NOM :

Prénom :

Examen, 1h30, Feuille A4 autorisée

Barème indicatif :

exercice	1	2	3	4	5	6	7	8
points	3	2.5	2.5	2	2,5	3	2	2,5

Exercice 1 On considère la fonction `mystere` suivante :

```
1  double mystere(double x, int n) {
2      if (n < 0) {
3          x = 1 / x;
4          n = -n;
5      }
6      double p = 1;
7      while (n > 0) {
8          int r = n % 2;
9          if (r == 1) {
10             p = p * x;
11         }
12         x = x * x;
13         n = n / 2;
14     }
15     return p;
16 }
```

1.1. Dessiner le graphe de contrôle correspondant aux instructions de cette fonction.
1.2. Combien y a-t-il de chemins (en considérant qu'on passe au plus une fois dans la boucle) ?
1.3. Compléter le tableau suivant en indiquant pour chaque variable (en ligne) l'ensemble de ses utilisations : on donnera le numéro de ligne en distinguant les utilisations dans des calculs (*c-use* (*computation*)) et les utilisations dans des conditions (*p-use* (*predicate*)). On donnera aussi l'ensemble des paires *def-use* pour chaque variable.

variable	*c-use*	*p-use*	paires *def-use*
x			
n			
p			
r			

1.4. Est-ce que le test `mystere(2, -1)` couvre toutes les instructions ? La réponse doit être argumentée. Dans la négative, on indiquera quels tests ajouter pour toutes les couvrir.
1.5. Est-ce que le test `mystere(2, -1)` couvre toutes les décisions ? La réponse doit être argumentée. Dans la négative, on indiquera quels tests ajouter pour toutes les couvrir.

Exercice 2 : Méthodes protected d'une classe
On veut afficher toutes les méthodes **`protected`** d'une classe donnée (et donc de ses super-classes). Les méthodes seront d'abord stockées dans une collection. Le premier argument de la ligne de commande est le nom de la classe, le second la collection à utiliser. Par exemple, on peut faire :

```
java Main java.util.Observable java.util.ArrayList
```

Le résultat sera alors :

```
protected void java.lang.Object.finalize() throws java.lang.Throwable
protected native java.lang.Object java.lang.Object.clone() throws java.lang.CloneNotSupportedException
protected synchronized void java.util.Observable.setChanged()
protected synchronized void java.util.Observable.clearChanged()
```

Compléter le programme du listing 1.

```
class Main {
    /** Ajouter dans la collection `methodes` toutes les méthodes protected
     * de la classe `classe` ou de ses superclasses. */
    static void collectProtectedMethods(Class<?> classe, Collection<Method> methodes) {
        ...
    }

    /** Afficher toutes les méthodes protected d'une classe dont le nom est
     * `args[0]`. Elles seront collectées dans la classe donnée par `args[1]`. */
    public static void main(String[] args) {
        try {
            ...
        } catch (Exception e) {
            System.out.println("Erreur : " + e + "\n");
            System.out.println("Usage : java Main classe collection");
            System.out.println("Exemple : java Main java.util.Observable java.util.ArrayList");
}    }    }
```

Listing 1: Squelette de la classe Main

Exercice 3 On considère le code Java suivant qui compile sans erreur.

```
1  import java.lang.annotation.*;

2  @Retention(RetentionPolicy.RUNTIME)

3  @Target({ElementType.CONSTRUCTOR, ElementType.METHOD})

4  @interface Mystere {

5      String p1() default "";

6      int p2();

7  }
```

3.1. Expliquer tous les éléments qui apparaissent sur ce code. On peut répondre sur le sujet.
3.2. Donner un exemple minimal d'utilisation de ce code.
3.3. À quel moment et de quelle manière `Mystere` peut être exploitée ?

Exercice 4 Répondre de manière concise et précise aux questions suivantes.
4.1. Acceleo permet de définir des *templates* et des *queries*. Expliquer ces deux notions.
4.2. Expliquer le principe de l'outil Pitest en s'appuyant sur le programme de l'exercice 1.

Filtres pour courrier électronique

On souhaite définir un mécanisme de filtrage de courrier électronique. Pour ce faire, le méta-modèle FILTRE est proposé (Figure 1). Un filtre est un ensemble de *règles*. Chaque règle est définie par des *motifs* qui spécifient les courriels concernés par la règle, et une *action* qui correspond au traitement à appliquer aux courriels ainsi sélectionnés. Un courriel doit vérifier tous les motifs d'une règle pour que l'action associée lui soit appliquée.

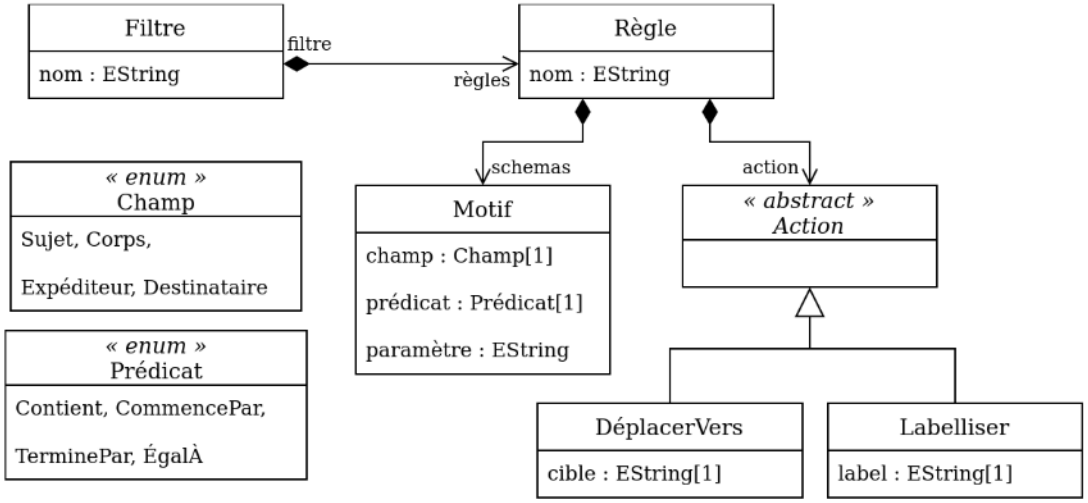

FIGURE 1 – Méta-modèle FILTRE

Exercice 5 Répondre succinctement aux questions suivantes portant sur Filtre (Figure 1).
5.1. Donner le nom des concepts *Ecore* correspondant aux éléments suivants : 1) `Règle`, 2) `prédicat` (dans l'élément `Motif`), et 3) `action` (entre les éléments `Règle` et `Action`).
5.2. Quel est le nom, le sens et l'utilité des flèches avec losange plein (♦) ?
5.3. Quel est le nom, le sens et l'utilité des flèches avec triangle vide (△) ?
5.4. Quelle relation relie les éléments `Filtre` et `Règle` ? Quelle est l'utilité de cette relation ?
5.5. Renseigner les *multiplicités* manquantes du méta-modèle.

Exercice 6 On considère le méta-modèle FILTRE (Figure 1).
6.1. Représenter sous la forme d'un *diagramme d'objet* conforme au méta-modèle FILTRE (Figure 1) le fitre nommé « *tranquilité* », composé des règles suivantes :

— Règle "étudiants" : lorsque le champ "Expéditeur" se termine par "@etu.enseeiht.fr", déplacer vers "Corbeille"
— Règle "urgence" : lorsque le champ "Sujet" contient "urgent", déplacer vers "Indésirables"

6.2. On veut pouvoir définir des motifs plus généraux. Plus précisément, une règle doit pouvoir définir un motif, qui est l'un des éléments suivants :

— un motif simple (tel que défini par `Motif` dans le méta-modèle actuel)

— la négation d'un motif

— la conjonction (ET) de plusieurs motifs

— la disjonction (OU) de plusieurs motifs

Modifier le méta-modèle FILTRE afin de prendre en compte ces exigences (ne dessiner que les nouveaux éléments et les éléments modifiés).

Exercice 7 On considère le méta-modèle FILTRE de la Figure 1.

Exprimer, à l'aide d'OCL et en précisant bien le *contexte*, une *requête* qui permet d'obtenir l'ensemble (potentiellement vide) des labels ajoutés par un filtre.

Exercice 8 On considère le méta-modèle GARDE donné Figure 2. Celui-ci représente un petit langage d'instructions gardées.

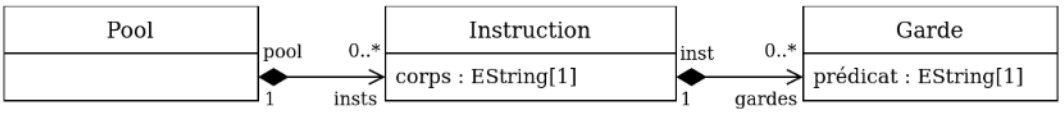

FIGURE 2 – Méta-modèle GARDE

La racine d'un modèle de GARDE est un "pool", autrement dit un ensemble d'instructions. Une instruction se compose d'un *corps* (le code à exécuter), et peut définir un ensemble de *gardes*. Chaque garde se compose d'un *prédicat* (instruction à valider pour que l'instruction soit exécutée). À noter qu'une instruction n'est exécutable que si *toutes* ses gardes sont vraies en même temps. Une instruction sans garde est toujours exécutable.

8.1. Quelle technologie peut-on utiliser pour transformer un modèle conforme à FILTRE (Figure 1) en un modèle conforme à GARDE ?

8.2. Décrire les règles d'une telle transformation sous la forme *entrée* $\rightarrow$ *sortie(s)*, en introduisant si nécessaire des requêtes intermédiaires. Le code des éléments `corps` de `Instruction` et `prédicat` de `Garde` est écrit en Java. On suppose l'existence d'une classe `Message` qui définit les accesseurs `getSubject()`, `getFrom()`, `getTo()` et `getBody()` pour obtenir, respectivement, le sujet, l'expéditeur, le destinataire et le corps du message et les méthodes `moveTo(d : String)` qui déplace le message dans le dossier `d`, et `tag(t : String)` qui ajoute le label `t` au message. La classe `String` définit les méthodes `startsWith(s : String)`, `endsWith(s : String)`, `contains(s : String)` et `equals(s : String)`.