M2 Informatique - HMIN304 - "Réutilisation et Composants"

Année 2017-2018. Session 1, janvier 2018.

Christophe Dony, Chouki Tibermacine

Durée : 2h00. **Documents non autorisés**. La précision et la concision des réponses sont notées ainsi que la lisibilité des codes (ne copiez pas de mémoire des phrases que vous ne comprenez pas ou qui ne sont pas en rapport avec la question). **Rédiger les réponses aux parties A et B de façon séparée (feuilles différentes)**.

PARTIE A

1 Réutilisation - (environ 5 points)

Considérons la hiérarchie de classes des éléments de stockage (Directory, File, Link), architecturée selon le schéma *Composite*, et dotée d'un mécanisme de visite, selon le schéma *Visiteur*, étudié dans le TD/TP no 1. Vous avez réalisé en TP plusieurs visiteurs dont `RazVisitor`. Soit (ci-dessous) le code de la classe `Visitor` et le code (incomplet) de la classe `Directory` dans lequel l'attribut `elements` sert à stocker les éléments contenus dans un *directory* (répertoire).

```
1  public abstract class Visitor {
2      public void visitFile(File f){}
3      public void visitDirectory(Directory f){}
4      public void visitLink(Link f){} }

5  public class Directory extends ElementStockage{
6      protected Collection<ElementStockage> elements ; ||
7      ... }
```

Questions

1. En terme de réutilisation (extension sans modification du code existant), dites quel est l'intérêt de l'architecture proposée par le schéma *Composite*. Dites ensuite quel est son point problématique auquel le schéma *Visiteur* apporte une solution.

2. Soit à réaliser un visiteur (`PrettyPrintVisitor` capable d'afficher au terminal le contenu d'un répertoire avec indentation, le décalage vers la droite indiquant l'inclusion d'un élément dans un directory. Par exemple, pour les objets donnés en listing 1, l'exécution du code donné en listing 2 donnera le résultat ci-contre.

```
1  Directory d = new Directory("UnProgramme") ;
2  Directory d2 = new Directory("src");
3  Directory d3 = new Directory("bin");
4  d.add(d2);
5  d.add(d3);
6  d2.add(new File("F1.java","..."));
7  d2.add(new File("F2.java","..."));
8  d3.add(new File("F1.class","..."));
9  d3.add(new Link("sources", d2));
```

Listing 1 – des éléments de stockage

```
1  Visitor pp = new
       PrettyPrintVisitor();
2  d.accept(pp);
```

Listing 2 –

Résultat :

```
Directory UnProgramme
    Directory src
        File F1.java
        File F2.java
    Directory bin
        File F1.class
        Link sources
```

(a) Le schéma visiteur dans sa version standard et pour les éléments de Stockage tels que présentés dans la section 1 ne permet pas de réaliser cette fonctionnalité. Expliquez quel est le problème.

(b) Pour remédier à ce problème, on propose ci-dessous la modification suivante de la classe `Visitor`.

```
1  public abstract class Visitor {
2      public void visitFile(File f){}
3      public void visitBeforeDirectory(Directory f){}
4      public void visitAfterDirectory(Directory f) {}
5      public void visitLink(Link f) {};
```

Donnez en Java la nouvelle version de la méthode `accept(Visitor v)` de la classe `Directory` compatible avec

Scanned by CamScanner

cette nouvelle version de la classe `Visitor`.

(c) Donnez le code Java de la classe `PrettyPrintVisitor`, dont je vous donne ci-dessous une des méthodes, celle qui réalise l'affichage des espaces avant le texte, via un attribut `int indent` destiné à stocker le niveau d'indentation courant. Inutile de recopier le texte de cette méthode dans votre réponse.

```
1    public void printIndent(){
2        //imprime 'indent' espaces en début de ligne
3        for (int i = 0; i < indent; i = i+1) System.out.print(" "); }
```

2 Composants JavaBeans et Aspects (Environ 4 points)

Plaçons nous dans le contexte de réalisation d'une classe `JBPoint` ("JavaBeans Points") définissant des points classiques (une abscisse et une ordonnée) mais compatibles avec l'environnement *JavaBeans*, donc qui pourront être disponible sur la palette, comme dans le TP *NetBeans* . Les `JBPoint` seront des *beans* de type "écouté", dont les propriétés `X` et `Y` représentant l'abcisse et l'ordonnée doivent être des propriétés liées ("*Bound Properties*").

Questions Java-Beans

1. Qu'est-ce que le problème du couplage en programmation par objet auquel l'approche par composants tente de répondre.
2. En quoi le modèle des JavaBeans induit-il du découplage. Dans votre réponse, prenez l'exemple des "propriétés liées" (*bound properties*) pour expliquer.
3. Lorsqu'on souhaite connecter visuellement un `JBPoint` avec un autre composant `c`, par exemple par ce que son `X` a changé, pourquoi l'environnement doit-il générer automatiquement un adapteur ?

Questions Programmation par Aspects

1. a) Donnez l'intérpet et décrivez conceptuellement un aspect `BoundPoint` permettant de rendre liées les propriétés de la classe Point.
   b) Donnez deux exemples de parties du code de l'aspect `BoundPoint`.

Voir page suivante pour partie B.

Scanned by CamScanner

PARTIE B

3 Composants (environ 10 points)

Questions

1. Choisir la seule affirmation juste. Commentez. Les message-driven beans sont des composants Java EE qui servent :
   1. à produire dynamiquement du contenu Web dans des messages de type HTTPResponse envoyés aux clients
   2. à implémenter une partie de la logique métier d'une application, exécutable par envoi synchrone de messages
   3. à traiter la réception asynchrone, dans une file, de messages envoyés par d'autres objets

2. Choisir la seule affirmation juste. Commentez. Avec Java EE, nous avons la possibilité :
   1. d'implémenter des composants distribués, mais qui ne peuvent jamais interagir avec des services Web,
   2. d'invoquer, dans le code des beans, les opérations de services Web, mais nous ne pouvons pas déployer des beans comme de nouveaux services Web,
   3. de définir un bean session comme un service Web et invoquer les opérations d'autres services Web

3. Les *beans Spring* sont obtenus par instanciation automatique et injection de leurs dépendances. Expliquer les 3 méthodes vues en cours pour déclarer des *beans*. Discuter les avantages et inconvénients de chacune des méthodes. Donner l'exemple d'un bean déclaré avec l'une des trois méthodes.

4. Choisir la seule affirmation juste. Commentez. Supposons l'existence de deux composants (*bundles*) *OSGi*. Si l'on souhaite les connecter en affectant la référence d'un objet (de type T) dans le deuxième bundle au champ d'un objet dans le premier bundle, on doit :
   1. déclarer un import-package dans le premier bundle (contenant l'interface requise T) et un export package dans le second (interface fournie T), puis laisser le framework rechercher les classes dans chaque bundle, qui doivent être instanciées,
   2. déclarer un import-package et un export-package (comprenant chacun le package de T), puis instancier les classes (implémentant T) manuellement à l'intérieur de chaque bundle,
   3. récupérer une référence de type T en utilisant une classe Factory ou l'annuaire de services, sans déclarer le package de T dans import-package et export-package,
   4. faire ces déclarations (import- et export- package, comprenant chacun le package de T) d'abord, et ensuite récupérer une référence de type T (avec une Factory ou l'annuaire de services),
   5. déclarer un require-bundle (comprenant un bundle ayant une classe qui implémente T), simplement. Le framework gère l'instanciation et l'injection des dépendances.

5. Décrire brièvement les 2 solutions développées dans *OSGi* et vues dans le cours pour gérer le problème d'indisponibilité (et d'attente efficace de la disponibilité) des services.

6. Un composant (plugin) *Eclipse* est un bundle *OSGi* qui déclare des extensions et éventuellement des points d'extension. Soit un plugin A qui déclare un point d'extension P, et un plugin B qui contribue par des extensions à ce point. Dans le contrat de ce point d'extension P, il est déclaré un type abstrait I. Laquelle de ces propositions vous semble correcte ? Commentez.
   1. C'est le plugin A qui déclare l'interface fournie I et le plugin B l'interface requise I,
   2. C'est le plugin B qui déclare l'interface fournie I et le plugin A l'interface requise I,
   3. Aucun des deux composants ne déclare une interface fournie ou requise. Les deux composants sont très fortement couplés (ils se connaissent mutuellement statiquement).

7. Quelles sont les différences entre les *scripts background* et les *scripts content* que nous pouvons définir dans un plugin de navigateur Web ? (Expliquer entre autres l'utilité et les privilèges de chacun). S'ils veulent partager des données, comment ces scripts communiquent-ils ?

8. Discuter les limites en termes de réutilisation en Java, qui ont poussé la proposition du système de modules dans le *JDK-9*. Expliquer entre autres les limites des systèmes de build comme *Maven* utilisés avec du code Java.

9. Quel impact a la définition d'un descripteur de modules Java-9 sur la visibilité des types qui y sont définis ? Quels sont les différents éléments, vus en cours, qui composent un descripteur de modules ? Les décrire brièvement. Donner un exemple de descripteur. Le JDK-9 a été entièrement restructuré en modules. Quel est le rôle des modules agrégateurs dans le JDK ? Quelle est la particularité de leurs descripteurs de modules ?

10. A quel besoin répondent les «services» dans le système de modules Java-9 ? Comment organise-t-on, en termes de modules, le code d'une application constituée d'un module fournissant un service et de deux autres modules utilisant ce service ? (expliquer entre autres où placer l'interface du service)

Scanned by CamScanner

PARTIEL COMPOSANTS Jan. 2018

Partie A

1 - Réutilisation

1) Le schema Composite permet d'ajouter de nouveaux éléments, simples ou composés d'éléments existants, ayant leurs opérations spécifiques en plus de celles héritées de la hiérarchie (réutilisation), sans modification des classes existantes.
Cependant, il n'est pas possible d'ajouter de nouvelles opérations aux éléments existants sans modifier le code et l'alourdir. Le patron visiteur permet d'ajouter ces nouvelles opérations sans modification du code de la hiérarchie.

2a) Lors du accept(v), on appelle la méthode visitDirectory(d) soit avant, soit après la boucle d'accept(v) sur les élements du dossier. Cela pose problème pour les "retours" d'indentation à la fin du dossier. Il faudrait un appel avant pour l'indentation et l'affichage du nom, puis un après, pour le retour d'indentation.

```
public class PrettyPrintVisitor extends Visitor {
    private int indent;

    @Override
    public void visitFile(File f) {
        printIndent();
        System.out.println("File " + f.getName());
    }

    @Override
    public void visitBeforeDirectory(Directory f) {
        printIndent();
        System.out.println("Directory " + f.getName());
        indent++;
    }

    @Override
    public void visitAfterDirectory(Directory f) {
        indent--;
    }

    @Override
    public void visitLink(SymLink f) {
        printIndent();
        System.out.println("Link " + f.name);
    }
```

```
    public void printIndent() {
        for (int i = 0; i < indent; i = i + 1)
            System.out.print(" ");
    }
}

public class Directory extends StorageElement {

    private Collection<StorageElement> elements;


    @Override
    public void accept(Visitor v) {
        v.visitBeforeDirectory(this);
        for (StorageElement element :
                elements) {
            element.accept(v);
        }
        v.visitAfterDirectory(this);
    }
}
```

2b)

```
public class Directory extends StorageElement {

   private Collection<StorageElement> elements;


   @Override
   public void accept(Visitor v) {
      v.visitBeforeDirectory(this);
      for (StorageElement element :
            elements) {
         element.accept(v);
      }
      v.visitAfterDirectory(this);
   }
}
```

2c)

```
public class PrettyPrintVisitor extends Visitor {
   private int indent;

   @Override
   public void visitFile(File f) {
      printIndent();
```

```
        System.out.println("File " + f.getName());
    }

    @Override
    public void visitBeforeDirectory(Directory f) {
        printIndent();
        System.out.println("Directory " + f.getName());
        indent++;
    }

    @Override
    public void visitAfterDirectory(Directory f) {
        indent--;
    }

    @Override
    public void visitLink(SymLink f) {
        printIndent();
        System.out.println("Link " + f.name);
    }

    public void printIndent() {
        for (int i = 0; i < indent; i = i + 1)
            System.out.print(" ");
    }
}
```

2 - Composants JavaBeans et Aspect

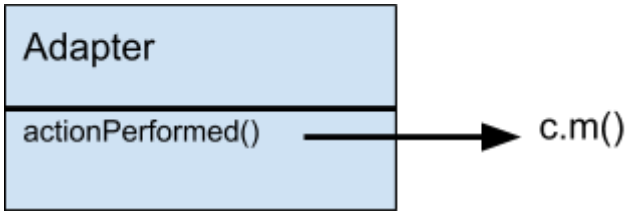
Partie Java Beans

1) Le couplage dénote une référence explicite, au sein d'un élément A, à un autre élément B nécessaire, par son nom ou son adresse.
Il est alors impossible de savoir que A nécessite B sans regarder le code, tout autant que de réutiliser A sans B (ou remplacer par autre chose) sans modifier le code de A

2) Les Java Beans sont des composants autonomes, qui communiquent par le biais du patron Observer/Observable. Un bean contenant une bound property tient une liste de listeners, qui seront prévenus du changement de la propriété. Ces composants sont paramétrables, par des données contenues dans des propriétés, et répondent à l'introspection, qui permet de connaître le contenu sans avoir le code. Les getters/setters permettent alors de savoir si les propriétés sont readable/writable.

Ainsi, il y a découplage car on peut utiliser les beans indépendamment. Il n'y a pas besoin de faire référence à un bean au sein d'un autre pour utiliser ses propriétés, mais simplement de s'enregistrer comme listener (recevra le message de this.notify())

3) Le composant "c" n'a pas forcément été prévu pour écouter des événements. Il faut alors un adapteur qui puisse être l'écouteur et contrôler le composant c.
Ex:



L'adapteur implémente alors l'interface ActionListener et Serializable? et permet à n'importe quel composant d'être écouteur de tout écouté (produit un résultat utile émettant des événements à d'autres).

Partie Aspects

1a) Un JBPoint a pour responsabilité principale d'être un composant graphique disposable sur une palette. Créer un aspect BoundPoint permet de définir une fonctionnalité orthogonale (par exemple une translation de coordonnées d'un point)

1b)

```
public aspect BoundPoint{
   pointcut changedX(Point p):
         target(p) && call(* p.setX(..));
   //idem avec Y
   around(Point p): changedX(p){
         System.out.println("x before "+p.getX());
         proceed();
         System.out.println("x after "+p.getX());
   }
   //idem avec Y
}
```

Partie B

3 - Composants (10pts)

1) 3

2) 3

3) Via le xml ; via annotations ; dans le code (classe config)

4) 4

5) Cours 3 p44 : 1-Programmer l'attente (Itération ou timer) ou 2-utiliser un Service tracker (évolution des service listener) ou 3-Exploiter le mécanisme de "Déclarative Services"

6) 2

7) Script "content":
Script "background" :
Communication: via des Objets "runtime.Port".

8) Limites Java

- Quand on réutilise des prog. on doit gérer beaucoup de packages qui proviennent de JARs différents.
- Gestion complexe des dépendances: Classpath (JAR) Hell
    - Dépendances non explicites : JAR ne dit pas de quel autre JAR il dépend.
    - Dépendances transitives (A dépend de B qui dépend de C)

Limites des systèmes de build comme Maven et Gradle: Outils utilisés de façon statique et externe à l'application.

- Dépendances explicitées en dehors de la définition des composants de l'application.
- A l'exécution, la notion de composant / dépendance disparaît.

9)

10)