IN201 : Conception et programmation orientées objet

Examen

Cet examen est composé de 8 exercices indépendants. Le barême indiqué pour chaque exercice l'est à titre indicatif et peut être modifié.
Les seuls documents autorisés pour cet examen sont :

- les cartes de référence distribuées en cours
- une feuille recto-verso de notes manuscrites au format A4

Les annales des examens des années précédentes sont interdites. Les téléphones portables doivent être éteints et rangés. L'utilisation d'un ordinateur durant l'examen est interdite.
Vous avez normalement la place de répondre sur ce document. Si vous devez utiliser une copie libre, n'oubliez pas d'y indiquer vos noms, prénoms et groupe de PC, ainsi que le numéro précis de la question à laquelle vous répondez.
Il est fortement conseillé de lire complétement un exercice avant d'y répondre.

Nom : _____

Prénom : _____

Groupe : _____

Question	Points	Score
1	5	
2	2	
3	2	
4	2	
5	3	
6	2	
7	2	
8	2	
Total :	20	

Commentaires éventuels :

1. On considère un ensemble de symboles représentés par le type **char** (disponible en UML et en Java). On utilisera « symbole » ou « caractère » indifféremment dans le texte qui suit.

Un *automate fini déterministe* (AFD) est un graphe *orienté* particulier dans lequel :

- il y a un nombre fini de nœuds
- les nœuds représentent des états
- chaque état possède un nom
- les arcs sont étiquetés par un seul caractère et sont appelés ici *transitions*
- un nœud particulier représente l'état initial de l'AFD
- un ensemble de nœuds particulier représente les états finaux de l'AFD
- deux transitions ayant pour origine le même nœud ne peuvent pas être étiquetées avec le même caractère

Les automates permettent de reconnaître des mots particuliers formés avec les caractères que l'on a à disposition (**char** ici). Pour reconnaître un mot, on part de l'état initial de l'automate et on suit les transitions en consommant les carcatères apparaissant dans le mot. Si on ne peut pas emprunter une transition (le caractère à consommer n'apparaît dans aucune transition ayant le nœud courant comme origine), le mot n'est pas reconnu. Si on arrive à consommer tous les caractères du mot à reconnaître, si on est dans un des états finaux de l'automate, alors le mot est reconnu.

Par exemple, considérons l'AFD qui reconnaît les mots respectant le motif $a^n bc$ où $n \geq 0$, i.e. les mots commençant par un certain nombre de a (ce nombre peut être nul) et qui finissent obligatoirement par bc. L'automate peut être représenté de la façon suivante (l'état initial est ici q_0 et l'état final est représenté par un double cercle) :

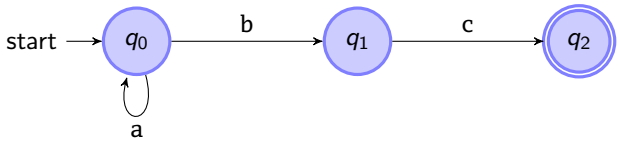

FIGURE 1 – Un automate reconnaissant les mots $a^n bc$ avec $n \geq 0$

En utilisant la notation $(q_i, \mathtt{xw}) \rightarrow (q_j, \mathtt{w})$ pour représenter le fait que l'on est passé de l'état q_i à l'état q_j en consommant le caractère x et qu'il reste le mot w à reconnaître à partir de q_j et en considérant que ϵ représente le mot vide (il n'y a plus de caractères à consommer), voici quelques exemples de mots reconnus ou non reconnus par l'automate précédent :

- aabc est un mot reconnu car $(q_0, \mathtt{aabc}) \rightarrow (q_0, \mathtt{abc}) \rightarrow (q_0, \mathtt{bc}) \rightarrow (q_1, \mathtt{c}) \rightarrow (q_2, \epsilon)$ et q_2 est un état final de l'AFD.
- ab n'est pas un mot reconnu car $(q_0, \mathtt{ab}) \rightarrow (q_0, \mathtt{b}) \rightarrow (q_1, \epsilon)$ et q_1 n'est pas un état final de l'AFD.
- abb n'est pas un mot reconnu car $(q_0, \mathtt{abb}) \rightarrow (q_0, \mathtt{bb}) \rightarrow (q_1, \mathtt{b})$ et il n'y a pas d'arc partant de q_1 étiqueté par b.

(1 pt) (a) proposer un diagramme UML présentant les classes `AFD`, `Etat` et `Transition`, leurs éventuels attributs, leurs méthodes, leur(s) constructeur(s) et les relations existant entre elles. Si vous souhaitez introduire des accesseurs et modifieurs sur des attributs qui sont nombreux, vous pouvez en écrire un seul et mettre « ... » pour les autres. On réfléchira plus particulièrement à quelle est (sont) la classe (les classes) devant contenir la méthode permettant de vérifier si un mot (représenté sous la forme d'une chaîne de caractères) est reconnu ou non par l'AFD.

Dans les questions suivantes, vous serez amenés à écrire du code Java. Il n'est pas nécessaire d'écrire la documentation Javadoc des classes, sauf si vous voulez préciser la signification d'un paramètre particulier. Comme précédemment, si vous devez écrire des accesseurs ou des modifieurs triviaux dans une classe, vous pouvez n'en écrire qu'un seul et préciser par un commentaire que les autres accesseurs et modifieurs seront écrits de la même façon.
Les méthodes suivantes de la classe `String` pourront être utiles :

- `length()` renvoie la longueur de la chaîne de caractères
- `charAt(int index)` renvoie le caractère situé à l'indice `index` dans la chaîne (les indices commencent à 0)

(½ pt) (b) écrire la classe `Transition` en Java

(1 pt) (c) écrire la classe classe `Etat` en Java

(1 pt) (d) écrire la classe `AFD` en Java

(1 pt) (e) écrire un programme dans une classe `Main` construisant l'automate présenté sur la figure 1 et affichant si le mot "aaaabc" est reconnu par l'automate.

(½ pt) (f) proposer une solution empêchant la construction d'automates non déterministes. On pourra éventuellement modifier la signature de méthodes ou de constructeurs existants.

2. On dispose d'une classe `Processor` possédant une méthode `process` prenant un **`int`** en paramètre et renvoyant un **`double`**. On souhaite tester la méthode `process`. Pour cela, on dispose d'un certain nombre de valeurs attendues en sortie en fonction de la valeur du paramètre de `process` :

entrée	résultat
1	5,900
2	−3,475
10	13,000
16	−23,675

(1 pt) (a) écrire une classe de test JUnit pour tester la méthode `process`. On suppose que l'on souhaite avoir une précision à 10^{-6} près pour `process` et que le constructeur de `Processor` ne prend pas d'argument.

(1 pt) (b) supposons que l'on ait un grand jeu de données à tester. Il n'est alors pas très judicieux de faire de un appel explicite à `process` pour chaque couple de données. En utilisant des structures de données adéquates pour stocker les jeux de tests, réécrire la classe de test pour simplifier la gestion de grands volumes de données.

(2 pt) **3.** On désire modéliser un système d'authentification d'utilisateurs souhaitant accéder à des documents classifiés selon différents niveaux. Un scénario particulier de récupération d'un document classifié « Confidentiel » par un utilisateur extérieur est le suivant :

- l'utilisateur se connecte au serveur d'authentification. Celui-ci lui demande alors son login et mot de passe ;
- l'utilisateur envoie son login et son mot de passe crypté au serveur ;
- celui-ci vérifie alors que le login et le mot de passe sont corrects ;
- le serveur d'authentification indique au système que l'utilisateur est bien « valide » en ouvrant une session sur le système pour l'utilisateur ;
- le système envoie un signal à l'utilisateur pour lui signaler qu'il est en attente d'une commande de sa part ;
- l'utilisateur demande au système l'accès et la sauvegarde d'un document classifié sur son compte (il précise le nom du fichier correspondant au document lors de la demande) ;
- le système demande le niveau de classification du document au serveur de fichiers le contenant, puis vérifie que l'utilisateur est habilité à récupérer ce type de document grâce au serveur d'authentification. Il renvoie ensuite le document à l'utilisateur.

Représenter sous forme d'un diagramme de séquence le scénario précédent.

4. On considère les deux classes `A` et `B` suivantes et le programme applicatif défini dans la classe `Main` :

```
1  class A {
2
3      void aff(double d) {
4          System.out.println("affichage de " + d + " depuis A");
5      }
6  }
7
8  class B extends A {
9
10     @Override void aff(double d) {
11         System.out.println("affichage de " + d + " depuis B.aff");
12     }
13
14     void affbis(double d) {
15         super.aff(d);
16         System.out.println("affichage de " + d + " depuis B.affbis");
17     }
18 }
19
20 class Main {
21
22     public static void main(String[] args) {
23         A objA = new A();
24         A objB = new B();
25
26         objA.aff(1.0);
27         objB.aff(2.0);
28         objB.affbis(3.0);
29     }
30 }
```

(1 pt) (a) corriger le programme défini dans `Main` pour qu'il puisse compiler et s'exécuter. On réécrira simplement la ou les lignes nécessitant d'être modifiées.

(1 pt) (b) qu'est-ce qui sera affiché sur la console lors de l'exécution de `Main` après correction ?

(3 pt) **5.** On cherche ici à écrire une application objet permettant d'optimiser des fonctions via des algorithmes génétiques. Les algorithmes génétiques sont des algorithmes de recherche ou d'optimisation utilisant le processus de sélection naturelle. Ils permettent de calculer la solution la plus optimale possible à un problème suivant un critère d'évaluation donné. Pour se faire, on va faire évoluer des populations de solutions à la manière de la théorie de l'évolution : en sélectionnant les solutions les meilleures (suivant le critère donné), en les mélangeant pour obtenir de nouvelles solutions et un ajoutant un degré d'aléa grâce à la mutation de certaines solutions.

Le *solver* permettant de résoudre un problème avec des algorithmes génétiques utilise plusieurs modules que nous détaillerons dans ce qui suit[1].

Le premier module concerne la modélisation du problème. La première chose à faire lorsque l'on utilise des algorithmes génétiques est de pouvoir représenter une solution à ce problème sous forme d'un chromosome constitué de gènes. Ces gènes représentent une donnée du problème et peuvent donc être typés : gène représentant un nombre, un entier, un réel, une chaîne de caractères, un booléen par exemple.

Les différents chromosomes sont ensuite regroupés dans une population qui est utilisée par le *solver*.

Pour que les algorithmes génétiques fonctionnent, on a besoin d'une fonction de *fitness* permettant de calculer sous forme d'une valeur entière quelle est la valeur d'une solution à partir de son chromosome. On peut bien sûr proposer plusieurs fonctions de *fitness*.

En utilisant cette fonction de *fitness*, on peut ensuite sélectionner quelles sont les solutions que l'on va retenir pour construire une prochaine population de chromosomes. Pour cela, différents mécanismes de sélection, appelés sélecteurs, sont disponibles : choix des meilleures valeurs absolues pour chaque chromosome, utilisation d'un système de tournoi entre chromosomes etc.

Enfin, on peut effectuer des opérations sur une population de chromosomes. Les opérations de base sont :
- la *reproduction* qui permet de copier une solution potentielle ;
- le *crossover* qui permet de mélanger les gènes de deux solution potentielles ;
- la *mutation* qui permet d'altérer aléatoirement la valeur d'un gène d'un chromosome.

On pourra bien sûr ajouter facilement de nouvelles opérations sur les chromosomes, comme des opérateurs de mutation gaussiens etc. Il faudra en tenir compte dans la conception du *solver*.

Proposer un diagramme de classes d'analyse représentant le domaine étudié. On fera apparaître les classes, les relations entre classes, les noms de rôles et les multiplicités des associations, et on justifiera l'utilisation de classes abstraites et d'interfaces. On pourra utiliser quelques attributs et représenter quelques méthodes si nécessaire. Enfin, on cherchera à avoir une solution la plus générique possible, i.e. permettant l'ajout de nouvelles fonctionnalités facilement (fonctions de *fitness*, sélecteurs, opérations sur les chromosomes).

1. Le terme module est utilisé ici de façon très générale, il ne faut pas le prendre au sens informatique.

6. On suppose ici que l'on s'intéresse à des algorithmes de plus court chemin développés dans le cadre d'un projet. Pour cela, on dispose de l'interface et des classes définies sur la figure 3 :

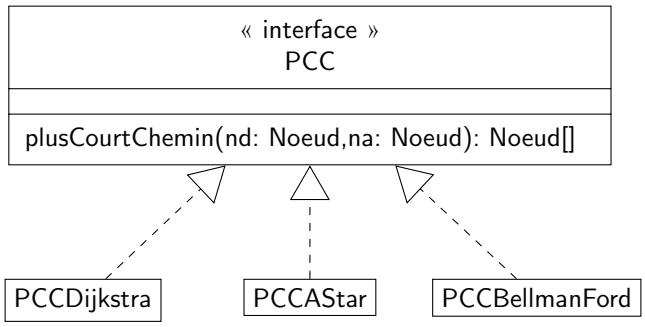

FIGURE 3 – L'interface PCC et les classes de calcul de plus court chemin

On suppose que l'on dispose d'une implantation en Java de l'interface PCC et des classes la réalisant, mais que l'on n'a pas accès à leur code source. On ne peut donc pas modifier les classes existantes.

On cherche à mesurer le temps d'exécution de chacun des algorithmes de plus court chemin et à afficher ce temps d'exécution sur la console. On veut donc ajouter cette mesure au calcul du plus court chemin. Pour cela, on peut utiliser la méthode *statique* `currentTimeMillis()` de la classe `System` qui renvoie un entier représentant le date courante en ms. En faisant un appel au début de la recherche du plus court chemin et à la fin, on peut ainsi calculer le temps d'exécution de l'algorithme et l'afficher sur la console.

(½ pt) (a) quel est le mécanisme le plus simple (vu en cours) permettant d'ajouter la mesure du temps de calcul du plus court chemin à chacun des algorithmes implantés ? Cette solution est-elle pertinente ? Pourquoi ?

(½ pt) (b) pour pallier le problème précédent, on peut utiliser un patron conception appelé *décorateur*. L'idée du décorateur est de pouvoir changer le comportement d'un objet en l'encapsulant dans un autre objet. Un diagramme de classe présentant le décorateur est donné sur la figure 4 et un diagramme de séquence expliquant son fonctionnement est donné sur la figure 5.

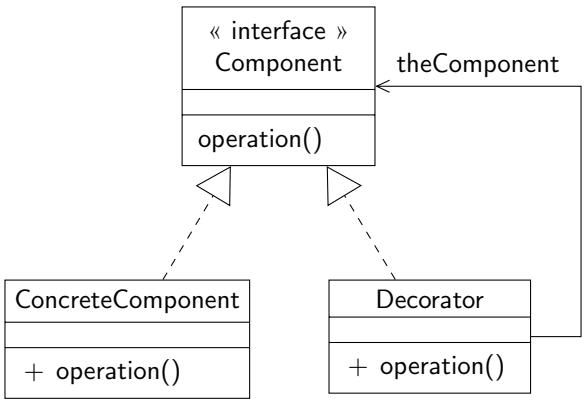

FIGURE 4 – Le patron de conception Décorateur

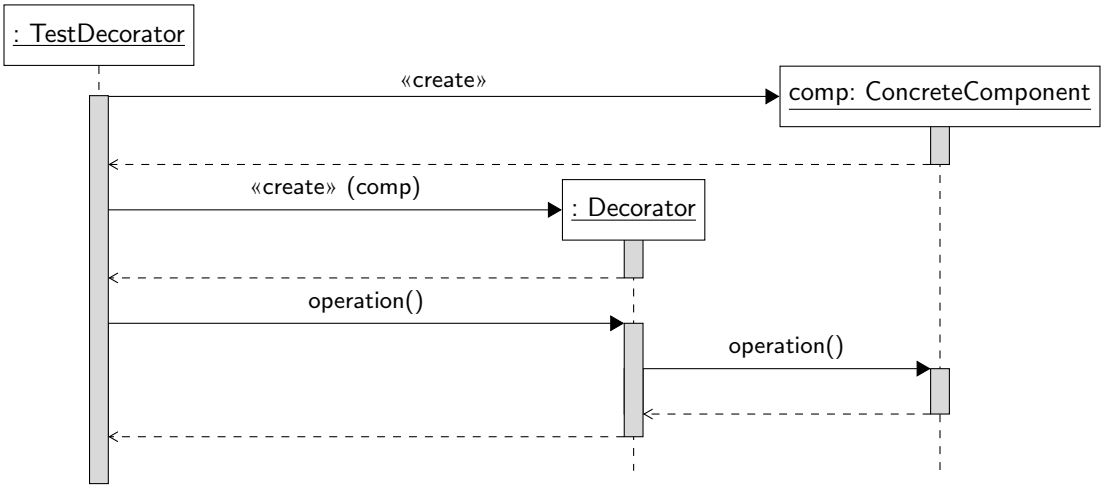

FIGURE 5 – Un scénario d'utilisation du décorateur

Adapter le patron de conception décorateur au problème posé en proposant un diagramme de classes.

(1 pt) (c) écrire la classe `DecoratorPCC` en Java.

(2 pt) **7.** Soit la classe `Fraction` suivante (la classe `DenominateurNulException` est une sous-classe directe d'`Exception`) :

```
 1 public class Fraction {
 2     private int num;
 3     private int den;
 4
 5     public void Fraction(int num, int den) {
 6         if (den == 0) throw new DenominateurNulException();
 7         this.num = num;
 8         this.den = den;
 9     }
10
11     public Fraction multiply(Fraction f) {
12         return new Fraction(this.num * f.num, this.den * f.den)
13     }
14
15     @Override public boolean equales(Object that) {
16         if (this == that) return true;
17
18         if (!(that instanceof Fraction)) return false;
19
20         return (that.num * this.den) == (that.den * this.num);
21     }
22 }
```

Quelles sont les 5 erreurs dans cette classe qui seront détectées par le compilateur Java ? On pourra soit indiquer **en rouge** directement dans le code source les erreurs ou se servir des numéros de ligne pour les préciser. On expliquera brièvement les erreurs.

8. On considère les classes `A` et `Main` suivantes :

Listing 6– La classe `A`

```
import java.io.IOException;

class A {

    void go() throws Exception {
        System.out.println("Demarrage methode go");

        try {
            meth();
            System.out.println("Fin bloc try");
        } catch (IOException e) {
            System.out.println("IOException attrapee");
        } finally {
            System.out.println("Execution bloc finally");
        }

        System.out.println("Fin methode go");
    }

    void meth() throws Exception {
        // cette methode peut lever une exception
    }
}
```

Listing 7– La classe `Main`

```
public class Main {

    public static void main(String[] args) throws Exception {
        A a = new A();
        a.go();
    }
}
```

(1 pt) (a) qu'est ce qui sera affiché sur la console si `meth` lève une `IOException` ?

(1 pt) (b) qu'est ce qui sera affiché sur la console si `meth` lève une `ArithmeticException` ?