Espace Malraux - Place Saint Clair
BP 86
14203 Hérouville Saint-Clair Cedex

le cnam Normandie

# 1) Questions de cours

## a) Patron de conception

Un patron de conception est un agencement caractéristique de modules ou composants logiciels en programmation par objets (classes instanciables, classes abstraites, interfaces) qui répond à un problème de conception récurrent. Il décrit les grandes lignes d'une solution reconnue comme une bonne pratique, modifiable et adaptable en fonction des besoins.

## b) Familles de patrons

Les problèmes à résoudre dans les logiciels relèvent de 3 catégories ou familles différentes :
- structuration des composants logiciels
- comportement des composants logiciels
- création des composants logiciels

Les patrons de structuration expliquent comment assembler des objets et des classes en de plus grandes structures, tout en les gardant flexibles et efficaces.

Les patrons de comportement mettent en place une communication efficace et répartissent les responsabilités entre les objets.

Les patrons de création fournissent des mécanismes flexibles de création d'objets.

## c) Exemples de patron

Dans la famille des patrons de structuration, on peut citer :
- adaptateur (adpater) : permet de convertir l'interface d'une classe en une autre interface attendue par le client ;
- composite : permet de décrire une hiérarchie d'objets et de manipuler de la même manière un élément unique, une branche ou l'ensemble de l'arbre ;
- décorateur (decorator) : offre une alternative à l'héritage pour ajouter des services à un objet dit décoré, en permettant d'attacher dynamiquement des responsabilités à l'objet décoré ;
- façade : fournit une interface unique sur un ensemble d'objets constituants d'un système ;
- procuration (proxy) : fournit un intermédiaire pour l'utilisation d'un objet, ce qui permet d'en contrôler l'utilisation.

Dans la famille des patrons de comportement, on peut citer :
- commande (command) : décrit une action à effectuer sous la forme d'un objet autonome qui contient tous les détails de cette action, ce qui permet de partager une même action pour plusieurs invocateurs, ou de conserver plusieurs actions à exécuter en séquence (macro-commande)… ;
- itérateur (iterator) : fournit une interface harmonisée d'accès séquentiel aux éléments d'une collection d'objets sans connaître la complexité de la structure de la collection ;
- mémento (memento) : permet de sauvegarder et de rétablir l'état précédent d'un objet sans révéler les détails de l'implantation de l'objet considéré ;
- observateur (observer) : établit une relation un à plusieurs entre des objets, où lorsqu'un objet change, plusieurs autres objets sont avisés du changement ;
- stratégie (strategy) : permet de définir différentes manières de réaliser une opération et d'en appliquer une dynamiquement en fonction du contexte (la situation de l'objet sur lequel l'opération doit se faire) ;
- visiteur (visitor) : sépare les algorithmes et les objets sur lesquels ils opèrent et permet d'ajouter des services à des classes d'objets sans changer ni hériter de ces classes.

Dans la famille des patrons de création, on peut citer :

- fabrique (factory) : permet d'instancier des objets dont le type est dérivé d'un type abstrait. La classe exacte de l'objet instancié dépend de paramètres fournis par l'objet à l'initiative de l'instanciation (l'appelant) mais n'est pas connue par l'appelant ;
- singleton : restreint l'instanciation d'une classe à un seul représentant.

## 2) Singleton

### a) Déclaration de la variable représentant la banque

```
private static Banque laBanque;
```

### b) Code de la méthode laBanque()

```
public static Banque laBanque() {
      if (laBanque == null) laBanque = new Banque();
      return laBanque;
}
```

## 3) Commande

### a) Définition de l'interface CommandeProgrammée.

```
public interface CommandeProgrammée extends Commande {
      Date date();
}
```

### b) Définition de la classe CommandePaiementCB.

```
public class CommandePaiementCB implements CommandeProgrammée {
   private int numéroCompte;
   private int montant;
   private String numéroCB;
   private Date date;
   public CommandePaiementCB( int numCompteCrédité,
                              int montantPayé,
                              String numCBDébiteur,
                              Date dateDuPaiement) {
      numéroCompte = numCompteCrédité;
      montant = montantPayé;
      numéroCB = numCBDébiteur;
      date = dateDuPaiement.copie();
   }
   public Date date() {
      return date;
   }
   public void execute() throws Exception {
      Banque.laBanque().paiement(numéroCompte, numéroCB, montant);
   }
   public String toString() {
      return "Le " + date
          + " paiement de " + String.format("%.2f", (float)montant/(float)100)
          + " sur le compte " + numéroCompte
          + " depuis le compte du porteur de la carte " + numéroCB;
   }
}
```

### c) Définition de la méthode paiementDifféré de la classe Banque.

```
public void paiementDifféré(int numCptBénéficiaire,
                            String numCBClient,
                            int montant,
                            Date quand) {
    System.out.print("Le " + Date.aujourdhui());
    System.out.print(", programmation d'un paiement de ");
    System.out.print(String.format("%.2f", (float)montant/(float)100));
    System.out.println(" le " +  quand);
    CommandePaiementCB cmd = new CommandePaiementCB(numCptBénéficiaire,
                                                    montant,
                                                    numCBClient,
                                                    quand);
    paiementsProgrammés.add(cmd);
}
```

### d) Définition de la méthode paiementNFois de la classe Banque.

```
public void paiementNFois(int numCptBénéficiaire,
                          String numCBClient,
                          int montant,
                          int nombre) throws Exception {
      int mntDUnPaiement = montant / nombre;
      int mntPremierPaiement = montant - (mntDUnPaiement * (nombre - 1));
      paiement(numCptBénéficiaire, numCBClient, mntPremierPaiement);
      Date quand = Date.aujourdhui().copie();
      for(int i = 1; i < nombre; i++) {
            quand.plusUnMois();
            paiementDifféré(numCptBénéficiaire,
                            numCBClient,
                            mntDUnPaiement,
                            quand);
      }
      System.out.println();
}
```

## 4) Exécution différée des commandes

### Définition de la méthode traitrePaiementsProgrammés de la classe Banque.

```
public void traitePaiementsProgrammésDuJour() throws Exception {
      Iterator<CommandeProgrammée> it = paiementsProgrammés.iterator();
      while(it.hasNext()) {
            CommandeProgrammée cmd = it.next();
            if (cmd.date().egale(Date.aujourdhui())) {
                  cmd.execute();
                  it.remove();
            }
      }
}
```