Université de Lille – Sciences et Technologies – Licence 3 Informatique 2017-2018

UE Conception Orientée Objet

Devoir Surveillé

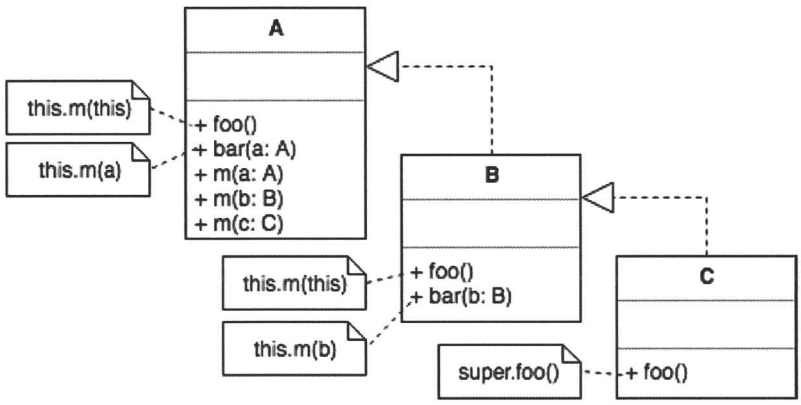
3h

Copie des diapositives de cours annotée autorisée.
Fiches « *design pattern* » du cours autorisées.
Dictionnaires de langue autorisés.
Autres documents interdits.

Sauf mention expresse, la javadoc et les tests des classes à écrire ne sont pas demandés.

Exercice 1 : Héritage

On donne le diagramme de classes suivant :



La méthode `toString` de chaque classe renvoie une chaîne de caractères correspondant au nom de la classe. En plus des portions de code indiquées sur le diagramme, chacune des autres méthodes commence par l'instruction :
"`System.out.println("`*NomDeClasse.nomMéthode(TypeParametres...)*`");`"
où *NomDeClasse* est évidemment remplacé par le nom de la classe où est déclarée la méthode, *nomMéthode* par le nom de cette méthode et *TypeParametres...* par le nom des types des paramètres de la signature de la méthode lorsqu'il y en a.
Par exemple, la méthode `public void m(A a)` de la classe `A` débute par la ligne :

```
System.out.println("A.m(A)");
```

Q 1. Indiquez précisément ce qu'affiche le programme suivant :

```
public static void main(String[] args) {
     List<A> listeA = new ArrayList<A>();
     listeA.add(new A());
     listeA.add(new B());
     listeA.add(new C());

     for(A a : listeA) {
         System.out.println("---"+a+".foo---");
         a.foo();
         System.out.println("---"+a+".bar---");
         a.bar(a);
     }
}
```

Exercice 2 :
On suppose définie une classe `Compte` :

Compte
- numero : String - solde : float
+Compte(numero : String) +debiter(montant : float) +crediter(montant : float) +getSolde() : float +getNumero() : String

Cette classe permet de modéliser des comptes en banque. `numero` représente le numéro du compte et `solde` représente le solde de ce compte. Les méthodes `debiter` et `crediter` permettent évidemment de respectivement réaliser les opérations de débit et crédit sur ce compte. Le paramètre de ces méthodes doit être positif sinon une exception `MontantNegatifException` est levée.

Dans le contexte d'utilisation de cette classe, on souhaite que des entités puissent être informées des opérations réalisées sur un compte. On peut par exemple citer :

- un *système de journal* qui enregistre toutes les opérations effectuées sur l'ensemble des comptes : on mémorise alors le montant et le compte concernés. On doit pouvoir récupérer toutes les opérations relatives à un compte donné enregistrées dans un journal. Un objet journal est modélisé par la classe `JournalOperations`.
- un *système d'alertes par SMS* qui permet d'envoyer un message à un téléphone mobile associé au numéro de compte pour chaque opération effectuée avec en plus un second message d'alerte de découvert si le solde du compte devient négatif suite à un débit. Les messages sont de la forme `"compte` *N* `crédité de` *X* `EUR."` ou `"Attention : le compte` *N* `est à découvert"`. Cette classe possède entre autres les méthodes suivantes (qui pourront être complétées par la suite):

AlerteSMS
- abonnes: Map<String,String> ...
+ AlerteSMS() + ajouteNumero(numeroCompte : String, numeroMobile : String) + envoiSMS(numeroMobile : String, message : String) + getNumeroMobile(numeroCompte : String) : String ...

  où `abonnes` associe un numéro de mobile à un numéro de compte. Les noms de méthodes sont suffisamment explicites pour qu'on en devine la fonction.

D'autres types d'entités doivent pouvoir être envisagées et ajoutées simplement à l'application. C'est pourquoi on souhaite gérer par un **mécanisme d'évènements** la transmission d'informations sur les opérations effectuées d'un objet compte vers les entités telles que le journal ou le système d'envoi de SMS : pour chaque opération réalisée le compte émet un évènement et les entités abonnées sont notifiées de cet évènement. Il suffira alors d'abonner à un compte un journal d'opérations ou un système d'alertes SMS pour que ceux-ci soient notifiés des opérations sur le compte.

Q 1. Donnez le code java de la classe `MontantNegatifException`

Q 2. Donnez le code java du test unitaire de la classe `Compte`

Q 3. Donnez les diagrammes UML détaillés de toutes les entités (classes et/ou interfaces) nécessaires à la mise en place du mécanisme d'évènements évoqués ci-dessus. Vous ferez apparaître dans vos diagrammes les relations d'héritage et d'implémentation. Les types `Compte`, `JournalOperations` et `AlerteSMS` doivent apparaître dans vos diagrammes. Pour la classe `AlerteSMS` vous complèterez le schéma donné ci-dessus.

Q 4. Donnez le code java de la classe représentant les évènements.

Q 5. Donnez le code java des tests unitaires vérifiant le déclenchement approprié des évènements.

Q 6. Donnez les codes java des méthodes de gestion des évènements de `AlerteSMS` (le code des autres méthodes n'est pas demandé).

Q 7. Donnez le code java des classes `Compte` et `JournalOperations`.

Q 8. Indiquez (en donnant des lignes de code java par exemple) ce qu'il faut faire pour qu'un objet journal d'opérations `jo` et un objet système d'alerte `alerte` soient notifiés des opérations réalisées sur un objet compte `compte`.

Exercice 3 : Parcours d'arbres

On s'intéresse à des arbres binaires dont les nœuds sont étiquetés par des objets d'un type `T` générique. On dispose de la classe `ArbreBinaire` définie ainsi :

ArbreBinaire<T>
- racine : T - gauche : ArbreBinaire<T> - droit : ArbreBinaire<T> + ARBRE_VIDE : ArbreBinaire
- ArbreBinaire() + ArbreBinaire(t : T, g: ArbreBinaire<T>, d: ArbreBinaire<T>) + racine() : T + gauche() : ArbreBinaire<T> + droit() : ArbreBinaire<T> + estVide() : boolean

Les méthodes `racine`, `gauche` et `droit` lèvent une exception `ArbreVideException` si elles sont appelées sur la constante statique `ARBRE_VIDE` représentant l'arbre vide.

On s'intéresse au parcours de tels arbres binaires. Différents parcours sont possibles : *infixe*, *postfixe* et *préfixe*. Lors du parcours d'un arbre, on souhaite pouvoir réaliser une opération sur chaque nœud de l'arbre. Cette opération transforme l'étiquette de type `T` du nœud en une valeur de type `R`.

Q 1. Donnez le code java d'une interface `Operation` qui définit une telle opération. Cette interface impose pour seule méthode la méthode qui permet de transformer la valeur de type `T` passée en paramètre en une valeur de type `R`.

Q 2. Définissez une opération et le(s) test(s) associé(s) dont le traitement affiche la valeur passée en paramètre et retourne cette même valeur (sans la transformer).

Q 3. Définissez une opération et le(s) test(s) associé(s) dont le traitement transforme une chaîne de caractères en un entier représentant la longueur de cette chaîne.

On définit pour ces parcours une interface `ParcoursAB` qui dispose d'une seule méthode `parcours` qui prend en paramètre un arbre binaire `ab` et une opération `op`. Le résultat de cette méthode est le nouvel arbre obtenu à partir de l'arbre `ab`, où chaque nœud rencontré (dans l'ordre du parcours) subit la transformation `op`.

Q 4. Donnez le code d'une classe `ParcoursABInfixe` qui permet le parcours des arbres binaires selon un parcours infixe en appliquant une opération, quelque soit le type des éléments des étiquettes des nœuds et du résultat de l'opération.

Q 5. On suppose définies de manière équivalente les classes `ParcoursABPrefixe` et `ParcoursABPostfixe`. Donnez les lignes de code qui, pour une variable—supposée initialisée—*ab* de type `ArbreBinaire` dont les nœuds sont étiquetés par des chaîne de caractères (`String`), permettent :

- un affichage préfixe de *ab*,
- un affichage postfixe de *ab*,
- d'obtenir un nouvel arbre *res* à partir de *ab* en conservant la structure de *ab* mais en remplaçant, place pour place, toutes les étiquettes des nœuds de *ab* par des entiers correspondant aux longueurs des chaînes étiquettes de ces nœuds.