Séance 11 – Exercices - Patrons de conception

Les exercices sont issus du livre de Laurent Debrauwer : Design patterns, les 23 modèles de conception

Cartes de paiement

Les clients d'une banque sont classés en deux catégories :
- ceux qui ont le droit au crédit;
- ceux qui n'ont pas le droit (débit immédiat sur leur compte)

Lors de la demande d'une carte de paiement, les premiers recoivent une carte de crédit (à débit différé sur leur compte, alors que les seconds peuvent seulement avoir une carte de crédit à débit immédiat sur leur compte.

1. Quel patron de conception permet-il de modéliser la création de la carte de paiement en fonction du client ?
2. Modélisez son utilisation par un diagramme de classes.

correction

Le patron *factory method.* La création de la carte est réalisée dans la sous-classe correspondant à la nature du client.

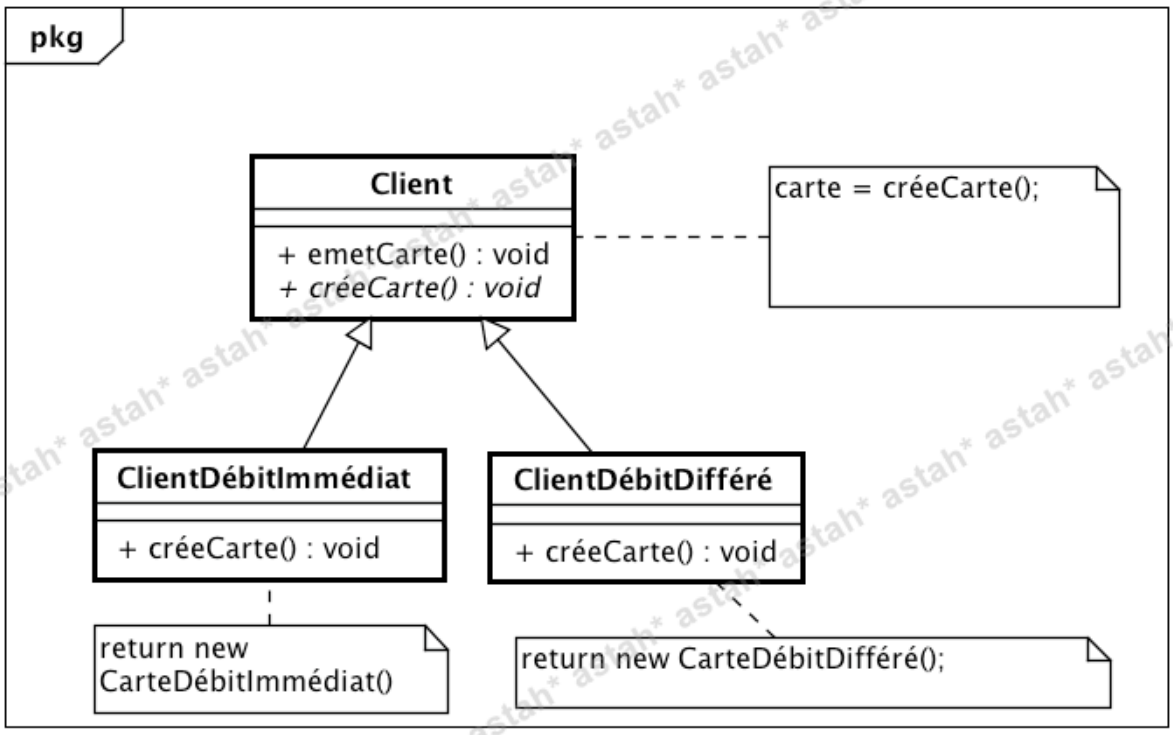


Figure 1. Le patron *factory method* appliqué à la méthode *créeCarte*

Il existe deux modèles de cartes de débit et de crédit, à savoir les cartes Visa et les cartes MasterCard.
Modélisez à l'aide d'un diagramme de classes la création d'une carte de paiement en fonction de sa famille (de crédit ou de débit) en utilisant le pattern `FabriqueAbstraite`.

correction

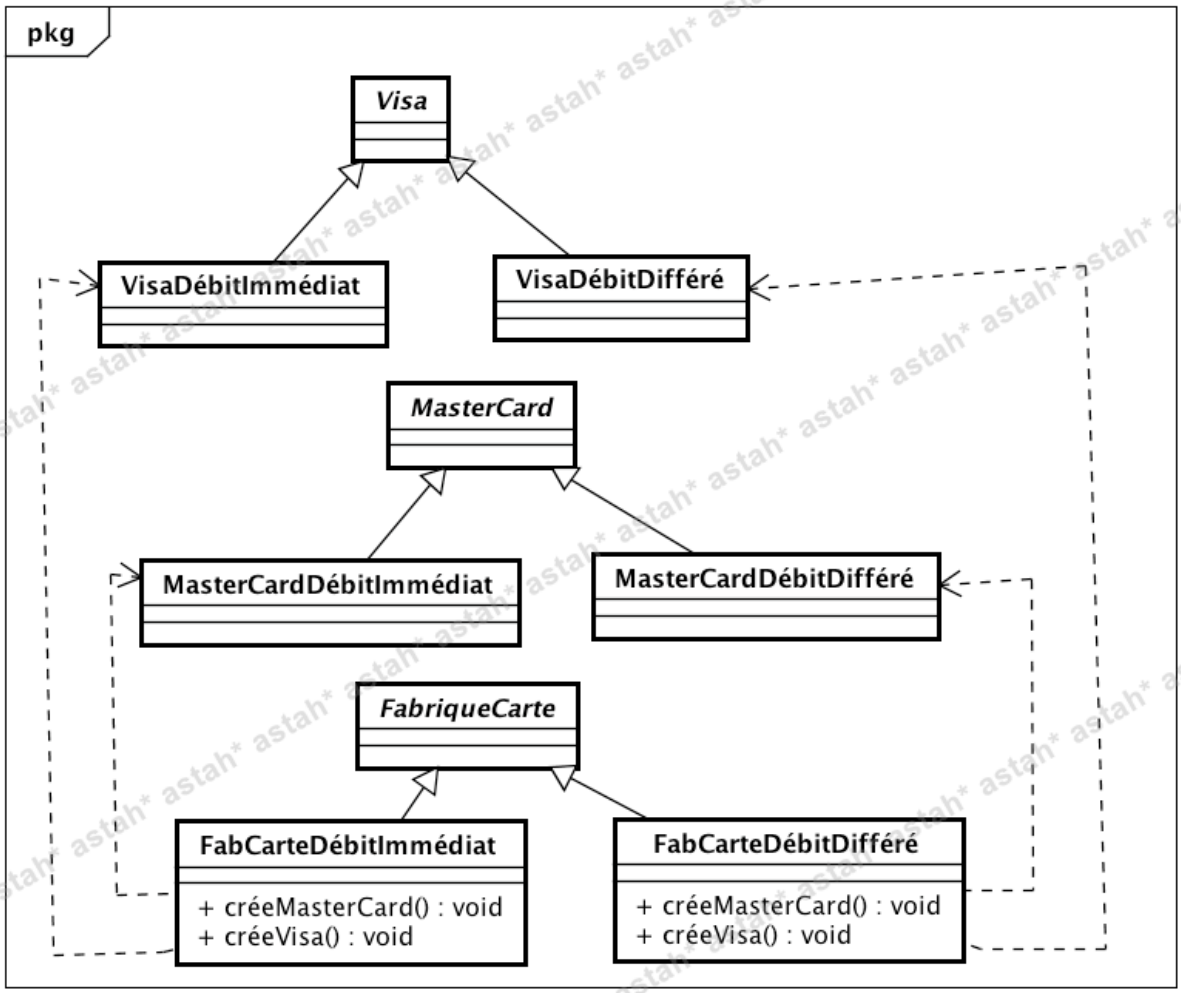


Figure 2. Le patron *FabriqueAbstraite* appliqué aux lignes de produit Visa et MasterCard

Lors d'un achat avec une carte de paiement, une autorisation doit être accordée. Si la carte est une carte de crédit à débit immédiat, l'autorisation est accordée si le solde du compte sur lequel la carte est débitée est suffisant. Si la carte est une carte de crédit à débit différé, l'autorisation est accordée si le montant mensuel des dépenses n'a pas dépassé le plafond.

1. Quel patron de conception permet-il de modéliser l'autorisation lors d'un achat avec une carte de paiement en fonction du modèle de la carte ?
2. Modélisez son utilisation par un diagramme de classes.

correction

Le patron *template method* permet de distinguer l'autorisation de paiement en fonction du modèle de carte. La méthode `autorisePaiement` est abstraite dans la classe CartePaiement. Elle est implantée différemment dans les deux sous-classes `CarteCrédit` et `CarteDébit` relativement à l'énoncé.

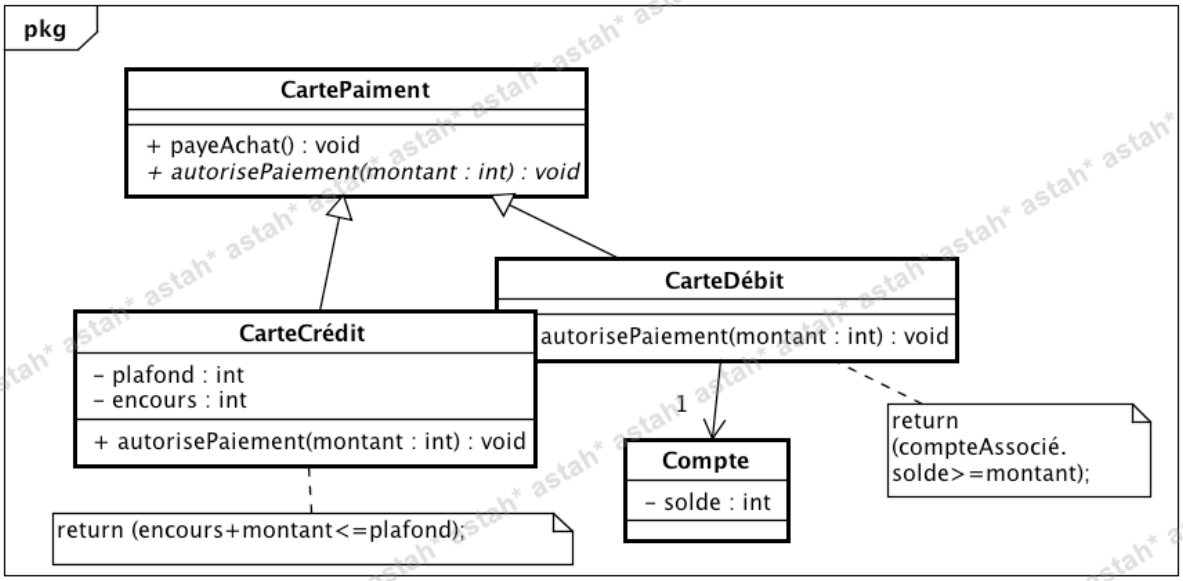


Figure 3. Le patron *template method* appliqué à la méthode *autorisePaiement*

Système de fichiers

Un répertoire contient des sous-répertoires et des fichiers. Un système de fichiers consiste en un ensemble de sous-répertoires et de fichiers contenus dans un répertoire « racine ». Les fichiers et les répertoires appartiennent à un utilisateur.

1. Quel patron permet-il de modéliser un système de fichiers ?

Un système de fichiers est basé sur une composition récursive : il faut donc utiliser le patron *Composite*.

2. Montrez cette modélisation à l'aide d'un diagramme de classes.

Un Répertoire est composé de Noeuds, qui peuvent être soit des Répertoires, soit des Fichiers. Dans la figure suivante, on montre aussi la relation entre Répertoire et Système de Fichiers.

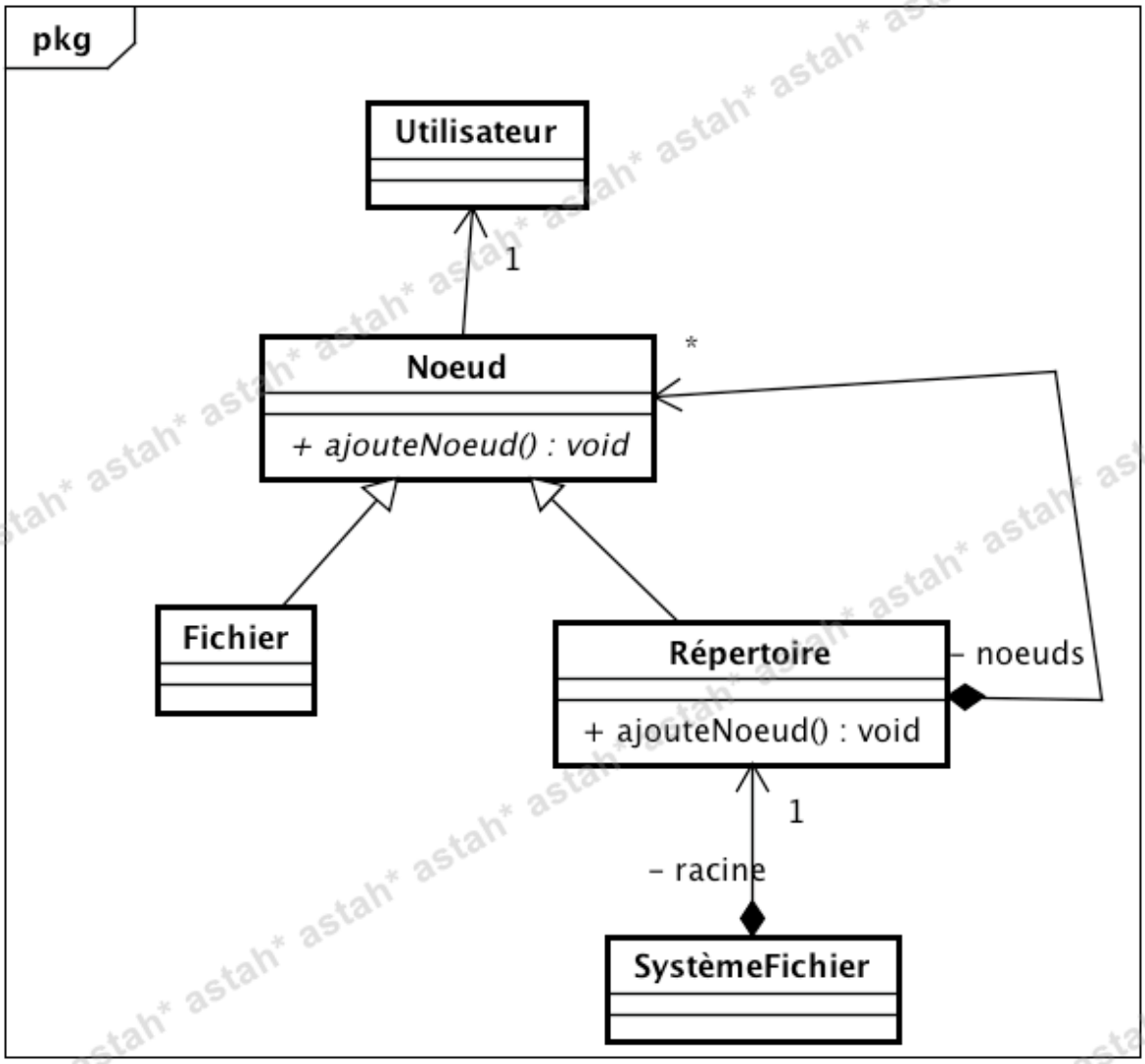


Figure 4. Le patron *composite* appliqué à un système de fichiers

Chaque fichier possède un attribut qui contient sa taille. Nous voulons maintenant connaître le nombre de fichiers et de répertoires ainsi que la taille globale du système de fichiers (les deux calculs doivent être séparés).

3. Quel patron permet-il de calculer ces informations en modifiant au minimum le diagramme des classes de la question 2.
4. Intégrez ce patron dans le diagramme de classes de la question 2
5. Le programme principal devra construire un exemple de système de fichiers et calculer le nombre de fichiers et de répertoires de cet exemple ainsi que sa taille globale.

Correction

Le patrons qui permet d'appliquer un certain nombre d'opérations sur des objets, sans en modifier la structure est le patron *Visiteur.*

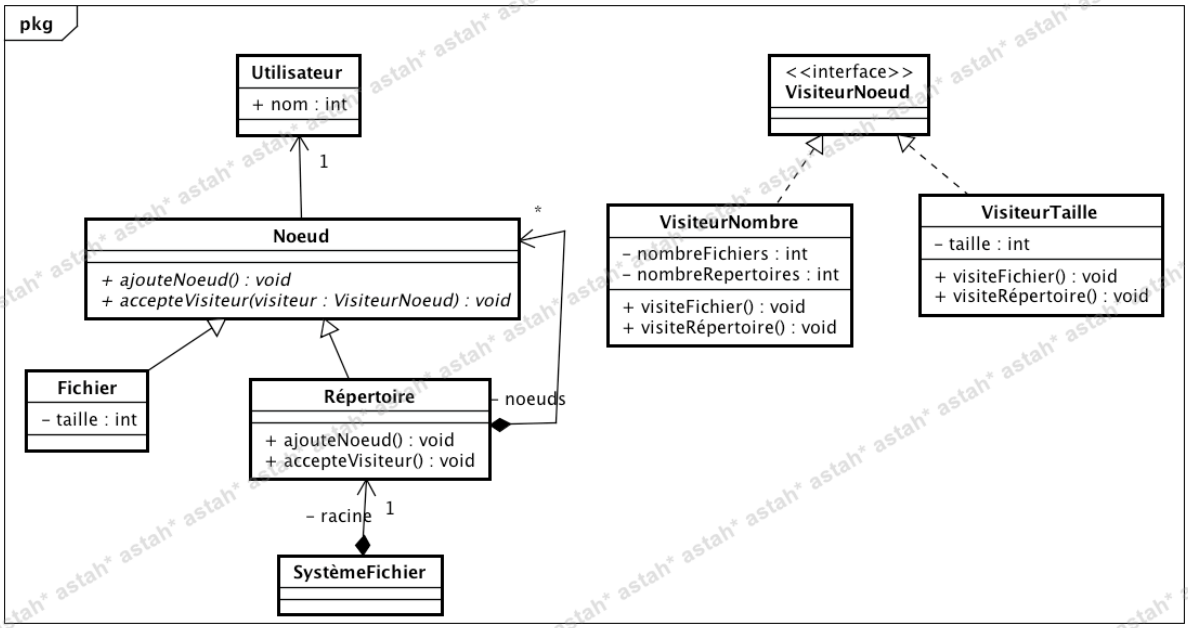


Figure 5. Le patron *visiteur* appliqué à un système de fichiers, pour compter les fichiers, et calculer la taille des répertoires.

Le programme est composé de plusieurs classes. Le programme principal est Test.java

```java
package patronsUML;

public class Fichier extends Noeud{
	protected int taille ;

	public Fichier(Utilisateur u, int taille) {
		super(u);
		this.taille = taille;
	}

	public int getTaille(){
		return taille;
	}

	@Override
	public void accepteVisiteur(VisiteurNoeud visiteur) {
		visiteur.visiteFichier(this);

	}

	@Override
	public boolean ajouteNoeud(Noeud noeud) {
		return false;
	}
}
```

```java
package patronsUML;

public abstract class Noeud {
	protected Utilisateur utilisateur;

	public Noeud(Utilisateur utilisateur){
		this.utilisateur = utilisateur ;
	}
```

```
	public abstract boolean ajouteNoeud(Noeud noeud);

	public abstract void accepteVisiteur(VisiteurNoeud visiteur);

}
```

```
package patronsUML;

import java.util.ArrayList;
import java.util.List;

public class Repertoire extends Noeud {

	protected List<Noeud> noeuds = new ArrayList<Noeud>();

	public Repertoire(Utilisateur u){
		super (u);
	}

	@Override
	public void accepteVisiteur(VisiteurNoeud visiteur) {
		visiteur.visiteRepertoire(this);
		for (Noeud noeud : noeuds) {
			noeud.accepteVisiteur(visiteur);
		}
	}

	@Override
	public boolean ajouteNoeud(Noeud noeud) {
		// TODO Auto-generated method stub
		return noeuds.add(noeud);
	}

}
```

```
package patronsUML;

public class SystemeFichier {
	protected Repertoire racine;

	public SystemeFichier(Repertoire racine){
		this.racine = racine;
	}

	public void accepteVisiteur (VisiteurNoeud visiteur){
		racine.accepteVisiteur(visiteur);
	}

}
```

```
package patronsUML;

public class Utilisateur {
	protected String nom ;

	public Utilisateur(String nom){
		this.nom = nom;
	}

	public String getNom() {
		return nom;
	}
```

```
}
```

```
package patronsUML;

public abstract class VisiteurNoeud {

		public abstract void visiteFichier(Fichier f);
		public abstract void visiteRepertoire (Repertoire dir);
}
```

```
package patronsUML;

public class VisiteurNombre extends VisiteurNoeud {

	protected int nombreFichiers = 0;
	protected int nombreRepertoires = 0;

	@Override
	public void visiteFichier(Fichier f) {
		nombreFichiers = nombreFichiers+1;
	}

	@Override
	public void visiteRepertoire(Repertoire dir) {
		nombreRepertoires = nombreRepertoires + 1;
	}

	public int getNombreFichiers(){
		return nombreFichiers;
	}

	public int getNombreRepertoire(){
		return nombreRepertoires;
	}
}
```

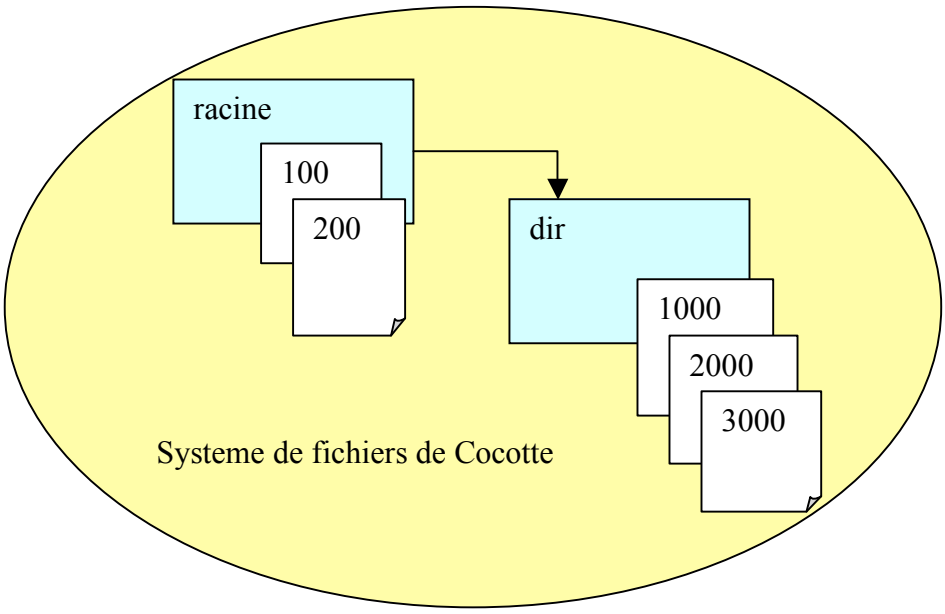
```
package patronsUML;

public class VisiteurTaille extends VisiteurNoeud {

	int tailleTotale = 0;

	@Override
	public void visiteFichier(Fichier f) {
		tailleTotale = tailleTotale+f.getTaille();
	}

	@Override
	public void visiteRepertoire(Repertoire dir) {
		//do nothing
	}

	public int getTailleTotale() {
		return tailleTotale;
	}

}
```

```
package patronsUML;
```

```java
public class Test {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Utilisateur moi = new Utilisateur("Cocotte");

		Repertoire racine = new Repertoire(moi);
		Repertoire dir = new Repertoire(moi);

		racine.ajouteNoeud(dir);
		racine.ajouteNoeud(new Fichier(moi, 100));
		racine.ajouteNoeud(new Fichier(moi, 200));
		dir.ajouteNoeud(new Fichier(moi, 1000));
		dir.ajouteNoeud(new Fichier(moi, 2000));
		dir.ajouteNoeud(new Fichier(moi, 3000));

		SystemeFichier systemfichiers = new SystemeFichier(racine);

		VisiteurNombre visiteurNombre = new VisiteurNombre();
		systemfichiers.accepteVisiteur(visiteurNombre);
		System.out.println("nombre de fichiers du syst√®me de fichiers :
"+visiteurNombre.getNombreFichiers());
		System.out.println("nombre de r√©pertoires du syst√®me de fichiers :
"+visiteurNombre.getNombreRepertoire());

		VisiteurTaille visiteurTaille = new VisiteurTaille();
		systemfichiers.accepteVisiteur(visiteurTaille);
		System.out.println("Taille du syst√®me de fichiers :
"+visiteurTaille.getTailleTotale());

	}

}
```

Que retourne le programme Test ? Dessinez l'arborescence des répertoires et fichiers.

```
nombre de fichiers du système de fichiers : 5
nombre de répertoires du système me de fichiers : 2
Taille du système de fichiers : 6300
```



États de la vie professionelle d'une personne

Une personne possède un cycle au long de sa vie professionnelle. Elle est d'abord étudiante, puis intègre la vie active, puis prend sa retraite.
Son comportement varie en fonction de l'état où elle se trouve. Notamment, elle se cotise pour sa retraite que durant la vie active. Les cotisations pour la retraite donnent lieu à l'ajout de points de retraite.

1. Quel patron est le mieux adapté pour décrire une personne au long de sa vie professionnelle ?
2. Concevez le diagramme de classes décrivant la personne en utilisant ce patron. Introduisez les méthodes `getNom` et `ajoutePoints` dont le comportement dépend de l'état. La méthode `getNom` renvoie le nom de l'état de la personne. La méthode `ajoutePoints` ajoute des points de retraite.

Correction
Le patron Etat, qui permet d'adapter le comportement des méthodes en fonction de l'état de l'objet (ici la Personne).
Par exemple, `ajoutePoints()` qui cumule les points de retraite ne possède d'implémentation que dans l'état Actif. Dans l'état Etudiant ou Retraité, la méthode ne fait rien. La méthode `ajoutePoints()` utilise la méthode `setPtsRetraite` de la classe Personne dont l'usage est réservé à la classe Etat et à ses sous-états.

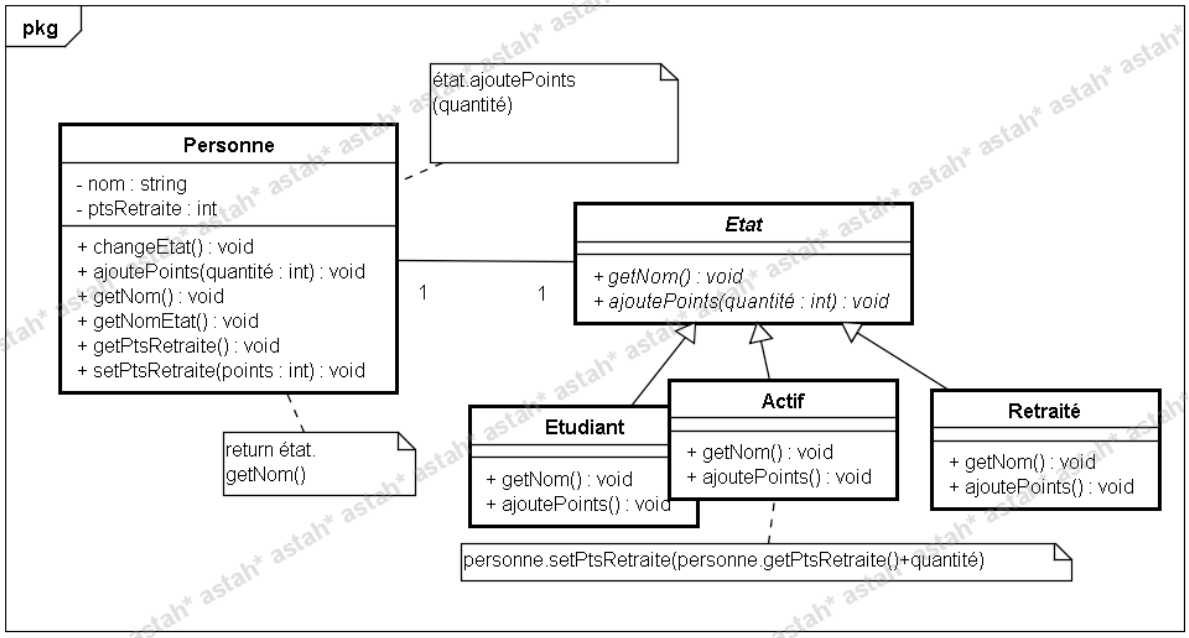


Figure 6. Le patron *état* appliqué à la vie professionnelle d'une personne

Il s'agit maintenant de créer une interface graphique qui affiche la personne ainsi que son état professionnel. Chaque fois que des données de la personne ou de son état son changées, les données affichées dans l'interface graphique sont mises à jour automatiquement. Il s'agit ici du nom de l'état et des points de retraite.

3. Quel patron est le mieux adapté pour concevoir cette interface graphique ?

4. Concevez le diagramme de classes correspondant, en intégrant le diagramme de la question 2.

Le patron observateur permet de réaliser la mise à jour automatique entre plusieurs objets (dont une IHM par exemple, qui reflète l'état du modèle de données) .

La méthode `ajoutePoints()` de la classe Actif utilise la méthode `notifie()` de la Personne pour informer tous les observateurs du changement de valeur dans le nombre de points.

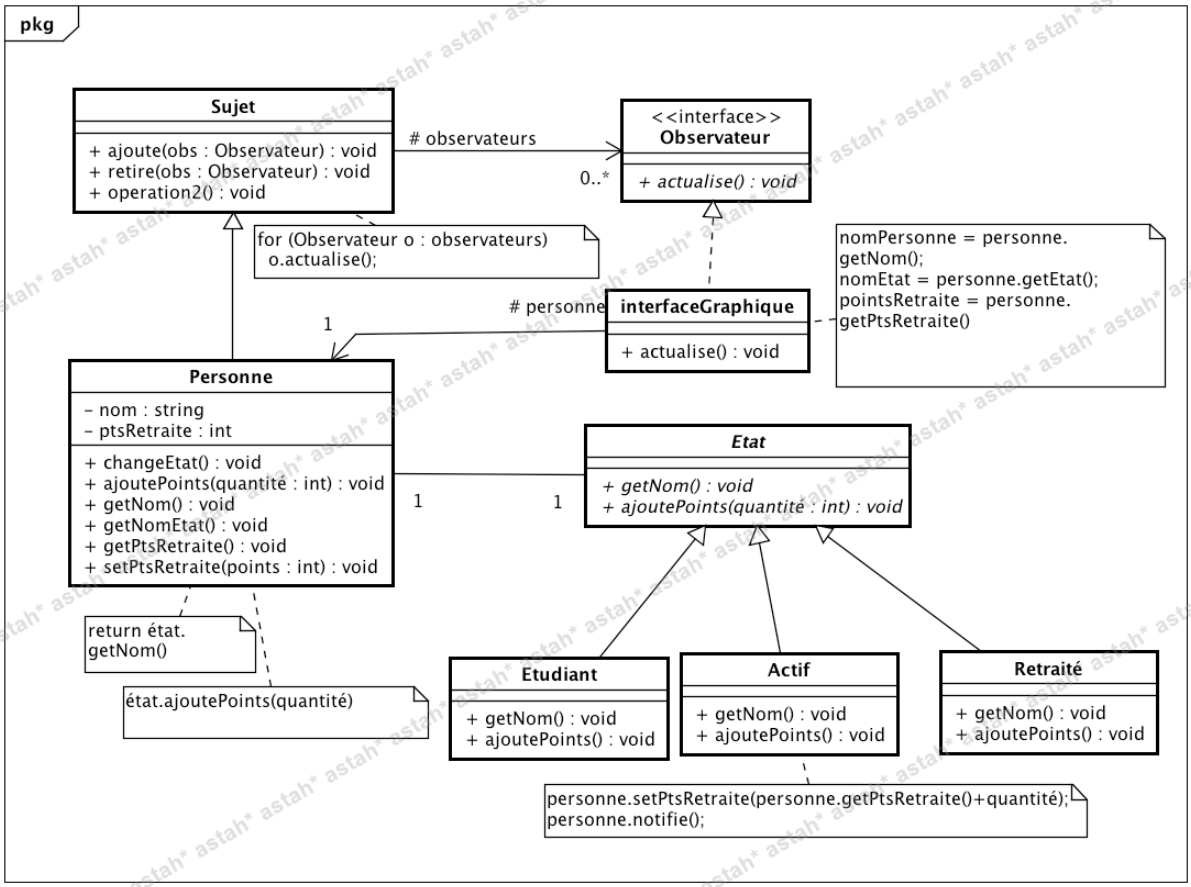


Figure 7. Le patron *observateur* appliqué à la vie professionnelle d'une personne