Examen de Génie Logiciel
1ère session - L3 informatique

Responsables : C. Roudet et B. Bordeaux

Documents autorisés : feuille A4 RV avec notes manuscrites + double page des DP

10 janvier 2025. Durée : 1h30

Exercice 1 : Question d'ordre général - CHOISIR l'une des 2 questions (5 pts / 20)

La qualité de la rédaction sera prise en compte ! FAITES DES PHRASES - 20 lignes MAXI !

1) Dans le cadre du développement logiciel, donner les principales **différences** entre le cycle en **V** et le cycle **itératif incrémental** (base des méthodes agiles). Pour cela, vous listerez leurs principaux **avantages** et **inconvénients** (sans hésiter à vous servir d'exemples). Vous penserez à bien repréciser ce qu'on entend par « effet tunnel ».

2) Qu'appelle-t-on **recette usine/métier** que certaines entreprises de maîtrise d'œuvre (MOE) comme Atol Conseil et Développement intègrent parfois dans le cycle de vie des applications qu'ils développent et déploient ? En quoi cela consiste-t-il, et à quel moment dans le cycle de vie cela intervient-il ? Justifiez son utilisation en vous servant, au besoin d'exemples et en indiquant sur quels outils il est possible de s'appuyer.

Exercice 2 : Design Patterns - CHOISIR l'une des 2 questions (4 pts / 20)

1) On veut programmer un **jeu de babyfoot** avec, au choix, des joueurs en 2D ou en 3D (leurs apparences et comportements diffèrent). La classe `Babyfoot` crée (et contient) un objet de la classe `Terrain` qui crée (et contient) à son tour les 11 joueurs de chaque équipe : 1 gardien de but, 2 arrières, 5 demis et 3 attaquants.
Proposer un diagramme **de classes UML** mettant en œuvre le DP **Fabrique abstraite** (Abstract Factory) et répondant à ce contexte.

2) On veut définir une classe **`Validateur`** capable de valider différentes saisies (sous forme de **`String`**), par exemple la saisie de **codes postaux** (comprenant un nombre fixe d'entiers) ou d'**adresses mail** (comprenant au moins un symbole @). Donner le diagramme de classes UML correspondant, conforme au pattern **Strategy**, en utilisant des commentaires UML pour détailler au besoin certaines méthodes ou constructeurs.

Exercice 3 : Tests fonctionnels, structurels et unitaires en Java (11 pts / 20)

Le problème des **tours de Hanoï** est un jeu de réflexion consistant à déplacer des disques de diamètres différents d'une tour de "**départ**" à une tour d' "**arrivée**", en passant par une tour "**intermédiaire**" et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ (où tous les disques se situent sur la tour de "**départ**"). Le nombre de disques est fonction de la hauteur des tours. La hauteur des tours, le nombre et le diamètre des disques sont **paramétrables** dans notre implémentation. A la page suivante, vous trouverez **deux des classes Java** composant le programme de résolution du jeu des tours de Hanoï.

../..

1- La classe **Disque** permet de représenter les disques que l'on empile sur les tours de Hanoï. Ces disques sont caractérisés par leur diamètre (nombre entier positif). Pour pouvoir comparer les disques entre eux, cette classe **implémente l'interface** *java.lang.Comparable*, et est définie comme ceci :

```
public class Disque implements Comparable<Disque> {
  private int diam;       //diamètre d'un des disques du jeu (à déplacer)

  public Disque(int t){
      diam = t;
  }

  public int compareTo(Disque d){
      Integer i = (Integer) diam;
      return i.compareTo((Integer) d.diam);  // comparaison signée détaillée
  }                                          // ci-dessous
} //fin de la classe Disque
```

La méthode `compareTo` de la classe `Integer` retourne les valeurs entières (int) suivantes :

- 0 si les 2 Integer (**`i`** et `(Integer)`**`d.diam`**) sont égaux ;
- une valeur inférieure à 0 si **`i`** est numériquement **inférieur** à l'argument (**`d.diam`**) ;
- et une valeur supérieure à 0 si **`i`** est numériquement **supérieur** à l'argument (**`d.diam`**).

2- La classe **Tour**, sur laquelle on empile les disques :

```
public class Tour {           //le jeu sera composé de 3 objets de ce type
  private Disque [] content;  //ce que contient la tour (rien ou des disques)
  private int index;  //indice du sommet de la tour : -1 ≤ index < size
  private int size;   //taille de la tour (nb de disques qu'elle peut contenir)
  public Tour(int size){
      this.size = size;
      content = new Disque [size];
      index = -1;                    //la tour est vide
  }

1  public boolean empiler(Disque obj) throws FullTowerException,
2                                            ForbiddenDiskException {
3   if(obj.getDiam() <= 0)
4        throw new ForbiddenDiskException();
5   else if(index == size - 1)                 //tour pleine
6        throw new FullTowerException();
7   else if(index >= 0 && content[index].compareTo(obj) <= 0)
8        throw new ForbiddenDiskException();
9   else {
10       index = index + 1;
11       content[index] = obj;
12       return true;
13  }
14 }

} //fin de la classe Tour
```

Questions de l'exercice 3 concernant la méthode `empiler` de la classe `Tour` :

I) Tests fonctionnels et unitaires

1) Déterminer les **classes d'équivalence** (test partitionnel) à considérer pour la méthode `boolean empiler(Disque obj)` de la classe `Tour`. Vous pouvez, pour cela, compléter le tableau suivant :

Configuration d'entrée	Classe valide	Classe invalide
Valeur de this.**size** (taille de la tour courante)		
Diamètre de **obj** si la tour est vide		
Diamètre de **obj** si la tour n'est pas vide (**s** = disque au sommet)		

2) Déterminer maintenant, par une **approche aux limites**, les Données de Tests (DT) à produire pour cette méthode. Pour chaque DT, indiquer le résultat attendu (« true » ou l'une des deux exceptions : `FullTowerException ou ForbiddenDiskException`).

Rappel : le **test des valeurs limites** (ou l'**approche aux limites**) n'est pas vraiment une famille de sélection de test, mais une tactique pour améliorer l'efficacité des DT produites par d'autres familles. Les erreurs se nichent généralement dans les cas limites, d'où le fait qu'on teste principalement les valeurs aux limites des domaines ou des classes d'équivalence.

3) Écrire maintenant, dans une **classe de test JUnit** nommée **`TestTour`** (en précisant si vous utilisez JUnit 3 ou 4), des **méthodes de test** pour tester la méthode `empiler` de la classe `Tour`, pour **deux** de vos DT (en les précisant et en testant **au moins un envoi d'exception**). **INUTILE d'écrire les « import Java » !**

II) Tests structurels et unitaires

1) Dessiner le **graphe de flot de contrôle** correspondant à la méthode `empiler` de la classe `Tour` (en précisant pour chaque état du graphe à quel numéro de ligne il est associé : cf. page 2 du sujet). Donner la **forme algébrique** associée à ce graphe de flot de contrôle.

2) Identifier les Données de Test (DT) minimales pour **couvrir toutes les instructions** (couverture de **tous les nœuds** du graphe de flot de contrôle). Vous ne préciserez le résultat attendu pour ces DT que si elles sont nouvelles (pas déjà rencontrées précédemment).

3) Ces DT assurent-elles la **couverture de tous les arcs du graphe** ? Sinon préciser la/les DT à considérer pour couvrir **tous les arcs**. Même consigne qu'à la question précédente pour le résultat attendu de ces DT.

4) Toutes les DT proposées jusque là vous semblent-elles suffisantes pour que les **tests structurels** (tests en boîte blanche) soient **complets** ? Justifiez votre réponse et en cas d'insuffisance des DT, précisez lesquelles il faudrait ajouter.

5) Expliquer ce qu'il se passe si l'on exécute le code suivant :

```
Tour t = new Tour(3);
Disque d = null;
t.empiler(d);
```

Proposer une solution pour remédier au problème rencontré, ainsi qu'un **test JUnit** permettant de s'assurer que le problème a bien été résolu.