M1 Informatique - HAI712I - Ingénierie Logicielle

Année 2022-2023, session 1 de janvier 2023.

Durée : 2h00. Documents non autorisés. La précision et la concision des réponses sont notées ainsi que la lisibilité des codes. Les exemple de code de l'énoncé sont en Java, mais vous pouvez utiliser tout autre langage de programmation équivalent de votre choix pour vos réponses.

Contexte prétexte

Un industriel permet à ses clients d'acheter des ordinateurs qu'ils montent eux-même à partir de composants qu'on appelle des "composantOrdis" [1]. Dans les *composantOrdis*, on distingue les *composantSimples* et les *montages*. Les *composantSimples* sont de type *rack* (ou *tour*, nom donné à l'armature externe), *carte-mère*, *processeur*, barrettes mémoire vive *(RAM)*, *disque-dur*, *DVD*, *carte-vidéo*, *alimentation*, *ventilateur*, *écran*, etc. Un montage (montage quelconque, carte-mère-montée, ordinateur, réseau d'ordinateurs) est une collection de composantsOrdis. Il est possible d'ajouter un nombre quelconque de *composantOrdis* à un montage. Une *carte-mère-montée* est composée des composants simples suivants : une carte-mère, un processeur, un ventilateur, une RAM, etc. Un ordinateur est un montage composé des composants simples et des montages suivants : un rack, une carte-mère-montée, un disque-dur, unécran, etc. Un montage est aussi une sorte de composantOrdi car un montage peut être utilisé dans un autre montage.

L'industriel dispose d'une application informatique *CompOrdiFramework* intégrant, entre autres, une classe pour chaque concept précédemment listé, permettant de réaliser des ordinateurs et des montages et, en premier lieu, de calculer le prix Toutes Taxes Comprises ("All Taxes Included"), dit `prixTTC`, de tout `composantsOrdi`. Cette application vise à être un framework extensible et réutilisable. Le `prixTTC` d'un `composantOrdi` (voir listing 1, est le produit de son prix hors taxe (dit *prixHT*) par les taxes (TVA). Toutes deux peuvent varier pour chaque sorte de composant et sont calculées par les méthodes `float getPrixHT()` (pour le prix hors taxe) et `float getTVA()` pour les taxes. Le prix hors taxes d'un montage est la somme des prix hors taxes de ses *composantOrdis*. Le code de `getTVA()` n'est jamais demandé.

Dans l'application, la classe abstraite `ComposantOrdi` détient une méthode publique : `float getPrixTTC()` qui calcule le prix TTC de tout composantOrdi. Elle est paramétrée par spécialisation. On pose comme contrainte à l'examen que **cette méthode ne soit jamais modifiée ni redéfinie**.

```
1  public abstract class ComposantOrdi {
2      protected double TVA = 19.6; //valeur par
           défaut
3      protected double getTVA() {return TVA;};
4      abstract protected double prixHT();
5      public double prixTTC(){
6        return this.prixHT() * this.TVA();}
7      ...
8      }
```

Listing 1 – Extrait de la classe de base de *CompOrdiFramework*

A Framework - Architectures extensibles (environ 6 points)

1. Nommez et Décrivez (diagramme de classes, texte) l'architecture logicielle de *compOrdiFramework* telle que décrit par le cahier des charges précédent. Placez y la classe des montages, d'autres classes de factorisation éventuelles et au moins une classe représentative des composants simples (`RAM` par exemple).
2. En prenant comme exemple la méthode `prixTTC()` de la classe `ComposantOrdi` expliquez le lien entre les concepts de fonction d'ordre supérieur d'une part et de liaison dynamique en programmation par objets d'autre part.
3. Donnez une définition (code) de la classe `Montage` et de ses méthodes utiles à l'énoncé (ne définissez pas les accesseurs).
4. Où se trouvent les points d'extensions et les inversions de contrôle du framework *compOrdiFramework* ?

1. le terme "composant" ne dénote ici rien d'autre que dans le langage courant ; on dit par exemple qu'une carte video est l'un des composants d'un ordinateur.

Où et comment s'y fait l'injection de dépendances ?

5. Votre code permet-il d'écrire le code suivant ? Si non modifiez le pour qu'il le permette.

```
new Ordinateur()
  .add(new CarteMereMontée()
    .add(new CarteMere())
    .add(new Processeur("Intel-i5"))
    .add(new RAM()))
  .add( new DisqueDur("xyz"));
```

B Amélioration de l'architecture (1) - Cohérence des montages (environ 3 points)

La constitution de montages cohérents est un problème que doit traiter l'application, par exemple il ne doit pas être possible d'associer n'importe quelle carte mère avec n'importe quel processeur. Par ailleurs, la sélection un par un des composants peut être fastidieuse et il est pratique de pouvoir proposer des montages types pré-conçus. On définit par exemple en conséquense une sous-classe de `Montage` nommée `Montage-Prédéfini1`. L'utilisateur peut choisit une configuration de ce montage au moment de la création en passant une lettre en argument au constructeur, comme ceci : `new Montage-Préféfini1(''A'')`. Il est possible de créer de nouvelles sortes de montages pré-définies.

1. Donner l'architecture du programme et les points principaux du code réalisation cette fonctionnalité en suivant le schéma "Fabrique Abstraite" *(AbstractFabrik)*.

C Extensibilité et Typage Statique - (environ 6 points)

Dans une version Java du programme, on ajoute une méthode de signature `boolean equiv (ComposantOrdi c, String critère)` sur la classe `ComposantOrdi`. Cette méthode booléenne dit si le `composantOrdi` receveur est équivalent au composantOrdi argument selon un `critère` donné, et donc, dit si l'on peut remplacer l'un par l'autre dans un montage (le code d'une telle méthode n'est pas demandé au début de l'exercice).

1. Comme le calcul de l'équivalence entre deux montages nécessite un code spécifique, la méthode `equiv` doit être redéfinie sur la classe `Montage`.
   On vous donne le choix entre les deux signatures suivantes pour la méthode `equiv` sur la classe `Montage` :
   - `boolean equiv (Montage c, String critère)`
   - `boolean equiv (ComposantOrdi c, String critère)`

   a) Les deux compilent-elles (sachant qu'on teste la compilation de façon excluvie, l'une ou l'autre, pas les deux ensemble) ?
   b) Laquelle choisiriez vous pour cette redéfinition et pourquoi ?
2. On définit dans le code la méthode que vous avez choisie (à la question précédente) et on ne définit pas l'autre. (Vous devez avoir répondu à la question précédente pour répondre à celle-ci).

```
RAM m1 = new RAM();
Montage m2 = new Montage();
ComposantOrdi m3 = m1;
ComposantOrdi m4 = m2;
m2.equiv(m4,"x");
m3.equiv(m4, "x");
m4.equiv(m4,"x");
m4.equiv((Montage)m4,"x");
m4.equiv(m3, "x");
```

Listing 2 – Test de l'équivalence de `composantOrdis`

Considérons alors les affectations et envois de messages[2] (appels de méthodes) du listing 2 . Pour chacun de ces 4 envois de messages du listing 2, indiquez, laquelle des deux méthodes `equiv` (celle sur `ComposantOrdi` ou celle sur `Montage`), est invoquée (on suppose qu'il n'y en a pas d'autre définie ailleurs).
Expliquez vos réponses en termes de redéfinition et de surcharge de méthodes en présence de typage statique, de liaison dynamique et de polymorphisme d'inclusion,

3. Commenter l'instruction `ComposantOrdi m4 = m2;` dans le contexte d'une discussion sur les frameworks.

4. considérons l'envoi de message `m4.equiv(m1,''x'');` ; il va invoquer la méthode `equiv` de `ComposantOrdi` ou celle de `Montage`. Dans les deux cas il faudra comparer le receveur qui sera un `montage` avec l'argument qui sera une `RAM`, la réponse dans ce cas sera donc *false*, mais comment savoir que l'argument est une RAM sans faire un test explicite de type.
Proposez une solution pour traiter ce problème sans faire aucun test explicite de type (pas de "*instanceof*", pas de transtypage, pas "swith case") qui rendent le code non extensible. Vous pouvez par contre ajouter de nouvelles méthodes au système. Décrivez votre solution (points clés du code).

D Amélioration de l'architecture (2) - (environ 5 points)

Suite aux évolutions du marché, les mémoires vives (RAM) voient leurs prix hors taxe varier suite à différents facteurs (par exemple variations liées aux transport, variations liées aux matières premières, ...). On souhaite pouvoir intégrer dans l'application les formules relatives à chaque variation dans le calcul du prix sans toucher au code existant. On ne sait jamais à l'avance quand il va aura une variation, ni quelles seront sa cause, sa durée et la façon dont le calcul de la variation devra être effectué. Par exemple la variation liée au transport tient compte du poids du composant et la variation liée à une matière première tient compte du prix de celle-ci.

On souhaite intégrer dans l'architecture de l'application la possibilité d'intégrer à tout moment de telles causes de variations.

1. Discutez de l'intérêt respectif des schémas *State* ou *Decorator* pour intégrer la gestion de cette fonctionnalité de façon modulaire et extensible dans l'application et indiquez celle que vous jugez la meilleure. Justifiez un choix entre l'un et l'autre.
2. Pour votre solution préférée, donnez les éléments clés du code de votre solution. Les classes clés, les envois de messages clés qui font que cela fonctionne et que le résultat est extensible et modulaire doivent être identifiés. Entre autres, donnez le code de la (ou des) méthode(s) prixHT() que vous placez dans le système.

2. L'argument `"x"` simule le critère de comparaison, il n'a aucune importance dans les questions de ces exercices.