Cours M3105 : Conception et programmation objet avancées
Le patron de conception Composite

Références : cours de D. Bouthinon, livre *Design patterns — Tête la première*,
E. & E. Freeman, ed. O'Reilly

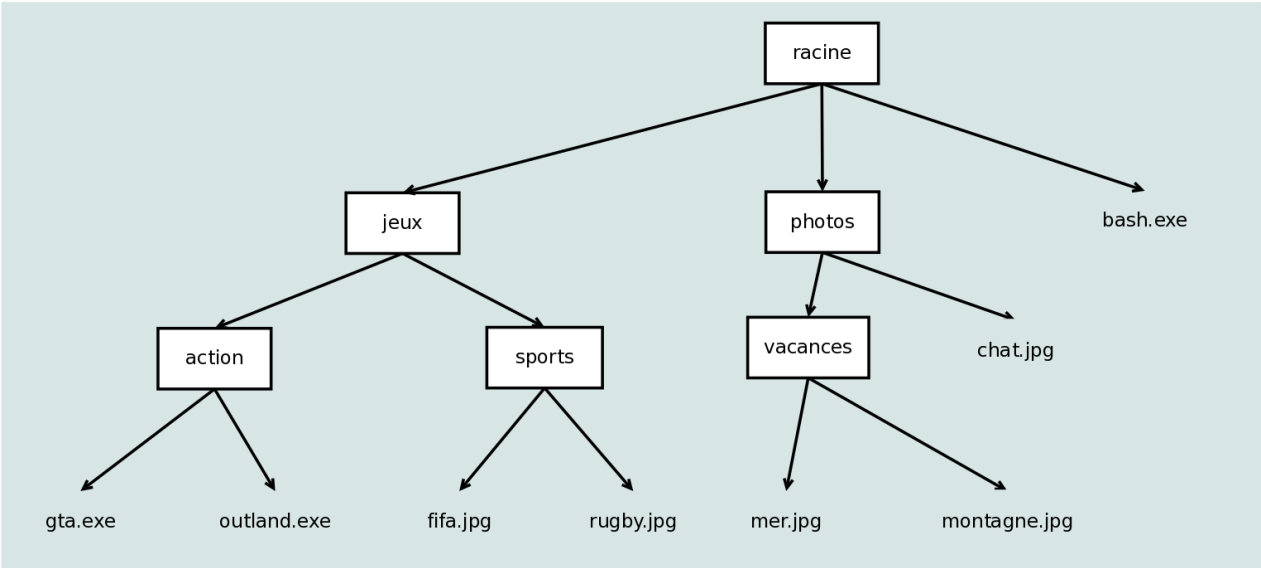
IUT Villetaneuse

2019-2020

Plan

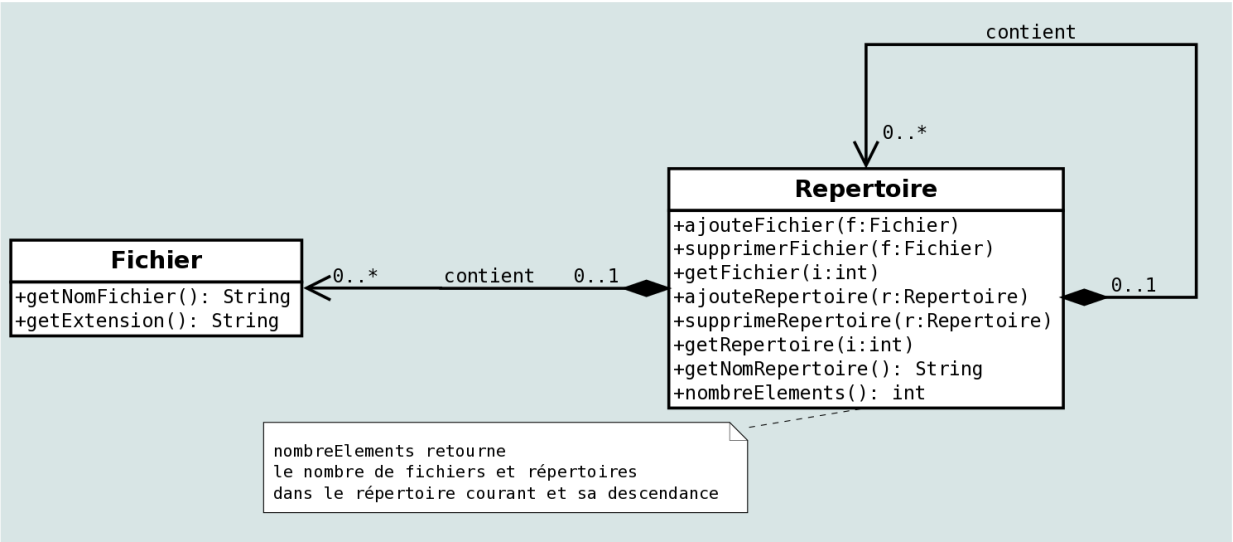
Gérer une arborescence contenant des éléments simples et composés

Exemple : gérer un disque dur

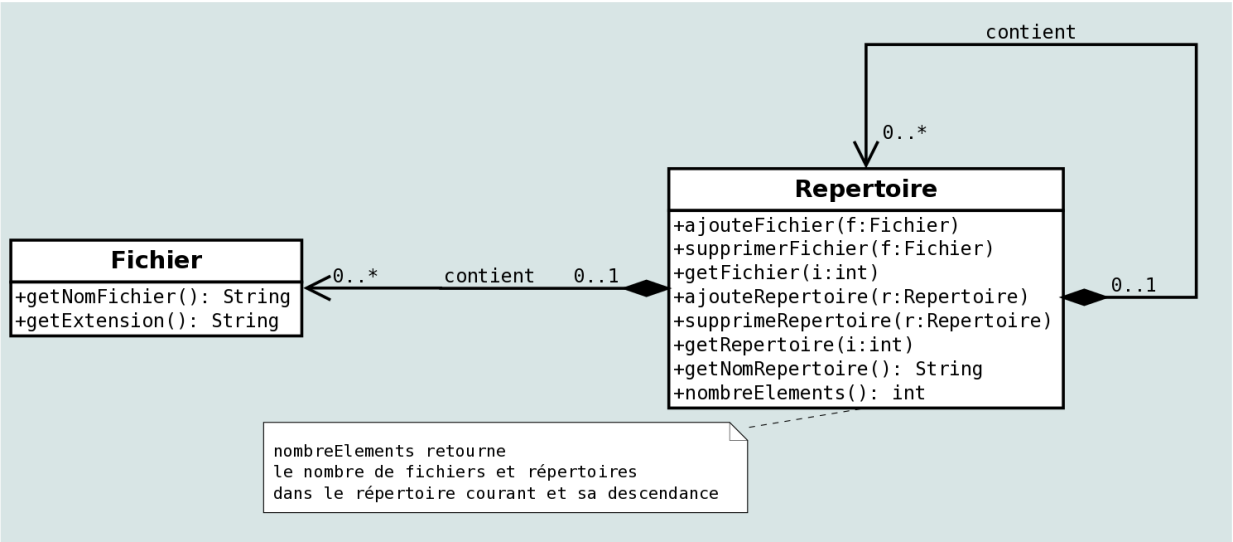


Un disque contient des répertoires (éléments composés ou "composites") et des fichiers (éléments simples ou "feuilles")

Distinguer les éléments simples (fichiers) des éléments composés (répertoires)

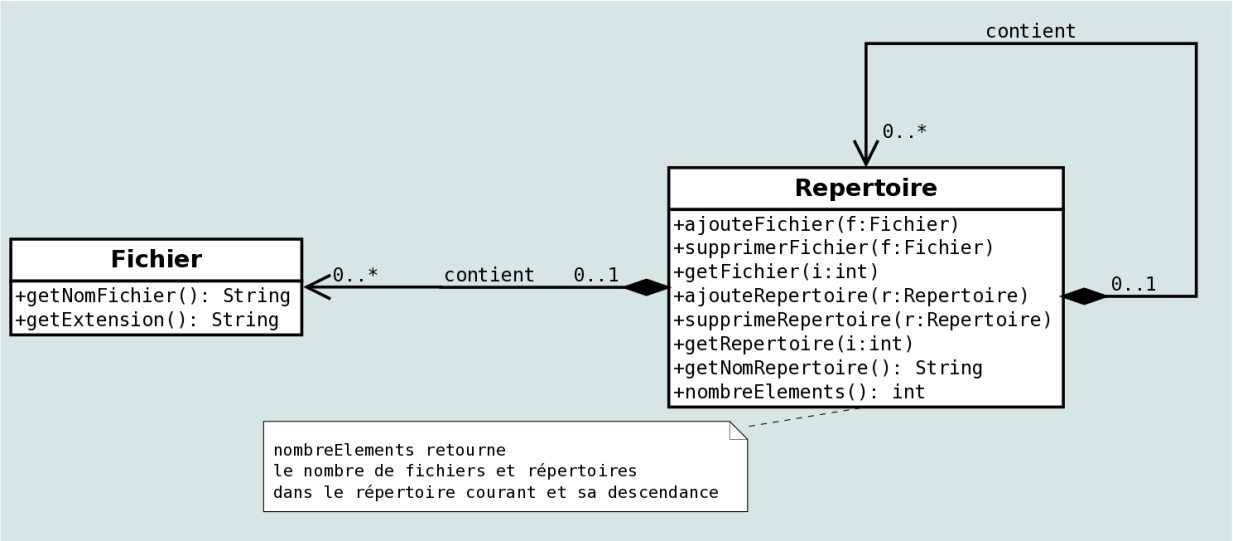


Distinguer les éléments simples (fichiers) des éléments composés (répertoires)



Problèmes : 1) non transparence du contenu d'un répertoire : il faut distinguer les fichiers des répertoires.

Distinguer les éléments simples (fichiers) des éléments composés (répertoires)



Problèmes : 1) non transparence du contenu d'un répertoire : il faut distinguer les fichiers des répertoires. 2) 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 répertoires (1)

```
 public class Repertoire
{
    private String nomRepertoire;
    private ArrayList<Repertoire> repertoires;//Un répertoire peut contenir des
répertoires
    private ArrayList<Fichier> fichiers;//Un répertoire peut contenir des
fichiers
        ...
    public void ajouteFichier(Fichier f) //méthode sur les fichiers
        { this.fichiers.add(f); }
        ...
    public void ajouteRepertoire(Repertoire r)//méthode sur les répertoires
        { this.repertoires.add(r); }
        ...
    public int nombreElements()//nombre d'éléments dans le répertoire courant
        { int n = this.repertoires.size() + this.fichiers.size();
        for (int i = 0; i < this.repertoires.size(); i++)
        n += this.repertoires.get(i).nombreElements();//on ajoute le nombre
d'éléments dans les répertoires de sa descendance
        return n;} }
```

Distinguer les fichiers des répertoires (2)

```
public class Fichier{
   private String nomFichier, extension;
   ...
   public String getNomFichier()
       { return this.nomFichier;}
   public String getExtension()
       { return this.extension; }
```

Tout client doit distinguer les opérations sur les fichiers des opérations sur les répertoires

```
 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 répertoire
action.ajouteFichier(gta) ;
gta.ajouteFichier(outland) ;//engendre une erreur de compilation: on ne peut
ajouter un fichier à un fichier
System.out.println(jeux.nombreElements()) ; }
```

Tout client doit distinguer les opérations sur les fichiers des opérations sur les répertoires

```
 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 répertoire
action.ajouteFichier(gta) ;
gta.ajouteFichier(outland) ;//engendre une erreur de compilation: on ne peut
ajouter un fichier à un fichier
System.out.println(jeux.nombreElements()) ; }
```

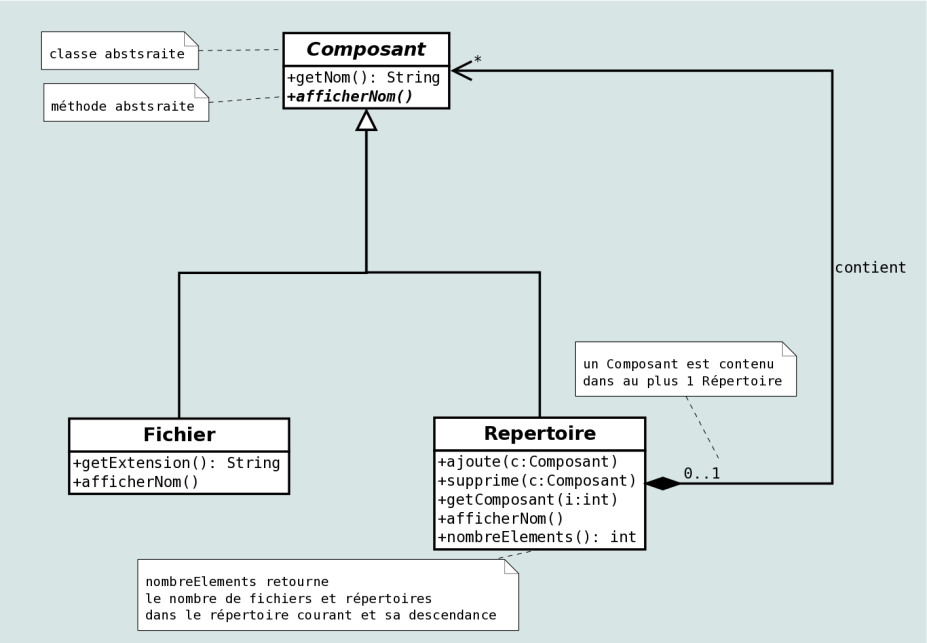
Sûreté : le compilateur peut détecter une opération illicite (ajout/suppression) sur un fichier.

Tout client doit distinguer les opérations sur les fichiers des opérations sur les répertoires

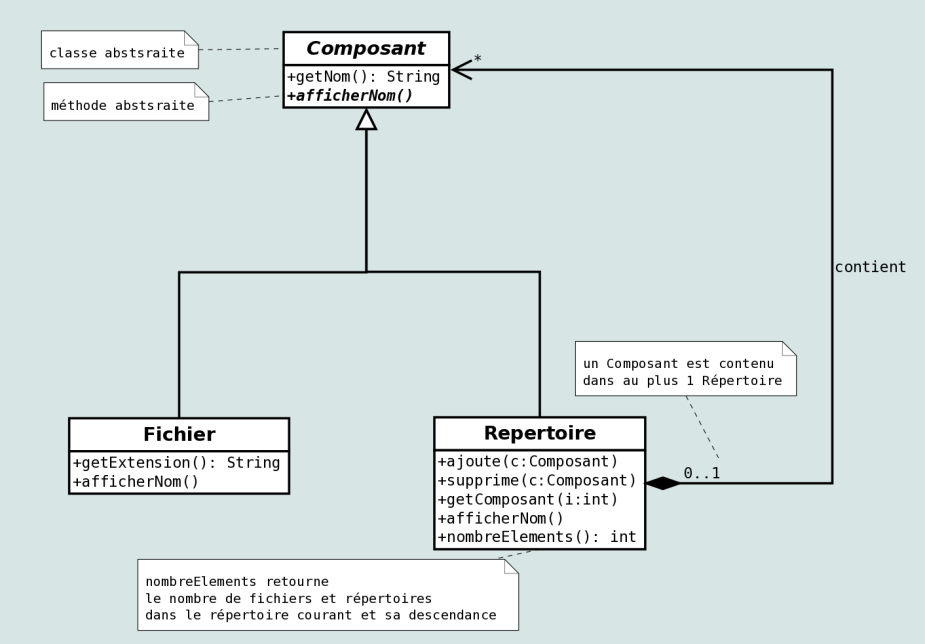
```
 public static void main(String[] args)
{     Repertoire jeux = new Repertoire("jeux") ;
Repertoire action = new Repertoire("action") ;
Fichier tennis = new Fichier("tennis", "exe") ;
Fichier gta = new Fichier("gta", "bat") ;
Fichier outland = new Fichier("outland", "exe") ;
jeux.ajouteFichier(tennis) ;//ajout d'un fichier
jeux.ajouteRepertoire(action) ;//ajout d'un répertoire
action.ajouteFichier(gta) ;
gta.ajouteFichier(outland) ;//engendre une erreur de compilation: on ne peut
ajouter un fichier à un fichier
System.out.println(jeux.nombreElements()) ; }
```

Sûreté : le compilateur peut détecter une opération illicite (ajout/suppression) sur un fichier. Manque de transparence : le client doit distinguer les éléments (fichiers, répertoires) contenus dans un répertoire.

Les éléments simples (fichiers) et composés (répertoires) sont déclarés du même type



Les éléments simples (fichiers) et composés (répertoires) sont déclarés du même type



Quasi-transparence : le client ne distinguera plus les fichiers des répertoires. L'ajout d'un nouveau type de contenu n'oblige pas à ajouter de nouvelles opérations pour ce type.

Les fichiers et répertoires sont des composants

```
 public abstract class Composant //un Composant est abstrait
    {private String nom //possède un nom ;
    public Composant(String unNom)
    {this.nom = unNom ;}
    public String getNom()
    {return this.nom ;}
    public abstract String afficherNom() ;
//et une méthode d'affichage du nom que les descendants (Fichier et Repertoire)
devront définir pour préciser si on affiche le nom d'un fichier ou d'un
répertoire
    }
```

La classe Repertoire distingue (en partie) les fichiers des répertoires

```
 public class Repertoire extends Composant //un Repertoire est un Composant
    {private ArrayList<Composant> composants //contient des Composant (s);
    ...
    public void ajoute(Composant c) //méthodes sur les composants
        {this.composants.add(c);}
    public void supprimer(Composant c)
            {this.composants.remove(c);}
    ...
    public String afficherNom() //définit la méthode abstraite héritée de
Composant
        {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) //il faut vérifier que c'est un Repertoire
pour ne pas appliquer une opération illicite sur un Fichier. La transparence
n'est pas totale
        n += ((Repertoire c)).nombreElements() ;}
        return n ;}
    }
```

La classe Fichier

```
 public class Fichier extends Composant //Un Fichier est un Composant
{
private String extension //Un Fichier possède une extension;
public Fichier(String unNom, String uneExtension)
    {super(unNom) ;
    this.extension = uneExtension ;}
public String getExtension()
    {return this.extension ;}
public String afficherNom() //définit la méthode abstraite héritée de
Composant{
    System.out.println("fichier : " + this.getNom()) ;}
}
```

Le client ne distingue plus les opérations sur les fichiers des opérations sur les répertoires

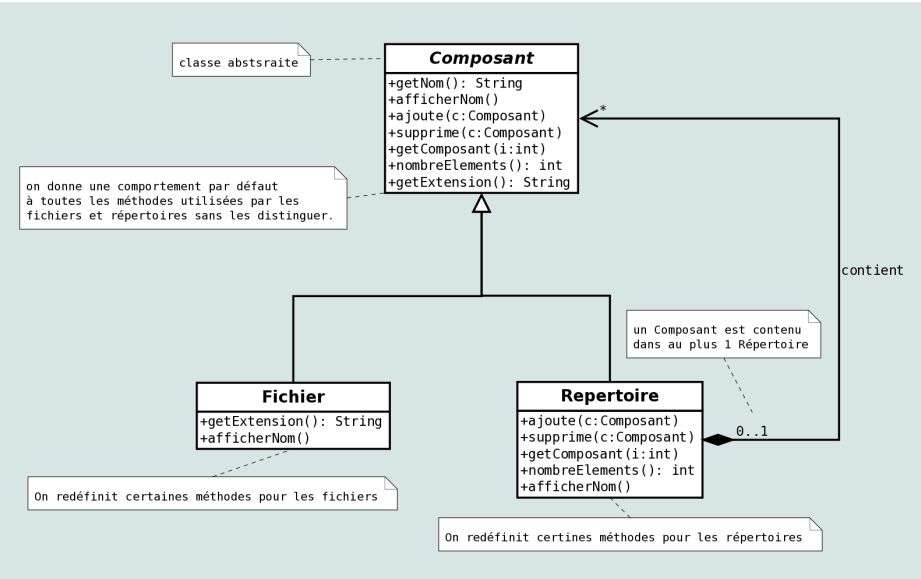
```
public static void main(String[] args)
   { Repertoire jeux = new Repertoire("jeux");
   Repertoire action = new Repertoire("action");
   Fichier tennis = new Fichier("tennis", "exe");
   Fichier gta = new Fichier("gta", "bat");
   Fichier outland = new Fichier("outland", "exe");
   jeux.ajoute(tennis) //ajout d'un fichier;
   jeux.ajoute(action)//ajout d'un répertoire;
   action.ajoute(gta);
   action.ajoute(outland);
   System.out.println(jeux.nombreElements());
   }
```

Le client ne distingue plus les opérations sur les fichiers des opérations sur les répertoires

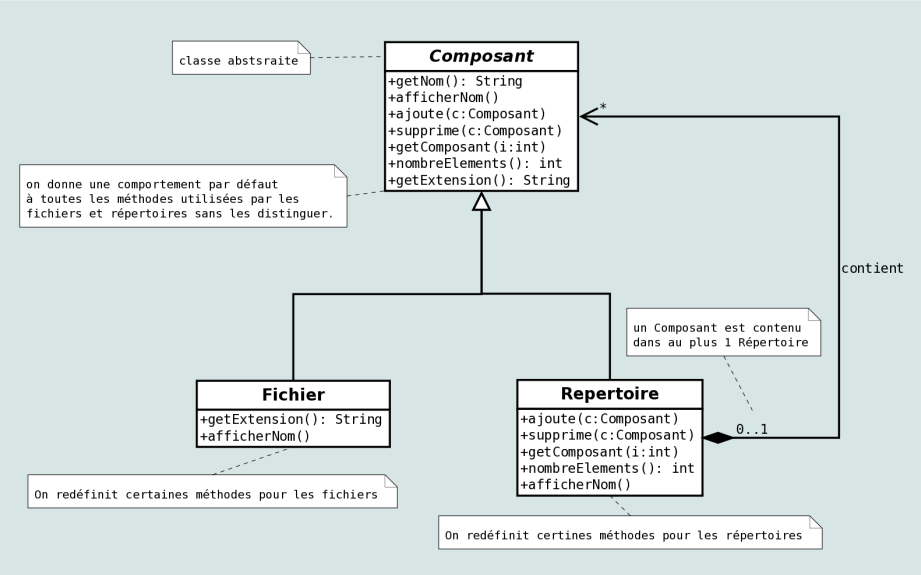
```
public static void main(String[] args)
   { Repertoire jeux = new Repertoire("jeux");
   Repertoire action = new Repertoire("action");
   Fichier tennis = new Fichier("tennis", "exe");
   Fichier gta = new Fichier("gta", "bat");
   Fichier outland = new Fichier("outland", "exe");
   jeux.ajoute(tennis) //ajout d'un fichier;
   jeux.ajoute(action)//ajout d'un répertoire;
   action.ajoute(gta);
   action.ajoute(outland);
   System.out.println(jeux.nombreElements());
   }
```

On a gagné en transparence

On propose les mêmes opérations pour tous les Composants (fichiers et répertoires)

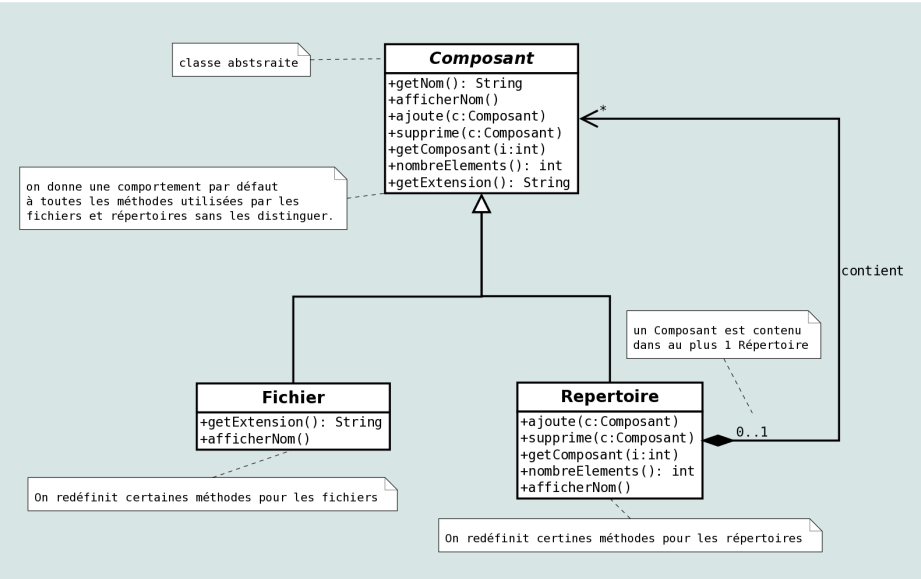


On propose les mêmes opérations pour tous les Composants (fichiers et répertoires)



Transparence totale : aucune classe ne distinguera les opérations sur les fichiers de celles sur les répertoires.

On propose les mêmes opérations pour tous les Composants (fichiers et répertoires)



Transparence totale : aucune classe ne distinguera les opérations sur les fichiers de celles sur les répertoires.Conception moins sûre : certaines erreurs n'apparaîtront qu'à l'exécution.

Les opérations sont définies par défaut dans la classe Composant

```
 public abstract class Composant
    {private String nom ;
    public Composant(String unNom)
        {this.nom = unNom ;}
//chaque méthode a un comportement par défaut qui pourra être redéfini dans
Fichier et Repertoire
    public String getNom()
        {return this.nom ;}
    public String afficherNom(){return "" ;}
//méthode pour les fichiers (comportement par défaut compatible avec les
répertoires)
    public String getExtension() {return "" ;}
//méthodes pour les répertoires (comportement par défaut compatible avec les
fichiers)
    public void ajoute(Composant c){}
    public Composant getComposant(int i){return null ;}
    public void supprime(Composant c){}
    public int nombreElements(){return 0 ;}
    }
```

La classe Repertoire ne distingue plus les fichiers des répertoires

```
 public class Repertoire extends Composant
    {private ArrayList<Composant> composants;
    ...
//on redéfinit ces méthodes pour leur donner un comportement propre à un
Repertoire
    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() //La méthode nombreElements() étant définie
pour les Fichiers et les Repertoires, il n'y a plus besoin de vérifier le type de
c.;}
        return n;}
    }
```

La classe Fichier

```
 public class Fichier extends Composant
    {private String extension ;
//On redéfinit ces méthodes pour leur donner un comportement propre à un Fichier
    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 ;}
    }
```

Pour le client, les fichiers et répertoires 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");
    jeux.ajoute(tennis);
    jeux.ajoute(action);
    action.ajoute(gta);
    action.ajoute(outland);
    System.out.println(jeux.nombreElements());
    Composant c = tennis.getComposant(0) //retourne null car tennis est un
fichier;
    int n = c.nombreElements() //A l'exécution déclenche une
NullPointerException car c == null. Aucune erreur à la compilation.;}
```

Pour le client, les fichiers et répertoires 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") ;
    jeux.ajoute(tennis) ;
    jeux.ajoute(action);
    action.ajoute(gta);
    action.ajoute(outland);
    System.out.println(jeux.nombreElements()) ;
    Composant c = tennis.getComposant(0) //retourne null car tennis est un
fichier;
    int n = c.nombreElements() //A l'exécution déclenche une
NullPointerException car c == null. Aucune erreur à la compilation.;}
```

Transparence totale, code moins sûr.

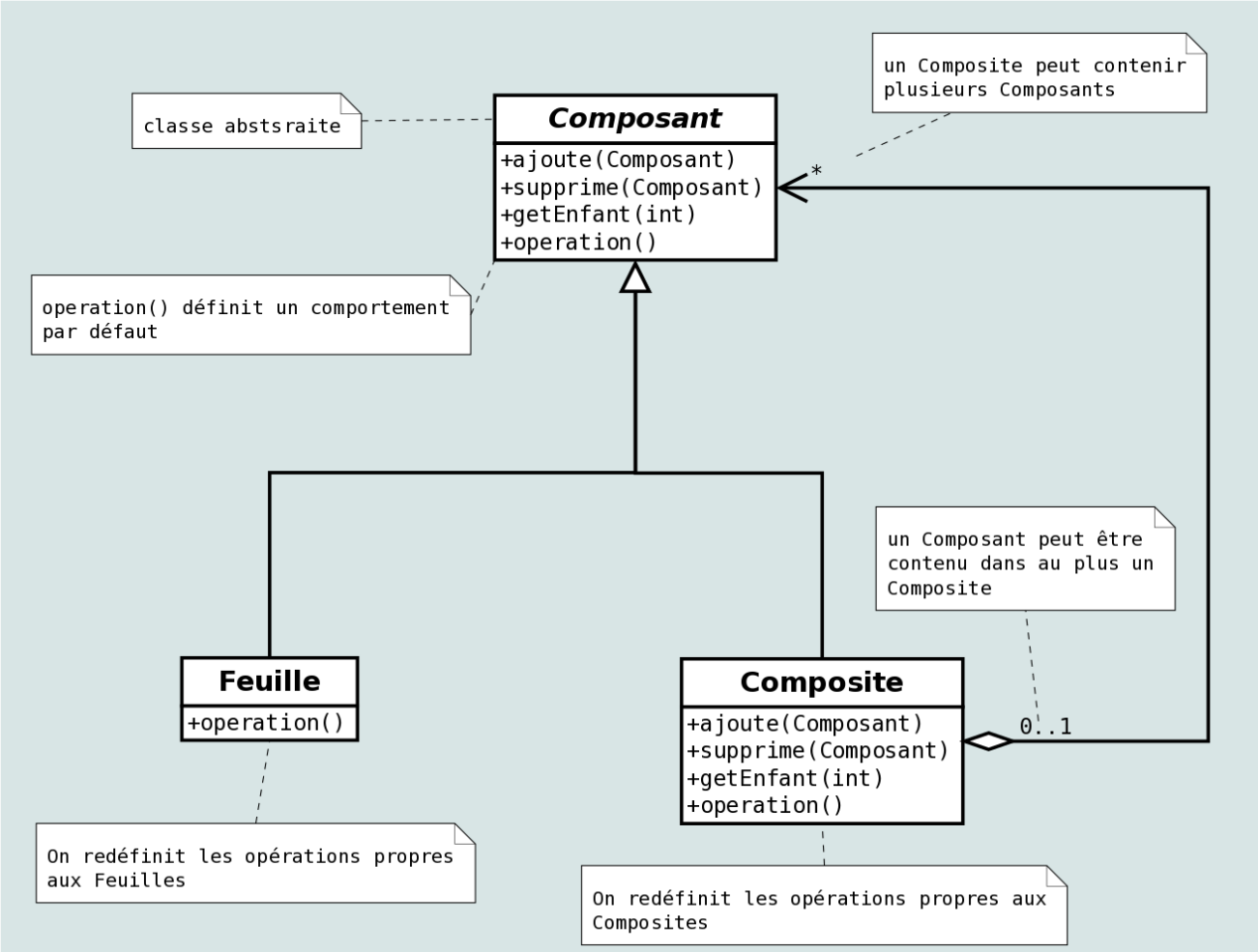
Définition

Le design pattern Composite

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.

Extrait de "Design Patterns : Elements of Reusable Object-Oriented Software" (E. Gamma, R. Helm, R. Johnson et J. Vlissides)

Structure du design pattern Composite



Principes généraux rencontrés

- 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.