M1 Informatique - HAI712I - Ingénierie Logicielle

Année 2024-2025, session 2 de mars 2025.

Durée : 2h00. Documents non autorisés. Les parties B, C, D sont indépendantes, ordonnez les comme il vous convient ; la partie A et en particulier la question A.1 doit être faite en premier. Le barême est globalement indicatif. Merci par avance de prendre en compte les points suivants. La précision et concision des réponses est prise en compte. Pour toute question, si vous un connaissez un schéma de conception qui répond à la question vous avez bien sûr le droit de le nommer et de l'utiliser. Ecrivez lisiblement et pas trop petit sous peine de ne pas être lus. Utilisez un brouillon pour préparer vos réponses, je ne lis pas les textes ou codes multi-raturés. Inscrivez vos noms et prénoms sur chaque copie, à l'endroit prévu à cet effet, et dans l'ordre demandé. Numérotez vos feuilles à l'emplacement prévu à cet effet.

Contexte prétexte aux questions

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 prixHT()` (pour le prix hors taxe) et `float TVA()` pour les taxes. Le prix hors taxes d'un montage est la somme des prix hors taxes de ses *composantOrdis*. Le code des méthodes `TVA()` n'est jamais demandé dans l'examen.

Dans l'application, la classe abstraite `ComposantOrdi` détient une méthode publique : `float prixTTC()` qui calcule le prix TTC de tout composantOrdi. On pose comme contrainte à l'examen que **cette méthode ne soit jamais modifiée (elle fait partie du framework et n'est pas accessible), ni redéfinie (c'est inutile).**

```
1  package compOrdiFramework;
2  public abstract class ComposantOrdi {
3      protected double TVA = 19.6; //par défaut
4      protected double TVA() {return TVA;};
5      abstract protected double prixHT();
6      public double prixTTC(){
7        return this.prixHT() * this.TVA(); }
8      ...
9    }
```

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

A Framework - Architectures extensibles (environ 6 points)

1. Un ordinateur, et plus généralement, tout montage est clairement représenté dans cette application par une arborescence d'objets qui relève du schéma de conception *Composite*.

   Selon ce schéma, décrivez (diagramme de classes UML précis) l'architecture logicielle de *compOrdi-Framework* respectant le cahier des charges précédent. Placez y la classe `Montage`, d'autres classes de

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.

factorisation éventuelles, la classe `Ordinateur` et au moins deux classes représentatives des composants simples, (`RAM` et `DisqueDur` par exemple).

2. Expliquez (pas plus de 3 lignes) en quoi son architecture permet d'étendre *compOrdiFramework* sans modifier son code ?
3. Quelle est l'instruction du framework qui réalise une inversion de contrôle ?
4. Donnez le code Java de l'intégration au framework de la classe `Clavier` (un ordinateur possède un clavier). La classe `Clavier` doit définir au moins une méthode invoquée via une invertion de contrôle du framework.
5. La documentation du schéma *Composite* stipule que : "*When dealing with tree-structured data, programmers often have to discriminate between a leaf and a node ...*". Commentez cette phrase dans le contextre de l'énoncé (qui sont les feuilles et qui sont les noeuds ... attention à ne pas confondre classes et instances dans votre réponse).
6. Expliquez comment la méthode **prixTTC()** de l'énoncé effectue la distinction entre feuille (*lead*) et noeud *node*.
7. 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.

B Variation sur l'architecture, contrôle des stocks (environ 3 points)

1. Pour ne pas risquer de rupture de stock, chaque classe de composant simple doit être doté d'un mécanisme qui empêche de créer plus d'instances de la classe qu'il n'y a de produits réels en stock. En appliquant le bon schéma de conception, donnez la partie du code de la classe `Ventilateur` réalisant cette fonctionnalité. Donc par exemple, on doit pouvoir positionner une variable indiquant le nombre de ventilateurs en stock dans le magasin et il doit être impossible de créer plus d'instances de la classe `Ventilateur` que ce nombre.
2. Est-il possible de factoriser la solution que vous avez mise en place sur la classe abstraite représentant les composants simples ? Si c'est possible, cela éviterait de remettre en place le dispositif au niveau de chaque sous-classe. Si oui dites comment (ne donnez pas de code).
3. Connaissez vous un langage où `new` est une vraie méthode et où il est possible de la redéfinir ? En quoi cela change-t-il l'implantation de la solution ? Donnez l'idée, pas de code.

C Extensibilité et Typage Statique - (environ 6,5 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 de cette méthode n'est pas demandé pour les premières question).

1. Considérons le cas des montages, l'équivalence entre deux montages se calcule en itérant sur les deux montages et en regardant s'il existe dans le second montage un équivalent à tout élément du premier (on se contentera de cet algorithme peu optimisé). Ce calcul est donc spécifique aux montage et necessite une méthode spécialisée définie sur la classe `Montage`, qui est rappelons-le une sous-classe de `ComposantOrdi`.

   En plus de celle sur `ComposantOrdi`, on propose donc de défnir une méthode `equiv` sur la classe `Montage` ; nous décidons sa signature comme ci-dessous puisque nous voulons comparer des montages entre eux :
   - `boolean equiv (Montage c, String critère)`

2. Considérons alors les affectations et envois de messages[2] (appels de méthodes) du listing 2 .

```
1  RAM m1 = new RAM();
2  Montage m2 = new Montage();
3  ComposantOrdi m4 = m2;
```

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

```
4  m2.equiv(m2,"x");
5  m2.equiv(m4,"x");
6  m2.equiv((Montage)m4,"x");
7  m4.equiv(m2,"x");
8  m4.equiv(m4,"x");
9  m4.equiv((Montage)m4,"x");
```

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

Pour chacun des envois de messages du listing 2, indiquez laquelle des deux méthodes `equiv` données dans l'énoncé est invoqué : soit celle de `ComposantOrdi` soit celle de `Montage` (on suppose qu'il n'y en a pas d'autre définie sur une autre classe à cet instant),

Expliquez chaque réponse de façon concise (utilisez bien les termes "envoi de message", "receveur", ne confondez pas une classe et ses instances, utilisez le vocabulaire du typage statique).

3. Dans l'idéal ce que l'on voudrait est que tous les envois de messages précédents du listing 2 invoquent la méthode `equiv(...)` de `Montage` puisque les variables m2 et m4 y référencent des montages.

   Si vous pensez que c'est nécessaire, changez l'entête de la méthode equiv(...) sur `Montage`.
4. Défininissez complètement (entête et corpps) la méthode `equiv(...)` de la classe `Montage`. Le corps réalisera l'algorithme précédemment évoqué).
5. considérons l'envoi de message `m1.equiv(m2,''x'');` ; il va invoquer la méthode `equiv` de `ComposantOrdi`. Dans la méthode, il faudra comparer le receveur qui sera une `RAM`, avec l'argument qui sera un `Montage`, la réponse dans ce cas sera donc *false*, mais comment savoir que l'argument est une instance de `Montage` 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 soit avec du texte soit avec du code soit les deux.

D Variation sur l'architecture (2) - (environ 4,5 points)

Suite aux évolutions du marché, les disques durs (classe `DisqueDur`) voient leurs prix hors taxe varier suite à différents facteurs (par exemple des variations liées aux transport, et des 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 y aura une variation, ni quelle sera sa cause, sa durée ou la façon dont le calcul de la variation devra être effectué. Par exemple la variation liée au transport tient compte du poids du composant transporté 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. Imaginez une modification de l'architecture de `CompOrdiFramework` pour intégrer cette fonctionnalité en l'appliquant au cas des disques durs.

   Donnez un diagramme UML de votre solution (ne montrez que la partie modifiée de l'architecture). N'oubliez pas la légende des flêches, les rôles et les cardinalitées.
2. Donnez le code de la méthode `prixHT()` correspondant à une augmentation de 20% du prix du transport des disques durs (sans réécrire la classe `prixHT()` des disques durs bien sûr)
3. Donnez un exemple de `main` dans lequel vous ajoutez une `variationTransport` et une `variationMatierePremie` à une instance de la classe `DisqueDur` référencée dans une variable nommée `dd`.
4. Faite un schéma décrivant en mémoire les objets que l'on peut atteindre via la variable `dd`.