Actions

Dans ce TD nous définissons les notions générales d'action et d'ordonnanceur d'actions (*scheduler*). Dans le TD suivant, nous exploiterons ces notions pour réaliser une application de simulation d'utilisation de ressources partagées, connue sous le nom du *problème de la piscine*.

Actions. Une **action** est définie comme un objet qui progresse de son état initial (*ready*) jusqu'à son état final (*finished*) par *appels successifs* de sa méthode `doStep()`. Une action commencée se trouve dans l'état *en cours* (*in progress*) tant qu'elle n'est pas terminée.
Certaines actions passent de l'état initial à l'état final par une seule exécution de `doStep()`, d'autres peuvent nécessiter plusieurs exécutions de cette méthode.
On peut considérer d'une certaine manière que le nombre de fois où il faut appeler la méthode `doStep()` correspond au "temps que prend l'action à s'exécuter". Chaque appel de `doStep()` représente le passage d'une unité de temps pendant laquelle une action peut avancer d'une et une seule étape.
Certaines actions (les *foreseeable action*) ne sont terminées qu'au bout d'un nombre connu par avance d'invocations de `doStep()`. Une action de ce type prend donc un "certain temps à s'exécuter".
Il peut exister d'autres actions pour lesquelles le nombre d'invocations de `doStep()` nécessaire avant d'atteindre l'état final n'est pas fixé ni connu à l'avance. Il est possible de déterminer si une action est complétée grâce à l'invocation de la fonction `isFinished()` qui retourne `true` si la méthode `doStep()` ne doit plus être invoquée. L'exécution de la méthode `doStep` pour une action complétée doit lever une exception.

Ordonnanceurs. On s'intéresse également à une catégorie particulière d'actions appelées **ordonnanceur** (*scheduler*). Un ordonnanceur est une action qui possède un ensemble d'actions (ces « *sous-actions* » peuvent donc être aussi des ordonnanceurs). Il a la responsabilité de faire progresser ces actions pas-à-pas jusqu'à ce qu'elles soient terminées. Un ordonnanceur est terminé quand toutes ses sous-actions sont terminées. Il faut prévoir la possibilité d'ajouter une nouvelle sous-action non terminée à un ordonnanceur tant qu'il est dans son état initial.
La méthode `doStep()` d'un ordonnanceur consiste à faire progresser sa « *prochaine* » action prévue d'un seul pas. Chaque appel à `doStep()` sur un ordonnanceur **fait avancer une seule de ses actions d'une et une seule étape**. Il faudra en général appeler de nombreuses fois la méthode `doStep()` d'un ordonnanceur pour terminer toutes les sous-actions, et donc l'ordonnanceur, mais on ne peut pas connaitre à l'avance ce nombre de fois.

Q 1 . Le code suivant est une implémentation possible de la notion d'action.

Lisez ce code, et proposez une suite de tests applicables à ce code pour vérifier que la notion d'action telle que décrite ci-dessus est correctement implémentée.

```
public enum ActionState { READY, IN_PROGRESS, FINISHED; }
public class BadAction {

  protected ActionState state;
  protected final boolean isScheduler;

  // these fields are only for foreseeable action (isScheduler == false)
  protected final int totalTime;
  protected int remainingTime;

  // this field is only used for schedulers (isScheduler == true), not for foreseeable
  protected final ArrayList<BadAction> actions = new ArrayList<BadAction>();

  /**
   * Either create a foreseeable action or a scheduler based on the value of
   * <code>timeToEnd</code>.
   *
   * @param timeToEnd
   *            For a foreseeable action, indicate the number of <code>doStep()</code> calls
   *            required. Value 0 tells action is a scheduler.
   */
  public BadAction(int timeToEnd) {
    this.totalTime = timeToEnd;
    this.state = ActionState.READY;
    if (timeToEnd == 0) {
      this.isScheduler = true;
    } else {
      this.isScheduler = false;
      this.remainingTime = timeToEnd;
    }
```

```
  }

  /**
   * Create a scheduler, warning : never a foreseeable action
   */
  public BadAction() {
    this(0);
  }

  public ActionState getState() {
     return this.state;
  }

  public boolean isFinished() {
     return this.state == ActionState.FINISHED;
  }

  public void doStep() throws ActionFinishedException {
    if (this.state == ActionState.FINISHED) {
       throw new ActionFinishedException("cannot doStep when finished.");
    }
    if (this.state == ActionState.READY) {
       this.state = ActionState.IN_PROGRESS;
    }
    // make one step
    if (! this.isScheduler) {
      this.remainingTime--;
    } else {
      if (! actions.isEmpty()) {  // happens if no action added to the scheduler before first doStep
         BadAction nextAction = actions.get(0);
         nextAction.doStep();
         if (nextAction.isFinished()) {
           this.actions.remove(0);
         }
      }
    }
    // stop condition
    boolean stop =false;
    if (! this.isScheduler) {
        stop = this.remainingTime <= 0;
      } else {
        stop = (this.state == ActionState.IN_PROGRESS && this.actions.isEmpty());
      }
    if (stop) {
       this.state = ActionState.FINISHED;
    }
  }

  //please don't use this if the action is not a scheduler !!!
  public List<BadAction> getScheduerActions() {
    return this.actions;
  }

  // please don't use this method if the action is not a scheduler !!!
  public void addAction(BadAction subAction) throws ActionFinishedException, SchedulerStartedException {
    if (this.state != ActionState.READY) {
       throw new SchedulerStartedException("Cannot add when scheduler is in progress or has finished");
    }
    if (subAction.isFinished()) {
      throw new ActionFinishedException("Cannot add an already finished action");
    } else {
      this.actions.add(subAction);
    }
  }
}
```

Q 2 . Au regard du code proposé ci-dessus, identifiez ses défauts et proposez des corrections applicables en quelques mots.

Q 3 . Proposez une nouvelle architecture corrigeant ces défauts et permettant d'anticiper l'ajout de nouveaux types d'actions, en plus des *foreseeable* et des ordonnanceurs.

Q 4 . Mettez en œuvre les tests que vous avez identifiés en question 1 pour cette proposition. Comment faire pour que ces tests soient factorisés et exécutés, quelque soit le type d'action considérée ?

Écrivez ces tests.

Q 5 . Comment vérifier qu'une `ForeseeableAction` de durée n est terminée après exactement n appels à `doStep()`?

Stratégies d'ordonnancement

Dans la description précédente d'ordonnanceur, il peut exister plusieurs interprétations possibles à l'expression « *la prochaine action prévue* » :

- *séquentiel* (*sequential scheduler*) : dans ce cas, la prochaine action prévue est celle qui vient d'être exécutée si elle n'est pas encore terminée et la suivante sinon. Ce type d'ordonnanceur termine donc l'action commencée avant de passer à la suivante. C'est ce type d'ordonnanceur qui était mis en œuvre dans la version `BadAction` ;
- *en temps partagé* (*fair scheduler*) : ici, la prochaine action prévue est celle qui suit l'action qui vient d'être exécutée. La progression des actions se fait en *parallèle*. L'ordonnanceur dispose d'une action courante a (non terminée) dont il invoque la méthode `doStep()`. Lors du prochain appel de la méthode `doStep()` de l'ordonnanceur, celui-ci fera avancer la première action non terminée qui suit a. L'ordonnanceur reviendra à a quand il aura invoqué une fois la méthode `doStep()` sur chacune de ses actions. Puis, il fera avancer la première action non terminée qui suit a et ainsi de suite.

Q 6 . Quels tests devez-vous ajouter pour prendre en compte ces différents ordonnanceurs ?

Q 7 . Reconsidérez votre architecture pour prendre en compte ces différents ordonnanceurs.

Q 8 . Écrivez le code Java des méthodes `doStep()` des différentes actions. Étudiez votre code et faites le nécessaire pour factoriser le maximum de code.

Q 9 . Pour les ordonnanceurs, on souhaite également vérifier que :

- il y a une action de plus à gérer après un appel à `addAction()` sans problème ;
- une exception `SchedulerStartedException` est déclenchée si l'on ajoute une action alors que l'ordonnanceur a démarré ;
- une exception `ActionFinishedException` est déclenchée si l'on ajoute une action terminée à un ordonnanceur ;
- quand un ordonnanceur est terminé, toutes ses actions sont terminées.

Ces tests doivent être vérifiés quelque soit l'ordonnanceur.

Quand on teste une classe, on essaie de la tester en isolation des autres classes. Quand on respecte cette pratique, si un test échoue, on sait que le problème est soit dans la classe testée, soit dans le test lui même. Il ne peut pas être ailleurs dans le système.

Comment rendre indépendant les tests sur les ordonnanceurs de toute autre classe d'actions ?

Écrivez les tests demandés pour les ordonnanceurs.

Q 10 . Écrivez un test qui vérifie qu'un ordonnanceur séquentiel traite bien ses sous-actions pas-à-pas, les unes après les autres.

Q 11 . Écrivez un test qui vérifie qu'un ordonnanceur en temps partagé traite bien ses sous-actions pas-à-pas « en parallèle ».