UE Conception Orientée Objet

Devoir Surveillé

1 h 30

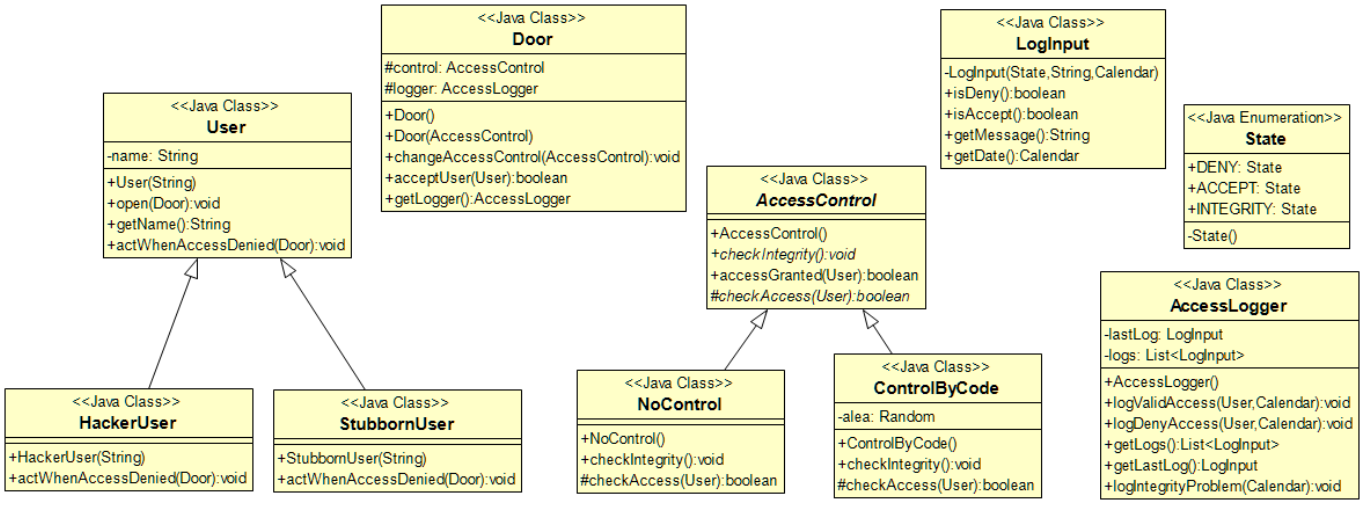
Copie des diapositives de cours annotée autorisée.
Dictionnaires de langue autorisés.
Autres documents interdits.

Indiquez sur votre copie *groupe INFO-X* ou *groupe MIAGE-X* où *X* est votre numéro de groupe.

Sauf mention expresse, la javadoc et les tests des classes à écrire ne sont pas demandés.

Exercice 1 :

On s'intéresse à une application de gestion de contrôle d'accès sur des portes dans un bâtiment. Les diagrammes de classe ci-dessous présentent les différents types impliqués:



- Les utilisateurs de type `User` peuvent ouvrir des portes de type `Door` grâce à la méthode `open`. Il y a différents types d'utilisateurs qui se différencient par un comportement que l'on pourrait appliquer dans le cas où l'accès à une porte est refusée. C'est le rôle de la méthode `actWhenAccessDenied()`. Il y a deux sous-classes de `User` proposées pour le moment, `HackerUser` et `StubbornUser`, mais d'autres sont possibles.

- Chaque porte possède un système de contrôle d'accès de type `AccessControl` et journalise les différentes tentatives d'accès dans un journal de type `AccessLogger` que l'on peut obtenir par la méthode `getLogger`. La dernière entrée d'un journal est accessible grâce à la méthode `getLastLog()`.

  C'est la méthode `acceptUser()` qui indique si l'utilisateur passé en paramètre est autorisé à ouvrir la porte.

- Les entrées d'un journal sont de type `LogInput`. On peut savoir si une telle entrée correspond à un accès autorisé ou refusé grâce aux méthodes `isAccept()` ou `isDeny()`.

- La classe `AccessControl` est abstraite. L'appel à la méthode finale `accessGranted()` permet de savoir si l'utilisateur passe avec succès ou non le contrôle d'accès.

  Pour éviter tout violation du système de contrôle, une vérification de l'intégrité du système est faite à chaque tentative d'accès[1]. Cette vérification est faite grâce à un appel à la méthode `checkIntegrity()` avant tout appel à la méthode `checkAccess()` qui doit quant à elle vérifier l'autorisation d'accès.

  La méthode `checkIntegrity()` signale que l'intégrité n'est pas vérifiée en levant une exception `IntegrityException`.

Q 1. Dans chacun des cas suivants, écrivez la ou les méthode(s) de test qui permettent de vérifier que :

[1] peu importe comment ou ce que fait cette vérification dans ce sujet.

Q 1.1. « *la méthode* `open` *de* `User` *lève une exception* `AccessDeniedException` *si et seulement si l'accès n'est pas accordé* ».

Faites en sorte d'assurer que les classes `HackerUser` et `StubbornUser` passent ce test, ainsi que toute autre sous-classe éventuelle de `User`.

Q 1.2. « *l'exécution de* `acceptUser` *provoque l'ajout d'une entrée dans le journal de la porte. Il s'agit d'une entrée de type* accept *si l'accès est autorisé et de type* deny *sinon* ».

Q 1.3. « *chaque appel à* `accessGranted()` *de* `AccessControl` *provoque un appel à* `checkIntegrity()` ».

Si vous trouvez une solution, faites en sorte que ce test vérifie en plus que « *l'appel à* `checkIntegrity()` *se fait toujours avant celui de* `checkAccess` », sinon ne testez que l'existence de l'appel à `checkIntegrity()`.

NB : n'écrivez pas plus de tests que ceux demandés ! Inutile de tester d'autres fonctionnalités du système que ce qui est demandé dans les questions ci-dessus !

Exercice 2 :

L'interface `Cumulable` permet de définir des éléments qu'il est possible *de cumuler* deux à deux. En voici le code :

```
public interface Cumulable<T> {
    T cumulate(T other);
}
```

Par exemple, on peut considérer une classe `Duration` pour représenter une durée en heures, minutes, secondes. La classe `Duration` pourrait implémenter l'interface `Cumulable` :

```
public class Duration implements Cumulable<Duration> { ...
```

Cumuler deux durées consiste à créer une nouvelle durée otenue en additionnant leurs nombres d'heures, minutes et secondes.
De même, une classe `StudentGroup` qui possède une liste d'étudiants peut implémenter cette interface. Cumuler deux groupes consiste à produire un nouveau groupe dont la liste est l'union des listes de ces deux groupes.

Pour manipuler les données cumulables (au sens `Cumulable`), on souhaite disposer d'un type `Aggregator` qui dispose de trois méthodes :

- `aggregate()` qui prend en paramètres deux données cumulables de même type et fournit la nouvelle donnée obtenue en résultat de leur cumul;
- `aggregateAll()` qui prend en paramètre une liste non vide de données cumulables et fournit en résultat la donnée obtenue en cumulant l'ensemble des éléments de cette liste. Une exception `RuntimeException` est levée si la liste en paramètre est vide;
- `iterAggregate()` qui prend en paramètre deux listes de données cumulables et fournit en résultat la liste obtenue en cumulant deux à deux les éléments de même position de chacune des listes. Une exception `RuntimeException` est levée si les deux listes ne sont pas de même longueur.

Q 1. Donnez le code java pour la classe `Aggregator` en respectant les contraintes suivantes :

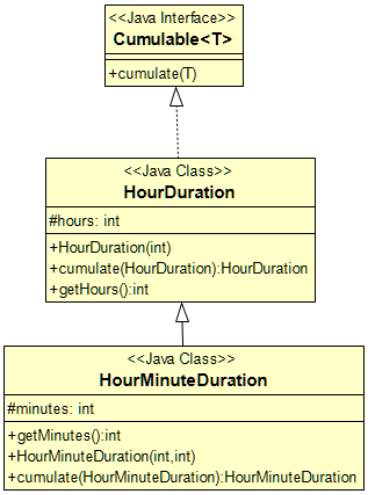
↪ Votre code doit couvrir les spécifications décrites ci-dessus ;

↪ Votre code doit passer avec succès les méthodes de tests données ci-dessous dans lesquelles `hourAggregator` est une référence vers un objet agrégateur correctement déclaré et initialisé (càd. les tests compilent et s'exécutent normalement).

Sachant que (voir diagramme ci-contre) :

▷ le type `HourDuration` représente des durées uniquement exprimées en heures. Elles se cumulent en ajoutant leurs heures ;

▷ le type `HourMinuteDuration` représente des durées exprimées en heures et minutes. Elles se cumulent en ajoutant leurs heures et leurs minutes.



Q 2. Calculez et commentez la *complexité cyclomatique* de chaque méthode de la classe `Aggregator`.

Q 3. Proposez et documentez une nouvelle version des tests unitaires décrits ci-dessous (refactoring) pour en améliorer la maintenabilité.

Il ne vous est pas demandé d'écrire de nouveaux tests ni de modifier la logique de ceux proposés[2] mais bien **uniquement** de proposer une réécriture de ces tests pour en améliorer le design.

Il n'est pas nécessaire de recopier tout le code. Vous pouvez ne fournir que les nouveaux éléments de code à introduire en documentant leur impact sur le code existant.

```
@Test
public void testAggregate() {
  HourDuration hd1 = new HourDuration(3);
  HourDuration hd2 = new HourDuration(5);

  HourDuration result = hourAggregator.aggregate(hd1, hd2);
  assertEquals(8, result.getHours());

  HourMinuteDuration hmd = new HourMinuteDuration(6,3);
  HourDuration result2 = hourAggregator.aggregate(hd1, hmd);
  assertEquals(9, result2.getHours());
}

@Test
public void testAggregateAll() {
  List<HourDuration> lhd = new ArrayList<>();
  // initialize with 10 first integers
  for(int i = 0; i < 11; i++) {
    lhd.add(new HourDuration(i));
  }
  HourDuration result = hourAggregator.aggregateAll(lhd);
  // sum of ten first integers = (10*11)/2
  assertEquals(55, result.getHours());
}

@Test
public void testIterAggregateWithSameTypeElements() {

  List<HourDuration> lhd = new ArrayList<>();
  for(int i = 0; i < 10; i++) {
    lhd.add(new HourDuration(i));
  }
  List<HourDuration> result = hourAggregator.iterAggregate(lhd,lhd);
  Iterator<HourDuration> it_result = result.iterator();
  for(int i = 0; i < 10; i++) {
```

[2]Par exemple vous devez maintenir le fait que les tests utilisent les classes `HourDuration` et `HourMinuteDuration`.

```
    assertEquals(2*i , it_result.next().getHours());
  }
}

@Test
public void testIterAggregateWithSubtypeElements() {

  List<HourDuration> lhd = new ArrayList<>();
  // initialize with  integers from 0 to 9
  for(int i = 0; i < 10; i++) {
    lhd.add(new HourDuration(i));
  }
  // initialize with  integers from 1 to 10
  List<HourMinuteDuration> lhmd = new ArrayList<>();
  for(int i = 0; i < 10; i++) {
    lhmd.add(new HourMinuteDuration(i+1,i*5));
  }

  List<HourDuration> result = hourAggregator.iterAggregate(lhd,lhmd);
  Iterator<HourDuration> it_result = result.iterator();
  for(int i = 0; i < 10; i++) {
    assertEquals(2*i +1 , it_result.next().getHours());
  }
}
```