine, se rhabiller, libérer sa cabine et rendre son panier.

Panier et cabine sont considérés comme des ressources.

Un nageur est caractérisée par (même ordre dans les constructeurs du code ci-après) :

- son nom,
- le gestionnaire des paniers à qui s’adresser,
- le gestionnaire des cabines à qui s’adresser,
- le temps qui lui est nécessaire pour se déshabiller,
- la durée pendant laquelle il va se baigner,
- le temps qui lui est nécessaire pour se rhabiller.

Il suffit, maintenant, qu’un scheduler 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 *Piscine* décrite ci-dessous :

```
public class Piscine {
    public static void main(String[] args) {
        GestionnaireDeRessources paniers = new GestionnaireDePaniers(3) ;
        GestionnaireDeRessources cabines = new GestionnaireDeCabines(2) ;
        Scheduler s = new Scheduler() ;
        s.ajouter(new Nageur("Jean" , paniers , cabines , 6 , 4 , 8)) ;
        s.ajouter(new Nageur("Paul" , paniers , cabines , 2 , 10 , 4)) ;
        s.ajouter(new Nageur("Bruno" , paniers , cabines , 10 , 18 , 10)) ;
        s.ajouter(new Nageur("Marcel" , paniers , cabines , 3 , 7 , 5)) ;
        s.ajouter(new Nageur("Anatole" , paniers , cabines , 18 , 3 , 3)) ;
        s.ajouter(new Nageur("Clement" , paniers , cabines , 3 , 6 , 10)) ;
        s.ajouter(new Nageur("Desire" , paniers , cabines , 6 , 5 , 7)) ;
        while (! s.termine()) s.faire() ;
    }
}
```

Q 10 . Après avoir indiqué comment elles s’intègrent dans l’ensemble des classes définies précédemment, codez les classes *Panier* et *Cabine* qui implémentent l’interface *Ressource* et les classes *GestionnaireDePaniers* et *GestionnaireDeCabines* qui sont les gestionnaires de ressources associées,

Q 11 . Donnez le code JAVA de la classe *Nageur*.

Il convient évidemment de prendre en compte la gestion de l’utilisation des ressources panier et cabine par le nageur.

¹Rien n’empêche donc un ordonnanceur d’avoir parmi ses «sous-actions» un autre ordonnanceur.

Annexe

```
package ressource;
/** La classe des utilisateurs de ressource.
 *
 * @param <R> le type de ressource utilisée
 */
public class RessourceUser<R extends Ressource> {

    /** La ressource utilisée */
    protected R ressource;
    /**
     * @return la ressource actuellement utilisée, <t>>null</t> si aucune
     */
    public R getRessource() {
        return this.ressource;
    }
    /** fixe la ressource utilisée
     * @param ressource la ressource
     */
    public void setRessource(R ressource) {
        this.ressource = ressource;
    }
    /**
     * réinitialise la ressource utilisée (mise à <t>>null</t>)
     */
    public void resetRessource() {
        this.ressource = null;
    }
}
```