

Examen (avec documents)

Corrigé

Préambule : Répondre de manière concise et précise aux questions. Ne pas mettre de commentaires de documentation sauf s'ils sont nécessaires à la compréhension.

Il est conseillé de répondre directement dans le sujet quand c'est possible. Sinon, il est conseillé de mettre une marque sur le sujet (par exemple une lettre majuscule : A, B, C, etc) et d'indiquer sur la copie la marque avec le texte (ou le code) associé.

Les exercices sont relativement indépendants.

Barème indicatif :

exercice	1	2	3	4	5
points	4	4	4	4	4

Vérificateur de propriétés sur une classe

Ces exercices ont pour but de définir un vérificateur de propriétés sur des classes Java. L'idée est de pouvoir facilement ajouter de nouvelles propriétés pour enrichir le vérificateur.

Nous commençons par définir quelques exemples de propriétés (exercice 1). Nous écrivons ensuite un évaluateur de telles propriétés (exercice 2). Nous proposons ensuite une description en XML pour représenter les propriétés qui pourra être exploitée pour construire une collection de propriétés à vérifier (exercice 3), description XML qui pourra être engendrée à partir d'une classe considérée comme modèle (exercice 4). Enfin, nous définissons une petite interface graphique pour cette application (exercice 5).

Exercice 1 : Quelques propriétés concrètes

L'objectif étant de pouvoir facilement ajouter de nouvelles propriétés, nous définissons une interface Propriete qui spécifie deux méthodes : `getTexte()` qui explique la vérification faite par la propriété et `verifier(Class<?> c)` qui évalue la propriété sur une classe. Les commentaires de documentation expliquent ces deux méthodes. Un exemple d'utilisation de propriétés concrètes est donné dans l'exercice suivant.

```
1 public interface Propriete {
2     /** Vérifier si cette propriété est satisfaite sur la classe c.
3         * @param c la classe à analyser
4         * @exception RuntimeException si la vérification échoue (le
5         * message explique l'échec).
6         */
7     void verifier(Class<?> c);
8 }
```

```
9     /** Obtenir l'objectif de cette propriété.
10     * @return l'objectif de cette propriété
11     */
12     String getTexte();
13 }
```

1.1 Écrire la classe `NomEnMajuscule` qui est une propriété vérifiant que l'initial du nom de la classe est une majuscule.

Solution :

```
1 public class NomEnMajuscule implements Propriete {
2
3     public void verifier(Class<?> c) {
4         char initiale = c.getName().charAt(0);
5         if (initiale < 'A' || initiale > 'Z') {
6             throw new RuntimeException(c.getName()
7                 + "_devrait_commencer_par_"
8                 + Character.toUpperCase(initiale));
9         }
10    }
11
12    public String getTexte() {
13        return "Est-ce_que_le_nom_commence_par_une_majuscule?";
14    }
15 }
```

1.2 Écrire la classe `MethodeExiste`, propriété qui vérifie qu'une méthode est bien déclarée dans la classe. Le constructeur de cette propriété prendra en paramètre le nom de la méthode (du type `String`) et les types de ses paramètres (un tableau de `Class<?>`).

Solution :

```
1 public class MethodeExiste implements Propriete {
2
3     private String nomMethod;
4     private Class<?>[] typeParams;
5
6     public MethodeExiste(String nomMethod, Class<?>... typeParams) {
7         this.nomMethod = nomMethod;
8         this.typeParams = typeParams;
9     }
10
11    public String getTexte() {
12        return "Est-ce_que_la_méthode_" + this.nomMethod
13            + "(" + this.typeParams2String() + ")"
14            + "_existe?";
15    }
16
17    private String typeParams2String() {
18        String resultat = "";
19        int n = 0;
20        for (Class<?> c : this.typeParams) {
21            if (n > 0) {
22                resultat += ",";
23            }
24        }
25    }
26 }
```

```
24         resultat += c.getName();
25         n++;
26     }
27     return resultat;
28 }
29
30 public void verifier(Class<?> c) {
31     try {
32         c.getDeclaredMethod(this.nomMethod, this.typeParams);
33     } catch (NoSuchMethodException e) {
34         throw new RuntimeException(e);
35     }
36 }
37
38 }
```

Exercice 2 : Vérificateur de propriétés

On considère maintenant l'interface `IVerificateur` ci-dessous. Sa méthode `evaluation` vérifie si une classe respecte chaque propriété d'une collection. La valeur retournée est une chaîne de caractères qui contient sur chaque ligne le texte de la propriété évaluée et le résultat de son évaluation sur la classe.

```
1 import java.util.Collection;
2
3 public interface IVerificateur {
4     String evaluation(Collection<? extends Propriete> props, Class<?> c);
5 }
```

Le listing 1 donne un exemple d'utilisation du vérificateur. La classe `Exemple1` est donnée au listing 2. Le résultat de l'exécution est donné au listing 3.

Listing 1 – "Exemple d'utilisation de la classe `Verificateur`"

```
1 import java.util.*;
2 public class ExempleVerificateur {
3     public static void main(String[] args) {
4         Collection<Propriete> props = new ArrayList<Propriete>();
5         props.add(new NomEnMajuscule());
6         props.add(new MethodeExiste("translater", Double.class, Double.class));
7         props.add(new MethodeExiste("distance", Exemple1.class));
8         props.add(new MethodeExiste("toString"));
9         props.add(new MethodeExiste("m", String.class, Integer.class));
10
11         IVerificateur v = new Verificateur();
12         String r = v.evaluation(props, Exemple1.class);
13
14         System.out.println(r);
15     }
16 }
```

Listing 2 – "La classe `Exemple1`"

```
1 public class Exemple1 {
2     public void translater(Double dx, Double dy) {}
}
```

```

3     private void distance(Exemple1 autre) {}
4 }

```

Listing 3 – "Résultats de l'exécution de la classe ExempleVerificateur"

```

1 Est-ce que le nom commence par une majuscule ? OUI
2 Est-ce que la méthode translater(java.lang.Double, java.lang.Double)
  existe ? OUI
3 Est-ce que la méthode distance(Exemple1) existe ? OUI
4 Est-ce que la méthode toString() existe ? java.lang.NoSuchMethodException
  : Exemple1.toString()
5 Est-ce que la méthode m(java.lang.String, java.lang.Integer) existe ?
  java.lang.NoSuchMethodException: Exemple1.m(java.lang.String, java.lang
  .Integer)
6 Fin.

```

2.1 Expliquer succinctement pourquoi il faut utiliser `Collection<? extends Propriete>` et pas seulement `Collection<Propriete>` pour le type du paramètre de évaluation.

Solution : Car `Collection<AttributsNonPublics>` n'étant pas un sous-type, on ne pourrait pas l'utiliser comme paramètre de évaluation. Utiliser le type joker (?) permet d'appeler évaluation avec toute `Collection` dont le type des éléments est au moins `Propriete`.

2.2 Écrire la réalisation `Verificateur` de l'interface `IVerificateur`.

Solution :

```

1 import java.util.*;
2
3 public class Verificateur implements IVerificateur {
4
5     public String evaluation(Collection<? extends Propriete> props, Class<?> c) {
6         String resultat = "";
7         for (Propriete p : props) {
8             try {
9                 resultat += p.getTexte() + " ";
10                p.verifier(c);
11                resultat += "OUI";
12            } catch (Exception e) {
13                resultat += e.getMessage();
14            }
15            resultat += "\n";
16        }
17        return resultat + "Fin.";
18    }
19
20 }

```

Exercice 3 : Propriétés en XML

On se propose de définir un fichier XML pour représenter les propriétés. L'objectif sera par exemple de pouvoir paramétrer une application de vérification grâce à un tel fichier XML. La DTD de ces fichiers XML est donnée ci-dessous.

```

1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2

```

```

3 <!ELEMENT proprietes ((nom-majuscule|meth-def)*)>
4 <!ELEMENT nom-majuscule EMPTY>
5 <!ELEMENT meth-def (param*)>
6 <!ATTLIST meth-def
7     nom CDATA #REQUIRED >
8
9 <!ELEMENT param EMPTY>
10 <!ATTLIST param
11     type CDATA #REQUIRED >

```

Le fichier XML du listing 4 est conforme à la DTD précédente. Il correspond à l'exemple du listing 1.

Listing 4 – Fichier de propriétés en XML

```

1 <?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
2
3 <!DOCTYPE proprietes SYSTEM "proprietes.dtd">
4
5 <proprietes>
6     <nom-majuscule />
7     <meth-def nom="translater">
8         <param type="java.lang.Double" />
9         <param type="java.lang.Double" />
10    </meth-def>
11    <meth-def nom="distance">
12        <param type="Exemple1" />
13    </meth-def>
14    <meth-def nom="toString" />
15    <meth-def nom="m">
16        <param type="java.lang.String" />
17        <param type="java.lang.Integer" />
18    </meth-def>
19 </proprietes>

```

3.1 On souhaite utiliser SaX pour lire le fichier XML et ajouter toutes les propriétés qu'il contient dans une collection de propriétés. Compléter le code donné au listing 5. Il s'agit en fait de compléter la classe ChargeurXMLHandler.

Listing 5 – Squelette de la classe ChargeurXML

```

1 import java.io.*;
2 import javax.xml.parsers.*;
3 import org.xml.sax.*;
4 import org.xml.sax.helpers.*;
5 import java.util.*;
6
7 public class ChargeurXML {
8
9     public void charger(InputStream in, Collection<? super Propriete> list)
10        throws SAXException, ParserConfigurationException, java.io.IOException
11    {
12        SAXParserFactory spf = SAXParserFactory.newInstance();
13        XMLReader xmlReader = spf.newSAXParser().getXMLReader();
14        xmlReader.setFeature("http://xml.org/sax/features/validation", true);

```

```
15     ChargeurXMLHandler handler = new ChargeurXMLHandler(list);
16     xmlReader.setContentHandler(handler);
17     xmlReader.parse(new InputSource(in));
18 }
19 }
20
21 class ChargeurXMLHandler extends DefaultHandler {
22
23     private Collection<? super Propriete> proprietes;
24
25     private String nomMethode;
26     private List<Class<?>> types = new ArrayList<Class<?>>();
27
28
29     public ChargeurXMLHandler(Collection<? super Propriete> proprietes) {
30         this.proprietes = proprietes;
31     }
32
33
34     public void startElement(String uriEspaceNom, String nom,
35                             String nomQualifie, Attributes attributs) throws SAXException {
36         if (nomQualifie.equals("meth-def")) {
37             this.types.clear();
38             this.nomMethode = attributs.getValue("nom");
39         } else if (nomQualifie.equals("param")) {
40             try {
41                 this.types.add(Class.forName(attributs.getValue("type")));
42             } catch (ClassNotFoundException e) {
43                 throw new SAXException(e);
44             }
45         } else if (nomQualifie.equals("nom-majuscule")) {
46             this.proprietes.add(new NomEnMajuscule());
47         }
48     }
49
50     public void endElement(String uriEspaceNom, String nom,
51                            String nomQualifie) {
52         if (nomQualifie.equals("meth-def")) {
53             this.proprietes.add(new MethodeExiste(this.nomMethode,
54                                                    this.types.toArray(new Class<?>[0])));
55         };
56     }
57
58
59 }
```

3.2 Expliquer pourquoi le type `Collection<? super Propriete>` est utilisé.

Solution : Ceci permet de fournir en paramètre n'importe quelle collection dont le type des éléments est `Propriete` ou une supertype, par exemple `ArrayList<Object>`.

Exercice 4 : Utiliser une classe existante comme modèle

On souhaite pouvoir utiliser une classe comme modèle pour construire le fichier de propriétés. Plus précisément, étant donnée une classe Java, on souhaite engendrer un fichier de propriétés

XML qui vérifie que chaque méthode de la classe modèle est bien présente dans la classe à tester.

Par exemple, les méthodes déclarées dans la classe `Modele1` (listing 6) correspondent aux propriétés décrites dans le fichier `exemple1.xml` (listing 4).

Listing 6 – La classe `Modele1`

```
1 public abstract class Modele1 {
2     abstract void translater(Double dx, Double dy);
3     abstract void distance(Exemple1 autre);
4     abstract public String toString();
5     abstract void m(String s, Integer i);
6 }
```

Compléter le code de la classe `ProprietesBuilder` (listing 7) qui utilise `JDom` pour construire le fichier de propriété.

Listing 7 – Squelette de la classe `ProprietesBuilder`

```
1 import java.io.*;
2 import java.util.*;
3 import org.jdom.*;
4 import org.jdom.output.*;
5
6 public class ProprietesBuilder {
7
8
9     public Element getElementMD(String nom, Class<?>... types) {
10         // Construire l'élément
11         Element eltProp = new Element("meth-def")
12             .setAttribute("nom", nom);
13         for (Class<?> t: types) {
14             Element eltType = new Element("param");
15             eltType.setAttribute("type", t.getName());
16             eltProp.addContent(eltType);
17         }
18         return eltProp;
19     }
20
21
22     public void genererXML(OutputStream out, Class<?> modele) {
23
24         Element racine = new Element("proprietes");
25         for (java.lang.reflect.Method m : modele.getDeclaredMethods()) {
26             // XXX seulement les méthodes privées ???
27             racine.addContent(getElementMD(m.getName(), m.getParameterTypes()));
28         }
29         Document document = new Document(racine,
30             new DocType("proprietes", "proprietes.dtd"));
31
32         ecrire(document, out);
33
34     }
35
36     public void ecrire(Document document, OutputStream out) {
37         try {
```

```

38         XMLOutputter sortie = new XMLOutputter(Format.getPrettyFormat());
39         sortie.output(document, out);
40     } catch (java.io.IOException e) {
41         throw new RuntimeException("Erreur_sur_écriture_", e);
42     }
43 }
44
45 }

```

Exercice 5 : Interface graphique simplifiée

On veut proposer une interface graphique en Swing qui permet à l'utilisateur de donner le nom de la classe à vérifier et le nom de la classe servant de modèle. Un bouton *vérifier* lance la vérification et affiche les résultats dans l'onglet *résultats*. Un bouton *quitter* permet de quitter l'application. Le squelette de la classe correspondante est donnée au listing 8.

5.1 Dessiner l'apparence de l'interface graphique lorsque l'application du listing 8 est lancée.

Solution :



5.2 Compléter l'application de manière à ce que les deux boutons deviennent actifs.

Listing 8 – Squelette de la classe VerificateurSwing

```

1  import java.io.*;
2  import java.util.*;
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class VerificateurSwing extends JFrame {
8
9      private JTextField nomClasse = new JTextField("Exemple1", 20);
10     private JTextField modeleClasse = new JTextField("Modele1", 20);
11     private JButton bVerifier = new JButton("Verifier");
12     private JButton bQuitter = new JButton("Quitter");
13     private JTextArea resultats = new JTextArea(10, 40);
14
15     public VerificateurSwing() {
16         super("Vérificateur");
17         JTabbedPane onglets = new JTabbedPane();
18         this.getContentPane().add(onglets);
19
20         JPanel parametres = new JPanel(new BorderLayout());
21         onglets.addTab("paramètres", parametres);
22

```



```
23 // Construire la partie supérieure
24 JPanel saisies = new JPanel(new GridLayout(2, 2));
25 saisies.add(new JLabel("Classe_à_vérifier:_"));
26 saisies.add(nomClasse);
27 saisies.add(new JLabel("Modèle_de_classe:_"));
28 saisies.add(modeleClasse);
29 parametres.add(saisies, BorderLayout.NORTH);
30
31 // Construire les boutons de commandes inférieurs
32 JPanel boutons = new JPanel();
33 boutons.setLayout(new FlowLayout());
34 boutons.add(bVerifier);
35 boutons.add(bQuitter);
36 parametres.add(boutons, BorderLayout.SOUTH);
37
38 // Construire l'onglet résultat
39 onglets.addTab("résultats", new JScrollPane(resultats));
40
41
42 // Définir les listeners
43 bQuitter.addActionListener(new ActionQuitter());
44 bVerifier.addActionListener(new ActionVerifier());
45
46
47 this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
48 this.pack();
49 this.setVisible(true);
50 }
51
52
53 private class ActionVerifier implements ActionListener {
54     public void actionPerformed(ActionEvent ev) {
55         try {
56             String nom = nomClasse.getText();
57             Class<?> c = Class.forName(nom);
58             String model = modeleClasse.getText();
59             Class<?> m = Class.forName(model);
60             ProprietesBuilder builder = new ProprietesBuilder();
61             // Écrire le XML dans un tableau
62             ByteArrayOutputStream bos = new ByteArrayOutputStream();
63             builder.genererXML(bos, m);
64             // Lire les propriétés XML
65             ArrayList<Propriete> proprietes = new ArrayList<Propriete>();
66             ByteArrayInputStream bis = new
67                 ByteArrayInputStream(bos.toByteArray());
68             new ChargeurXML().charger(bis, proprietes);
69             // Vérifier
70             String res = new Verificateur().evaluation(proprietes, c);
71             resultats.setText(res);
72         } catch (Exception e) {
73             resultats.setText(e.toString());
74         }
75     }
76 }
```

```
77
78 private class ActionQuitter implements ActionListener {
79     public void actionPerformed(ActionEvent ev) {
80         System.exit(0);
81     }
82
83 }
84
85
86 public static void main(String[] args) {
87     EventQueue.invokeLater(new Runnable() {
88         public void run() {
89             new VerificateurSwing();
90         }
91     });
92 }
93 }
```