Corrigé - Examen Ingénierie Logicielle (HAI721) - Session 2

Durée : 2h00. Précision et concision notées. Codes en Java (comme l'énoncé).

A Framework - Architectures extensibles (6 points)

1. Pattern **Composite**.

Diagramme UML (texte) :

```
ComposantOrdi (abstract)
  - prixTTC() : double
  + TVA() : double (protected)
  + prixHT() : double (abstract protected)

  ↑
ComposantSimple (abstract, optionnelle pour factoriser composants simples)
  ↑
RAM
  - prixBase : double
  + prixHT() : return prixBase

  ↑ (from ComposantOrdi)
Montage
  - composants : List<ComposantOrdi>
  + add(ComposantOrdi c)
  + prixHT() : somme des composants.prixHT()

  ↑
Ordinateur
```

2. L'architecture permet l'extension via sous-typage (nouvelles sous-classes de `ComposantOrdi` implémentent `prixHT()` sans modifier code existant ; polymorphisme assure intégration). Respecte Open/Closed.

Code pour `Clavier` :

```
public class Clavier extends ComposantSimple {
    private double prixBase = 50.0;

    @Override
```

```
    protected double prixHT() {
        return prixBase;
    }
}
```

(Méthode `prixHT()` invoquée via inversion de contrôle dans `prixTTC()`.)

3. L'appel `this.prixHT()` dans `prixTTC()` (framework appelle méthode des extensions).

4. Dans les données arborescentes, il faut distinguer feuilles (terminaux) et branches (composites). Ici, feuilles : instances de `ComposantSimple` (e.g., RAM sans enfants). Noeuds : instances de `Montage` (e.g., Ordinateur avec enfants).
`prixTTC()` distingue via polymorphisme : appelle `prixHT()`, qui est fixe pour feuilles, récursif (somme enfants) pour noeuds ; pas de test explicite.

5. `prixTTC()` est fonction d'ordre supérieur : compose avec `prixHT()` (fournie par sous-classes). Liaison dynamique résout runtime quelle `prixHT()` exécuter (selon type concret), rendant extensible.

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

1. Pattern **Singleton** (par type, multiton si multi-instances limitées). Pour limite stock :

```
public class DisqueDur extends ComposantSimple {
    private static int stock = 0; // Setter pour initialiser
    private static int count = 0;

    private DisqueDur() { /* privé */ }

    public static DisqueDur create() {
        if (count >= stock) throw new RuntimeException("Stock épuisé");
        count++;
        return new DisqueDur();
    }

    @Override
    protected double prixHT() { /* impl */ }
}
```

2. Oui, factoriser via classe abstraite `ComposantSimple` avec attributs/méthodes statiques protégés, redéfinis par sous-classes (chaque sous-classe gère son stock indépendamment).

3. En Smalltalk ou Ruby, `new` est méthode redéfinissable. Change : implémenter directement dans `new` le contrôle (pas besoin constructeur privé + factory).

C Extensibilité et Typage Statique (6 points)

1. `equiv(ComposantOrdi c, String critere)` : redéfinition (override, même signature).
`equiv(Montage c, String critere)` : surcharge (overload, signature différente par argument).
Typage statique : surcharge résolue compile-time (types statiques), override runtime (type dynamique receveur). Permet spécialisation sans casts.

2. (Listing 2 : similaire à S1, envois : m2.equiv(m4,"x"); m3.equiv(m4,"x"); m4.equiv(m4,"x"); m4.equiv((Montage)m4,"x"); m4.equiv(m3,"x");)

- m2.equiv(m4,"x") : surcharge `equiv(Montage, String)` (types statiques Montage, Montage).
- m3.equiv(m4,"x") : `ComposantOrdi.equiv` (type statique receveur ComposantOrdi).
- m4.equiv(m4,"x") : override `Montage.equiv(ComposantOrdi, String)` (liaison dynamique, receveur Montage).
- m4.equiv((Montage)m4,"x") : surcharge `equiv(Montage, String)` (cast force type statique argument).
- m4.equiv(m3,"x") : override `Montage.equiv(ComposantOrdi, String)` (liaison dynamique).

Redéfinition : héritage, résolution dynamique. Surcharge : pas héritage, résolution statique par signatures.

3. Illustre polymorphisme dans frameworks : référence abstraite (`ComposantOrdi`) pointe instance concrète (`Montage`), permet extension sans modifier code client (Open/Closed).

4. Solution : **double dispatch**.
Ajouter méthode abstraite `boolean equivTo(ComposantOrdi other, String crit)` dans `ComposantOrdi` .
Dans `equiv(ComposantOrdi c, String crit) : return c.equivTo(this, crit);` (inversion).
Sous-classes redéfinissent `equivTo` pour types spécifiques (e.g., dans `RAM` : if other instanceof Montage return false; else comparer).
Pour m1.equiv(m2,"x") : appelle m2.equivTo(m1,"x") → exécute `Montage.equivTo` sans test dans `RAM` .

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

1. Pattern **Decorator** pour ajouter variations dynamiquement.

UML (modifié) :

```
ComposantOrdi (abstract)
  ↑
VariationPrix (abstract decorator)
  - wrapped : ComposantOrdi
  + prixHT() : wrapped.prixHT() + variation()
```

```
  ↑
VariationTransport
  + variation() : poids * facteur

  ↑
VariationMatierePremiere
  + variation() : prixMatiere * quantite
```

(Appliquer à RAM : decorators wrap RAM.)

2. Exemple main :

```
ComposantOrdi ram = new RAM();
ram = new VariationTransport(ram); // Ajoute variation transport
ram = new VariationMatierePremiere(ram); // Ajoute variation matière
double prix = ram.prixTTC();
```

3. Schéma mémoire (texte) :
VariationMatierePremiere → VariationTransport → RAM
(Chaîne decorators, chaque wrappe précédent ; appels prixHT() récursifs.)

4. Composite : arborescence pour structure hiérarchique (noeuds/feuilles même interface, récursion).
Decorator : arborescence linéaire (chaîne wrappers autour objet base, ajoute comportements).
Les deux construisent arbres, mais Composite pour composition, Decorator pour décoration/extensibilité.