UNIVERSITÉ DE ROUEN **Master GIL, 2e année**
FACULTÉ DES SCIENCES

POA
Annales d'Examens

Enoncé 1

Exercice I (questions de cours)

1. Quels sont les trois modes de tissage proposés par AspectJ ? Préciser, dans chaque cas, quel outil réalise le tissage, ce qu'il reçoit en entrée et renvoie en sortie.
2. Quelles entités les signatures suivantes reconnaissent-elles ?
   — @Business* Customer+
   — !public * Account.*()
3. On considère le code suivant (présenté en cours) :

```
1  import java.lang.reflect.Field;
2  import java.util.List;
3
4  public class FieldSpy<T> {
5      public boolean[][] b = {{ false, false }, { true, true } };
6      public String name  = "Alice";
7      public List<Integer> list;
8      public T val;
9
10     public static void main(String... args) {
11         try {
12             Class<?> c = Class.forName(args[0]);
13             Field f = c.getField(args[1]);
14             System.out.format("Type: %s%n", f.getType());
15             System.out.format(
16                 "GenericType: %s%n", f.getGenericType());
17
18             // production code should handle these exceptions
19         } catch (ClassNotFoundException x) {
20             x.printStackTrace();
21         } catch (NoSuchFieldException x) {
22             x.printStackTrace();
23         }
24     }
25 }
```

   — Quel affichage produit chacune des commandes suivantes ?
   (a) java FieldSpy FieldSpy b
   (b) java FieldSpy FieldSpy name
   (c) java FieldSpy FieldSpy list
   (d) java FieldSpy FieldSpy val
   — Que fait la méthode getGenericType en toute généralité ?
4. Quelle sont les trois catégories d'advice de type after ? Préciser en quelques mots ce qui les distingue.

5. Que produit la ligne suivante ?
   declare @type : Account+ : @Domain
6. Quelles sont les valeurs du type énuméré java.lang.annotation.RetentionPolicy ? Dans quel contexte utilise-t-on les valeurs de cette énumération ? Et dans quel but ?
7. Quelles sont les caractéristiques de la méthode aspectOf ? A quoi sert-elle ? Préciser la manière de l'invoquer et sa valeur de retour, selon le mode d'instanciation de l'aspect cible.

Exercice II

On a présenté en cours un aspect permettant de tracer les entrées et sorties de méthodes exécutées par une application. Cet aspect a été conçu de manière à indenter l'affichage. Dans cet exercice, on demande de coder le même aspect en affichant, en plus, le temps pris par l'exécution de chaque méthode. Lors de l'exécution du simulateur de distributeur de boissons, votre aspect doit produire l'affichage ci-après :

```
Entering [DrinksMachineModel.fillCash(..)]
    Entering [StdDrinksMachineModel.getCashNb(..)]
        Entering [MoneyAmount.getNumber(..)]
        Exiting [MoneyAmount.getNumber(..)] after : 6 microseconds
    Exiting [StdDrinksMachineModel.getCashNb(..)] after : 69 microseconds
    Entering [MoneyAmount.addElement(..)]
        Entering [StdStock.getNumber(..)]
        Exiting [StdStock.getNumber(..)] after : 0 microseconds
    Exiting [MoneyAmount.addElement(..)] after : 55 microseconds
    Entering [StdDrinksMachineModel.setChanged()]
    Exiting [StdDrinksMachineModel.setChanged()] after : 3 microseconds
Exiting [DrinksMachineModel.fillCash(..)] after : 306 microseconds
```

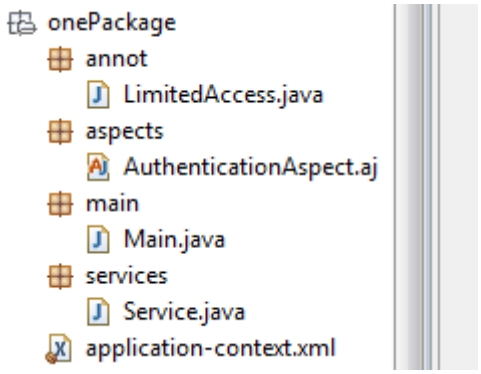
1. Pour la première version de votre aspect, vous complèterez le code suivant en développant un advice de type around :

```
1  public aspect TimedIndentLogging {
2      private Logger logger =
3          Logger.getLogger(TimedIndentLogging.class.getName());
4      {
5          Handler handler = logger.getParent().getHandlers()[0];
6          handler.setFormatter(new IndentFormatter());
7      }
8      private int depth = 0;
9
10     pointcut traced() : // définition de la coupe;
11
12     // partie à compléter (définition d'un advice de type around)
13 }
```

   Vous trouverez en annexe, l'implémentation de la classe IndentFormatter. Vous pouvez utiliser la méthode System.nanoTime() pour récupérer le nombre de nanosecondes (10^{-9}s) écoulées depuis le 1er janvier 1970.
2. Pour la seconde version, vous proposerez un aspect n'utilisant aucun advice de type around.

Exercice III

On considère le contexte décrit par le diagramme suivant :



Un service est défini par l'interface Service dont le code est le suivant :

```
1  package onePackage.services;
2
3  import onePackage.annot.LimitedAccess;
4
5  public interface Service {
6      @LimitedAccess
7      public void m();
8  }
```

Une implémentation en est fournie par la classe OneService dont on ne possède que le code binaire. L'annotation LimitedAccess indique que l'accès à la méthode annotée peut être restreint, si nécessaire. Et l'on va, précisément, développer un aspect destiné à exécuter un protocole d'authentification lorsqu'une méthode ainsi annotée doit être exécutée. On travaille sous Spring. On va donc écrire l'aspect en @AspectJ et l'instancier, ainsi que le service, sous la forme d'un bean.

1. Compléter le code source de l'aspect AuthenticationAspect, donné ci-après, de manière à intercepter l'exécution de toute méthode annotée par l'annotation LimitedAccess. La méthode privateAuth sera alors exécutée. Si elle retourne true la méthode interceptée reprendra son cours, sinon, elle est éludée et la valeur null est retournée.

```
1  package onePackage.aspects;
2
3  import ...
4
5  @Aspect
6  public class AuthenticationAspect {
7
8      // partie à compléter
9
10     /* procédure d'authentification */
11     private boolean privateAuth() {
12         System.out.println("Authentification...");
13         return ...;
14     }
15 }
```

2. Remplir le fichier xml qui suit de manière à définir un bean instance de OneService, un bean instance de AuthenticationAspect et à demander à Spring de tisser des aspects dans les proxy enveloppant ses beans.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    <!-- partie à compléter -->
</beans>
```

3. Donner le code source de la fonction main. Sans aspect, celui-ci produira l'affichage suivant (la méthode m de OneService réalise juste un affichage sur la sortie standard) :

```
exécution de m
```

Avec l'aspect, votre code devra produire, selon que l'authentification a été validée ou non :

```
Authentification...
validée !
exécution de m
```

ou

```
Authentification...
non validée !
```

Annexe

```java
import ...

public class IndentFormatter extends Formatter {
    public synchronized String format(LogRecord record) {
        StringBuffer sb = new StringBuffer();
        int depth = ((Integer) record.getParameters()[0]);
        sb.append(getIndent(depth));
        String message = formatMessage(record);
        sb.append(message);
        sb.append(System.getProperty("line.separator"));
        return sb.toString();
    }
    private String getIndent(int n) {
        if (n == 0) {
            return "";
        } else {
            return "␣␣␣␣" + getIndent(n - 1);
        }
    }
}
```

Enoncé 2

Exercice I (questions de cours)

1. Quelles sont les trois variables spéciales auxquelles peuvent accéder les advices ?
2. Expliquez précisément les contraintes à respecter pour que la portion de code suivante soit compilable :

```
Object around() throws T : <coupe> {...}
```

   où `T` désigne un type héritant directement ou indirectement de `Exception`.

3. Dans quelles circonstances, et pourquoi, est-il intéressant d'utiliser l'identificateur `value` pour définir un élément d'un type d'annotation ? Illustrer votre explication d'un petit exemple.
4. Donner l'équivalent, avec la syntaxe @AspectJ, de la déclaration AspectJ suivante :

```
public pointcut accountOperation(Account account, float amount) :
        execution (public * banking.domain.Account.*(float))
        && this(account)
        && args(amount);
```

5. A quoi sert le mot clé `perthis` ? Préciser le mécanisme qu'il induit et ses implications pour la méthode `aspectOf`.
6. On considère un type `Typ`. On ne dispose d'aucune instance de la classe `Class`. On demande de fournir quatre moyens potentiels différents pour obtenir l'instance de la classe `Class` associée au type `Typ`. Préciser dans chaque cas les contraintes à respecter et donner des exemples.

Exercice II

On souhaite étudier les possibilités de mise en œuvre du patron de conception "singleton" à l'aide d'aspects. On considère la situation initiale suivante :

```
public class A {}

public class Main {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();

        System.out.println("a1 : " + a1 + " ; a2 : " + a2 );
    }
}
```

Bien sûr, à l'exécution ce code affiche deux adresses mémoires distinctes.

Une implémentation possible du patron "singleton" est la suivante :

```
public class A {
    private static A a;

    private A() {}

    public static A getA() {
        if (a == null) {
```

```
8               a = new A();
9           }
10          return a;
11      }
12  }
13
14  public class Main {
15      public static void main(String[] args) {
16          A a1 = A.getA();
17          A a2 = A.getA();
18
19          System.out.println("a1 : " + a1 + " ; a2 : " + a2 );
20      }
21  }
```

Cette fois, une exécution provoque un double affichage d'une même adresse. Mais le code des deux classes `A` et `Main` a dû être modifié pour mettre en œuvre le patron.

1. Développer un aspect `SingletonAspect` qui, associé à la première version des classes `A` et `Main`, permet d'obtenir à l'exécution le même effet que la deuxième version avec un flot d'exécution similaire.
2. On souhaite généraliser l'aspect précédent pour permettre d'appliquer le patron à n'importe quels types. Pour cela on demande d'écrire un premier aspect qui contient tout le code du mécanisme du patron puis un second aspect où sont précisés les types pour lesquels on veut que le patron soit appliqué.

   On considère maintenant une autre implémentation du patron "singleton", à travers une troisième version des classes `A` et `Main` :

```
1   public class A {
2       private static A a = new A();
3
4       private A() {}
5
6       public static A getA() {
7           return a;
8       }
9   }
10
11  public class Main {
12      public static void main(String[] args) {
13          A a1 = A.getA();
14          A a2 = A.getA();
15
16          System.out.println("a1 : " + a1 + " ; a2 : " + a2 );
17      }
18  }
```

3. Développer un nouvel aspect `SingletonAspectBis` qui, toujours associé à la première version des classes `A` et `Main`, permet d'obtenir à l'exécution le même effet que cette troisième version avec un flot d'exécution similaire.
4. Comme pour la question 2, développer deux aspects permettant de généraliser le mécanisme de la question précédente pour appliquer le patron à n'importe quels types.

On veut maintenant lier la mise en place du patron à la présence d'une annotation (`@Singletonable`). On considère le code suivant :

```
@Singletonable
public class A {}

public class B {}

public class C extends A {}

public class Main {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();
        B b1 = new B();
        B b2 = new B();
        C c1 = new C();
        C c2 = new C();

        System.out.println("a1 : " + a1 + " ; a2 : " + a2 );
        System.out.println("b1 : " + b1 + " ; b2 : " + b2 );
        System.out.println("c1 : " + c1 + " ; c2 : " + c2 );
    }
}
```

L'exécution de ce code produit trois affichages : le premier contient deux fois la même adresse, le deuxième contient deux adresses distinctes et le troisième contient deux fois la même adresse.

5. Donner le code définissant l'annotation `@Singletonable` et l'aspect qui permettent d'obtenir ce comportement.
6. On supprime l'annotation de la classe `A`, mais on veut conserver un comportement identique à l'exécution. Autrement dit, la mise en place du patron par le biais d'une annotation est entièrement réalisée depuis des aspects. Donner le code de ces aspects.

Enoncé 3

Exercice I

1. A quoi servent les instances de la classe `java.util.logging.Handler` et quelles en sont les principales caractéristiques ?
2. Répondre par *vrai* ou *faux* aux 3 questions suivantes, en fournissant une ou deux lignes de justification à chaque fois.
   (a) Si, à l'aide d'une déclaration inter-type, un aspect ajoute un nouveau champ privé à un type `T` alors seules les instances de `T` pourront accéder à ce champ.
   (b) Il est possible d'annoter un champ depuis un aspect.
   (c) La coupe suivante ne sélectionne aucun point de jonction

   ```
   execution(* Account.*(..)) && call(* Account.*(..))
   ```
3. Quel affichage produira l'exécution du code suivant :

```
1 public class Main {
2     public static void main(String[] args) {
3         A a = new A();
4         B b = new B();
5         a.meth01();
6         b.meth02();
7     }
8 }
```

sachant que les types `A` et `B` sont indépendants l'un de l'autre, que les méthodes `A.meth01()` et `B.meth02()` ne font rien et que l'aspect ci-après a été tissé par-dessus ?

```
1 public aspect Coucou pertarget(c2()) {
2     pointcut c1() : call(* A.meth01());
3     pointcut c2() : call(* B.meth02());
4
5     after() : c1() {
6         System.out.println("coucou");
7     }
8 }
```

Justifier votre réponse.

Exercice II

On souhaite développer une fonctionnalité qui permette de réaliser la sauvegarde d'objets dans une base de données lorsque leur état a été modifié. Concrètement, un développeur utilisant cette fonctionnalité se contentera d'annoter les classes dont il veut que les instances soient régulièrement sauvegardées. Un aspect repérera les changements de valeur des champs de ces objets et enregistrera ces derniers auprès d'un `Saver` qui réalisera les sauvegardes proprement dites.

Le développeur écrira, par exemple, la classe suivante :

```
1 package project.data;
2
3 import functionality.Savable;
4
```

```java
@Savable
public class Supply {
    private String desc;

    @Override
    public String toString() {
        return "Supply [desc = " + desc + "]";
    }

    public void meth(String desc) {
        this.desc = desc;
    }
}
```

L'annotation `@Savable` indique que les instances de cette classe doivent être sauvegardées quand leur état est modifié. Du coup la classe `Main` ci-après :

```java
package project.main;

import functionality.Saver;
import project.data.Supply;

public class Main {
    public static void main(String[] args) {
        Supply supply = new Supply();
        System.out.print("sauvegarde 01 : "); Saver.save();
        supply.meth("boulons");
        System.out.print("sauvegarde 02 : "); Saver.save();
        supply.meth("écrous");
        System.out.print("sauvegarde 03 : "); Saver.save();
        supply.meth("écrous");
        System.out.print("sauvegarde 04 : "); Saver.save();
    }
}
```

produira la sortie suivante :

```
sauvegarde 01 : Supply [desc = null] est sauvegardé
sauvegarde 02 : Supply [desc = boulons] est sauvegardé
sauvegarde 03 : Supply [desc = écrous] est sauvegardé
sauvegarde 04 : Aucune sauvegarde n'est nécessaire.
```

Remarque : En principe, la classe `Saver` n'est pas connue du développeur. Il ne devrait donc pas l'utiliser dans la méthode `main`. Au lieu de cela, un thread devrait tourner en tâche de fond et faire les appels à la méthode `save`. Mais la mise en place de ce mécanisme dépassant le cadre de cet exercice, on se satisfera du `main` tel qu'il est écrit.

Dans le paquetage `functionality` sont donc définis l'annotation `@Savable` et la classe `Saver` de la façon suivante :

```java
package functionality;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
public @interface Savable {}
```

```java
package functionality;

import java.util.HashSet;
import java.util.Set;

public class Saver {
    public interface Savable {}

    static private Set<Savable> savables = new HashSet<Savable>();

    static void add(Savable savable) {
        savables.add(savable);
    }

    public static void save(){
        if (savables.isEmpty()) {
            System.out.println("Aucune␣sauvegarde␣n'est␣nécessaire.");
            return;
        }
        for (Savable savable : savables) {
            assert savable != null;
            // faire une sauvegarde
            System.out.println(savable + "␣est␣sauvegardé");
            savables.remove(savable);
        }
    }
}
```

Il ne reste plus qu'à développer, dans le paquetage `functionality`, l'aspect `SavableManagerAspect` qui permet de produire l'affichage vu précédemment quand la méthode `main` est exécutée. C'est le travail qui vous est demandé.

On notera qu'un objet mérite d'être sauvé dès sa création et qu'il faut, bien sûr, le sauver quand un, au moins, de ses champs a changé de valeur, mais qu'une sauvegarde n'est pas utile quand on affecte à un champ la valeur qu'il possède déjà. Cette vérification nécessite des opérations d'introspection.

Exercice III

Une équipe de développement a fait l'observation suivante : de nombreuses applications accèdent à des dépôts (*repository*, en anglais) de données et les classes présentes dans ces dépôts fournissent toujours des méthodes qui retournent des `Collection`s de données. Quand on appelle ces méthodes on doit systématiquement vérifier que l'objet retourné n'est pas `null` avant de l'exploiter comme une collection. Ces tests polluent le code et sont très souvent inutiles. Aussi le chef de projet décide-t-il d'une pratique de développement : « les méthodes des classes situées dans un paquetage nommé repository ou dans l'un de ses sous-paquetages, directs ou indirects, et dont le type de retour est `Collection<?>` ou un de ses sous-types ne devront jamais retourner la valeur `null`. »

1. Ecrire un aspect `NullDetector` qui permet de produire un message de journalisation (un log) chaque fois que cette règle est violée. Cet aspect ne doit pas contenir d'advice de type around et, dans le contexte suivant :

```
1  package project.repository;
2
3  import java.util.Set;
4
5  public interface Repository<E> {
6      Set<E> contents();
7  }
```

```
1  package project.repository;
2
3  import java.util.Set;
4
5  public class StubRepository<E> implements Repository<E> {
6      public Set<E> contents() {
7          return null;
8      }
9  }
```

```
1  package project.main;
2
3  import project.repository.Repository;
4  import project.repository.StubRepository;
5
6  public class Main {
7      public static void main(String[] args) {
8          Repository<Object> rep = new StubRepository<Object>();
9          int size = rep.contents().size();
10         System.out.println("Taille␣de␣la␣collection␣:␣" + size);
11     }
12 }
```

l'exécution du code tissé produira sur la sortie erreur standard, le résultat ci-après :

```
1  ...en-tête du message de journalisation...
2  ATTENTION: Détection d'une requête de dépôt retournant null :
3          execution(StubRepository.contents())
4  Exception in thread "main" java.lang.NullPointerException
5          at project.main.Main.main(Main.java:9)
```

2. On veut maintenant développer le même outil mais dans un environnement Spring (sans disposer du compilateur d'AspectJ).
   (a) Ecrire une version @AspectJ de l'aspect `NullDetector`.
   (b) Donner le contenu (sans l'en-tête) d'un fichier `application-context.xml` qu'utilisera Spring pour construire les beans dont vous aurez besoin et tisser votre aspect.
   (c) Réécrire la classe `Main` donnée à la question précédente pour récupérer les objets utiles à l'application *via* le "bean container" de Spring.

   l'exécution de cette version produira la sortie suivante :

```
...en-tête du message de journalisation...
ATTENTION: Détection d'une requête de dépôt retournant null :
        execution(Repository.contents())
Exception in thread "main" java.lang.NullPointerException
        at project.main.Main.main(Main.java:...)
```

   (d) Expliquer en 3 ou 4 lignes d'où vient la différence avec la sortie produite à la question 1 (`Repository` au lieu de `StubRepository`, ligne 3).

3. On souhaite une troisième version pour un environnement java simple (ni AspectJ, ni Spring). Pour cela on exploite la classe `java.lang.reflect.Proxy`. La classe `Main` devient :

```
1  package projet.main;
2
3  import java.lang.reflect.Proxy;
4
5  import project.proxy.NullDetectionHandler;
6  import project.repository.Repository;
7  import project.repository.StubRepository;
8
9  public class Main {
10     public static void main(String[] args) {
11         Repository<Object> rep = new StubRepository<Object>();
12         Repository<Object> repProxy = ...
13         int size = repProxy.contents().size();
14         System.out.println("Taille de la collection : " + size);
15     }
16 }
```

   (a) Compléter les ... de la ligne 12.
   (b) Donner le code de la classe `NullDetectionHandler` permettant d'obtenir une journalisation similaire à celle des questions précédentes.

   On revient dans le contexte de la question 1 mais, au lieu de journaliser l'apparition d'une valeur `null`, on veut lui substituer une collection vide.

4. Réécrire l'aspect `NullDetector` (avec un advice around, cette fois) pour mettre en œuvre cette solution.