conception et prog objet avancés

DEV.3.4– CPOO

Florent Madelaine

IUT de Sénart Fontainebleau
Département Informatique

Année 2022-2023
Cours 4

Interface

En gros

comportement = des méthodes = interface

Question

Comment découper les interfaces ?

Réponse

Pour faciliter le travail en cas de changement, on ne laisse ensemble dans une interface que les méthodes en lien avec 1 comportement.

Exemple

être duplicable (Clonable) n'est pas le même comportement qu'être sauvable/récupérable (serializable). Donc on a deux interfaces.

SOLID

Jargon

Ce principe de découpage des interfaces correspond au **I** de SOLID.

- **I** interface-segregation principle

C'est l'analogue du S pour les classes.

- **S** Single responsibility principle

`https://en.wikipedia.org/wiki/SOLID`

Les notions pour l'évaluation

1. Usage : DCU, synopsis.
2. Structure : DC sans les méthodes, DO.
3. Comportement : DC avec méthodes, Diagramme de Séquence.
4. Abstraction : DC avec classes abstraites et interfaces, API, couplage faible.
5. Patrons : solutions à des problèmes typiques.

Patron de conception

Il s’agit de techniques plus ou moins complexes adressant des problèmes rencontrés en génie logiciel.

Vous en avez déjà manipulé même si vous ne le saviez pas.

UML est un langage pour communiquer

- Toutes les facilités offertes par UML ne sont pas forcément utilisées.
- Il convient d'être pragmatique et d'utiliser ce qui apporte quelque chose à la conception et documentation du projet.
- Le plus important reste le logiciel ou prototype livré.

UML pour nous c'est la démarche suivante

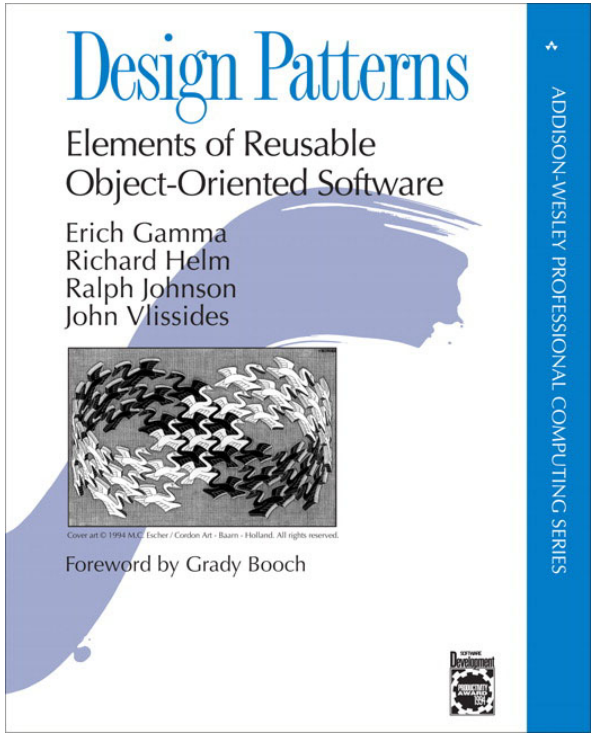
1. Un diagramme simplifié de cas d'usage (DCU)
   (pour les rôles + sorte d'index du projet)
2. Des exemples (synopsis)
3. Un diagramme de classes de conception simplifié
   (juste le modèle, pas de méthodes, on teste avec des diagrammes objets correspondant aux exemples)
4. Diagrammes de séquences systèmes (DSS)
   (pour des exemples, permet de trouver les méthodes du modèle)
5. Un diagramme de classes de conception complet (DC)
   (juste le modèle, mais cette fois avec les méthodes)
6. Un diagramme de classes d'analyse
   (précise, ajoute la vue le contrôleur etc)
7. Diagrammes de séquences
   (pour des exemples complets, permet de trouver les méthodes du diagramme d'analyse)

UML n’est pas magique

- Ce n’est pas parce que vous causez l’UML que vous savez concevoir correctement.
- (analogue : vous écrivez sans fautes mais vous n’avez pas d’idée pour votre dissertation)
- Mais vous n’êtes pas seuls !
- Vous arrivez après d’autres informaticiens qui ont fait évoluer de nombreuses façons de répondre proprement et de manière ingénieuse à tout un tas de questions récurrentes.
- Pour le génie logiciel, cela donne ce qu’on appelle en jargon les patrons de conceptions ou *design patterns*.

moyen-âge : 1994 et la bande des 4

- Le livre propose 23 patrons de conception
- Depuis, il y en a d'autres
- Certains patrons sont là pour pallier à des choses qui manquent dans les langages
- (première version de java = 96, langages objets historiques comme C++ datent des années 80 et ne proposent pas forcément tout ce qu'on souhaiterait)
- tous décrivent des constructions, structures ou comportement qui dépassent l'héritage ou les interfaces.



Le ***Gang of Four*** *(GoF)*

Les types de patrons de conception

Dans le GoF, les motifs sont partagés en 3 catégories.

Construction manières de créer de nouveaux objets

Structuration manières de structurer des programmes

Comportement manières d'interagir entre objets

Il y a d'autres catégories, comme

Architecture comme micro-services ou modèle-vue-contrôleur

Et certains patrons de conception plus récent ou moins connus n'ont pas forcément de catégorie (*Hic sunt dracones*).

`https://en.wikipedia.org/wiki/Software_design_pattern`

Patrons que vous avez utilisés

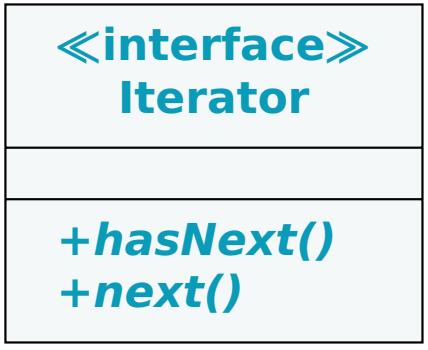
Nous allons commencer par évoquer les patrons que vous avez déjà utilisés sans le savoir.

- iterator
- observer

Problème de l’itération

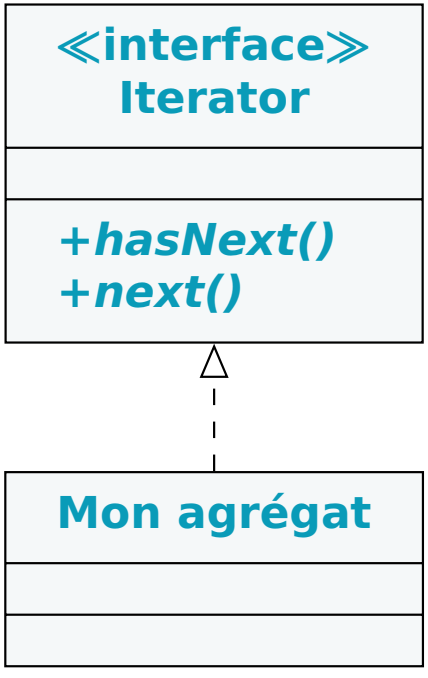
- Quand on considère un objet complexe qui est un agrégat de plusieurs autres objets, un utilisateur veut pouvoir explorer en les traversant un à un sans forcément savoir comment ça fonctionne vraiment.

Notez l'existence en java de



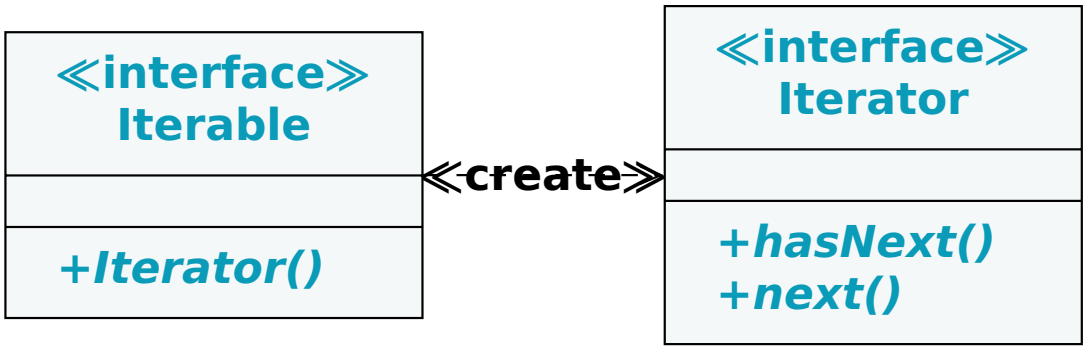
Regardez dans le paquet `Java.util`, l'interface est en fait un générique `iterator<T>`

Objet agrégat implémente iterator



Si la classe gère à la fois la navigation et la structure de données sous-jacente, on ne peut pas permettre plusieurs navigations simultanées sans dupliquer l'objet (qui peut être gros).

Notez l'existence en java de



Regardez dans le paquet `Java.util`, les interfaces sont en fait des génériques `iterator<T>` et `iterable<T>`

Problème de l'itération II

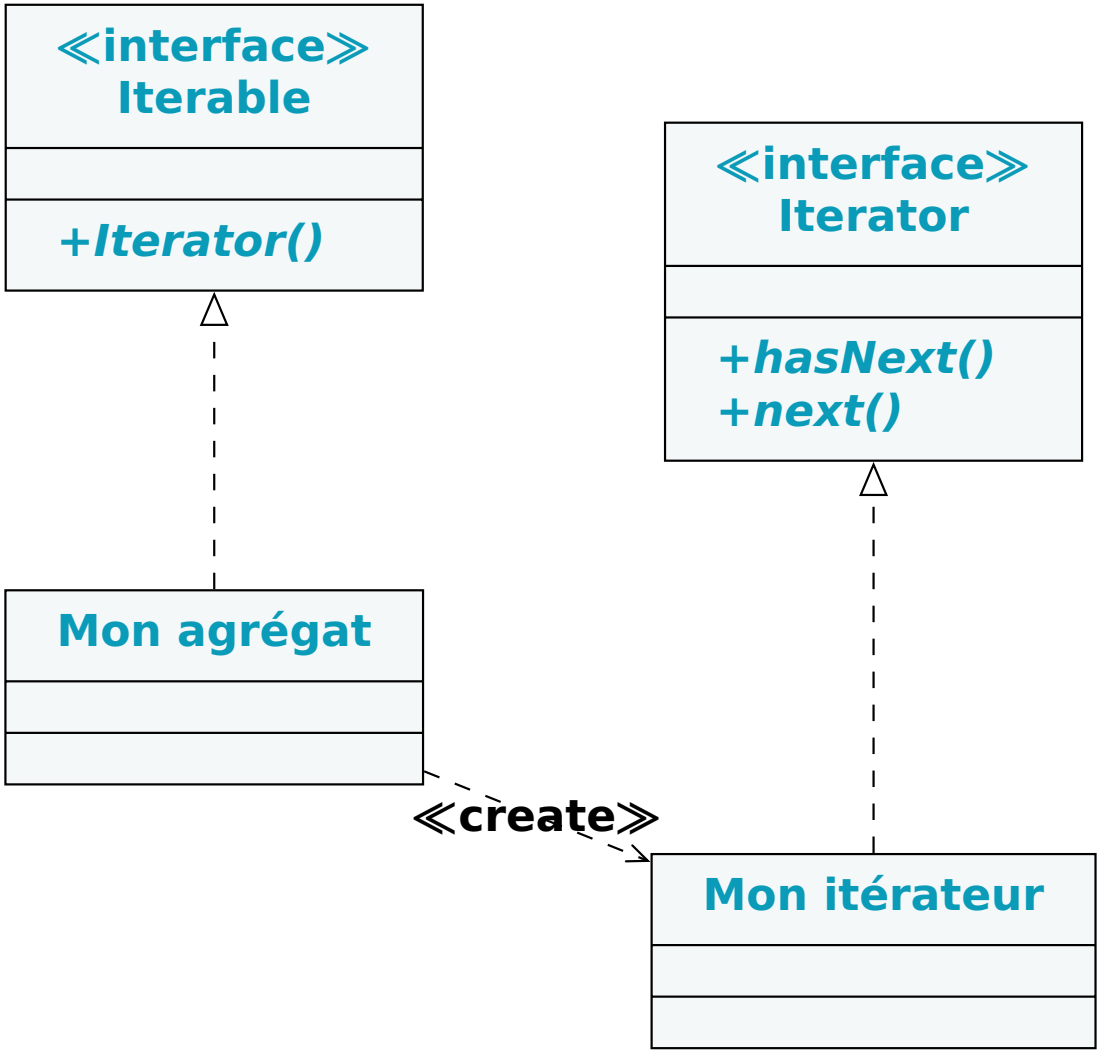
- Quand on considère un objet complexe qui est un agrégat de plusieurs autres objets, un utilisateur veut pouvoir traverser sans forcément savoir comment ça fonctionne vraiment.
- On souhaite pouvoir faire plusieurs navigations simultanément sans pour autant dupliquer les données.

Solution au problème de l'itération

- On découple la partie structure de donnée / sauvegarde de l'objet complexe de la partie navigation / itération.
- On considère une interface Iterable.
- Notre classe d'agrégat implémente cette interface
- Un agrégat retourne un objet iterator à la demande.

Patron de conception Iterator

Objet agrégat implémente iterator



Problème

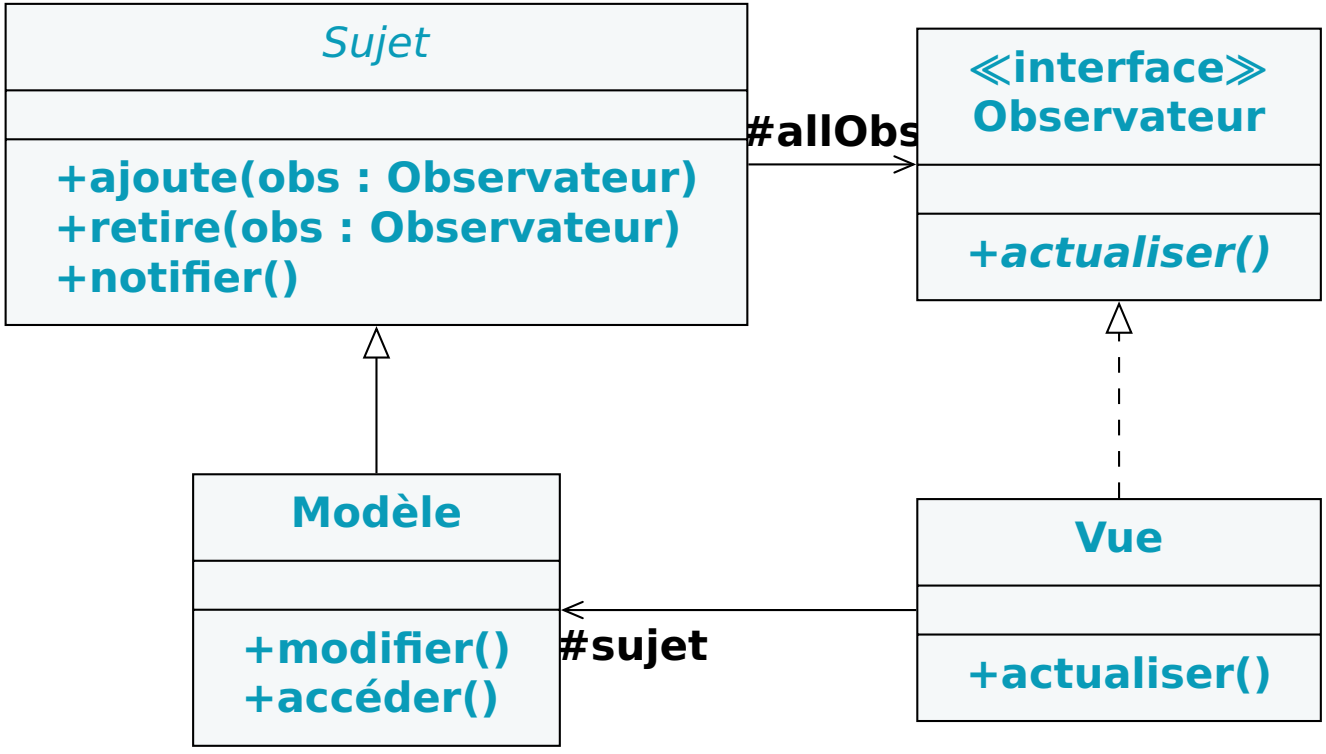
Ce qu'on veut

Propager les modifications du modèle vers les modifications des vues.

Solution

Idée

- Le modèle maintient une liste de ses observateurs.
- À chaque modification de son état ils sont notifiés.



Solution

Idée

- Le modèle maintient une liste de ses observateurs.
- À chaque modification de son état ils sont notifiés.

```
public class Modèle extends Sujet {
  ...
  public modifier() {
    ...
    notifier(
  }
  ...
}
```

```
public class Sujet {
  ...
  public notifier() {
    for (Observateur o: allObs) {
      o.actualiser();
    };
  }
  ...
}
```

Et en Java ?

- Il y a la classe `java.util.Observable` qui permet de modéliser la partie des données observables du modèle.
- Il y a l'interface `java.util.Observer`

Bref, tout comme le patron iterator, le patron Observer est nativement disponible en java.

ce que dit la doc

- interface observer
  - update(Observable o, Object arg)
    This method is called whenever the observed object is changed.
- classe observable
  - addObserver(Observer o)
    Adds an observer to the set of observers for this object [...]
  - deleteObserver(Observer o)
    Deletes an observer [...]
  - notifyObservers()
    If this object has changed [...] then notify all of its observers [...]
  - etc

pour plus de détails voir la doc,

`https://docs.oracle.com/javase/8/docs/api/java/util/Observer.html`

`https://docs.oracle.com/javase/8/docs/api/java/util/Observable.html`

Exemple

```
import java.util.Observable;
import java.util.Observer;

class MessageBoard  extends Observable
{
    public void changeMessage(String message)
    {
        setChanged();
        notifyObservers(message);
    }

}

class Student implements Observer
{
    @Override
    public void update(Observable o, Object arg)
    {
        System.out.println("Message board changed: " + arg);
    }
}

public class MessageBoardTest
{
        public static void main(String[] args)
    {
        MessageBoard board = new MessageBoard();
        Student bob = new Student();
        Student joe = new Student();
        board.addObserver(bob);
        board.addObserver(joe);
        board.changeMessage("More Homework!");
    }
}
```

Autres avatars d'*Observer*

- Dans d'autres langages / contextes, on retrouve la même chose sous le nom de `Listener`.
- C'est typique de la programmation événementielle et des interfaces graphiques.

exemple avec swing, vous utilisez ActionListener

Patrons de comportement

Les deux patrons que l'on vient de voir sont des exemples de patrons de comportement qui sont nativement disponibles en java.

iterator

observer

Il y en a d'autres comme :

Memento *grosso modo* sauver / restaurer un objet. Par exemple par serialisation, ou encore si on veut interagir avec d'autres applis dans des langages différents via un fichier xml ou json ou encore dans une base de données).

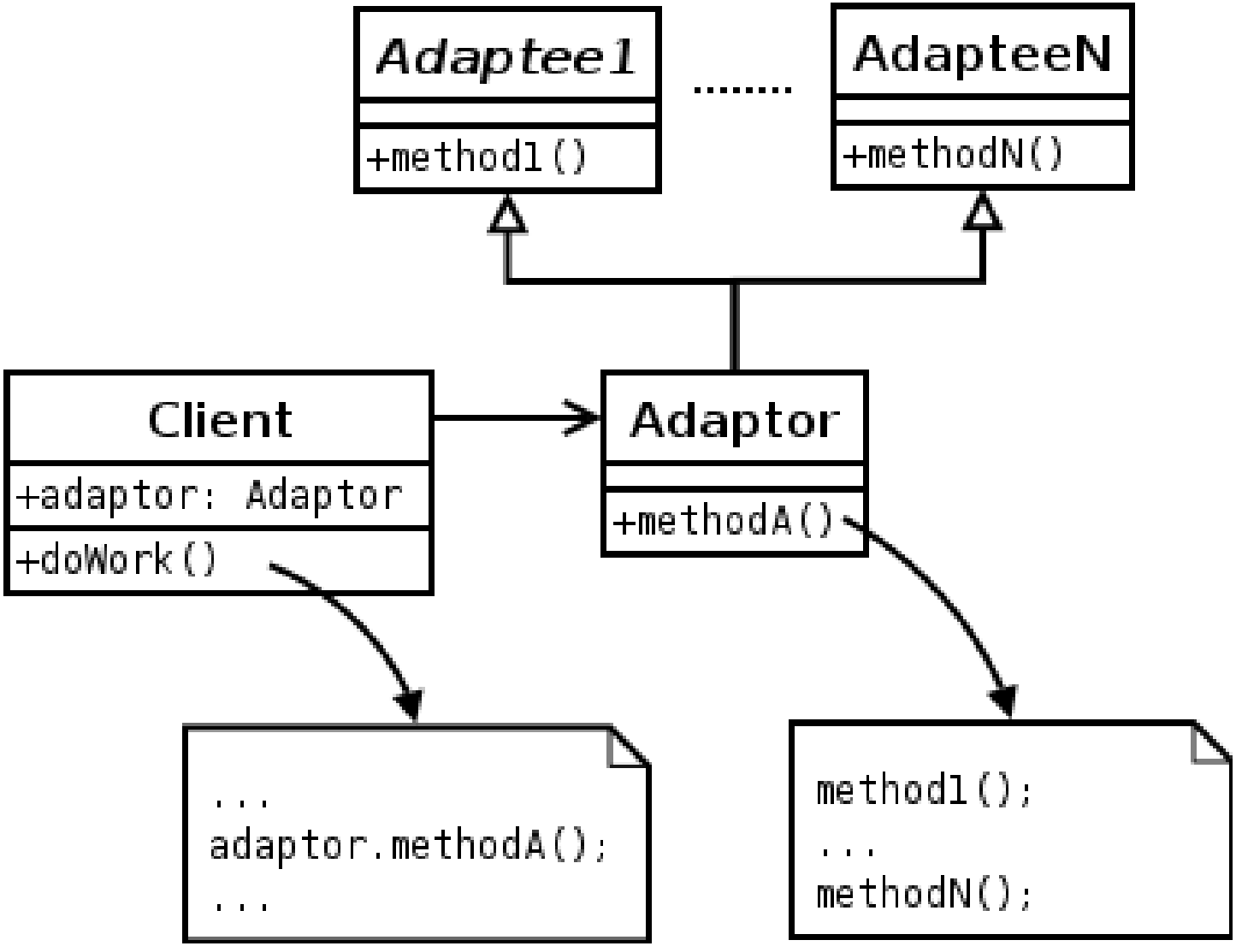
Null object pour éviter d'avoir a tester si on a une référence vers null ou d'avoir des NullPointerException. C'est un objet spécial « qui ne fait rien ».

https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html
https://stackoverflow.com/questions/35689410/ways-to-implement-null-object-pattern

Problème

On dispose d'une nouvelle application qu'on ne peut pas changer (développée par un tiers) qui nécessite une connexion à des données sous une certaine forme.
La base de donnée de l'entreprise ne peut pas être changée facilement (c'est un autre service qui le gère, peut-être est-ce même externalisé).
Votre chef dit : débrouille toi, il faut que ça marche hier

Adapter



By GilHamilton at English Wikipedia [Public domain], via Wikimedia Commons

Remarque

En bases de données, on peut voir une vue comme une sorte d'adapteur.

- En pratique on prétend qu'une vue est comme une table
- Mais une vue (non matérialisée) est donnée par une requête (sur des tables ou même d'autres vues) qu'on va calculer à la volée quand on en aura besoin.

Variantes

Adapter est un motif « de plomberie ». Il y en a d'autres. Tous servent à découpler différentes partie d'un développement pour plus de modularité.

Facade pour faire une API simplifiée qui cachent des choses

Bridge pour découpler deux parties relativement indépendantes

Façade

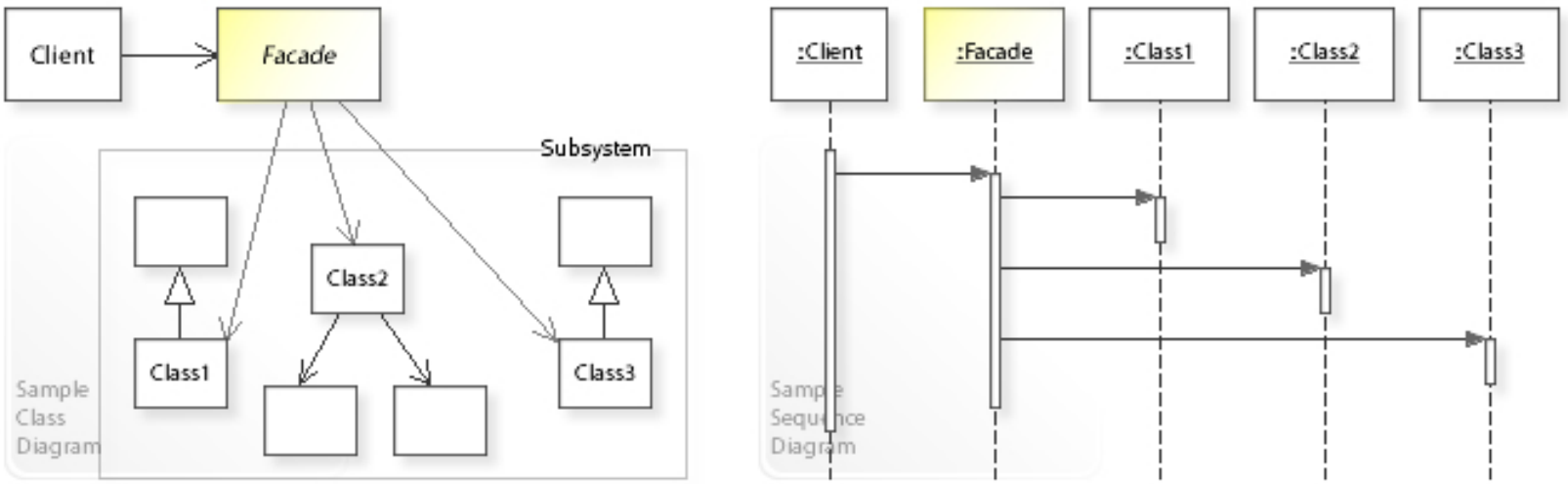
Problème

- Besoin d'une interface qui implémente certaines fonctions.
- Ces fonctions sont implémentées mais par différentes classes/interfaces.
- Ou bien il ne manque pas grand chose pour les implémenter depuis ces fonctions.

Solution

Regrouper dans une classe fournissant l'interface tout ce dont on a besoin en le « piochant »ailleurs.

Facade



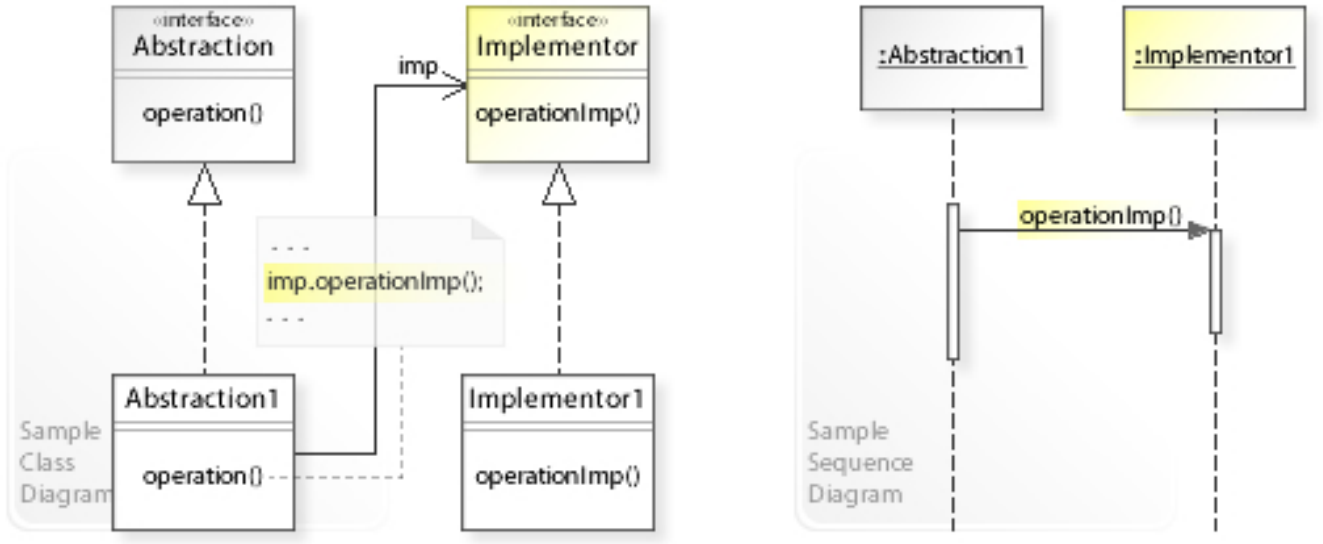
By Vanderjoe [CC BY-SA 4.0], from Wikimedia Commons

Facade pour faire une API simplifiée qui cachent des choses

Exemple

```
/* Complex parts */

class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}

class HardDrive {
    public byte[] read(long lba, int size) { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}

/* Facade */

class ComputerFacade {
    private CPU processor;
    private Memory ram;
    private HardDrive hd;

    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }

    public void start() {
        processor.freeze();
        ram.load(BOs.jpgOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
}

/* Client */

class You {
    public static void main(String[] args) {
        ComputerFacade computer = new ComputerFacade();
        computer.start();
    }
}
```

Bridge



By Vanderjoe [CC BY-SA 4.0], from Wikimedia Commons

Exemple

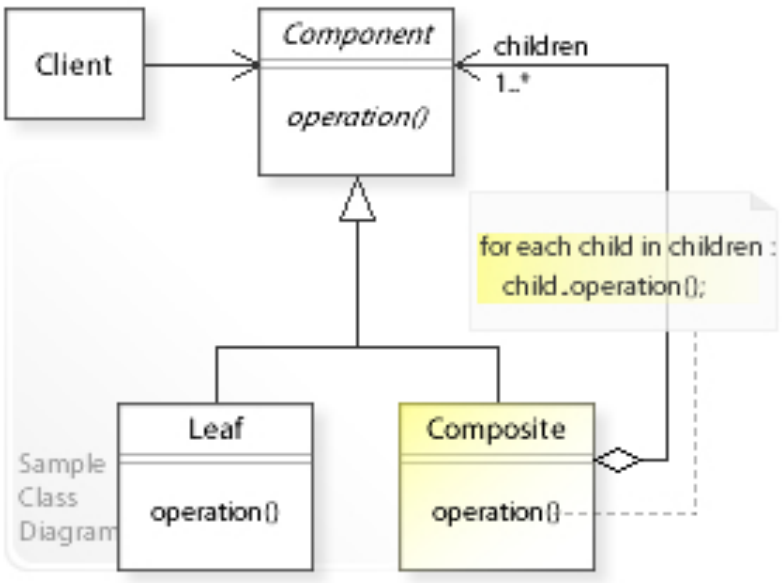
En IHM c'est ce qu'on peut utiliser si on veut découpler l'application de l'interface.
L'interface ne peut pas vraiment savoir si de l'autre côté du bridge il y a une application ou simplement une fausse application qui utilise des données de tests (par exemple dans une base de données).

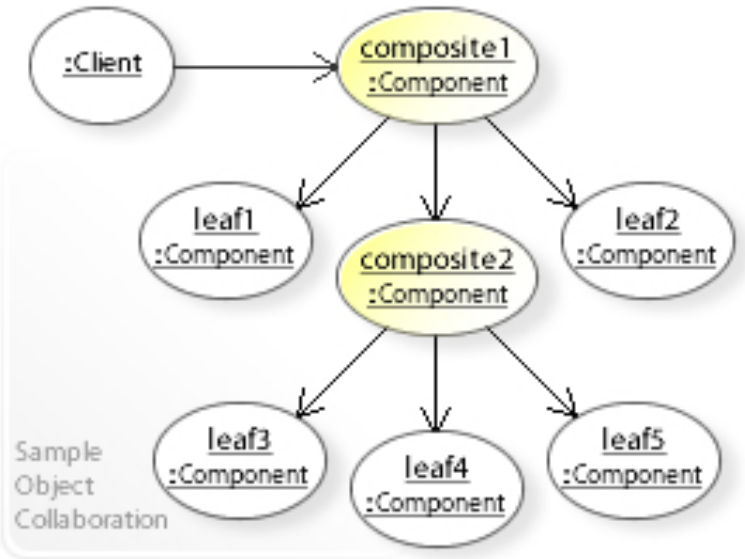
Problème

On veut traiter de manière uniforme les feuilles d'une structure arborescente comme les noeuds internes.
Ceci permet de faciliter l'écriture des méthodes récursives qui calculent des choses des feuilles vers la racine.

- Le cas de base se retrouve dans la classe feuille.
- Le cas de récurrence dans la classe noeud interne.

Idée





By Vanderjoe [CC BY-SA 4.0], from Wikimedia Commons

Exemple

On veut modéliser des fichiers et des répertoires. Sur un élément du système de fichier, on veut connaître le nombre de fichiers qu'il regroupe (en se comptant lui-même).

Solution

Component

```
interface  EverythingIsAFile{
    public int countFiles();
}
```

leaf

```
class DataFile implements EverythingIsAFile {
    public int countFiles(){
        return 1;
    }
}
```

composite

```
import java.util.*;

class Directory implements EverythingIsAFile {
    private List<EverythingIsAFile>  ls = new ArrayList<EverythingIsAFile>();
    public int countFiles() {
        int count = 1 ; // Me the directory, I count for 1.
        for (EverythingIsAFile item : ls) {
            count += item.countFiles();  // Delegation
        }
        return count;
    }
    public void mv(EverythingIsAFile item) {
        ls.add(item);
    }
    public void rm(EverythingIsAFile item) {
        ls.remove(item);
    }
}
```

La prochaine fois

D’autres motifs comme

singleton

factory