Design Pattern structuraux

Jonathan Lejeune

Programmation Objet
Sorbonne Université/LIP6

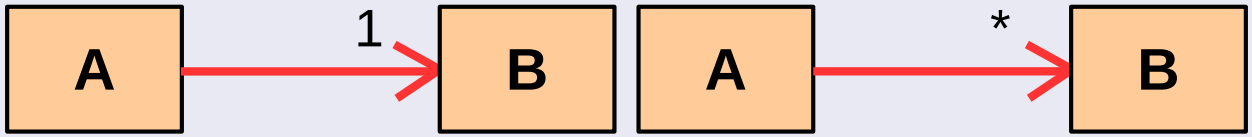
sources et images

- refactoring.guru
- cours précédents de Y. Thierry-Mieg

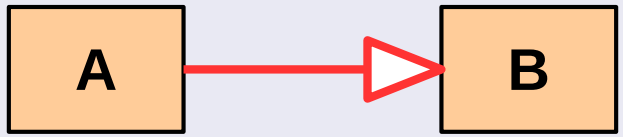
Rappels des relations entre types

Lien d'**utilisation** : *A utilise un B*

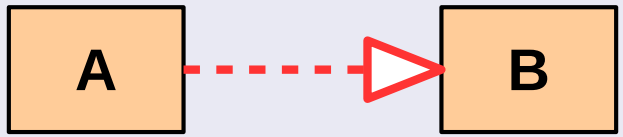
Lien de **composition** : *A contient un ou N B*



Lien d'**héritage** : *A est un B* ou *la représentation de A étend celle de B*



Lien d'**implémentation** : *A implante B*



Design Pattern (ou patron de conception)

Qu'est-ce que c'est ?

Méthode standard de bonne pratique de programmation objet réutilisable sur des problèmes de conception reconnus récurrents.

Qu'est-ce que ça n'est pas ?

Un algorithme avec un ensemble d'actions bien définies

$\Rightarrow$ le pattern doit être adapté au code auquel on souhaite l'appliquer

Origine

- Le GOF (Gang Of Four)

  Gamma, Helm, Vlissides, Johnson (1995). Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2
- Définition de 23 patterns dont la majorité sont devenus des standards
- Forte influence sur les APIs des langages objets (dont java)

Objectif des DPs

Produire un code bien organisé

- Flexibilité $\Rightarrow$ Avoir un code modulaire qui s'adapte facilement au besoin du programmeur
- Maintenabilité $\Rightarrow$ Factoriser le code (pas de copier/coller)
- Extensibilité $\Rightarrow$ Étendre une application sans modifier l'existant
- Configurabilité $\Rightarrow$ Avoir un code facilement paramétrable
- Élégance $\Rightarrow$ Avoir un code à la fois robuste et simple à comprendre

Séparer les problèmes

Utilisation massive des abstractions et des interfaces

Principes communs des DPs

Principe 1 : Favoriser la composition à l'héritage

- La composition est un lien dynamique (plus flexible)
  exemple : mécanisme de délégation
- L'héritage est un lien statique (moins flexible)

Attention : Favoriser ne veut pas dire remplacer systématiquement

Principe 2 : Utiliser les types abstraits

Éviter d'utiliser le type de la classe d'implémentation

ex : utiliser `List<T>` au lieu de `ArrayList<T>`

Principe 3 : Encapsuler fortement

Permettre l'isolation et ainsi être plus modulaire

Les familles de DPs

Les patterns de création

- Mécanismes de création/d'instanciation d'objet
- Focus : constructeurs, clonage

Factory,***Builder***,*FactoryMethod*, *Prototype*, ***Singleton***

Les patterns structuraux

- Assemblage, agencement et structure des objets
- Focus : dépendances, composition, héritage

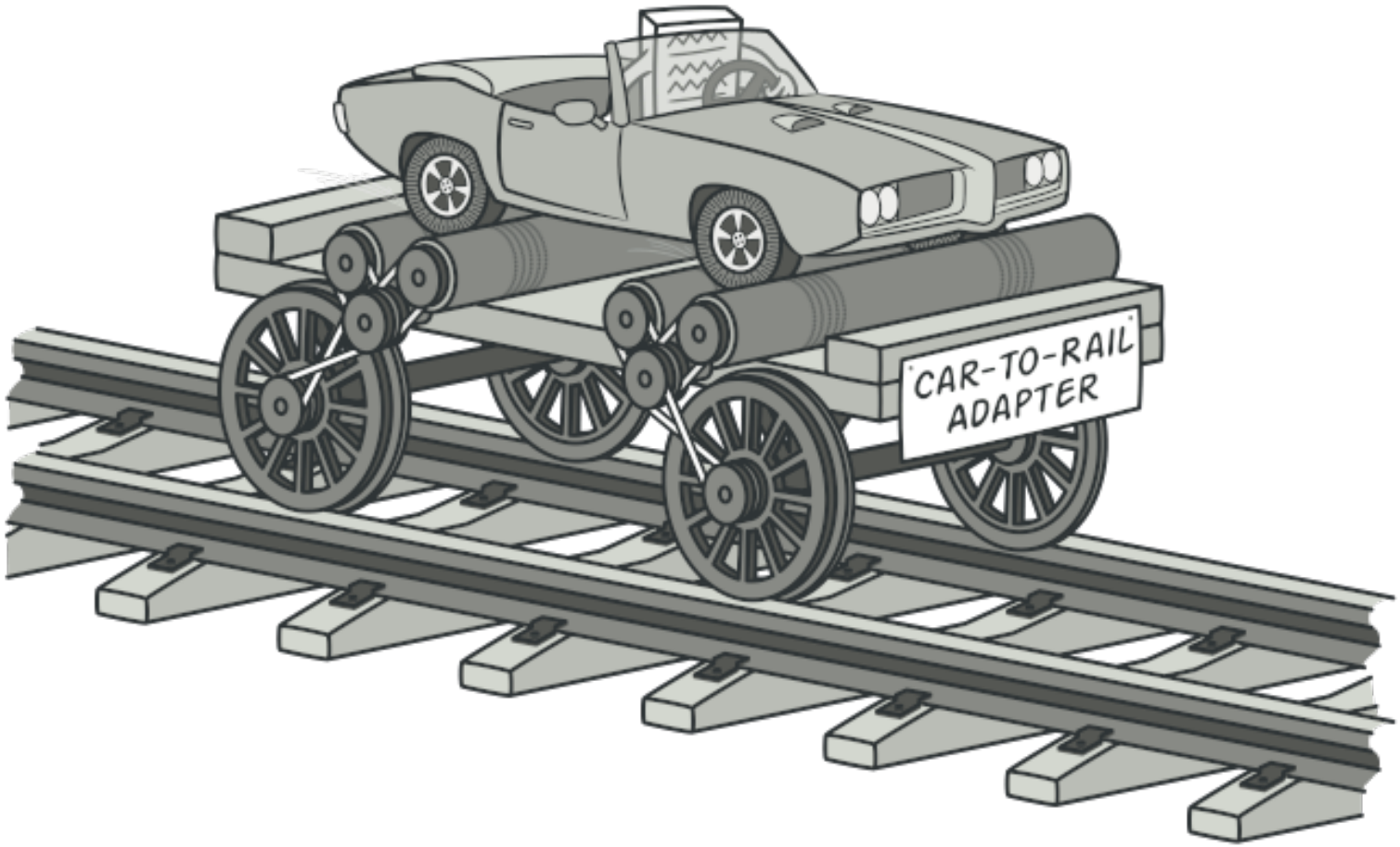
Adapter, *Bridge*, ***Composite***, ***Decorator***, *Facade*, *Flyweight*, *Proxy*

Les patterns comportementaux

- Communication et affectation des responsabilités entre objets
- Focus : méthodes offertes et leurs implémentations

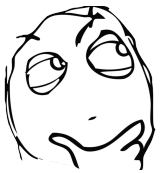
ChainOfResponsibility, ***Command***, ***Iterator***, *Interpreter*, *Mediator*, *Memento*, ***Observer***, ***State***, ***Strategy***, *TemplateMethod*, ***Visitor***

Adapter



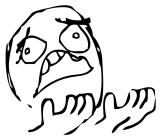
Crédit image : refactoring.guru

Scénario illustratif de la problématique



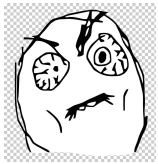
On code une librairie pour faire des traitements analytiques sur des fichiers XML

On veut ensuite l'enrichir avec une librairie analytique tierce

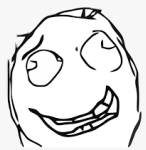


Mais la librairie tierce est faite pour les fichiers JSON

Solution : télécharger le code source et modifier la librairie JSON



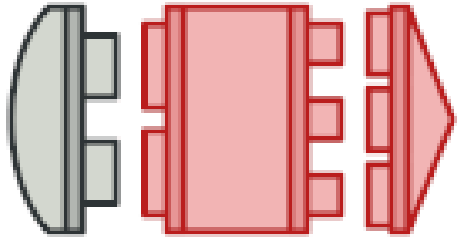
Le code source n'est pas disponible



Solution : hériter et redéfinir des méthodes

Certaines classes sont déclarées final

Le design-pattern Adapter



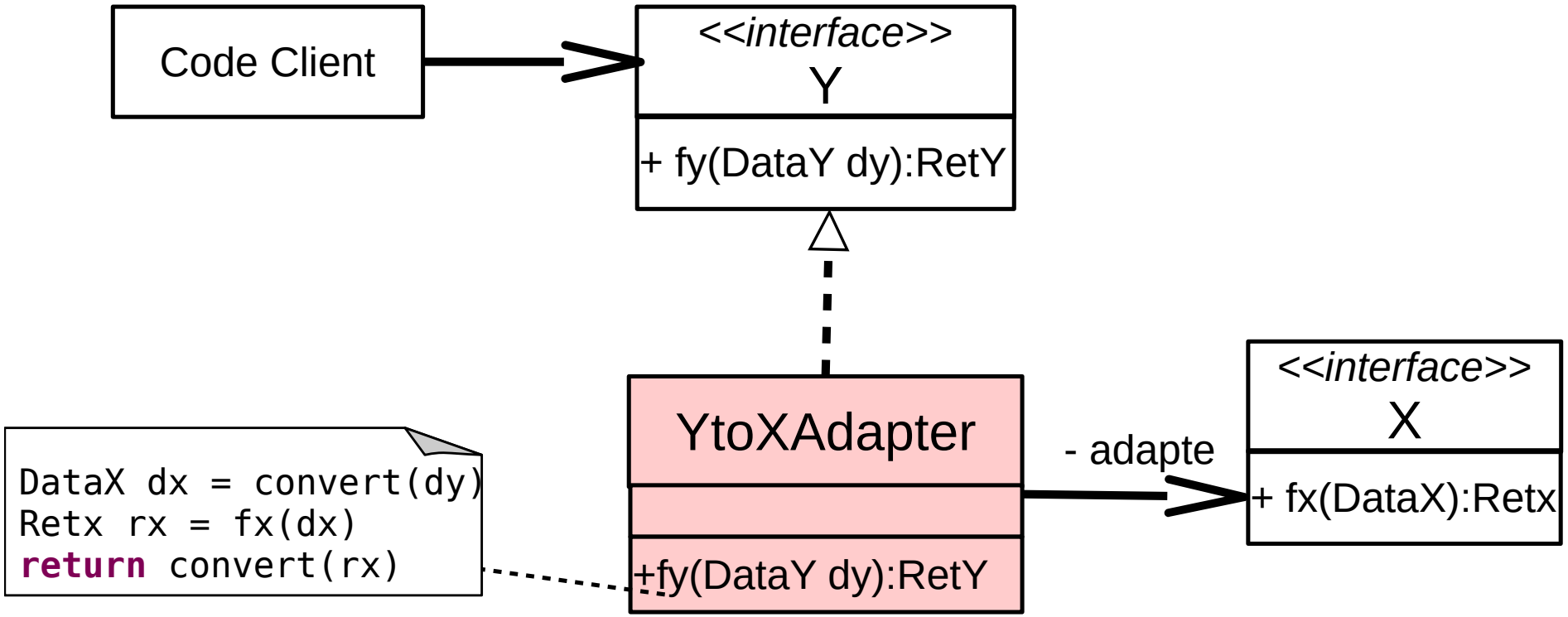
Définition

- Patron de conception structurel
- Conversion d'une API d'interface X existante pour se conformer à une API d'interface Y attendue par le client.

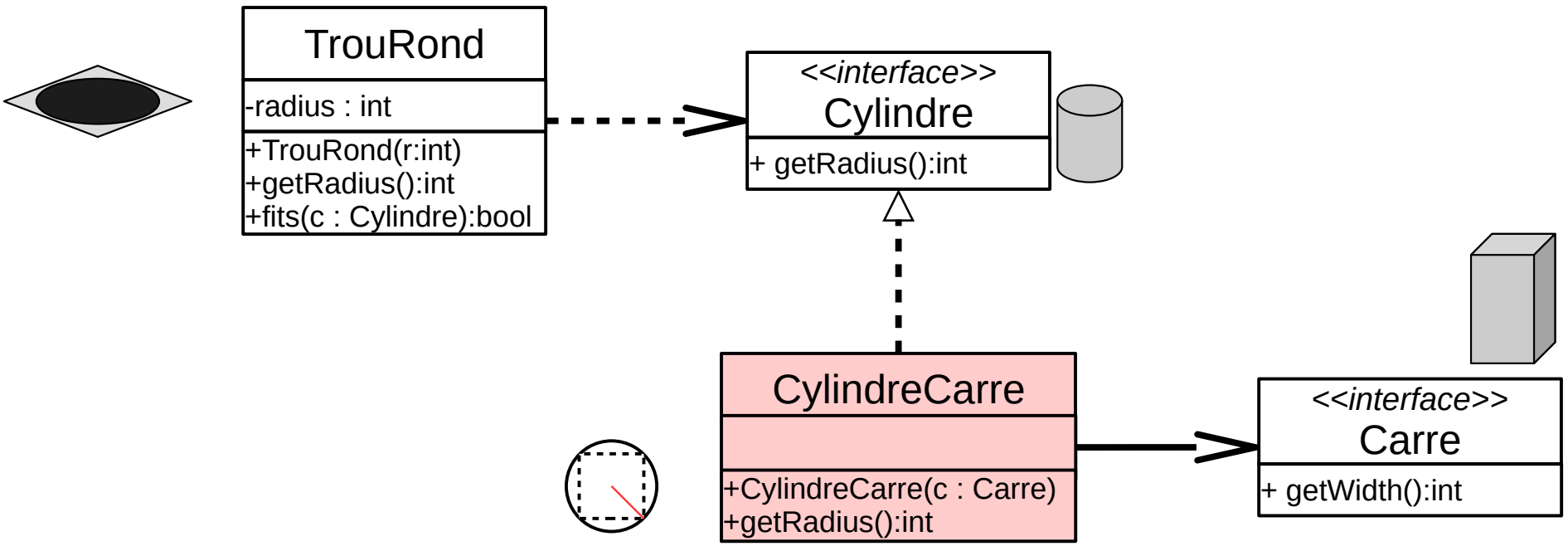
Étapes de mise en place

1) S'assurer qu'il existe une incompatibilité entre X et Y
2) Fournir une classe étendant Y et qui possède une référence de type X
3) Implanter une par une les méthodes pour convertir le traitement Y vers l'interface X
4) Instancier l'adapter en lui fournissant une instance de X

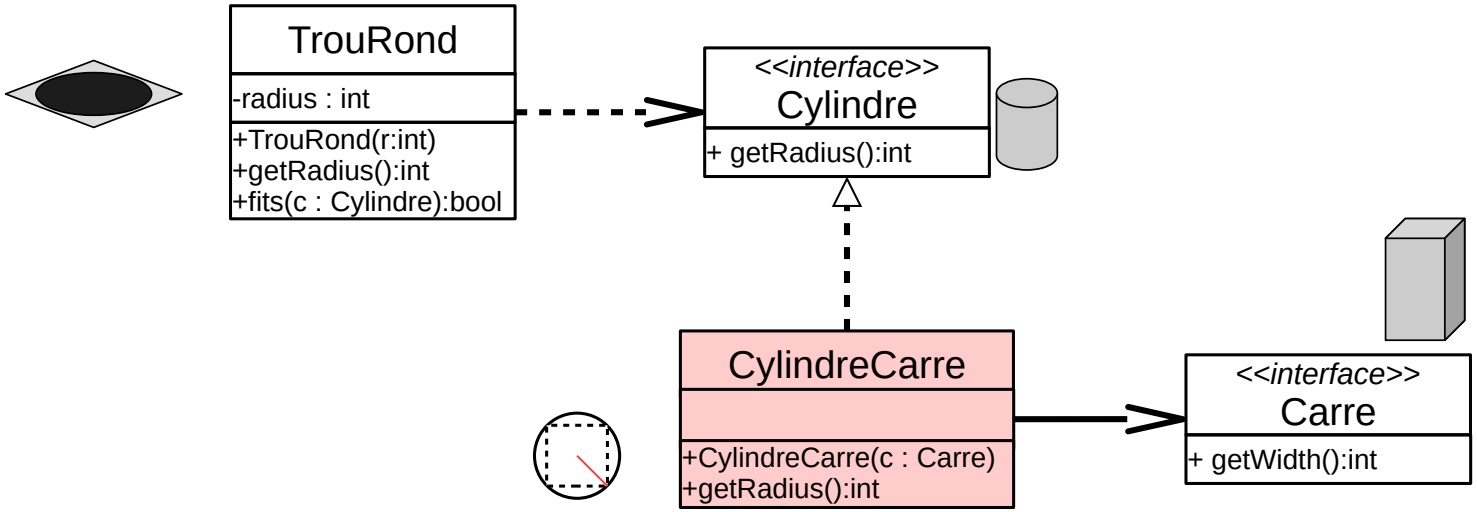
Le design-pattern Adapter : schéma UML



Le design-pattern Adapter : exemple

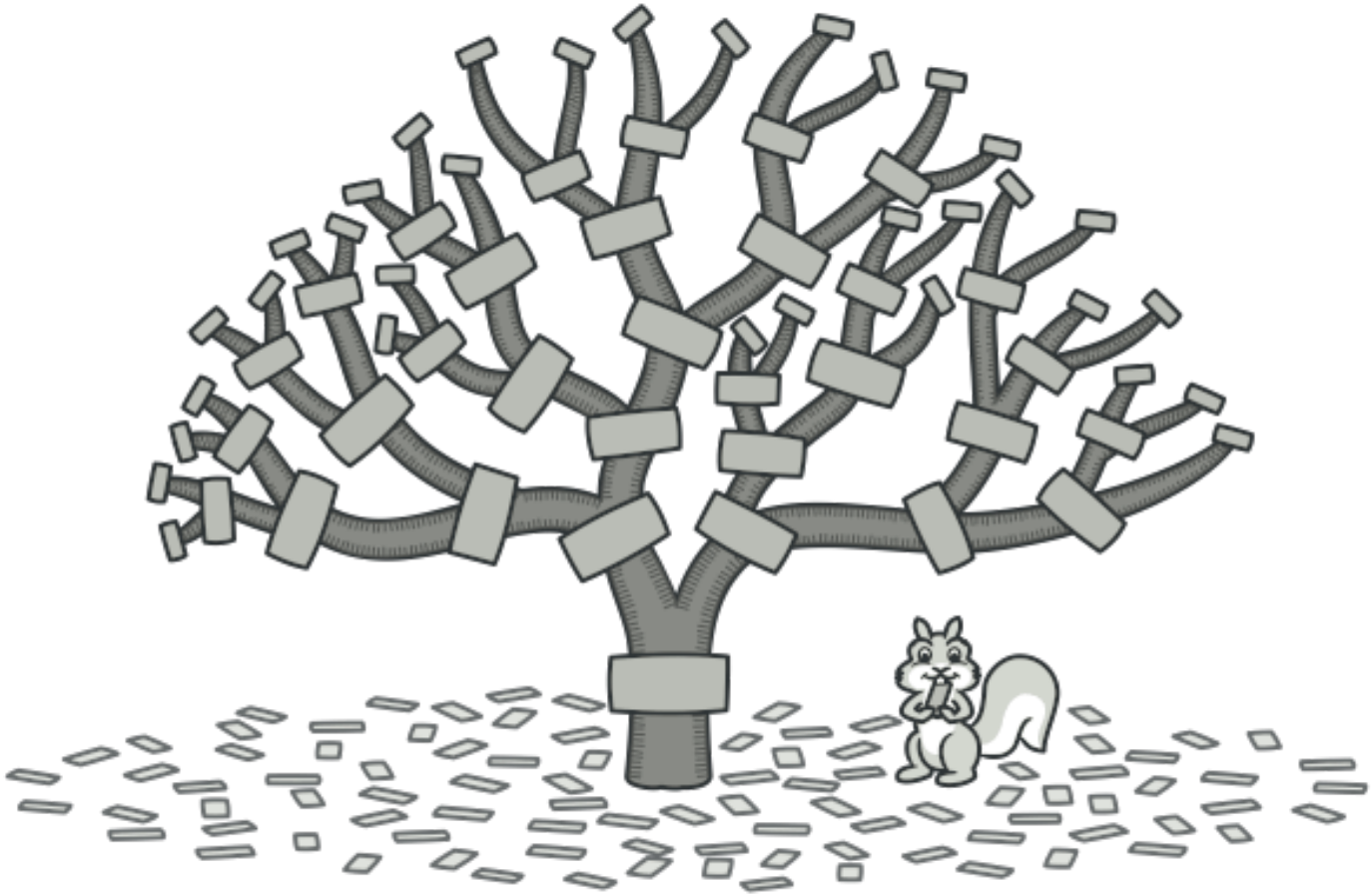


Le design-pattern Adapter : exemple



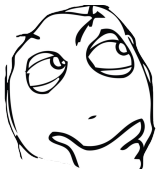

```
public class CylindreCarre implements Cylindre{
  private Carre adapte;

  public CylindreCarre(Carre c){
    this.adapte=c;
  }

  public int getRadius(){
    return (int) (adapte.getWith()/ Math.sqrt(2));
  }
}
```

Composite

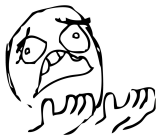


Crédit image : refactoring.guru

Scénario illustratif de la problématique

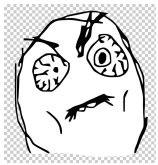


On souhaite coder une application graphique de dessin de différentes formes (carré, cercle, triangle . . .)

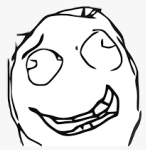


Comment gérer le déplacement et le positionnement des dessins ?

Solution : Parcourir en direct les éléments à modifier



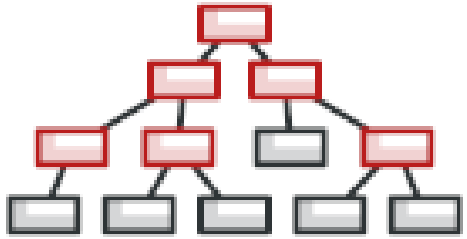
Mais il est possible de faire des groupes de dessins qui eux-même sont des dessins



Solution : Modifier les dessins en direct ou les dessins appartenant au même groupe

Il n'y a pas de limite dans la profondeur des groupes : on peut avoir des groupes de groupes de groupes .... de dessins

Le design-pattern Composite



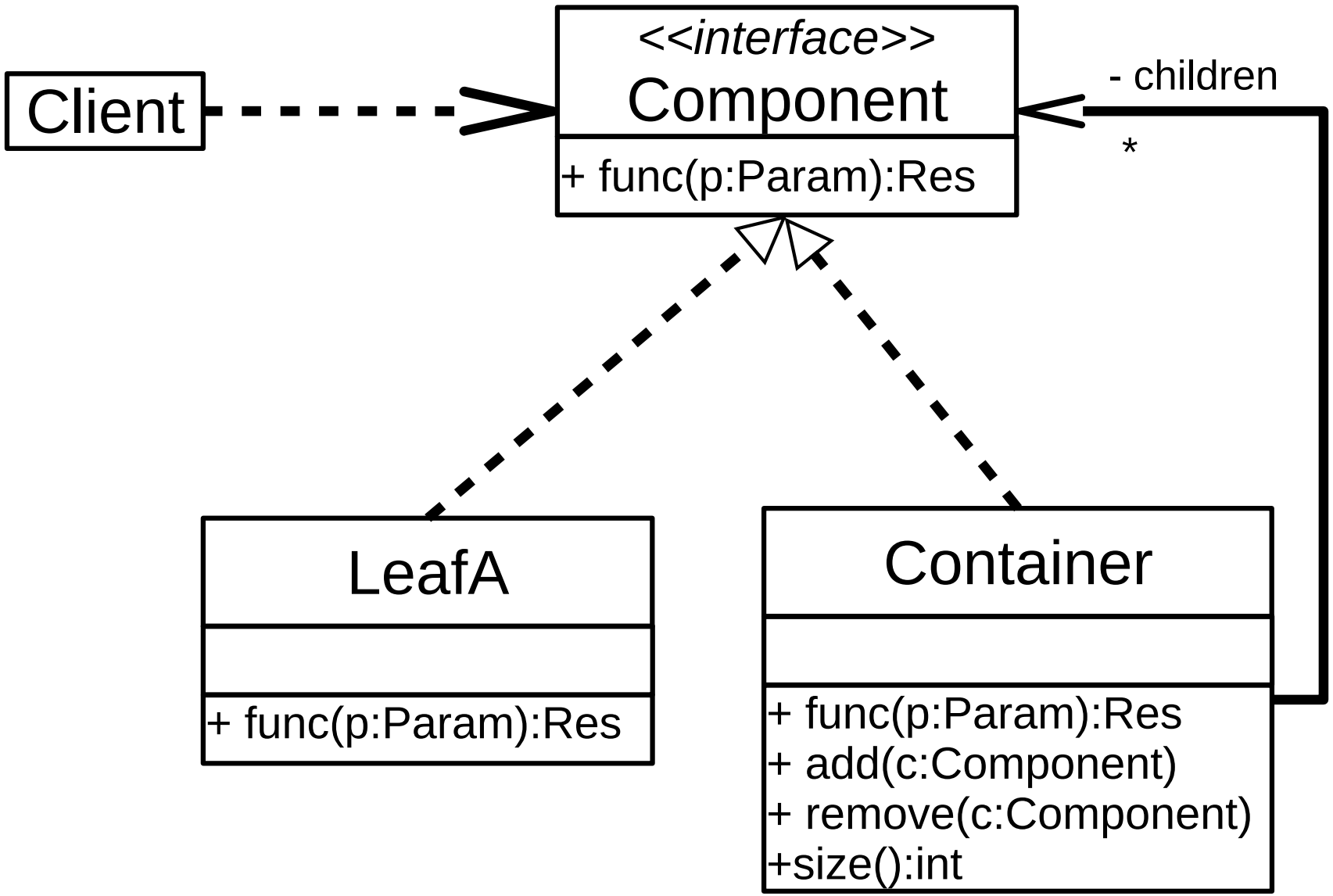
Définition

- Patron de conception structurel
- Structurer des objets pour former des hiérarchies arborescentes tout en permettant une manipulation homogène des feuilles et des nœuds.

Étapes de mise en place

1) S'assurer que l'application puisse se découper en composants arborescents
2) Déclarer une interface commune aux composants
3) Déclarer les classes des composants simples (feuilles)
4) Déclarer les classes des composants conteneurs (nœuds)

Le design-pattern Composite : schéma UML



Le design-pattern Composite : exemples concrets

Système de fichiers

- feuille : fichier simple
- nœud : répertoire

API graphique

- feuilles : bouton, label, formes, widgets
- nœuds : layout, Panel, Fenêtre, ...

Expressions arithmétiques ou booléenne

- feuilles : variable, valeur littérale
- nœuds : opération unaire ou binaire

Traitement de programme (AST)

- feuilles : variable, littéral,...
- nœuds : structures de contrôle, invocations ...

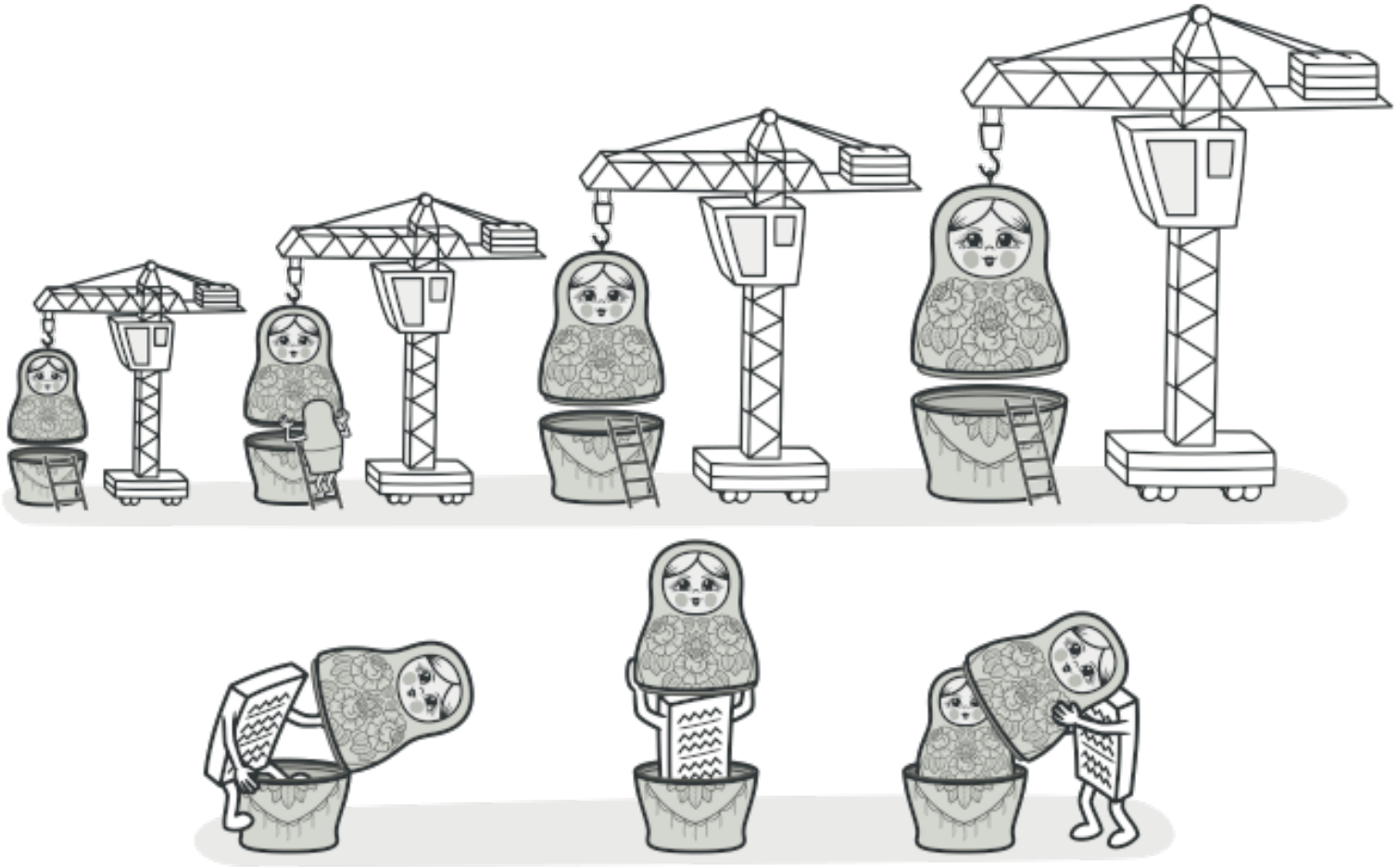
Le design-pattern Composite : exemple du filesystem

```
public interface Link {
  public int getSize();
}
```

```
public class File
    implements Link {

  private int size =0 ;

  public write(byte[] b){
    ...
    size+=b.length;
  }
  public int getSize(){
    return size;
  }
}
```

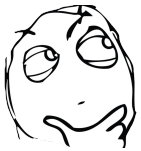
```
public class Directory
    implements Link {

  private List<Link> children
    = new ArrayList<>();

  public int getSize(){
    int cpt = 0;
    for(Link l : children){
        cpt+=l.getSize();
    }
    return cpt;
  }
}
```

Decorator



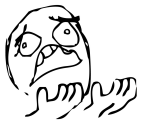
Crédit image : refactoring.guru

Scénario illustratif de la problématique

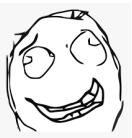


On souhaite définir un catalogue de voitures de différents modèles

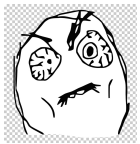
Solution : Définir une interface `Voiture` et implanter une classe pour chaque modèle



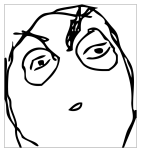
On souhaite ajouter une option diesel



Solution : Pour chaque modèle, écrire une sous-classe Diesel et une sous-classe Essence



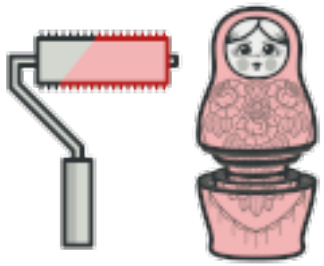
On souhaite ajouter l'option peinture métal



Solution : Pour chaque modèle et pour chaque motorisation essence et diesel écrire une classe avec peinture métal et sans peinture métal

Ah et on a aussi une option climatisation, turbo, boite auto, radar de recul et régulateur de vitesse

Le design-pattern Decorator



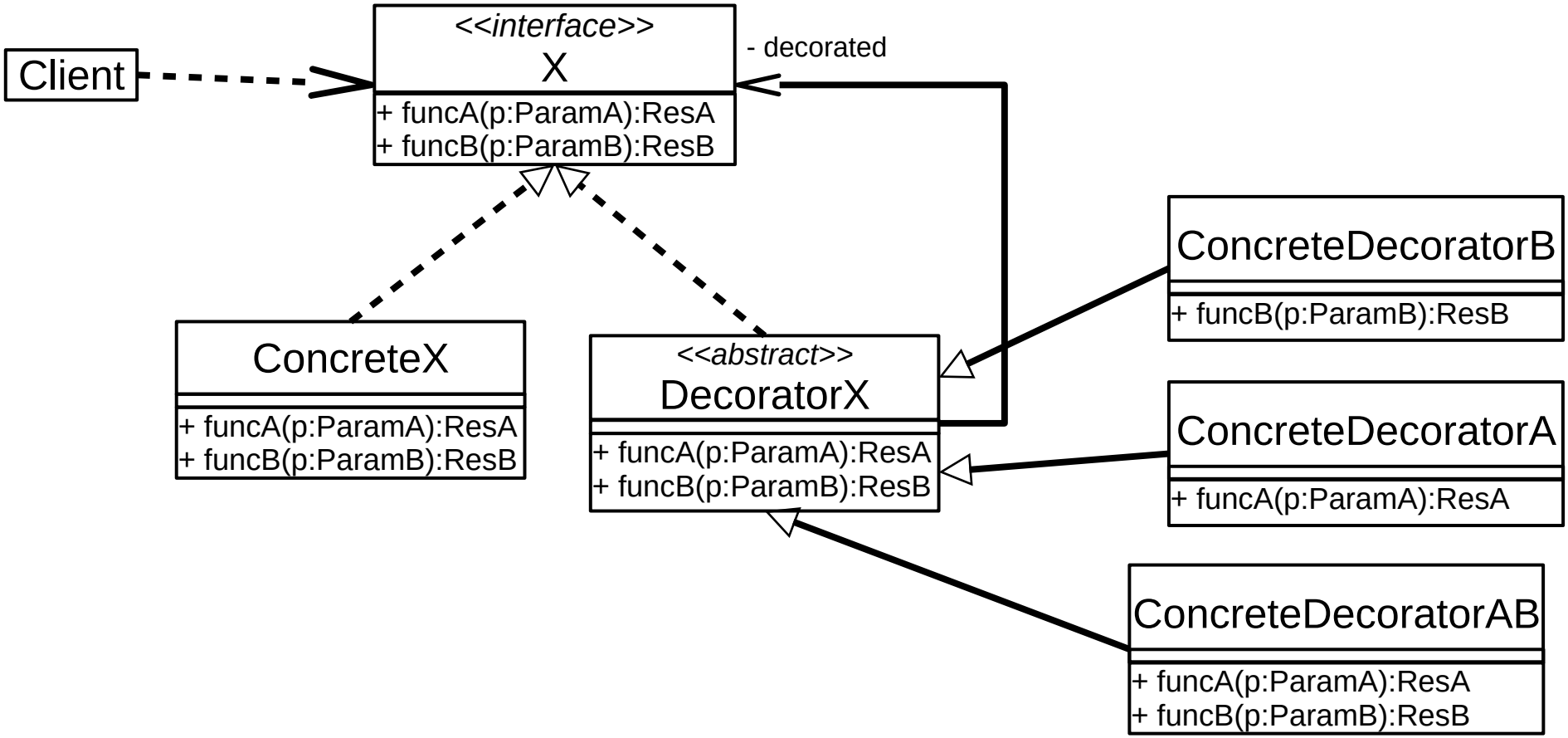
Définition

- Patron de conception structurel
- Ajouter modifier dynamiquement des responsabilités à des objets

Étapes de mise en place

1) S'assurer d'avoir un composant primaire offrant une interface *X*
2) Définir un décorateur abstrait *DecX* : implante `X` et délègue tous les appels sur un attribut de type *X*
3) Définir les décorateurs concrets qui sont des sous-classes de *DecX* et qui modifient les fonctionnalités qu'ils sont censés décorer
4) Le client compose à la volée les différents décorateurs

Le design-pattern Decorator : schéma UML



Le design-pattern Decorator : exemple (étape 1)

Définition de l'interface

```
public interface Voiture {
  public int getPrix();
  public int getPuissance();
}
```

Définition de l'implantation de base

```
public class ModeleA implements Voiture{
  public int getPrix() {
    return 8000 ;
  }
  public int getPuissance() {
    return 50;
  }
}
```

Le design-pattern Decorator : exemple (étape 2)

Définition d'un décorateur abstrait

```
public abstract class VoitureDecorator implements Voiture {
  private Voiture decorated;
  public   VoitureDecorator(Voiture d){
    this. decorated = d ;
  }
  public int getPrix(){
     return decorated.getPrix();
  }
  public int getPuissance(){
     return decorated.getPuissance();
  }
}
```

Le design-pattern Decorator : exemple (étape 3)

Définition d'un décorateur Diesel

```
public class Diesel extends VoitureDecorator {
 public Diesel(Voiture d){  super(d);  }

 public int getPrix(){   return 2000 + super.getPrix();  }

 public int getPuissance(){ return 30+super.getPuissance();}
}
```

Définition d'un décorateur Turbo

```
public class Turbo extends VoitureDecorator {
 public Turbo(Voiture d){  super(d);  }

 public int getPrix(){  return 1000 + super.getPrix();  }

 public int getPuissance(){
     return (int) (1.50 *  super.getPuissance());
  }
}
```

Le design-pattern Decorator : exemple (étape 4)

Composition par le client

```
//avoir une voiture Diesel
Voiture v1 = new Diesel(new ModeleA());

//avoir une voiture Turbo
Voiture v2 = new Turbo(new ModeleA());

//avoir une voiture Turbo Diesel
Voiture v3 = new Turbo(new Diesel(new ModeleA()));

System.out.println(v1.getPrix()+"␣"+v2.getPrix()+"␣␣"+v3.
    getPrix());
System.out.println(v1.getPuissance()+"␣"+v2.getPuissance()+"
    ␣"+v3.getPuissance());
```

Affichage produit

```
10000 9000 11000
80 75 120
```

Le design-pattern Decorator : synthèse

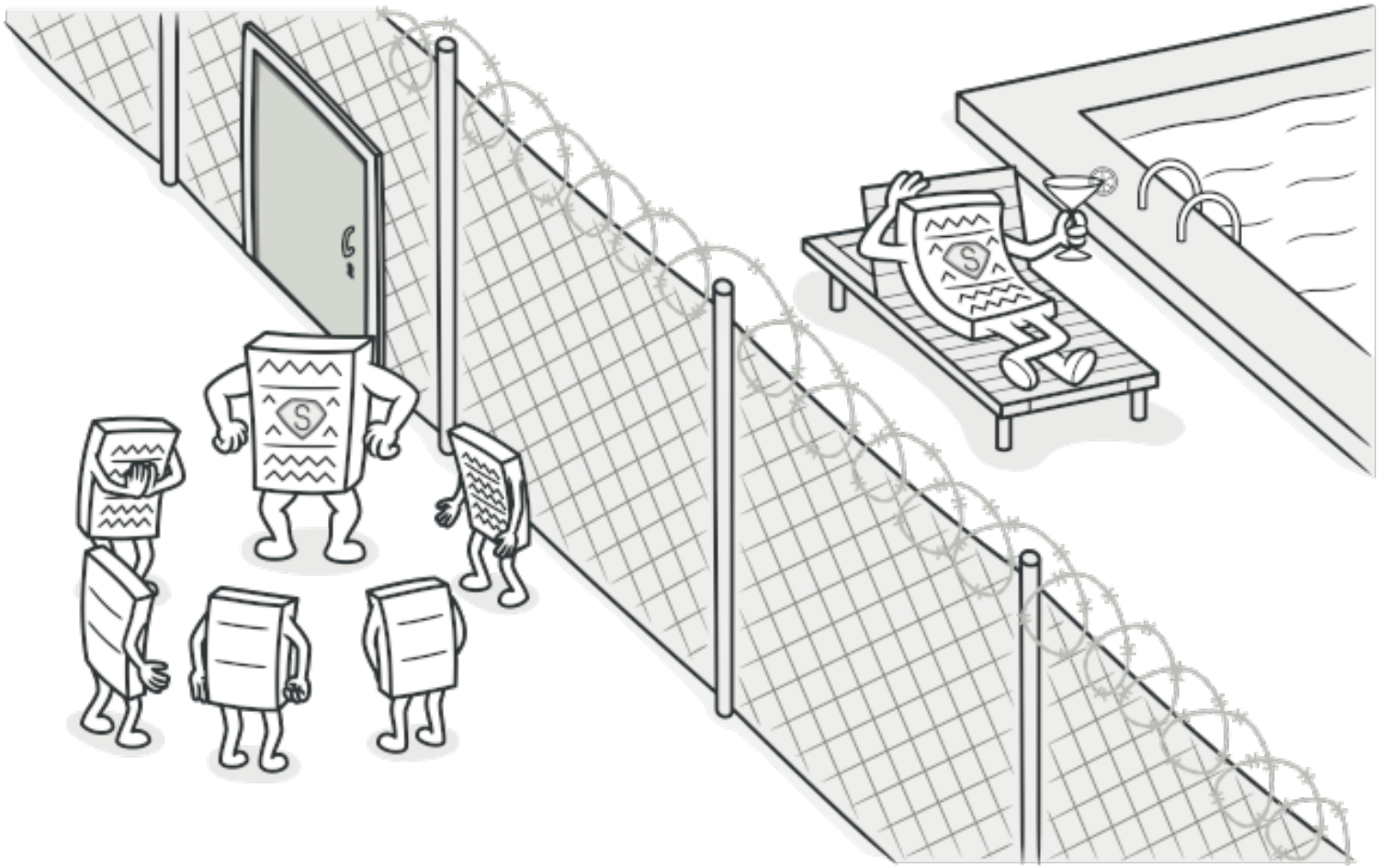
Points forts

- Il permet d'ajouter ou de supprimer dynamiquement des responsabilités par composition de classes
- Pas besoin de définir de nouvelle classe à chaque composition
- Un pattern puissant implanté dans les APIs standards (ex : les I/O java)

Limites

- Rien n'empêche d'appliquer plusieurs fois un décor
  $\Rightarrow$ on peut faire un diesel de diesel de diesel
- L'ordre d'instanciation des décors peut avoir un impact
  $\Rightarrow$ puissance d'un TurboDiesel $\neq$ puissance d'un DieselTurbo
- Difficile d'enlever un décor intermédiaire dans la pile des décors
  $\Rightarrow$ à partir d'un TurboDieselClim difficile d'avoir un TurboClim

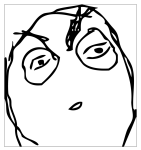
Proxy



Crédit image : refactoring.guru

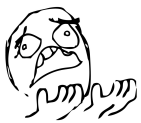
Scénario illustratif de la problématique

On possède un objet lourd à forte responsabilité gérant une base de données.

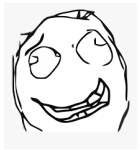


Comment gérer l'initialisation d'un tel objet ?

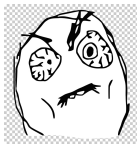
Solution : chaque client doit initialiser de manière paresseuse la base de données, ça duplique pas mal le code mais si il y a peu de clients ça passe.



L'objet est utilisé par beaucoup de client, de plus tous les clients n'ont pas les mêmes droits d'accès à cet objet



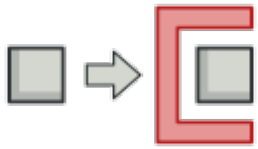
Solution : Modifier la classe de l'objet directement et ajouter des filtres de sécurité sur chaque méthode



La classe de l'objet est développée par un tiers et n'est pas open-source

Et j'oubliais : Les objets clients et l'objet en question ne sont pas forcément sur la même machine physique

Le design-pattern Proxy



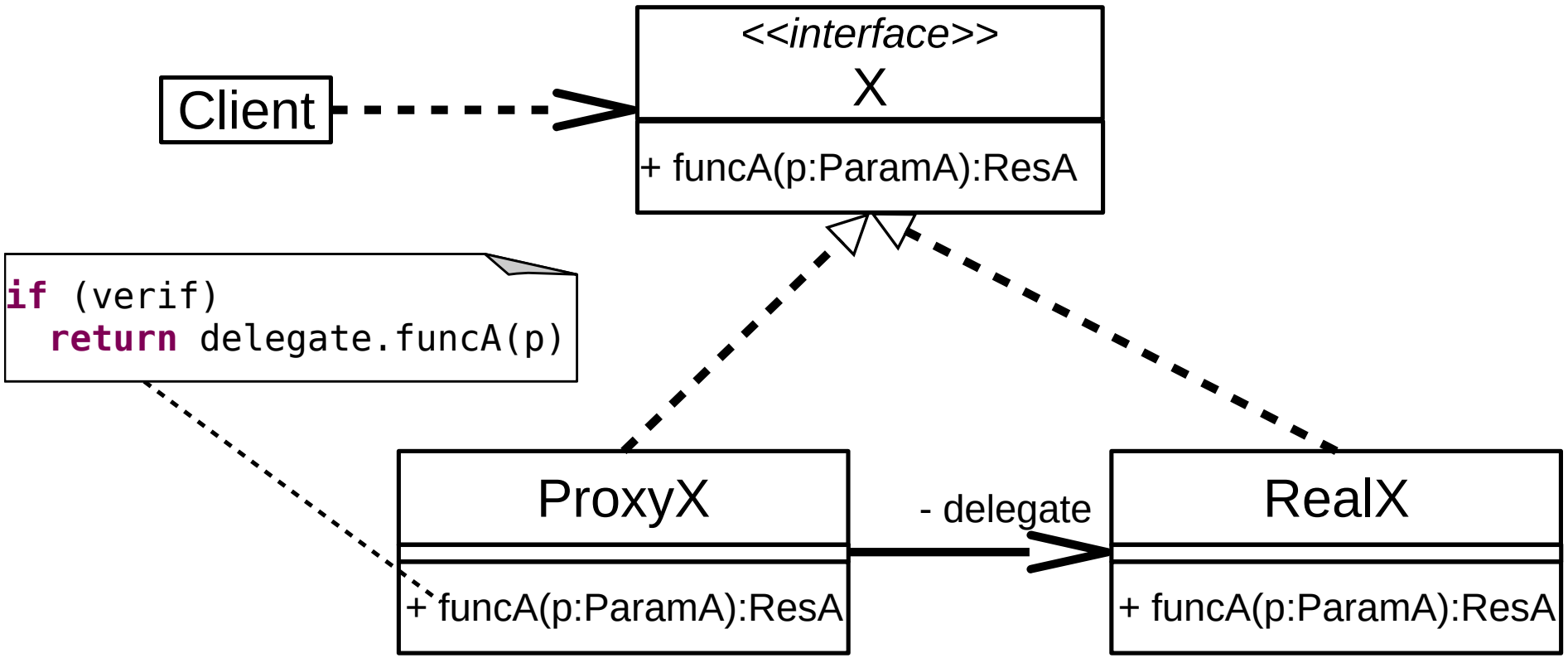
Définition

- Patron de conception structurel
- Utilisation d'un objet mandataire dialoguant avec un objet principal et qui offre tous les deux la même interface.

Étapes de mise en place

1) Créer éventuellement une interface exposant les méthodes publiques de l'objet principal
2) Créer une classe *Proxy* implantant l'interface et référençant l'objet principal
3) Implanter les méthodes du proxy en fonction de son rôle
4) Implanter un mécanisme permettant au client de référencer un proxy ou bien l'objet principal

Le design-pattern Proxy : schéma UML



Le design-pattern Proxy : exemples applicatifs

Proxy de sécurité

- Le proxy a la responsabilité de contrôler les accès à l'objet principal
  exemple : bloquer des messages sur un forum, gérer les droits d'accès
- L'objet principal ne gère pas la sécurité

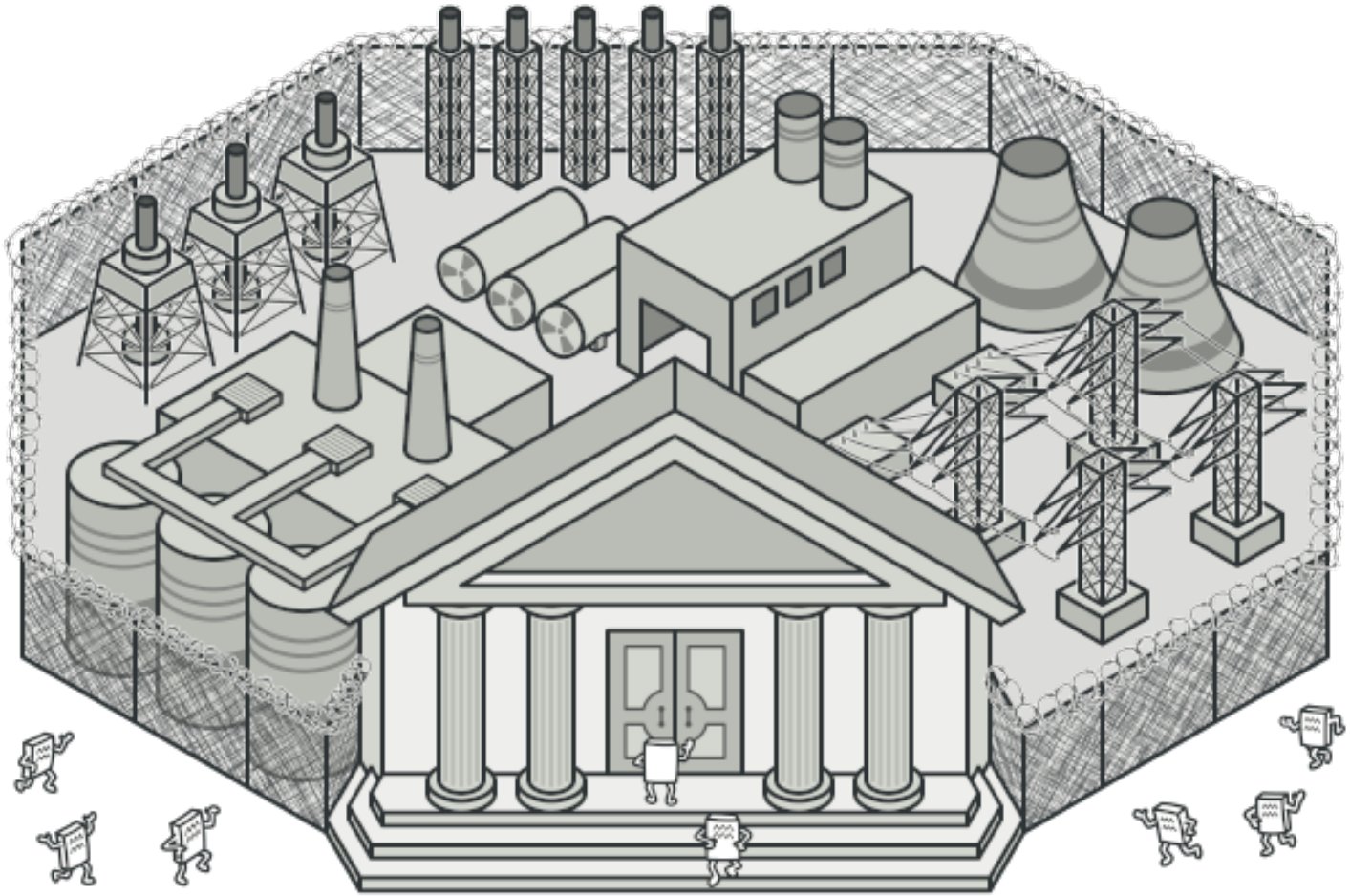
Proxy distant

- Le proxy et l'objet principal ne sont pas sur la même machine hôte
- Le proxy implante la communication réseau pour dialoguer avec l'objet principal exemple : Remote Procedure Call

Proxy Smart Reference

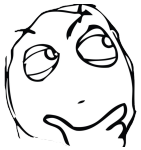
- Spécifique aux langages non munis de Garbage Collector
- Le proxy fait office de référence sur l'objet principal
- Maintien d'un compteur de référence global à mise à jour à la création, copie ou destruction d'un proxy sur l'objet

Facade



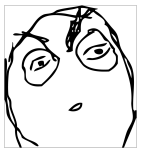
Crédit image : refactoring.guru

Scénario illustratif de la problématique

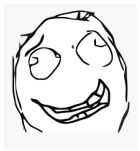


On souhaite réaliser une application de traitement vidéo. Fonctionnalité numéro 1 : convertir des vidéos

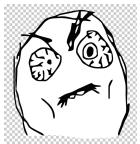
Solution : Télécharger et utiliser dans mon application des librairies existantes de Codec



Fonctionnalité numéro 2 : On souhaite améliorer la qualité des vidéos ainsi que le son



Solution : Télécharger et utiliser des librairies de traitement d'image et de son. Faire le lien avec les différents codecs.



On veut aussi que les vidéos puissent être envoyées par mail, mises sur un réseau social ou être sauvegardées sur disque

Solution : Ajouter des librairies d'interfaçage et d'export et lier le tout

On ne souhaite pas être dépendant de l'ensemble de ces librairies

Le design-pattern Façade

Définition

- Patron de conception structurel
- Offrir une interface simplifiée d'un sous-système complexe d'objets

Étapes de mise en place

1) Vérifier qu'il est possible de produire une interface simplifiée du sous-système en question
2) Définir une classe qui implante l'interface de façade et qui fait le lien entre les fonctionnalités et les différents éléments du sous-système
3) S'assurer que le client ne soit dépendant que de l'interface façade
4) Subdiviser éventuellement en sous-façades si trop de méthodes

Le design-pattern Façade : schéma UML