IN201: Conception et Programmation Orientées Objet

Examen: problème

Cette partie de l'examen est composée de 5 questions. Le barême indiqué pour chaque question l'est à titre indicatif et peut être modifié.
Les seuls documents autorisés pour cet examen sont:

- les notes distribuées en cours (avec vos notes manuscrites)
- les cartes de référence distribuées en cours

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.

Test et réflexion

Florence reflected in the river (photography by Francine Lévesque, 2005)

On souhaite ici développer un système évolué de vérification des TPs effectués par les étudiants de SUPAERO lors du module IN201. Ce système devra être par exemple capable de vérifier que les bonnes pratiques de conception et de développement introduites dans le cours sont respectées: attributs privés, pas de constructeur par défaut etc. Ces vérifications seront effectuées sous la forme de tests JUnit.
Nous supposerons ici qu'on s'intéresse à la classe `Point` développée en cours dont le diagramme UML est donné ci-dessous:

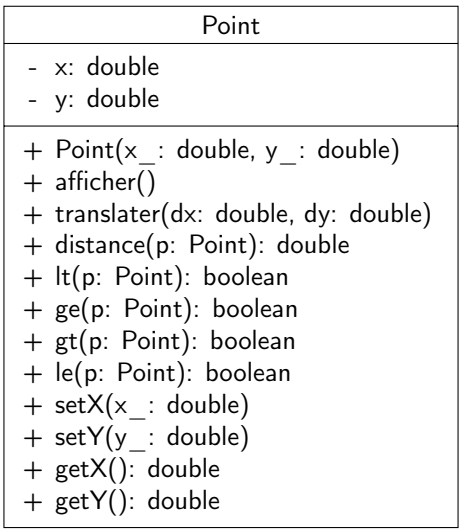

Figure 1: La classe `Point`

(1½ pt) **1.** pour pouvoir établir si par exemple la classe possède ou non des attributs non privés, il faut pouvoir représenter ces attributs pour pouvoir connaître leur visibilité. Or, les seules représentations que nous avons à disposition ici sont les notions objet (classe, objet, interface etc.). Il faut donc représenter des concepts objet avec des concepts objet... On appelle cela de la *méta-modélisation*. Un méta-modèle d'UML représentera par exemple les notions de classes, d'attributs, de méthodes sous forme de classes, d'interfaces etc.

Proposer un diagramme de classes ne faisant apparaître que les notions suivantes: classes, héritage, attributs, méthodes, visibilité. On ne s'intéressera donc pas ici aux interfaces, ni aux associations entre classes.

Solution:

Un diagramme de classes est proposé sur la figure 2.

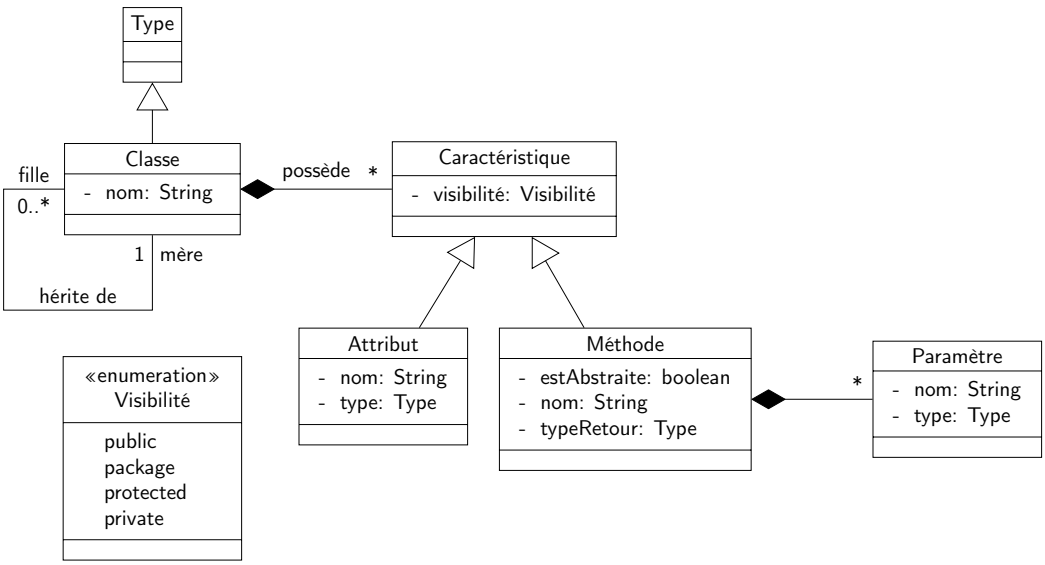

Figure 2: Diagramme de classe « méta-niveau » présentant quelques notions objets

Ce diagramme n'était pas forcément très évident à construire, il demandait un peu de réflexion. Cependant, il n'influait pas sur la suite du problème. Quelques remarques:

- j'ai utilisé des attributs lorsqu'il me semblait que cela permettait d'avoir un diagramme plus simple.
- il fallait pouvoir manipuler des types. J'ai donc créé une classe `Type` dont hérite `Classe` (les autres types étant les interfaces et les types primitifs, non représentés ici).
- l'héritage est représenté par une association de `Classe` sur elle-même, les multiplicités étant évidentes (une classe n'hérite que d'une seule classe, mais peut être un sous-type de plusieurs).
- une classe possède des caractéristiques qui ont une visibilité. J'ai choisi une énumeration pour représenter la visibilité.
- en ce qui concerne les mathodes et attributs, cela n'était pas très compliqué.

Si vous êtes intéressés, vous trouverez une présentation du méta-modèle détaillé d'UML dans [**?**].

2. l'API de Java fournit en fait une classe générique appelée `Class` représentant les classes et un paquetage permettant de travailler avec les caractéristiques des classes (attributs, méthodes etc.), `java.lang.reflect`. Nous allons les utiliser et les présenter dans ce qui suit.

La classe `Class<E>` est une classe générique. Le paramètre formel `E` est le type de la classe représentée par l'objet de type `Class<E>`. Par exemple, une instance de `Class<Point>` représente la classe `Point`.

La classe `Object` possède une méthode `getClass` permettant de récupérer un objet de type `Class` représentant la classe de l'objet considéré.

(1 pt) (a) la méthode `getDeclaredFields` de `Class` renvoie un tableau d'objets de type `java.lang.reflect.Field` représentant les attributs de la classe. `Field` est une classe possédant une méthode `getModifiers()` renvoyant un entier représentant la visibilité de l'attribut. Cet entier peut être comparé à des entiers définis sous forme d'attributs publiques et *statiques* de la classe `java.lang.reflect.Modifier`: `PRIVATE`, `PROTECTED`, `PUBLIC` entre autres.

Écrire une classe de test JUnit `PointTest` qui vérifie que la classe `Point` ne possède que des attributs privés.

Solution:

Le début de la classe `PointTest` concernant la question est présenté sur le listing 1. Quelques remarques:

- si l'on déclarait un attribut de type `Class<Point>` pour stocker l'objet de type `Class` correspondant à la classe `Point`, il fallait transtyper le retour de `getClass`, car le type de ce dernier est `? extends Point` (attention au sous-typage avec les types génériques).
- j'ai utilisé la boucle **for** vue en cours pour parcourir le tableau.

Listing 1: La classe `PointTest`

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3  import java.lang.reflect.*;
4
5  public class PointTest {
6
7      private Point p;
8      private Class<Point> pclass;
9
10     @Before public void setUp() {
11         this.p = new Point(0, 0);
```

```
12          this.pclass = (Class<Point>) this.p.getClass();
13      }
14
15      @After public void tearDown() {
16      }
17
18      @Test public void testAttributs() {
19          Field[] attributs = this.pclass.getDeclaredFields();
20
21          for (Field f : attributs) {
22              assertEquals("L'attribut " + f + " n'est pas prive !",
23                           Modifier.PRIVATE, f.getModifiers());
24          }
```

(1 pt) (b) de la même façon, la méthode `getConstructors` de `Class` renvoie un tableau d'objets de type `java.lang.reflect.Constructor` représentant les constructeurs de la classe. `Constructor` possède des méthodes intéressantes:

- `isSynthetic()` renvoyant un booléen permet de savoir si un constructeur donné a été synthétisé automatiquement par le compilateur
- `getGenericParameterTypes()` renvoyant un tableau d'objets de type `java.lang.reflect.Type` permet de connaître les types des paramètres d'un constructeur (dans l'ordre d'apparition dans la signature du constructeur)

Compléter la classe de test JUnit précédente pour vérifier que `Point` ne possède ni constructeur synthétisé par le compilateur ni constructeur par défaut. On écrira simplement la ou les méthodes de test correspondantes.

Solution:

Les deux méthodes sont présentées sur le listing 2. Rien de particulier, il fallait juste appliquer les méthodes décrites ci-dessus.

Listing 2: Les méthodes de la classe `PointTest` concernant le constructeur de `Point`

```
1       @Test public void testConstructeurSynthetique() {
2           Constructor[] cons = this.pclass.getConstructors();
3
4           for (Constructor c : cons) {
5               assertFalse(c.isSynthetic());
6           }
7       }
8
9       @Test public void testConstructeurDefaut() {
10          Constructor[] cons = this.pclass.getConstructors();
11
12          for (Constructor c : cons) {
13              assertFalse(c.getGenericParameterTypes().length == 0);
14          }
15      }
```

(1½ pt) **3.** on souhaiterait maintenant collecter les erreurs produites par les tests et les stocker dans un fichier. On ne s'intéresse dans cette question qu'à la méthode de test permettant de vérifier la visibilité des attributs de `Point` développée précédemment. On suppose que l'on dispose d'une classe `LogFile` qui possède une méthode `write(String s)` et un constructeur prenant un nom de fichier en paramètre. L'appel à `write` permet d'écrire la chaîne de caractères passée en paramètre dans le fichier dont le nom est fourni lors de la construction de l'instance.

Lorsqu'une méthode de type **`assert`** échoue, une instance de `AssertionError` est levée par l'appel à cette méthode. Cette instance peut contenir un message sous forme d'une instance de `String` accessible par la méthode `getMessage` de `AssertionError` (le message peut éventuellement être **`null`**).

Modifier la méthode développée précédemment pour écrire les messages d'erreur dans le fichier `point.log`. On n'oubliera pas de garantir le bon fonctionnement de JUnit qui doit récupérer l'instance de `AssertionError` correspondant à l'erreur.

Solution:

La méthode est présentée sur le listing 3. Il fallait procéder tranquillement:

1. création de l'instance de `LogFile`
2. récupération des attributs
3. pour chaque attribut:
   - encapsuler l'appel à `assertEquals` dans un bloc **try**
   - récupérer l'instance de `AssertionError` levée lors de l'échec du test et utiliser son message pour écrire dans le *log*
   - ne pas oublier de relancer l'instance de `AssertionError` pour que JUnit puisse fonctionner

Listing 3: La méthode `testAttributs` utilisant `LogFile`

```
1     @Test public void testAttributs() {
2         LogFile lf = new LogFile("point.log");
3
4         Field[] attributs = this.pclass.getDeclaredFields();
5
6         for (Field f : attributs) {
7             try {
8                 assertEquals("L'attribut " + f + " n'est pas prive !",
9                              Modifier.PRIVATE, f.getModifiers());
10            } catch (AssertionError ae) {
11                String s = ae.getMessage();
12
13                if (s != null) {
14                    lf.write(s);
15                }
16
17                throw ae;
18            }
19        }
20    }
```

4. on s'intéresse maintenant à la qualité des tests produits dans les tests JUnit concernant la classe `Point`. Pour vérifier la qualité d'un test donné, une technique qui peut être utilisée est celle du test par mutation (*mutation testing*): on n'utilise pas la classe `Point`, mais une version modifiée de la classe `Point`. Cette version modifiée ne devrait pas passer les tests de `PointTest`. Par exemple, on pourrait modifier la méthode `translater` de la classe `Point` pour que l'argument `dy` ne soit pas pris en compte:

```
public void translater(double dx, double dy) {
  this.x = this.x + dx;
  this.y = this.y + dx;
}
```

Dans ce cas, le test concernant `translater` écrit a priori dans `PointTest` devrait échouer s'il est bien écrit...

(½ pt) (a) écrire un mutant pour `Point` qui ait la méthode `translater` présentée ci-dessus. On cherchera à écrire le moins de code possible.

Solution:
Il paraît évident que l'on va profiter ici du mécanisme d'*héritage* de Java: on sous-classe `Point` en `PointMutant` et on redéfinit seulement la méthode `translater` dans `PointMutant`. La classe `PointMutant` est présentée sur le listing 4. Il fallait faire attention à l'appel au constructeur de `Point` dans celui de `PointMutant`. Comme les attributs de `Point` sont privés, il fallait faire appel à la méthode `translater` de `Point` via **super**.

Listing 4: La classe `PointMutant`

```
1  public class PointMutant extends Point {
2
3      public PointMutant(double x, double y) {
4          super(x, y);
5      }
6
7      @Override
8      public void translater(double dx, double dy) {
9          super.translater(dx, dx);
10     }
11 }
```

(½ pt) (b) modifier la classe de test JUnit `PointTest` afin de pouvoir écrire une classe de test JUnit `PointMutantTest` utilisant le mutant et faisant passer tous les tests définis dans `PointTest`. Écrire la classe `PointMutantTest`.

Solution:
Dans un premier temps, il faut modifier la classe `PointTest`. L'idée est de pouvoir faire hériter la classe `PointMutantTest` de `PoinTest`: `PointMutantTest` héritera de toutes les méthodes de test de `PointTest`. Pour pouvoir faire passer les tests définis dans `PointTest` par un objet de type `PointMutant`, il faut pouvoir modifier l'attribut `p` servant pour les tests dans `PointTest` depuis `PointMutantTest`. Le corrigé du TP sur l'héritage nous proposait deux solutions pour cela:

- modifier la visibilité de l'attribut en le mettant **protected**
- utiliser une *factory method*, i.e. une méthode permettant de construire l'objet à affecter à l'attribut. On peut alors redéfinir cette méthode dans `PointMutantTest` pour changer le type de l'objet à tester. C'est cette solution que j'ai choisie dans les listings 5 et 6[1].

Listing 5: La classe `PointTest` avec attribut « modifiable »

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  public class PointTest {
5
6      private Point p;
7      public static final double EPS = 10E-6;
8
9      @Before public void setUp() {
10         this.p = createPoint(0,0);
11     }
12
13     protected Point createPoint(double x, double y) {
14         return new Point(x, y);
15     }
16
17     @After public void tearDown() {
18     }
```

```
19
20      @Test public void testTranslater() {
21          double oldX = this.p.getX();
22          double oldY = this.p.getY();
23
24          this.p.translater(3,6);
25
26          assertEquals(oldX + 3, this.p.getX(), EPS);
27          assertEquals(oldY + 6, this.p.getY(), EPS);
28      }
29  }
```

Listing 6: La classe `PointMutantTest`

```
1  public class PointMutantTest extends PointTest {
2
3      @Override
4      public Point createPoint(double x, double y) {
5          return new PointMutant(x, y);
6      }
7  }
```

(c) en fouillant un peu sur le site d'IBM DeveloperWorks, vous tombez sur un article de Brian Goetz sur les *proxies* dynamiques [**?**]. Un *proxy* est un représentant d'un objet: il permet d'« intercepter » les appels à une ou plusieurs méthodes d'un objet, d'effectuer un traitement puis d'appeler la méthode sur l'objet et de récupérer le résultat avant de le renvoyer. Vous découvrez également que Java propose déjà une classe permettant de faire des *proxies* dynamiques. On pourrait donc créer un *proxy* pour un point qui modifierait le comportement de la méthode `translater` par exemple.

Dans ce qui suit, on appelera objet original l'objet à partir duquel on a créé le *proxy* et méthode originale une méthode que l'on appelle sur le *proxy* (et qui doit donc nécessairement être une méthode de l'objet original).

Toutes les nouvelles classes et interfaces présentées dans cette question appartiennent au paquetage `java.lang.reflect`.

(1 pt) (a) en vous plongeant plus en détail dans l'article, vous découvrez que le mécanisme de *proxy* dynamique repose sur la notion d'*invocation handler*. Un *invocation handler* est une classe réalisant l'interface `InvocationHandler` qui possède une seule méthode dont la signature est la suivante:

```
Object invoke(Object proxy, Method method, Object[] args) throws Throwable
```

À chaque fois qu'on appelle une méthode sur le *proxy*, la méthode `invoke` de l'*invocation handler* associé au *proxy* est appelée. C'est donc dans cette méthode que l'on va faire les traitements supplémentaires à ajouter à la méthode de l'objet original. Les paramètres de la méthode `invoke` sont créés lors de l'appel de la méthode sur le *proxy* et ont la signification suivante:

- `proxy` représente le *proxy* sur lequel on a appelé la méthode originale
- **`method`** est un objet de type `Method` représentant la méthode originale. `Method` est une classe dont les instances sont des objets représentant des méthodes. Elle possède une méthode permettant d'exécuter la méthode représentée sur un objet particulier:

  ```
  public Object invoke(Object obj, Object[] args) throws Exception
  ```

  où `obj` est l'objet sur lequel appeler la méthode et `args` les arguments de la méthode. Les exceptions qui peuvent être levées le sont lorsque par exemple l'objet passé en paramètre n'est pas du bon type et n'a pas la méthode représentée par l'objet de type `Method`.
  Un appel d'une méthode `m` sur un objet `o` avec les paramètres `arg1`, ..., `argn` devient donc `oM.invoke(o, args)` si `oM` est un objet de type `Method` représentant `m` et `args` un tableau contenant `arg1`, ..., `argn`.

- `args` représente les arguments de la méthode
- un objet de type `Throwable` peut être propagé par la méthode en cas de problème.

`Method` possède également une méthode `getName` qui renvoie le nom de la méthode sous forme d'un objet de type `String`.

Écrire une classe `TranslaterHandler` représentant un *invocation handler* pour modifier la méthode translater d'un point comme présenté ci-dessus. On réfléchira en particulier aux attributs nécessaires à la classe.

Solution:
Ce n'était pas très compliqué, mais il fallait lire attentivement l'énoncé. Dans un premier temps, il fallait réfléchir à ce dont on avait besoin pour construire l'*invocation handler*. Comme on souhaite de toute façon translater un point, il faut connaître un point. Le seul attribut de la classe sera donc une instance de `Point` qui est l'objet réel à translater.
Reste à implanter la méthode `invoke`. Dans celle-ci, on va successivement:

1. vérifier que le nom de la méthode à invoquer est bien `translater` (c'est une vérification minimale...). Si ce n'est pas le cas, on appelle la méthode sur le point sans rien changer;
2. construire un tableau d'arguments en utilisant deux fois la coordonnée en x du vecteur de translation;
3. appeler `translater` sur le point en utilisant ces nouveaux arguments.

La classe `TranslationHandler` est présentée sur le listing 7.

Listing 7: La classe `TranslationHandler`

```
1  import java.lang.reflect.*;
2
3  public class TranslationHandler implements InvocationHandler {
4
5      private Point point;
6
7      public TranslationHandler(Point p) {
8          this.point = p;
9      }
10
11     public Object invoke(Object proxy, Method method, Object[] args)
12         throws Throwable {
13
14         // si la methode n'est pas translater, on l'appelle sur le
15         // point sans changement. On pourrait egalement faire une
16         // verification sur le nombre d'arguments
17         if (! method.getName().equals("translater")) {
18             return method.invoke(this.point, args);
19         }
20
21         // on construit un tableau d'arguments avec deux fois la
22         // coordonnee de translation en x
23         Object[] nargs = { args[0], args[0] };
24
25         return method.invoke(this.point, nargs);
26     }
27 }
```

(1 pt) (b) on va maintenant chercher à construire un *proxy* dynamique pour un point en utilisant le *handler* précédemment développé. Pour cela, on va utiliser la méthode *statique* de la classe `Proxy` suivante:

```
Object newProxyInstance(ClassLoader loader,
                        Class<?>[] interfaces,
                        InvocationHandler h)
```

où

- `loader` est ce que l'on appelle un *class loader*. On peut l'obtenir en appelant `getClass().getClassLoader()` sur un objet. *Vous n'avez pas besoin d'en savoir plus.*
- `interfaces` est une tableau d'objets de type `Class` représentant les interfaces que l'on veut que le *proxy* réalise. Dans notre cas, on supposera qu'on dispose d'une interface `IPoint` contenant toutes les méthodes publiques de `Point`.
  On rappelle que pour créer et instancier un tableau dans une même expression, on peut écrire `new int[] {1, 2 , 3 , 4}` (l'exemple pris ici est celui d'un tableau d'entiers contenant les valeurs 1, 2, 3 et 4 dans cet ordre).
- le type de retour de `newProxyInstance` est `Object` (attention !)

Réécrire la classe de test `PointMutantTest` en utilisant le mécanisme de *proxy*. On supposera que le point utilisé dans `PointTest` est déclaré en utilisant `IPoint`.

Solution:

Là encore, il fallait faire attention et bien lire l'énoncé. Si l'on reprend ce qui a été fait, on va redéfinir la méthode `createPoint` pour qu'elle renvoie un point qui « ne se translate pas correctement ». Pour cela, on va:

1. créer un « vrai » point en utilisant la méthode `createPoint` de `createPoint` de `PointTest`. Attention, il y a du transtypage normalement, car on travaille avec `IPoint`.
2. utiliser `newProxyInstance` avec une instance de `TranslationHandler` qui est construite avec le point précédemment créé. Là encore, il ne fallait pas oublier de transtyper le résultat.

La classe `PointMutantTest` est présentée sur le listing 8. Vous pourrez vérifier facilement que tout cela fonctionne.

Listing 8: La classe `PointMutanTest` utilisant le *handler*

```
1  import java.lang.reflect.*;
2
3  public class PointMutantTest extends PointTest {
4
5      @Override
6      public IPoint createPoint(double x, double y) {
7          Point point = (Point) super.createPoint(x, y);
8
9          return (IPoint) Proxy.newProxyInstance(point.getClass().getClassLoader(),
10                                                 new Class[] { IPoint.class },
11                                                 new TranslationHandler(point));
12     }
13 }
```

5. le travail effectué pour la classe `Point` pourrait être étendu afin d'écrire une classe de test permettant de vérifier qu'une classe quelconque n'a pas d'attributs non privés par exemple. On pourrait alors écrire une classe de test JUnit `PrivateAttributesTest` vérifiant la non-existence d'attributs autres que privés dans une classe donnée. Encore faut-il pouvoir passer la classe en question à `PrivateAttributesTest`...

(1 pt) (a) est-il possible pour résoudre ce problème de rendre `PrivateAttributesTest` générique, le paramètre formel de type de `PrivateAttributesTest` représentant la classe dont on veut vérifier la visibilité des attributs?

Solution:

Il est toujours possible de rendre `PrivateAttributesTest` générique, mais comment alors instancier le paramètre de type générique? Supposons que l'on dispose de la classe

JUnit `PrivateAttributesTest<E>` qui permettent bien de vérifier que `E` ne possède pas d'attributs non privés, on pourrait créer une classe `PointAttributesTest` qui hériterait de `PrivateAttributesTest<Point>` et cela fonctionnerait.

Si l'on regarde de plus près la classe `PrivateAttributesTest<E>`, elle devrait posséder un attribut de type `Class<E>` représentant la classe à tester. Or, comment créer une instance de cette classe? On ne peut pas utiliser **new** `E().getClass()` car:

- on ne sait pas s'il existe un constructeur de `E` sans paramètres
- surtout **new** `E()` est interdit (pas de réification des types génériques en Java!)

(1 pt) (b) on suppose maintenant que l'on a un attribut de type `Class` dans `PrivateAttributesTest` représentant la classe à tester. *On supposera que l'on peut initialiser cet attribut via un constructeur de `PrivateAttributesTest`*. Écrire la classe `PrivateAttributesTest`.

Solution:

Si l'on dispose d'un attribut de type `Class` représentant la classe à tester, la solution est assez simple et est présentée sur le listing 9.

Listing 9: La classe `PrivateAttributesTest`

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  public class PrivateAttributesTest {
5
6      private Class classeATester;
7
8      @Test public void testAttributs() {
9          Field[] attributs = this.classeATester.getDeclaredFields();
10
11         for (Field f : attributs) {
12             assertEquals("L'attribut " + f + " n'est pas prive !",
13                          Modifier.PRIVATE, f.getModifiers());
14         }
15     }
16 }
```