Cours - Patron de conception - #4

Composite

Guillaume Santini

14 janvier 2024

Plan

Motivations

Objectifs : Gestion transparente de structures arborescentes

- Compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé.
- Permet aux clients de traiter de la même façon les objets individuels (nœuds) et les combinaisons de ces derniers (sous-arbres).

Motivations

Objectifs : Gestion transparente de structures arborescentes

- Compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé.
- Permet aux clients de traiter de la même façon les objets individuels (nœuds) et les combinaisons de ces derniers (sous-arbres).

Principe SOLID

- Open/Closed,
- Inversion des dépendances.

Motivations

Objectifs : Gestion transparente de structures arborescentes

- Compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé.
- Permet aux clients de traiter de la même façon les objets individuels (nœuds) et les combinaisons de ces derniers (sous-arbres).

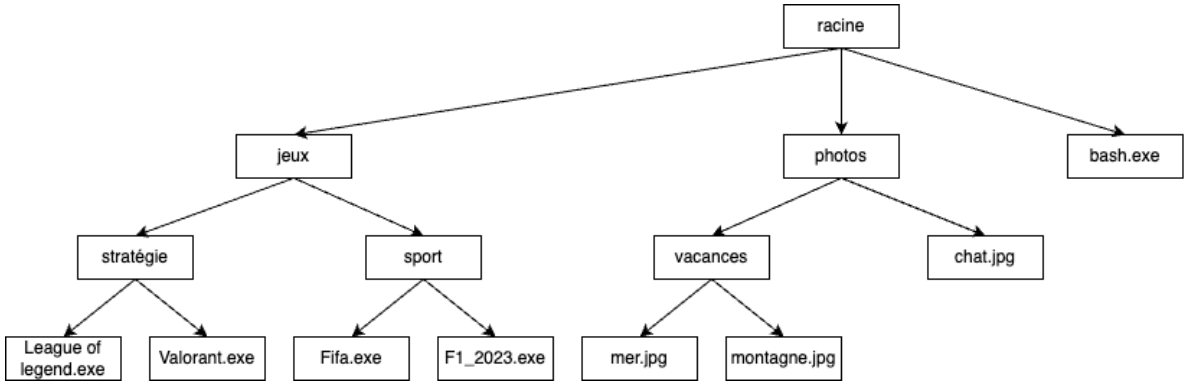
Principe SOLID

- Open/Closed,
- Inversion des dépendances.

Autres principes

- Transparence dans la manipulation des objets individuels ou des ensembles d'objets.

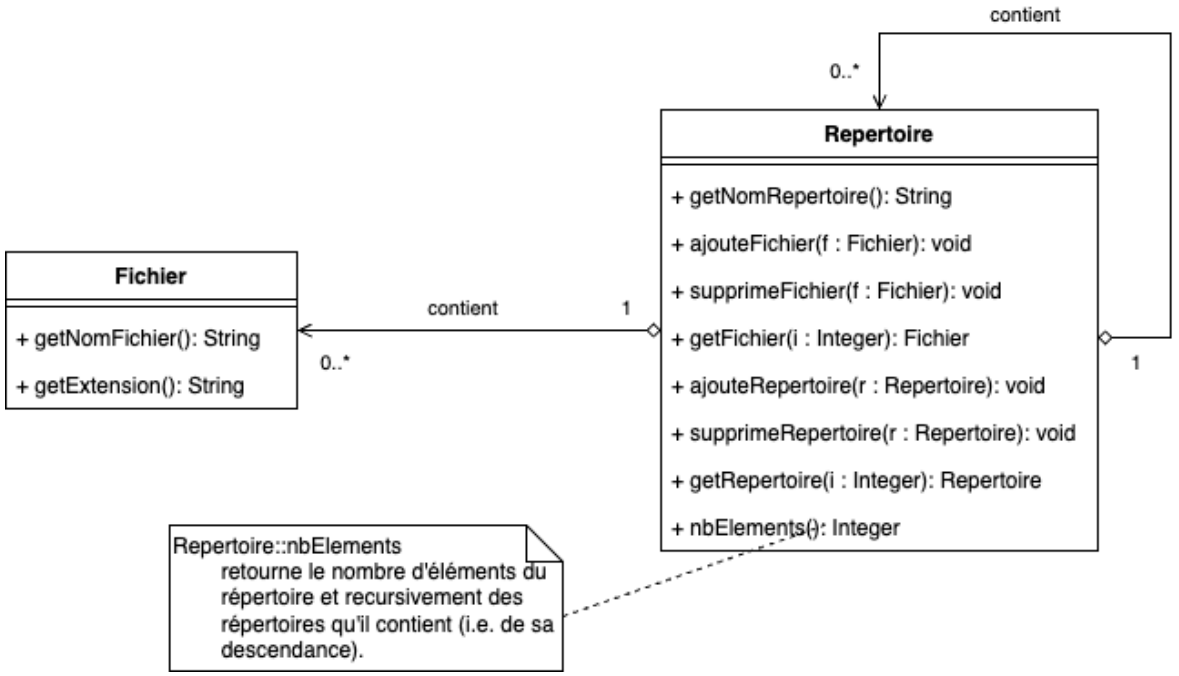
Cas d'étude



Gestion/représentation d'un disque dur

- Les fichiers et répertoires se structurent en arborescence,
  - les feuilles de l'arbre sont les fichiers,
  - les autres nœuds sont les répertoires, qui sont des éléments *composites* pouvant eux-mêmes être composés de fichiers et de sous-répertoires.
- Certaines opérations doivent pouvoir être effectuées aussi bien sur les feuilles que sur les composites (*i.e.* les opérations classiques sur les systèmes de fichiers).

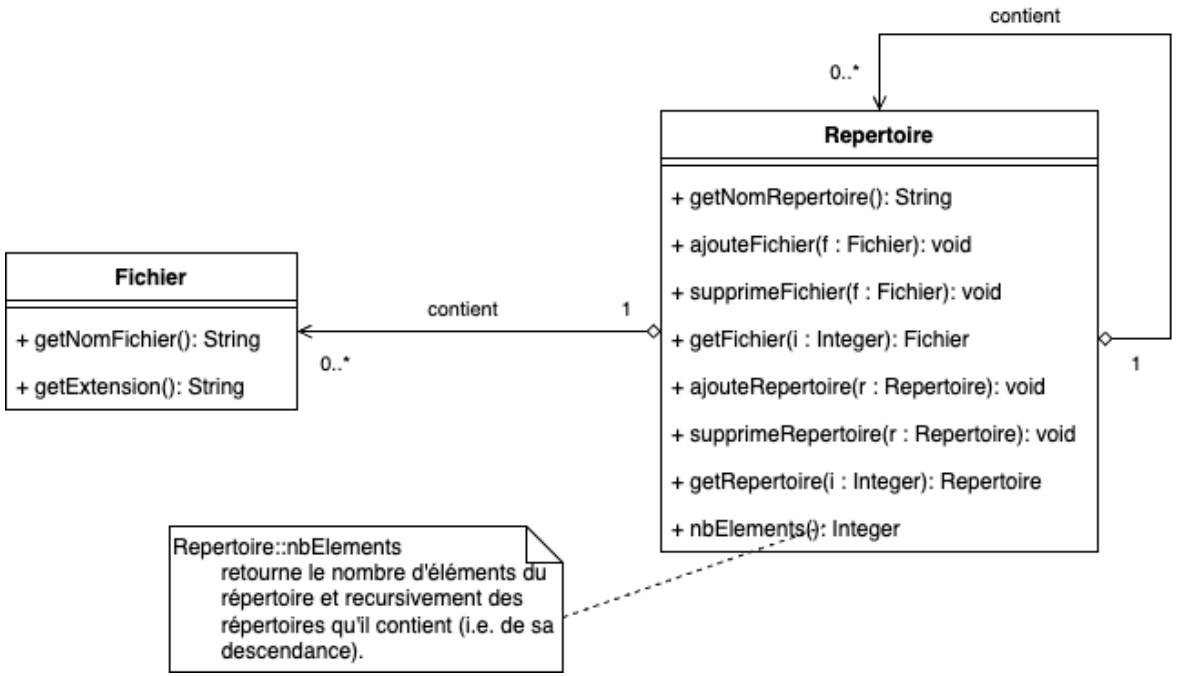
Distinguer les éléments simples (Fichiers) des éléments composés (Repertoires)



Principe SOLID Inversion des dépendances

- non transparence du contenu d'un répertoire : il faut distinguer les `Fichiers` des `Repertoires`.

Distinguer les éléments simples (`Fichiers`) des éléments composés (`Repertoires`)



Principe SOLID Open/Closed

- l'ajout d'un nouveau type de contenu (ex : `Alias`) oblige à ajouter de nouvelles opérations pour ce type (`ajouteAlias(a Alias)` :, ...)

Distinguer les Fichiers des Repertoires #1

```
public class Repertoire {
  private String nomRepertoire ;

  private ArrayList<Repertoire> repertoires;  // contient des repertoires

  private ArrayList<Fichier> fichiers ;       // contient des fichiers

  public void ajouteFichier(Fichier f) {        this.fichiers.add(f); }

  public void ajouteRepertoire(Repertoire r){    this.repertoires.add(r); }

  public int nombreElements(){
    int n = this.repertoires.size() + this.fichiers.size() ;
    // on ajoute le nombre d'elements des repertoires de sa descendance
    for (int i = 0; i < this.repertoires.size(); i++)
      n += this.repertoires.get(i).nombreElements() ;
    return n ;
  }
}
```

Principe SOLID Inversion des dépendances

- L'implémentation de la classe `Repertoire` dépend des classes concrètes `Fichier` et `Repertoire` et non d'abstractions.
- Il y a deux méthodes différentes pour la même opération d'ajout.

Distinguer les Fichiers des Repertoires #2

```
public class Repertoire {
  private String nomRepertoire ;

  public String getNomRepertoire() {
    return nomRepertoire ;
  }
}

public class Fichier {
  private String nomFichier;
   private String extension ;

  public String getNomFichier() {
    return this.nomFichier ;
  }

   public String getExtension() {
    return this.extension;
  }
}
```

Pas de factorisation pour les opérations communes

- Les Fichiers comme les Repertoires on un nom,
- La même opération getNom() est codée deux fois.

Distinguer les Fichiers des Repertoires #2

```
public static void main(String[] args) {
  Repertoire jeux      = new Repertoire("jeux");
  Repertoire action    = new Repertoire("action");

  Fichier tennis       = new Fichier("tennis", "exe");
  Fichier gta          = new Fichier("gta", "bat");
  Fichier outland      = new Fichier("outland", "exe");

  jeux.ajouteFichier(tennis);          // ajout d'un fichier
  jeux.ajouteRepertoire(action);       // ajout d'un repertoire
  action.ajouteFichier(gta);
  gta.ajouteFichier(outland);          // engendre une erreur de compilation:
                                       // on ne peut ajouter un Fichier a un Fichier
  System.out.println(jeux.nombreElements());
}
```

Sureté

- le compilateur peut détecter une opération illicite (ajout/suppression) sur un `Fichier`.

Distinguer les Fichiers des Repertoires #2

```
public static void main(String[] args) {
  Repertoire jeux      = new Repertoire("jeux");
  Repertoire action    = new Repertoire("action");

  Fichier tennis       = new Fichier("tennis", "exe");
  Fichier gta          = new Fichier("gta", "bat");
  Fichier outland      = new Fichier("outland", "exe");

  jeux.ajouteFichier(tennis);          // ajout d'un fichier
  jeux.ajouteRepertoire(action);       // ajout d'un repertoire
  action.ajouteFichier(gta);
  gta.ajouteFichier(outland);          // engendre une erreur de compilation:
                                       // on ne peut ajouter un Fichier a un Fichier
  System.out.println(jeux.nombreElements());
}
```

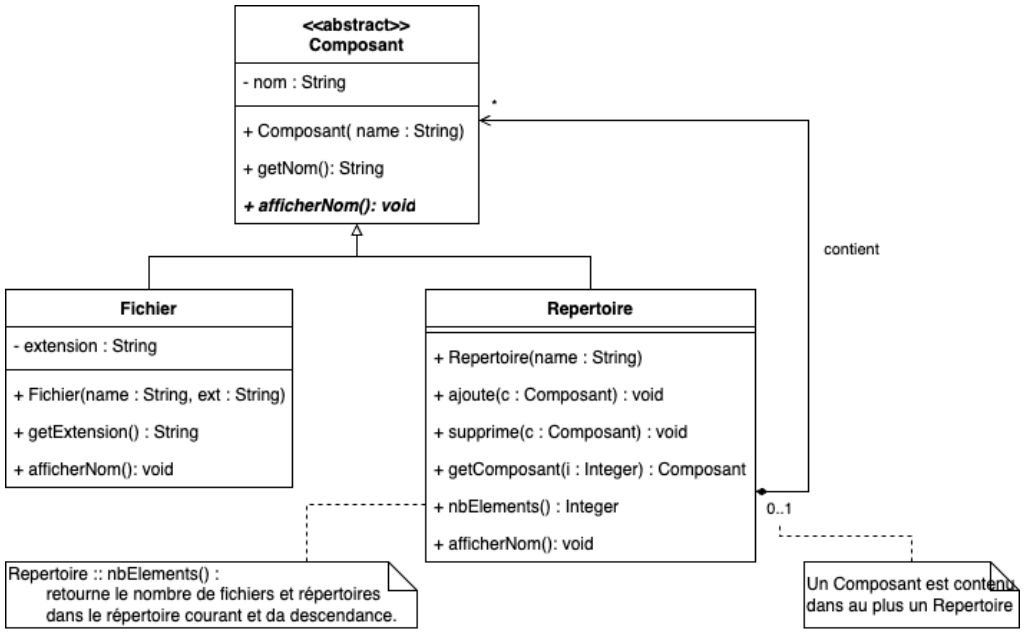
Sureté

- le compilateur peut détecter une opération illicite (ajout/suppression) sur un Fichier.

Principes SOLID Inversion des dépendances

- le client doit distinguer les éléments (Fichiers, Repertoires) contenus dans un Repertoire pour chaque opération.

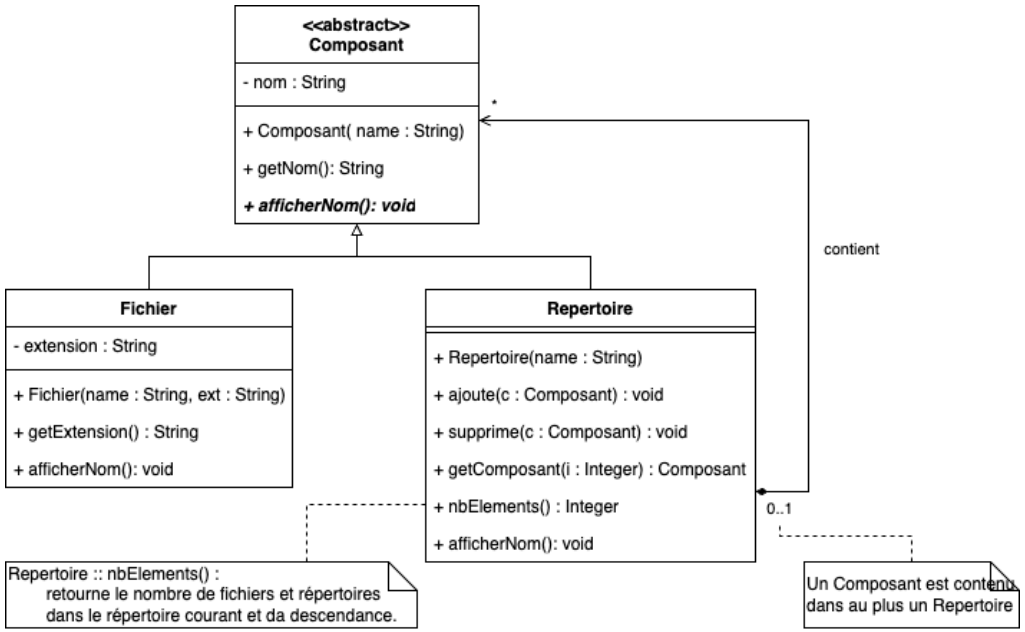
Les éléments simples (`Fichiers`) et composés (`Repertoires`) sont déclarés du même type



Principe de transparence

- le client n'a plus a distinguer les `Fichiers` des `Repertoires` lors des opérations d'*ajout*, de *suppression* ou d'obtention du nom (`getNom()`).

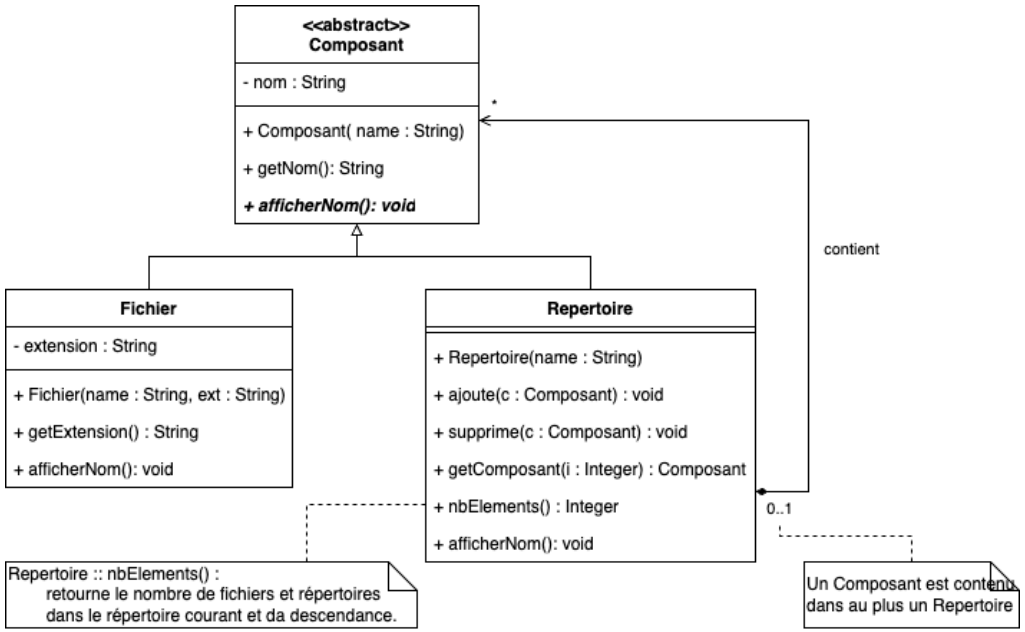
Les éléments simples (`Fichiers`) et composés (`Repertoires`) sont déclarés du même type



Principes SOLID Inversion des dépendances

- Les implémentations des méthodes `ajoute(c Composant)` et `supprime(c Composant)` ne dépendent plus de types concrets (`Fichier` et `Repertoire`) mais d'un type abstrait (`Composant`).

Les éléments simples (`Fichiers`) et composés (`Repertoires`) sont déclarés du même type



Principes SOLID Open/Closed

- L'ajout d'un nouveau type comme `Alias` est simplifiée. Il suffit de spécialiser la classe `Composant`.

Les Fichiers et Repertoires sont des Composants

```
public abstract class Composant {   //un Composant est abstrait

  private String nom;                // possede un nom

  public Composant(String unNom) {
    this.nom = unNom;
  }
  public String getNom() {
    return this.nom;
  }

  // Methode d'affichage du nom que les descendants
  // (Fichier et Repertoire) devront definir pour preciser
  // si on affiche le nom d'un fichier ou d'un repertoire

  public abstract String afficherNom() ;
}
```

Factorisation du code

- toutes les caractéristiques communes aux composants sont introduites dans la classe mère `Composant`.
- la classe est abstraite et n'est pas instanciable,
- le contrat `afficherNom()` reste sans implémentation, car l'affichage dépend de la nature de l'instance.

La classe Repertoire est plus transparente

```
public class Repertoire extends Composant {

  private ArrayList<Composant> composants ;

  public void ajoute(Composant c){ this.composants.add(c); }

  public void supprimer(Composant c){ this.composants.remove(c); }

  public String afficherNom(){ System.out.println("repertoire :" + this.getNom(); }

  public int nombreElements(){
    int n = this.composants.size() ;
    for (int i = 0 ; i < this.composants.size() ; i++) {
      Composant c = this.composants.get(i) ;
      if (c instanceof Repertoire)
        n += ((Repertoire c)).nombreElements() ;
    }
    return n ;
  }
}
```

Principe SOLID Inversion des dépendances

- Les `Repertoires` ne contiennent plus que des `Composants`.
- Les implémentations des méthodes `ajoute()` et `supprime()` ne dépendent plus de types concrets mais du type abstrait `Composant`.

La classe Repertoire n'est pas complètement transparente

```
public class Repertoire extends Composant {

  private ArrayList<Composant> composants ;

  public void ajoute(Composant c)}}}@>{ this.composants.add(c);

  public void supprimer(Composant c){ this.composants.remove(c); }

  public String afficherNom(){ System.out.println("repertoire␣:" + this.getNom(); }

  public int nombreElements(){
    int n = this.composants.size() ;
    for (int i = 0 ; i < this.composants.size() ; i++) {
      Composant c = this.composants.get(i) ;
      if (c instanceof Repertoire)
        n += ((Repertoire c)).nombreElements() ;
    }
    return n ;
  }
}
```

La transparence n'est pas totale.

- Il faut vérifier le type de l'objet contenu :
  - si c'est un `Repertoire` on peut appliquer la récursivité,
  - si c'est un `Fichier` il ne faut pas appliquer une opération illicite sur celui-ci comme `nbElements()`.

Les classes filles spécialisent la structure et les comportements si nécessaire

```
public class Fichier extends Composant {

  private String extension ;

   public Fichier(String unNom, String uneExtension) {
     super(unNom) ;
     this.extension = uneExtension ;
   }
   public String getExtension() {
     return this.extension;
   }
   public String afficherNom() {
     System.out.println("fichier : " + this.getNom()) ;
   }
}
```

Les comportements spécifiques sont introduits dans les classes filles.

- ajout d'élément structurel (`extension`) :
- ajout de composant fonctionnels (`getExtension()`) :
- définition des méthodes abstraites (`afficherNom()`) :

Le client ne distingue plus les opérations sur les `Fichiers` des opérations sur les `Repertoires`

```
public static void main(String[] args) {

  Repertoire jeux = new Repertoire("jeux") ;
  Repertoire action = new Repertoire("action") ;

  Fichier tennis = new Fichier("tennis", "exe") ;
  Fichier gta = new Fichier("gta", "bat") ;
  Fichier outland = new Fichier("outland", "exe") ;

  jeux.ajoute(tennis);
  jeux.ajoute(action);

  action.ajoute(gta) ;
  action.ajoute(outland) ;

  System.out.println(jeux.nombreElements()) ;

  gta.ajoute( new Fichier("Fifa"));
}
```

Principe SOLID Inversion des dépendances

- Plus de transparence dans l'utilisation de la classe,
- Seuls des `Composants` sont passés en paramètres.

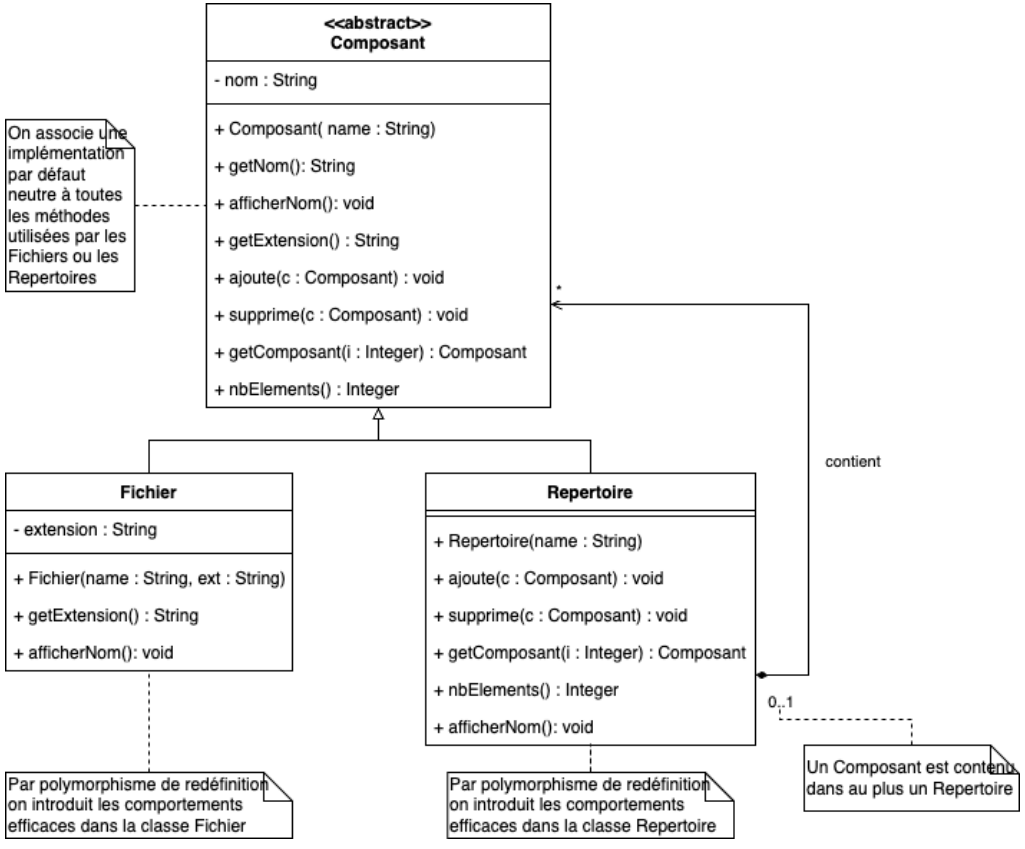
Le client ne distingue plus les opérations sur les Fichiers des opérations sur les Repertoires

```
public static void main(String[] args) {

  Repertoire jeux = new Repertoire("jeux") ;
  Repertoire action = new Repertoire("action") ;

  Fichier tennis = new Fichier("tennis", "exe") ;
  Fichier gta = new Fichier("gta", "bat") ;
  Fichier outland = new Fichier("outland", "exe") ;

  jeux.ajoute(tennis);
  jeux.ajoute(action);

  action.ajoute(gta) ;
  action.ajoute(outland) ;

  System.out.println(jeux.nombreElements()) ;

  gta.ajoute( new Fichier("Fifa"));
}
```

Conception sûre

- Les opérations illicites seront identifiées à la compilation.

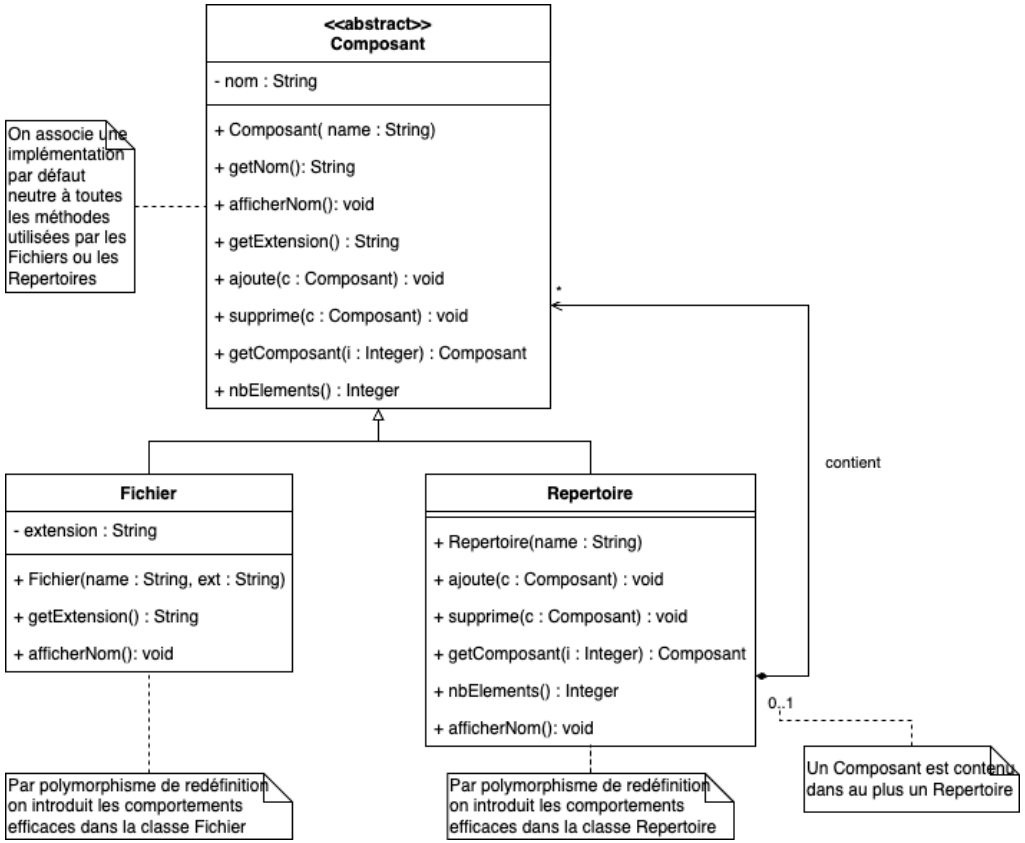
Proposer les mêmes opérations pour tous les Composants



Transparence complète

- Toutes les méthodes pourront être invoquées sur n'importe quel `Composant`, `Fichier` comme `Repertoire`.

Proposer les mêmes opérations pour tous les Composants



Une conception moins sûre

- Certaines opérations illicites ne seront identifiées qu'à l'exécution.

La classe Composant introduit un comportement par défaut pour toutes les opérations

```
public abstract class Composant {

   private String nom ;

   public Composant(String unNom){
      this.nom = unNom ;
   }
   public String getNom()
      return this.nom ;
   }
   public String afficherNom()
      return "" ;
   }
   public String getExtension()
      return "" ;
   }
   public void ajoute(Composant c){}

   public Composant getComposant(int i){
      return null ;
   }
   public void supprime(Composant c){}

   public int nombreElements(){
      return 0 ;
   }

}
```

Une implémentation pour les méthodes :

- communes des `Fichiers` et des `Repertoires`.
- à redéfinir dans `Fichier` avec un comportement par défaut compatible avec les `Repertoires`.
- à redéfinir dans `Repertoire` avec un comportement par défaut compatible avec les `Fichiers`.

Fichier introduit ses comportements effectifs

```
public class Fichier extends Composant {

  private String extension ;

  public Fichier(String unNom, String uneExtension) {
    super(unNom) ;
    this.extension = uneExtension ;
  }
  public String afficherNom() {
    System.out.println("fichier␣:" + this.getNom());
  }
  public String getExtension() {
    return this.extension ;
  }
}
```

Définition des comportements effectifs (polymorphisme de redéfinition)

Redéfinition des comportements par défaut introduits dans `Composant`.

Repertoire introduit ses comportements effectifs

```
public class Repertoire extends Composant {

  private ArrayList<Composant> composants ;

  public void ajoute(Composant c) {
    this.composants.add(c);
  }
  public void supprimer(Composant c) {
    this.composants.remove(c);
  }
  public String afficherNom() {
    System.out.println("repertoire␣:" + this.getNom();
  }
  public int nombreElements() {
    int n = this.composants.size() ;
    for (int i = 0 ; i < this.composants.size() ; i++) {
      Composant c = this.composants.get(i) ;
      n += c.nombreElements();
    }
    return n ;
  }
}
```

Définition des comportements effectifs (polymorphisme de redéfinition)

Redéfinition des comportements par défaut introduits dans `Composant`.

La classe Repertoire ne distingue plus les Fichiers des Repertoires

```
public class Repertoire extends Composant {

  private ArrayList<Composant> composants ;

  public void ajoute(Composant c) {
    this.composants.add(c);
  }
  public void supprimer(Composant c) {
    this.composants.remove(c);
  }
  public String afficherNom() {
    System.out.println("repertoire :" + this.getNom();
  }
  public int nombreElements() {
    int n = this.composants.size() ;
    for (int i = 0 ; i < this.composants.size() ; i++) {
      Composant c = this.composants.get(i) ;
      n += c.nombreElements();
    }
    return n ;
  }
}
```

Transparence totale

`nombreElements()` est définie pour `Fichier` et `Repertoire` : plus besoin de vérifier le type de `c`.

Pour le client, les Fichiers et Repertoires sont vus comme des Composants

```
public static void main(String[] args) {

  Composant jeux    = new Repertoire("jeux") ;
  Composant action  = new Repertoire("action") ;
  Composant tennis  = new Fichier("tennis", "exe") ;
  Composant gta     = new Fichier("gta", "bat") ;
  Composant outland = new Fichier("outland", "exe") ;
  System.out.println(jeux.nombreElements()) ;

  jeux.ajoute(tennis) ;
  jeux.ajoute(action);
  action.ajoute(gta);
  action.ajoute(outland);

  Composant c = gta.getComposant(0);

  int n = c.nombreElements();
}
```

Transparence totale

- Certaines opérations sont transparente du fait de l'introduction de comportement par défaut,
- `gta.getComposant(0)` retourne `null` car `gta` est un `Fichier`.

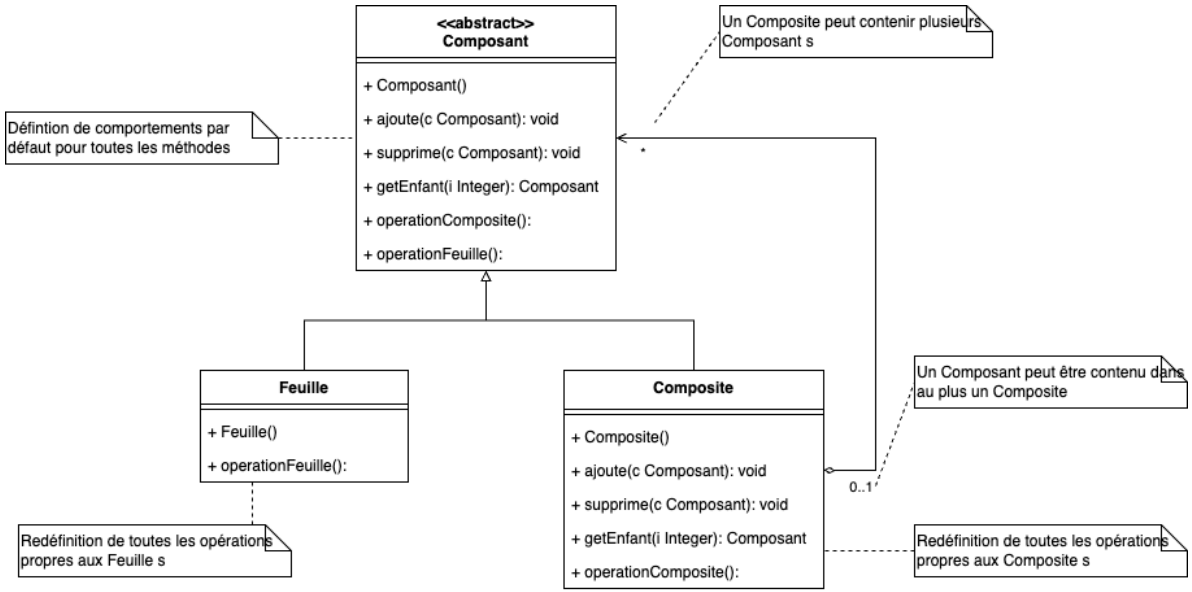
Pour le client, les Fichiers et Repertoires sont vus comme des Composants

```
public static void main(String[] args) {

  Composant jeux    = new Repertoire("jeux") ;
  Composant action  = new Repertoire("action") ;
  Composant tennis  = new Fichier("tennis", "exe") ;
  Composant gta     = new Fichier("gta", "bat") ;
  Composant outland = new Fichier("outland", "exe") ;
  System.out.println(jeux.nombreElements()) ;

  jeux.ajoute(tennis) ;
  jeux.ajoute(action);
  action.ajoute(gta);
  action.ajoute(outland);

  Composant c = gta.getComposant(0);

  int n = c.nombreElements();
}
```

Une conception moins sûre

- Des opérations illicites ne provoquent aucune erreur à la compilation.
- `c.nombreElements()` déclenche à l'exécution une `NullPointerException` car `c == null`.

Le patron de conception Composite



Définition

- Il compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé.
- Il permet aux clients de traiter de la même façon les objets individuels et les combinaisons de ces derniers.

Principes de conception

Principes généraux mis en œuvre

- **Transparence :** Permet de traiter de la même façon plusieurs types d'objets (`Feuilles` et `Composites`).
- **Sûreté :** Un programme est sûr lorsqu'il est capable de détecter et de résister aux erreurs. Il est plus sûr de détecter les erreurs à la compilation qu'à l'exécution.
- **Transparence/Sûreté :** Une augmentation de la transparence entraîne parfois une diminution de la sûreté : on ne détecte plus les erreurs propres à certains types d'objets que la transparence ne permet plus de distinguer.

Principes de conception

Principes SOLID respectés

- **Dependency Inversion :** La classe abstraite Composant permet aux classes clientes de ne pas avoir à distinguer les `Feuilles` des `Composites`,
- **Open/Closed :** Plusieurs type de `Feuilles` peuvent être définis sans avoir à modifier les programmes pré-existants.

Credits

- Support soumis à copyleft : 
- Le cours de D. Bouthinon,
- **Design patterns — Tête la première**, E. & E. Freeman, ed. O'Reilly.