Patrons Décorateur et Template
Les énumérations en Java

Loïc Mazo

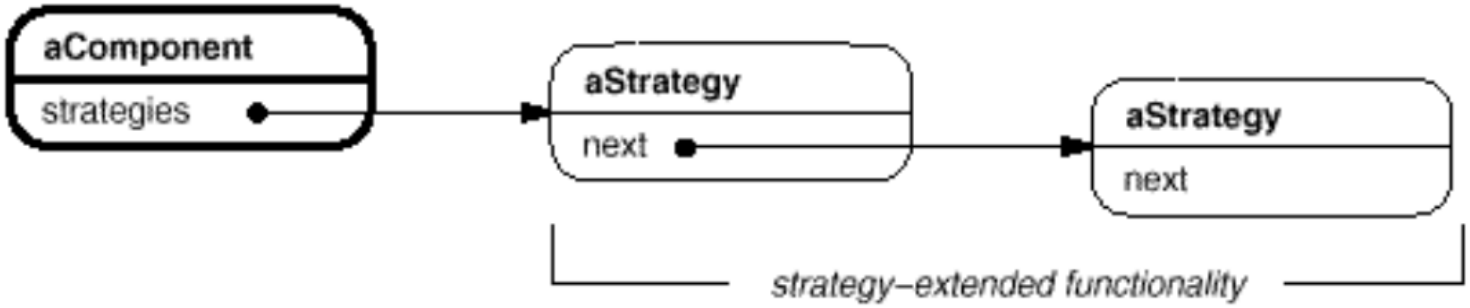
Automne 2020

Le patron Décorateur

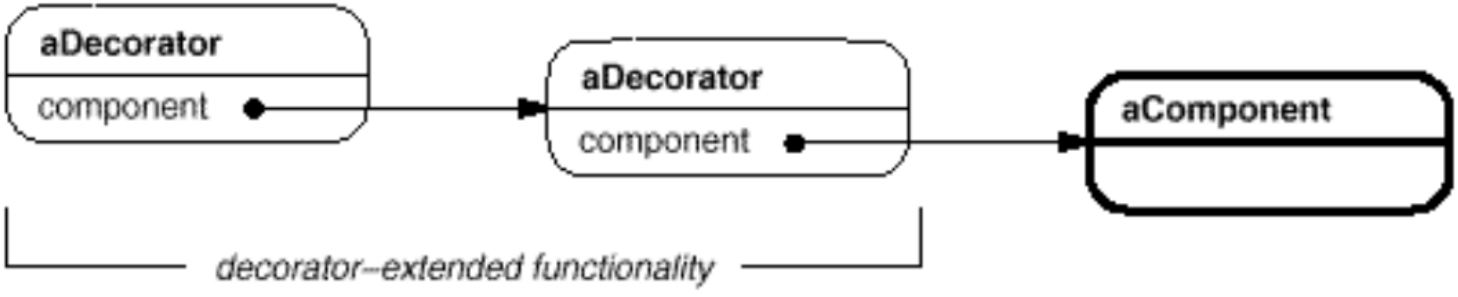
Patrons Décorateur

Changer dynamiquement le comportement d'un objet

- Stratégie : en changeant ses « tripes »



- Decorateur : en modifiant sa peau



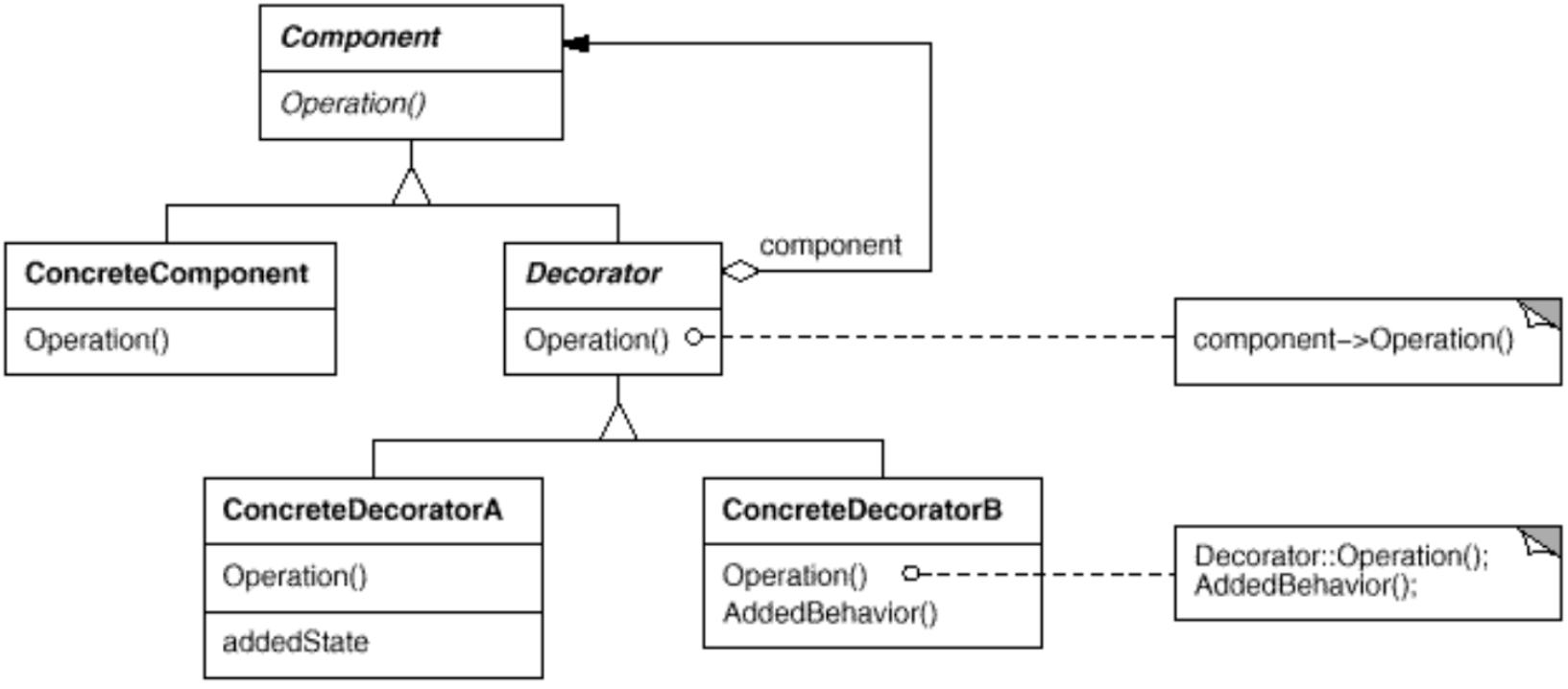
Patron **Décorateur**

Problème Étendre dynamiquement le comportement d'un objet.

Par exemple, dans système de widgets, ajouter à un panneau une barre de défilement, une bordure, une barre de titre, etc.

Solution Envelopper l'objet dans un autre objet, « décorateur »,
- qui transmet les requêtes des clients au décoré,
- en les modifiant si besoin.

Décorateur = variante du Composite

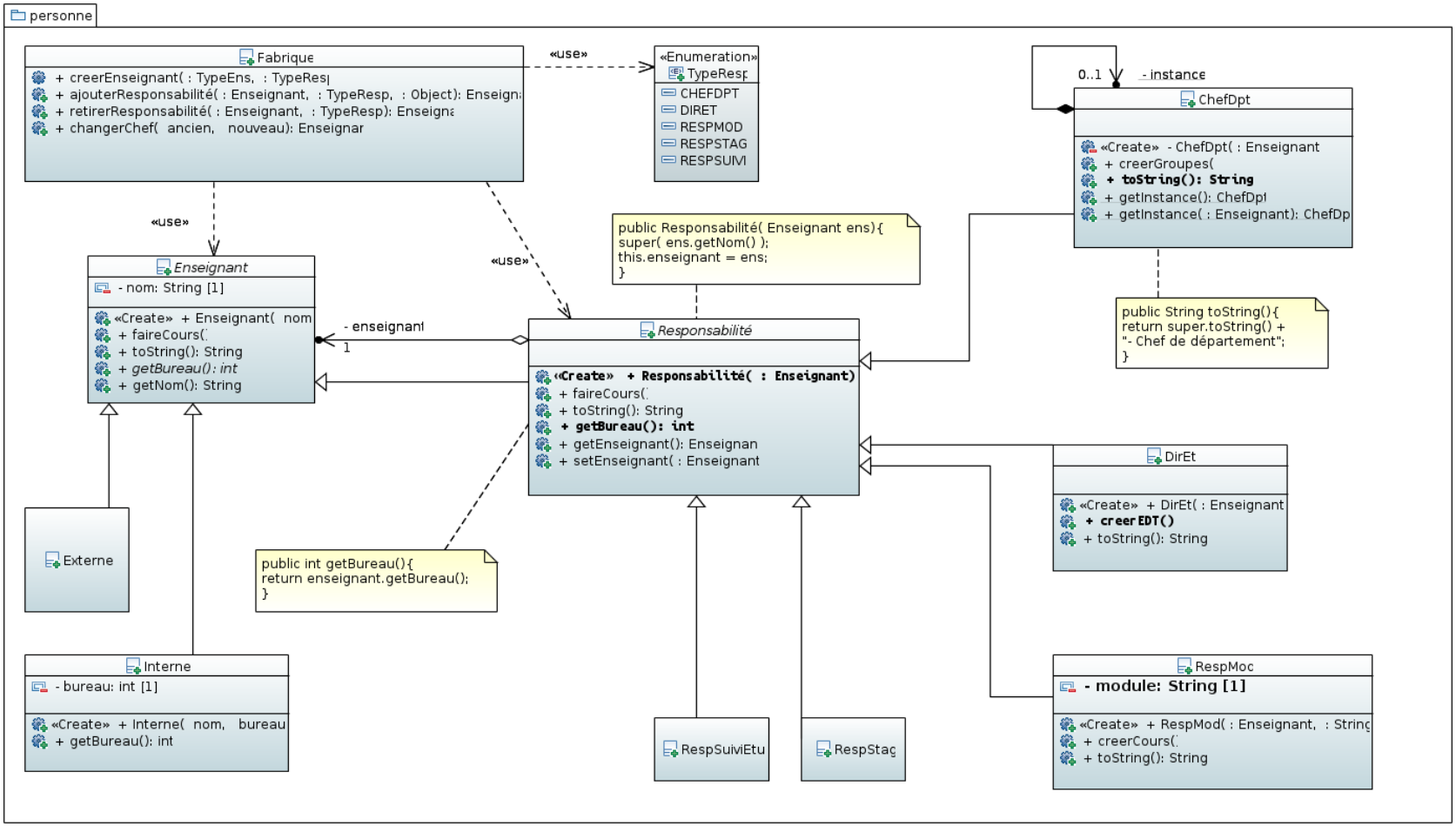


(source : GoF).

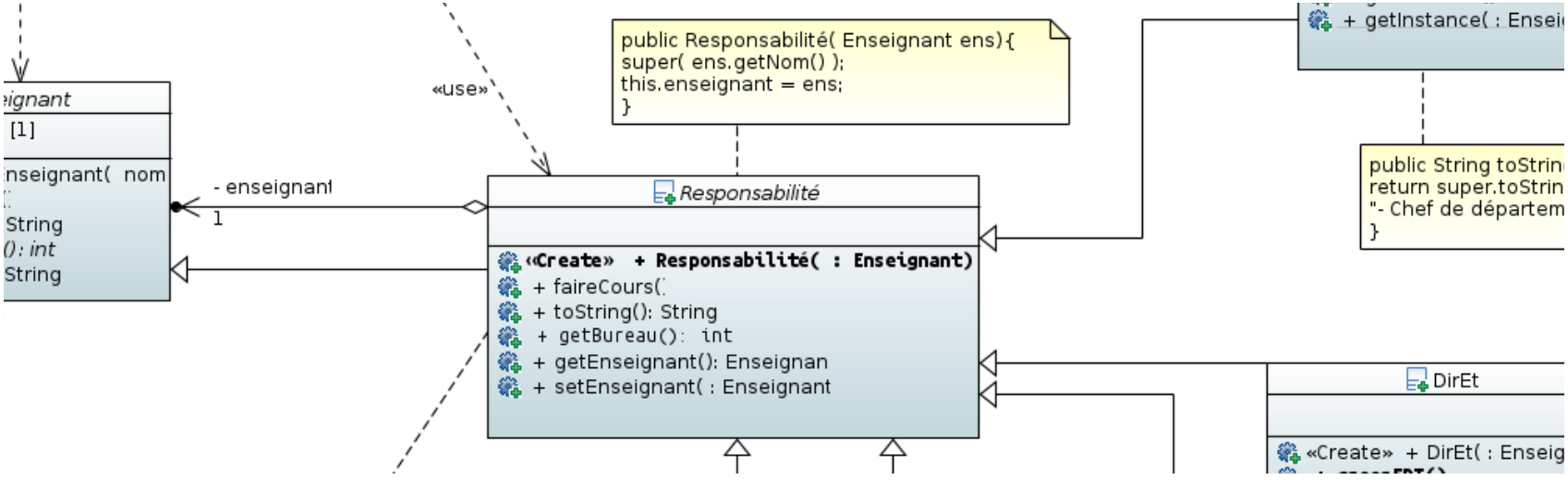
Exemples :

- système de fenêtres (avec ou sans décorations),
- gestion de parc de véhicules (modèles/options),

Décorateur

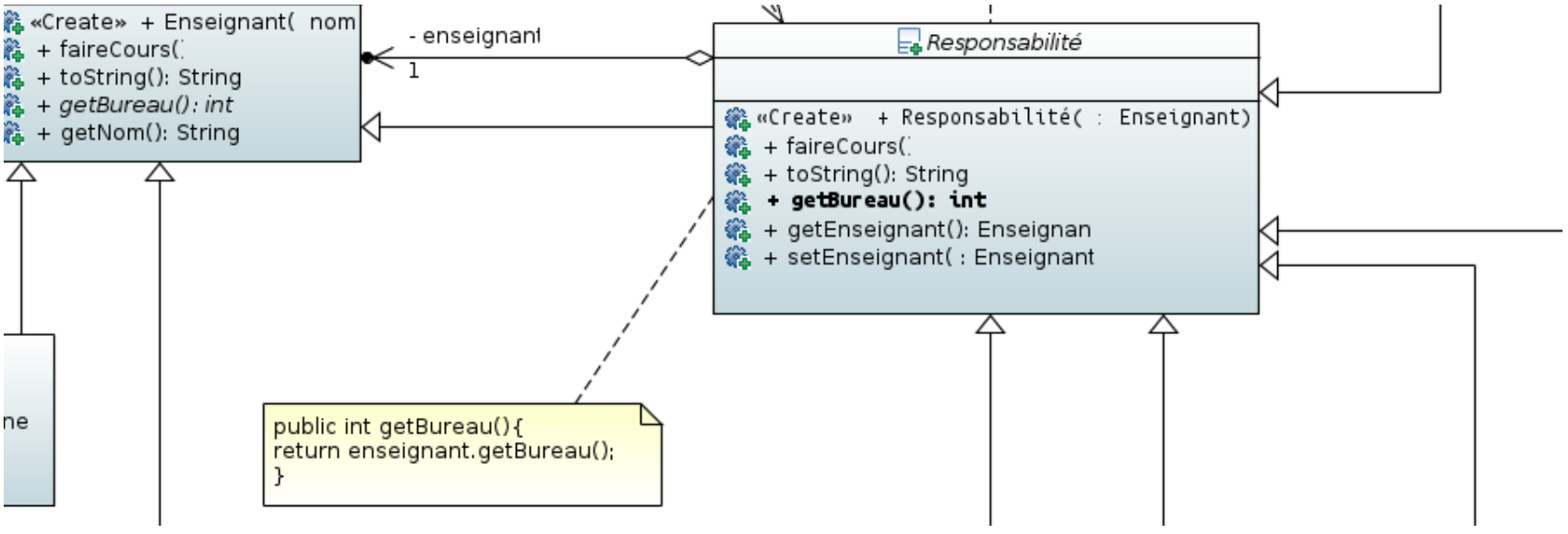

personne
Fabrique
+ creerEnseignant(: TypeEns, : TypeResp
+ ajouterResponsabilité(: Enseignant, : TypeResp, : Object): Enseign
+ retirerResponsabilité(: Enseignant, : TypeResp): Enseigna
+ changerChef(ancien, nouveau): Enseignar
«use»
«Enumeration»
TypeResp
CHEFDPT
DIRET
RESPMOD
RESPSTAG
RESPSUIVI
0..1
- instance
ChefDpt
«Create» - ChefDpt(: Enseignant
+ creerGroupes(
+ toString(): String
+ getInstance(): ChefDpt
+ getInstance(: Enseignant): ChefDp
«use»
«use»
Enseignant
- nom: String [1]
«Create» + Enseignant(nom
+ faireCours(
+ toString(): String
+ getBureau(): int
+ getNom(): String
- enseignant
1
public Responsabilité(Enseignant ens){
super(ens.getNom());
this.enseignant = ens;
}
Responsabilité
«Create» + Responsabilité(: Enseignant)
+ faireCours(
+ toString(): String
+ getBureau(): int
+ getEnseignant(): Enseignan
+ setEnseignant(: Enseignant
public String toString(){
return super.toString() +
"- Chef de département";
}
DirEt
«Create» + DirEt(: Enseignant
+ creerEDT()
+ toString(): String
Externe
public int getBureau(){
return enseignant.getBureau();
}
Interne
- bureau: int [1]
«Create» + Interne(nom, bureau
+ getBureau(): int
RespSuiviEtu
RespStag
RespMoc
- module: String [1]
«Create» + RespMod(: Enseignant, : String
+ creerCours(
+ toString(): String

Décorateur



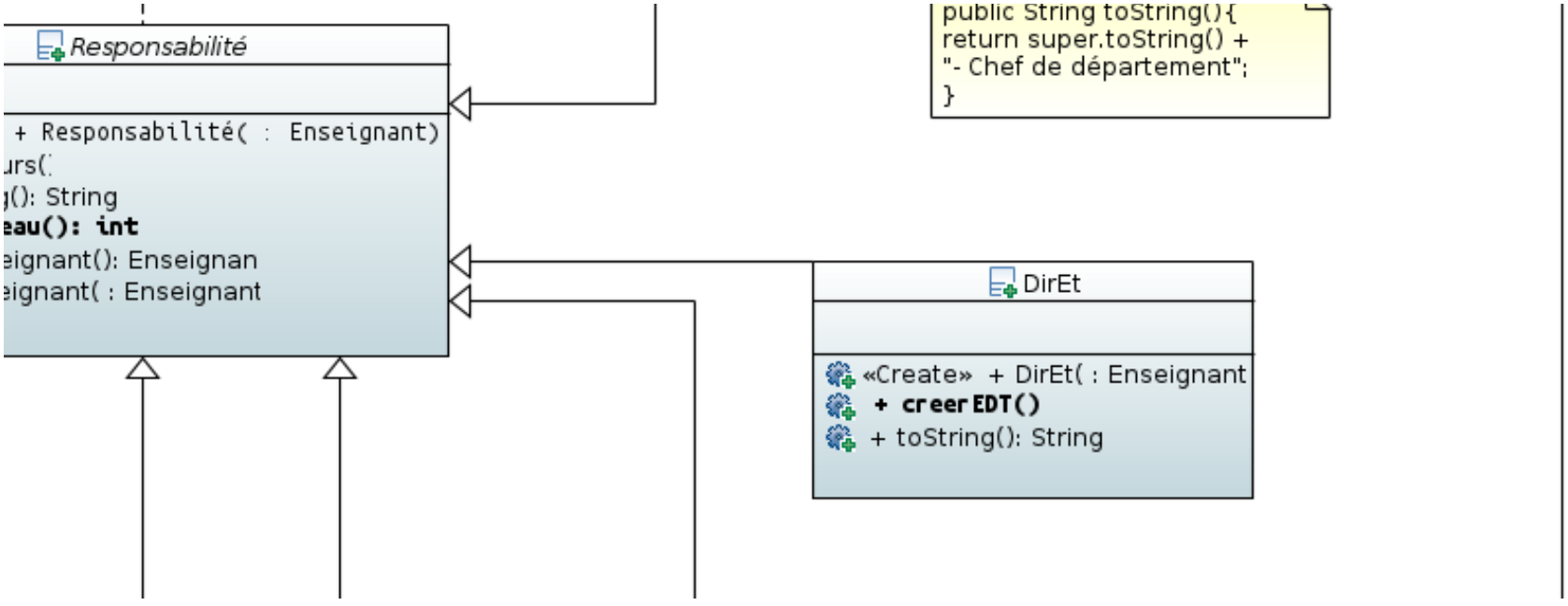
Nécessité de pouvoir appeler immédiatement le constructeur `super`

Décorateur



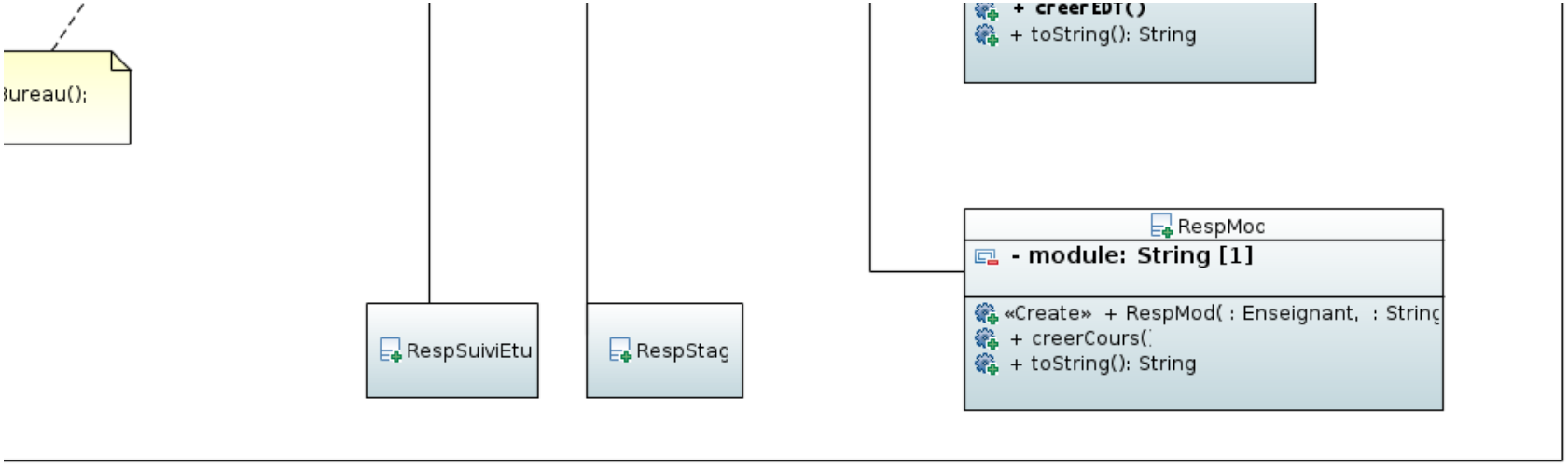
Par défaut, la décoration transmets les messages à l'objet décoré

Décorateur



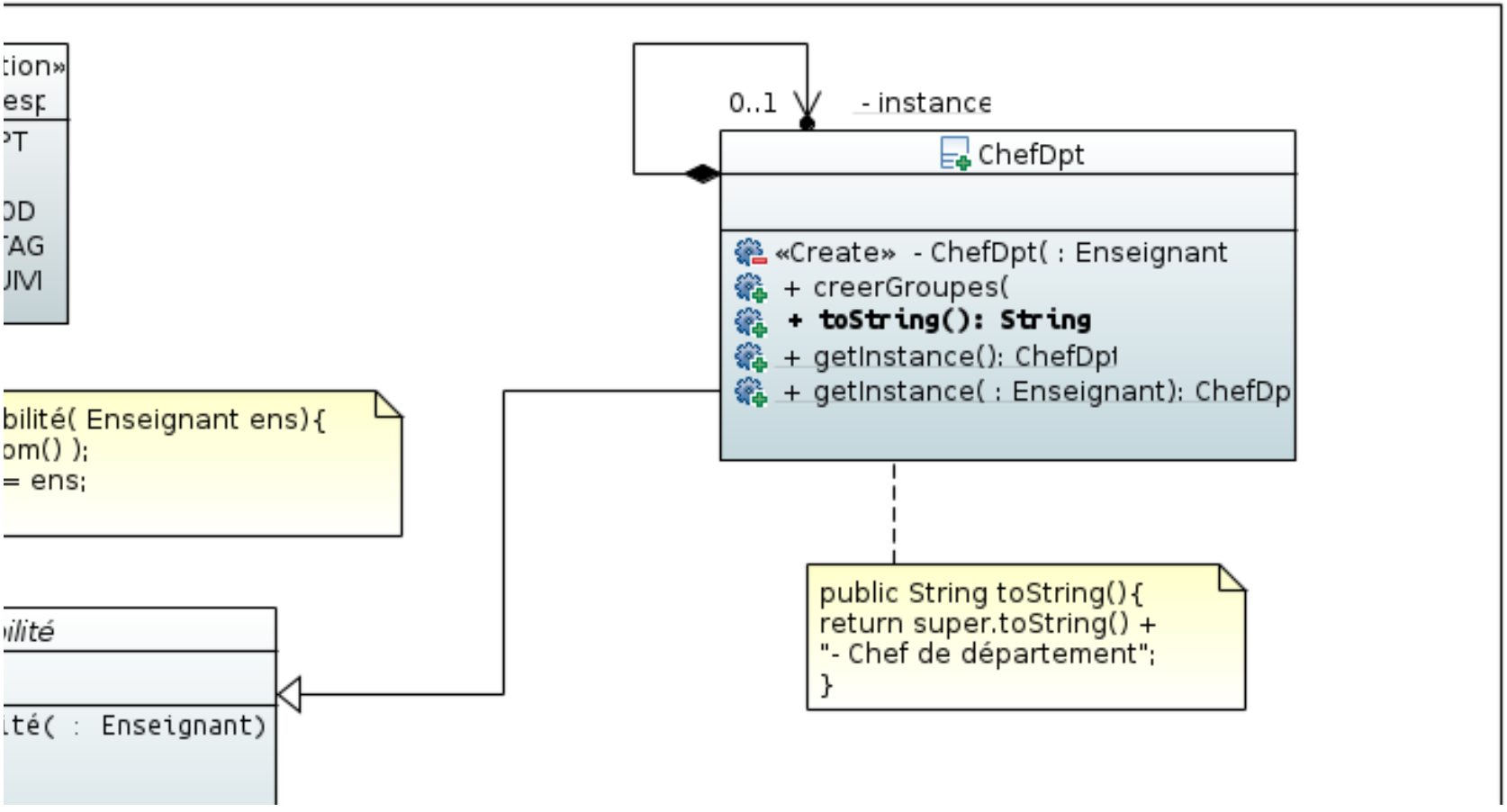
La décoration peut ajouter des comportements

Décorateur



La décoration peut ajouter des attributs

Décorateur



La décoration peut modifier des comportements

Décorateur : usage basique

- Ajouter des décorations à un `Enseignant` *h* :

```
h = new RespMod( h, "M31" );
h = new DirEt( h );
```

- Retirer une décoration :

```
h = ( (Responsabilité) h).getEnseignant();
```

Et pour retirer une décoration interne ?

- basique : retirer les décorations jusqu'à celle recherchée ; l'enlever et remettre les autres décorations.
- mieux : parcourir la structure composite (une chaîne) jusqu'au maillon recherché ; ouvrir la chaîne, enlever le maillon, refermer la chaîne.

Patron **Décorateur**

Discussion

- ☹ Redéfinition des méthodes du décoré
- ☺ On peut empiler les couches de décoration
- ☺ On peut retirer une décoration
- ☺ Renforce la cohésion
- ☺ Réduit la complexité : moins de classes

Le patron Plan (*template*)

Patron **Template**

Problème Définir le squelette d'un algorithme à l'aide d'une opération.

Solution Faire appel dans l'algorithme à des opérations abstraites.

Les sous-classes peuvent/doivent redéfinir certaines étapes sans changer la structure de l'algorithme.

public abstract class MyTemplateImageLoader

```
public final void myTemplateMethod( String file ) {

    Window w = creerFenetre( );

    lireImage( w, file ) ;

    fixerPosition( w );

    fixerTaille( w );

    fixerFermeture( w );

    afficher( w );
}

...
```

public abstract class MyTemplateImageLoader (suite)

```
protected W. creerFenetre(Str. s){return new Window(null);}

protected abstract void fixerPosition(W. w);

protected void fixerTaille(W. w){((Window) w).pack()}

private void lireImage(Obj. w,Str. s) throws IOException {
        JLabel img = new JLabel(new ImageIcon(file));
        hookOperation( img );
        w.add(img);
}

protected void hookOperation(Jlabel img) { }

protected void fixerFermeture(W. w) { }

private afficher( w ) { (w.setVisible( true ); }
```

// doit être redéfini -peut être r. -ne peut pas -ne fait rien (peut être r.)

public class MyConcreteImageLoader extends MyTemplateImageLoader

```
@Override
protected Window creerFenetre( String s) {
    return new JFrame ( s );
}

@Override
protected void fixerPosition(Window w) {
    w.setLocation( 500, 1000 );
}

@Override
protected void hookOperation( JLabel img ) {
    img.setBorder(new LineBorder( Color.BLACK, 100) );
}

public static void main( String s ){
    new MyConcreteImageLoader("img.png");
```

Patron **Template**

Discussion

- Permet de factoriser des comportements similaires (évite la redondance de code)
- Permet un contrôle fin de ce qui peut être modifié dans l'algorithme
- Attention à ne pas imposer trop de redéfinitions aux clients.

Les énumérations

Aparté : les énumérations Java

Une énumération est une classe définissant ses propres instances, exhaustivement :

```
public enum Couleur {PIQUE, COEUR, CARREAU};

Couleur p = Couleur.PIQUE; // OK
```

Il est impossible de créer d'autres instances (pas de `new`) :

```
Couleur TREFLE = new Couleur(); // ERREUR
```

Énumérations : contrôle du flux de programme

Comme les valeurs des énumérations sont des constantes, elles peuvent servir de condition dans des `switch-case` :

```
Couleur c = ... ;

switch(c) {
  case PIQUE: ... ; break;
  case COEUR: ... ; break;
  case CARREAU: ... ; break;
  default: ... ; // c == null ?
}
```

Énumérations : itérations

Les énumérations possèdent la méthode `values` :

```
/**
 * @return les instances de cette énumération
 */
public static E[] values()
```

où `E` est le type associé à l'énumération.

La méthode `values()` permet d'itérer sur l'énumération :

```
for (Couleur c: Couleur.values()) {
    ... ;
}
```

FIN