



Initiation aux Design Patterns



François ANDRE



Objectif des design patterns

Les **design patterns** (ou *modèles de conception*) proposent des **réponses efficaces à base de modélisation objets à des problèmes fréquemment rencontrés.**

Ainsi, chaque design pattern sera définie par

- Son **nom**
- Le **problème qu'il cherche à résoudre**
- Le diagramme UML proposé pour répondre au problème

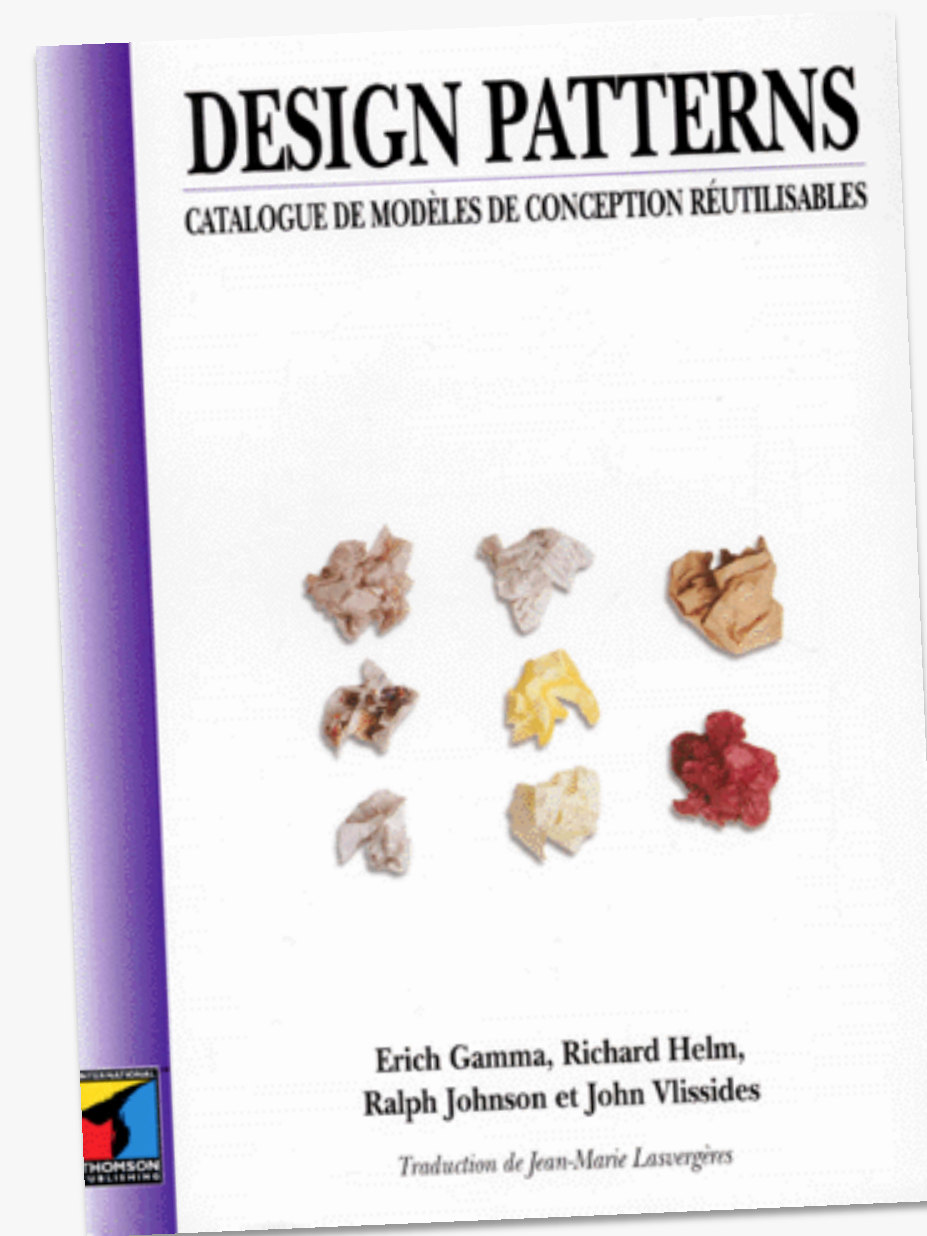
Objectif des design patterns

En informatique, les 23 premiers design patterns ont été mis en évidence dans le livre

Design patterns. Catalogue des modèles de conception réutilisables

D'autres modèles de conception ont été développés ultérieurement par d'autres auteurs. Citons par exemple

- L'inversion de contrôle
- Le modèle-vue-contrôleur
- Les Enterprise Integration Patterns
- ...



-
- On trouve sur Internet différentes implémentations de ces différents modèles de conception pour la plupart des langages objets.
 - On trouve également la notion d'**antipattern** qui correspond à la fausse bonne idée, c'est à dire l'erreur commune que l'on peut rencontrer en programmation objet qui semblait être pourtant une solution efficace au premier abord.
 - L'étude des deux concepts - patterns et antipatterns - s'avère utile et fructueuse. Entre autre il permet aux développeurs et aux concepteurs de mettre en place un vocabulaire complémentaire dans le domaine de la modélisation.



Design Patterns abordés

■ Nous allons étudier les Design Patterns suivants:

- Singleton
- Iterator
- Observer
- Decorator
- Visitor

Creational

Behavioral

Behavioral

Structural

Behavioral

Introduction



Problème à résoudre

Restreindre le nombre d'instance d'une classe.
Généralement à une seule instance.



Exemple

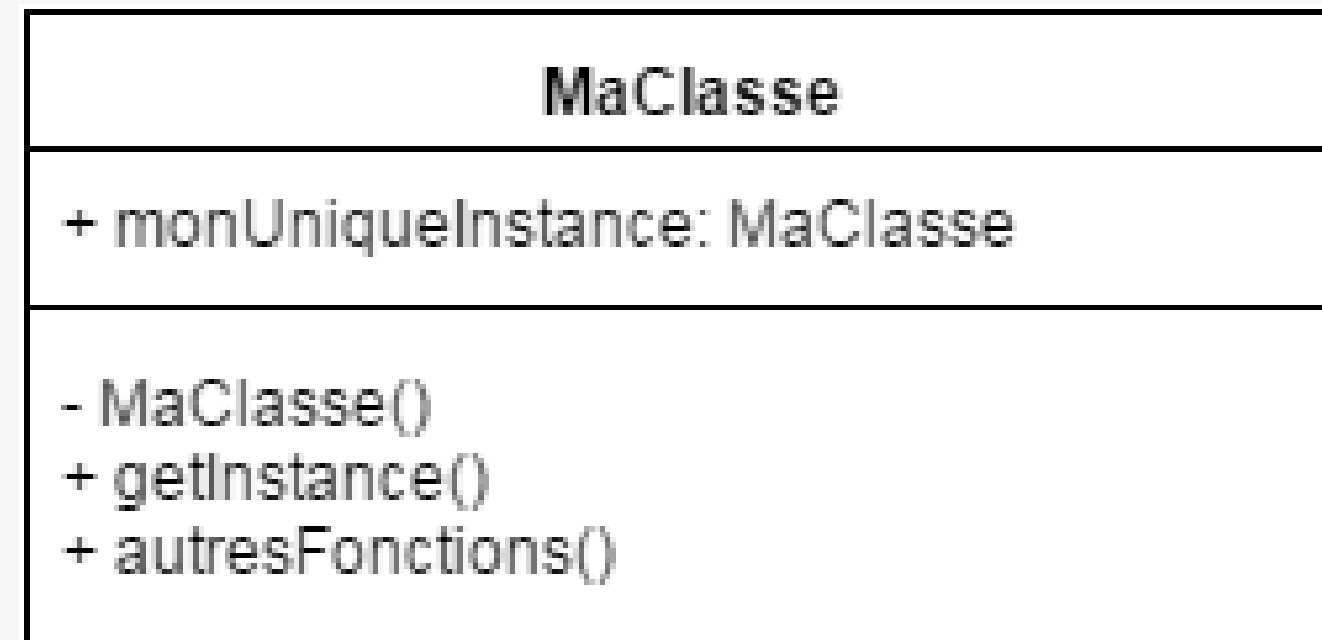
Les bases de données ont souvent un nombre maximal de connexions autorisées.

Afin d'éviter d'épuiser les connexions disponibles - mais aussi pour des raisons de performances - il peut être préférable de laisser ouverte une connexion vers une base de données et de faire en sorte que chaque appel à la base de données utilise cette connexion.



Implémentation

Diagramme UML





Implémentation

Le Singleton

```
package designpatterns.singleton;
public class Connexion {
    private static Connexion connexionInstance;
    private Connexion() { }
    public static Connexion getInstance() {
        if (connexionInstance == null) {
            connexionInstance = new Connexion();
        }
        return connexionInstance;
    }
    public void executeSQL(String request) { // Do something }
}
```



Implémentation

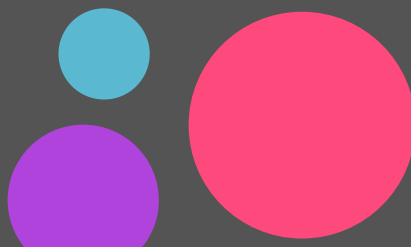
Utilisation

```
public static void main(String[] args) {  
    Connexion instance = Connexion.getInstance();  
    instance.executeQuery("select * from MA_TABLE");  
}
```

-
- Dans un Singleton, l'usage veut que la méthode permettant de récupérer l'instance unique soit appelée *getInstance*. Parfois le nom de la classe comporte également le suffixe "Singleton". Exemple: `Calendar.getInstance()`;
 - Il est important de noter l'absence de constructeur public pour empêcher une instantiation directe de la classe
 - L'implémentation ci-dessus est très simple. Elle souffre de certains défauts, notamment la non prise en compte des accès concurrents (multithreading)



Iterator





Problème à résoudre

Parcourir un ensemble de données.



Exemple

Liste des températures

Supposons qu'une classe Thermomètre stocke les différentes valeurs de la température toute les heures dans une chaîne de caractères où chaque température est séparée par une virgule :

ex: 21,22,23,25,22,20 ...



Exemple

Approche brutale

Si une classe, Graphique, veut tracer un graphique des températures une première approche est que Graphique récupère la chaîne des températures et la décompose en se basant sur la virgule afin de pouvoir effectuer le tracé.



Exemple

Limitations de l'approche brutale

- L'approche brutale est limitée car elle va à l'encontre de l'encapsulation des données en programmation objet. En effet, avec cette approche il est nécessaire pour la (ou les) classes utilisatrices de connaître la manière interne dont Thermomètre stocke ses données.
- Ainsi, le traitement de Graphique ne va plus fonctionner si Thermomètre change le caractère de séparation ou s'il change sa manière de stocker (utilisation d'un tableau).
- De plus si une autre classe veut également récupérer les données, on peut s'attendre à ce qu'elle mette en place le même genre de code que Graphique et nous aurons sûrement des doublons.



Exemple

Iterator

Pour être le plus efficace possible, il faut que Thermomètre ne permettent plus l'accès direct à son mécanisme de données mais via un objet qui aura la seule responsabilité d'égrener les données. Cet objet va implémenter l'interface Iterator.



Implémentation

L'interface Iterator est la suivante :

Iterator <<interface>>
+ hasNext(): boolean + next(): Object

Elle est présente nativement dans Java (<https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html>) et est le moyen privilégié de parcourir les objets de type *List* ou *Set*.

Cette version de l'Iterator utilise la notion de *generics* non encore étudiée. Elle rajoute également la méthode *remove()* qui est optionnelle.



Implémentation

La classe Thermomètre

```
package designpatterns.iterator;
import java.util.Iterator;

public class Thermometre {

    private String values = "";
    private final static String SEPARATOR = ",";

    public Thermometre() {

    }

    public Thermometre(int fakeValueNumbers) {
        for (int i = 0; i < fakeValueNumbers; i++) {
            addValue(randInt(20, 30));
        }
        System.out.println("The values are " + values);
    }

    public void addValue(int value) {
        if (values.length() != 0) {
            values = values + SEPARATOR;
        }
        values = values + value;
    }
}
```

...



Implémentation

La classe Thermomètre

```
...  
  
    public static int randInt(int min, int max) {  
        return min + (int) (Math.random() * ((max - min) + 1));  
    }  
  
    public Iterator getChronologicalIterator() {  
        String[] aux = values.split(SEPARATOR);  
        return new ArrayListIterator(aux);  
    }  
  
    public Iterator getAntiChronologicalIterator() {  
        String[] aux = values.split(SEPARATOR);  
        // Reverse array  
  
        for (int i = 0; i < aux.length / 2; i++) {  
            String temp = aux[i];  
            aux[i] = aux[aux.length - i - 1];  
            aux[aux.length - i - 1] = temp;  
        }  
        return new ArrayListIterator(aux);  
    }  
}
```



Implémentation

Iterator

```
package designpatterns.iterator;
import java.util.Iterator;

public class ArrayIterator implements Iterator {

    private String[] values;
    private int index = 0;

    public ArrayIterator(String[] values) {this.values = values;}

    @Override
    public boolean hasNext() {return (index < values.length);}

    @Override
    public Object next() {
        String result = values[index];
        index++;
        return result;
    }

    @Override
    public void remove() {
        // TODO Auto-generated method stub
    }

}
```

Remarque : normalement, cette classe devrait être une classe interne de la classe Thermomètre mais je l'ai extraite pour simplifier la lecture.



Implémentation

Utilisation

```
package designpatterns.iterator;

import java.util.Iterator;

public class Test {

    public static void main(String[] args) {
        Thermometre thermometre = new Thermometre(10);
        System.out.println("Chronological");
        Iterator iterator = thermometre.getChronologicalIterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
        System.out.println("Anti-Chronological");
        iterator = thermometre.getAntiChronologicalIterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Remarque : on voit bien l'intérêt de l'Iterator qui conserve un traitement identique dans la classe utilisatrice, *Test*, alors que le parcours des données peut être totalement différent et à la charge de la classe connaissant les données et la manière la plus efficace de les traverser.



Problème à résoudre

Communiquer avec des objets lointains ou inconnus...



Problème à résoudre

A et B sont séparés par de multiples objets

Lorsqu'un objet A invoque une méthode sur un objet B, on dit qu'il y a dialogue, qu'un message est envoyé de A à B.

Toutefois la communication classique peut être inadaptée dans deux cas que l'on rencontre fréquemment



Problème à résoudre

A et B sont séparés par de multiples objets

On peut envisager que dans notre application A ne possède pas une référence directe sur B et que les objets A et B sont séparés par de multiples objets: C, D, E, F.

Pour que A puisse dialoguer avec B, le message va devoir passer par C, D, E, F avant d'atteindre B.

Une des conséquences est que l'on doit alors ajouter dans C, D, E et F des méthodes uniquement pour faire passer la communication entre A et B sans que C, D, E et F ne soient forcément concernées par la nature du message. Un peu comme si les tuyaux reliant la chaudière à la baignoire passaient au beau milieu du salon...



Problème à résoudre

A et B ne se connaissent pas

Il se peut qu'au moment de son codage, A ne connaissent pas les différents objets B avec lesquels il devra communiquer.



Exemple

Notre application comporte une page de connexion via laquelle un utilisateur administrateur indique son login et son mot de passe.

En cas de réussite, plusieurs parties de notre application doivent se mettre à jour:

- La barre de menu doit afficher la partie dédiée aux administrateurs
- La barre d'en-tête doit afficher le nom de l'utilisateur



Exemple

Approche brutale

Notre objet `connectionPage` pourrait avoir une référence sur l'instance de l'objet `MenuBar` et une autre vers l'objet `HeaderBar`/

A l'issue d'une connexion réussie, `connectionPage` pourrait appeler explicitement des méthodes de ces objets:

- `menuBar.displayAdministrationMenu()`
- `header.displayUserName();`



Exemple

Limitations de l'approche brutale

L'approche brutale est limitée car elle suppose que notre objet `connectionPage`

- possède une référence vers tous les objets qui sont concernés par l'événement *un utilisateur s'est connecté* ce qui peut mener à du code spaghetti.
- sache quel est la manière dont chacun de ces objets réagissent à l'événement (l'un affiche un menu, l'autre un nom..)

De plus, si un nouvel objet veut réagir à la connexion d'un utilisateur, il est nécessaire de modifier la classe `ConnectionPage` ce qui n'est pas forcément possible, celle-ci pouvant par exemple venir d'une librairie tierce.



Exemple

L'Observer

Le design pattern Observer ressemble à la notion de l'abonnement à une revue:
Lorsqu'un client veut recevoir une revue,

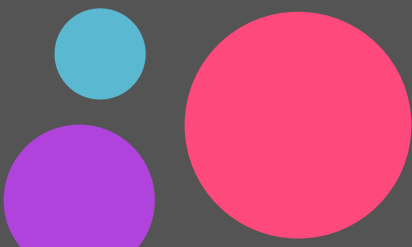
1. il s'inscrit au préalable auprès de l'éditeur de la revue.
2. chaque fois qu'une revue est publiée, l'éditeur l'envoie dans la boîte aux lettres de tous les abonnés
3. Les abonnés décident de ce qu'ils doivent faire de la revue (la lire de suite, l'amener au bureau,...)

On voit bien que ce mécanisme d'abonnement apporte :

- Il définit une interface, *l'abonné*, qui a une méthode *deposeBoiteAuxLettres*.
- L'éditeur ne connaît de l'abonné que cette méthode *deposeBoiteAuxLettres* et ne sait pas ce que vont faire les abonnés de la revue



Decorator





Problème à résoudre

Attacher dynamiquement de nouvelles responsabilités à un objet.



Decorator

Lorsqu'on a un objet de base et que l'on souhaite lui ajouter de nouveaux comportements le premier réflexe du développeur est l'héritage.

Toutefois, dans certains cas :

- Cette approche n'est pas possible (ex: classes ou méthodes *final*)
- Cette approche n'est pas la meilleure.

Le design pattern *Decorator* est ainsi parfois une astucieuse alternative.



Exemple

Supposons que notre objectif est d'imprimer des factures. Nous avons un objet de base représentant le corps d'une facture.

Nous savons que nous allons devoir ajouter plusieurs comportements à notre impression de facture. En effet, dans certains cas nous allons ajouter un en-tête, dans d'autre un pied de page, parfois une carte, parfois plusieurs de ces composants.



Exemple

Approche brutale

On pourrait commencer par faire

- une classe *FactureBasique*,
- une classe *FactureAvecEntete*,
- une classe *FactureAvecPiedDePage*,
- une classe *FactureAvecEnteteEtPiedDePage*
- ...



Exemple

Limite de l'approche brutale

On voit bien que **la stratégie est mauvaise** car elle aborde la complexité du problème de la mauvaise manière.

En effet, **les classes vont se multiplier** pour couvrir tous les cas et l'ajout d'un nouveau comportement obligerait à créer un très grand nombre de classe.



Exemple

Decorator

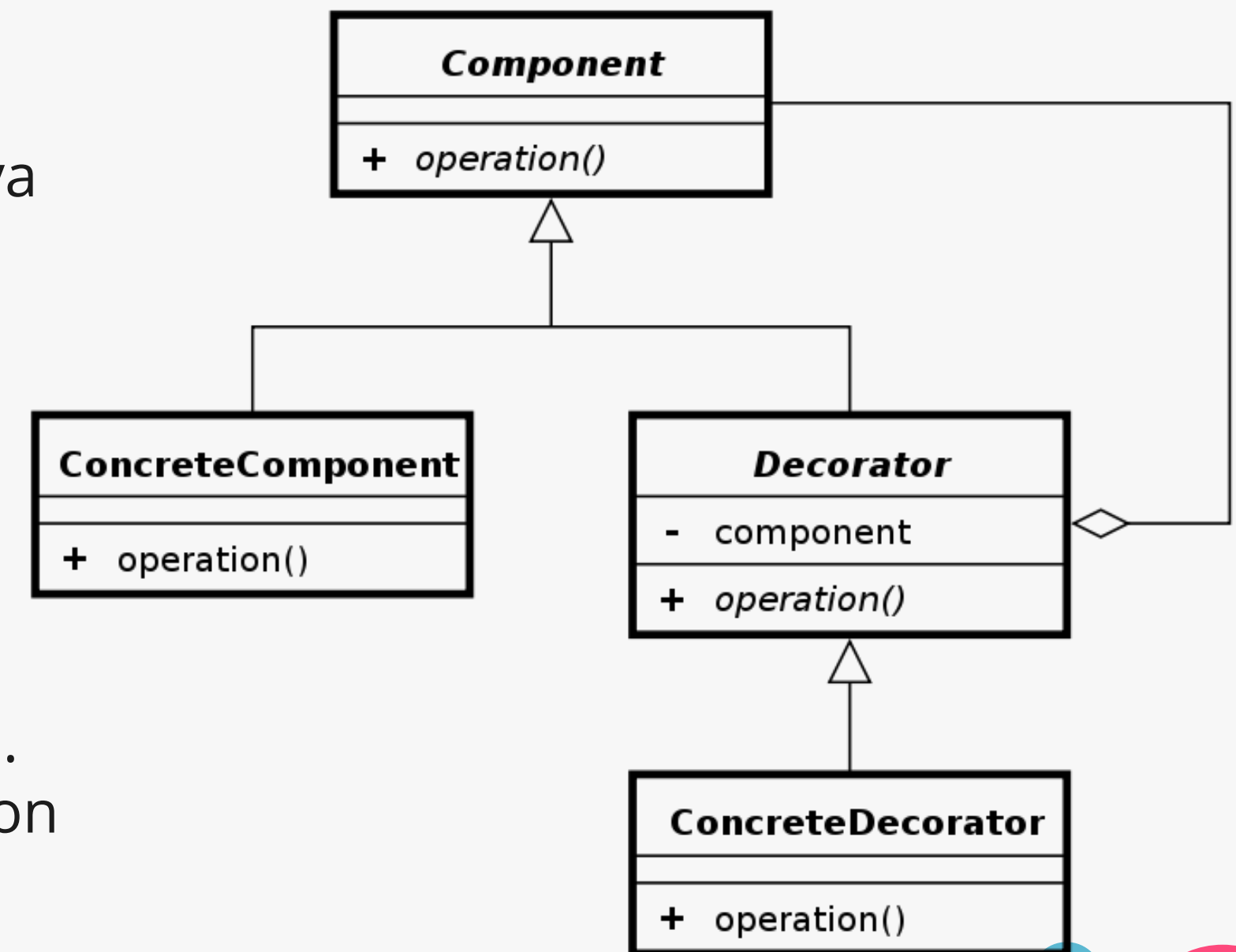
Pour être le plus efficace, au lieu d'utiliser un ajout de comportement de manière verticale avec l'héritage, le *Decorator* va proposer une **démarche plus horizontale - basée sur la composition** - où les différents composants vont s'emboîter de manière sélective en ajoutant chacun un comportement.

Exemple

Decorator

Les points importants à noter pour comprendre le mécanisme sont les suivants :

- On définit une interface générique (*Component*) qui va définir le comportement de base de notre objet.
- Il y a au moins une implémentation directe de cette interface (*ConcreteComponent*)
- Les autres classes vont décorer la classe *ConcreteComponent* en définissant une chaîne de *Component*. Le *Decorator* va stocker la chaîne des précédents *Component* et chaque *ConcreteDecorator* va ajouter un traitement à celui retourné par cette chaîne. Chaque *ConcreteDecorator* va recevoir la chaîne dans son constructeur.





Exemple

Interface Facture

```
package designpatterns.decorator;
```

```
public interface Facture {
```

```
    String print();
```

```
}
```



Exemple

Classe DefaultFacture

```
package designpatterns.decorator;

public class DefaultFacture implements Facture {

    @Override
    public String print() {
        StringBuilder sb = new StringBuilder();
        sb.append("-----\n");
        sb.append("Voici ma facture \n");
        sb.append("-----\n");
        return sb.toString();
    }
}
```



Exemple

Classe AbstractDecoratedFacture

```
package designpatterns.decorator;
```

```
public abstract class AbstractDecoratedFacture implements Facture {  
    protected Facture previousFacture;
```

```
    public AbstractDecoratedFacture(Facture previousFacture) {  
        this.previousFacture = previousFacture;  
    }  
}
```

```
}
```

Le *Decorator* est abstrait et implémente le *Component* et introduit le constructeur prenant un *Component* comme argument .



Exemple

Exemples de ConcreteDecorator

Nous allons introduire 3 comportements - en-tête, pied de page et carte.
Il nous suffira d'une classe par comportement.

```
package designpatterns.decorator;

public class FactureAvecEntete extends AbstractDecoratedFacture {

    public FactureAvecEntete(Facture previousFacture) {
        super(previousFacture);
    }

    @Override
    public String print() {
        StringBuilder sb = new StringBuilder();
        sb.append("-----\n");
        sb.append("Voici l'en-tête \n");
        sb.append("-----\n");
        sb.append(previousFacture.print());
        return sb.toString();
    }
}
```



Exemple

Exemples de ConcreteDecorator

```
package designpatterns.decorator;

public class FactureAvecPiedDePage extends AbstractDecoratedFacture {

    public FactureAvecPiedDePage(Facture previousFacture) {
        super(previousFacture);
    }

    @Override
    public String print() {
        StringBuilder sb = new StringBuilder();
        sb.append(previousFacture.print());
        sb.append("-----\n");
        sb.append("Voici le pied de page \n");
        sb.append("-----\n");
        return sb.toString();
    }
}
```



Exemple

Exemples de ConcreteDecorator

```
package designpatterns.decorator;

public class FactureAvecCarte extends AbstractDecoratedFacture {

    public FactureAvecCarte(Facture previousFacture) {
        super(previousFacture);
    }

    @Override
    public String print() {
        StringBuilder sb = new StringBuilder();
        sb.append(previousFacture.print());
        sb.append("-----\n");
        sb.append("Voici la carte \n");
        sb.append("-----\n");
        return sb.toString();
    }
}
```

Exemple

Exemples de ConcreteDecorator

```
package designpatterns.decorator;
```

```
public class Test {
```

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

```
        Facture defaultFacture = new DefaultFacture();
```

```
        Facture factureAvecEntete = new FactureAvecEntete(new DefaultFacture());
```

```
        Facture factureAvecPiedDePage = new FactureAvecPiedDePage(new DefaultFacture());
```

```
        Facture factureAvecEnteteEtPiedDePage = new FactureAvecPiedDePage(new FactureAvecEntete(new DefaultFacture()));
```

```
        Facture factureAvecCarteEtPiedDePage = new FactureAvecPiedDePage(new FactureAvecCarte(new DefaultFacture()));
```

```
        System.out.println("Facture par défaut :");
```

```
        System.out.println("");
```

```
        System.out.println(defaultFacture.print());
```

```
        System.out.println("");
```

```
        System.out.println("Facture avec en-tête :");
```

```
        System.out.println("");
```

```
        System.out.println(factureAvecEntete.print());
```

```
        System.out.println("");
```

```
        System.out.println("Facture avec pied de page :");
```

```
        System.out.println("");
```

```
        System.out.println(factureAvecPiedDePage.print());
```

```
        System.out.println("");
```

```
        System.out.println("Facture avec en-tête et pied de page :");
```

```
        System.out.println("");
```

```
        System.out.println(factureAvecEnteteEtPiedDePage.print());
```

```
        System.out.println("");
```

```
        System.out.println(factureAvecCarteEtPiedDePage.print());
```

```
        System.out.println("");
```

```
        System.out.println(factureAvecEnteteEtPiedDePage.print());
```

```
        System.out.println("");
```

```
    }
```

```
}
```

La philosophie centrale du *Decorator* qui apparait clairement lors de l'instanciation :

```
Facture factureAvecEnteteEtPiedDePage = new FactureAvecPiedDePage(new FactureAvecEntete(new DefaultFacture()));
```

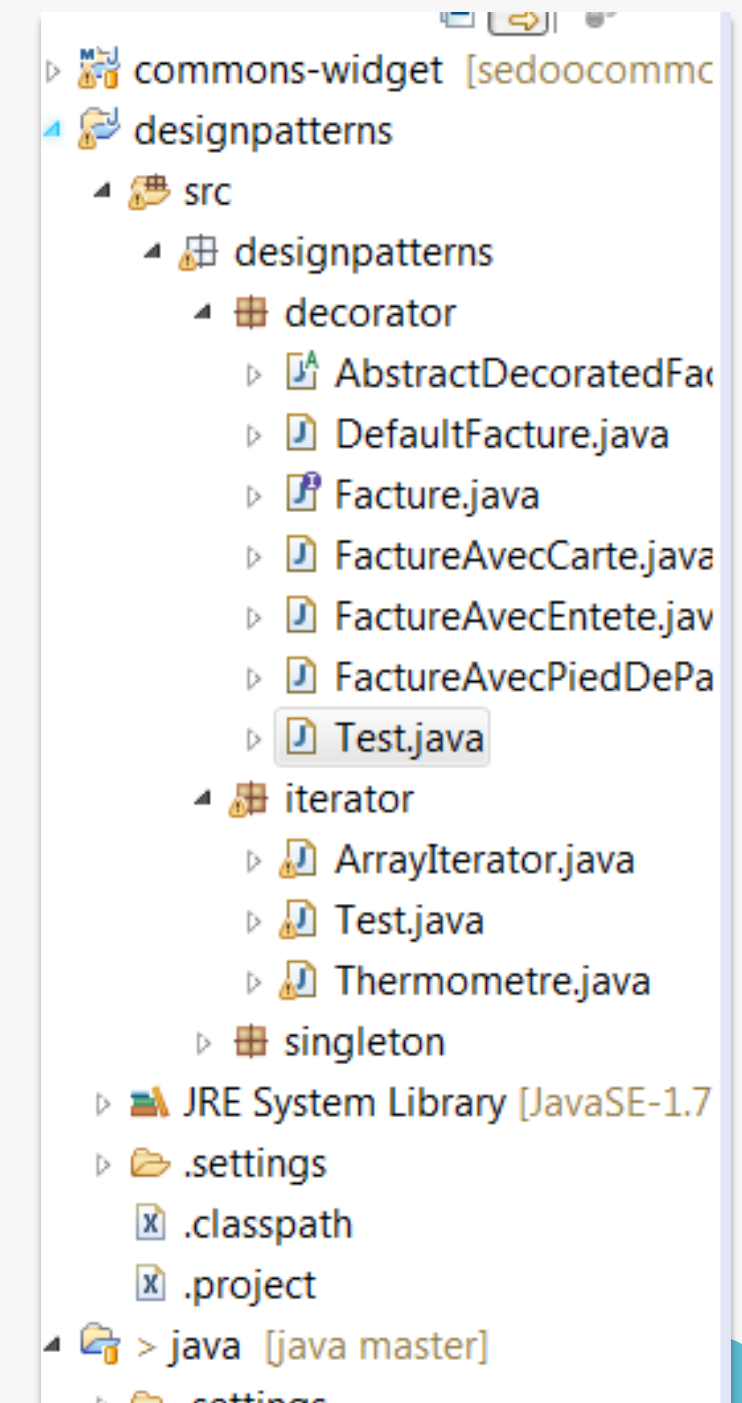
On voit bien l'enchaînement des *ConcreteDecorator* qui se termine par le *ConcreteComponent*.

Conclusion

Dans Eclipse, le concept de décorateur est très utilisé. L'exemple de l'arbre des projets est très représentatif.

Chaque fichier est associé à une icône - qui est notre *ConcreteComponent* - et chaque module complémentaire (Git, Java, M2E,...) va ajouter des *ConcreteDecorator* qui vont chacun ajouter une surcouche à l'icône de base: une croix sur les fichiers ne compilant pas, un "J" et un "M" sur le projet Java mavenisé...

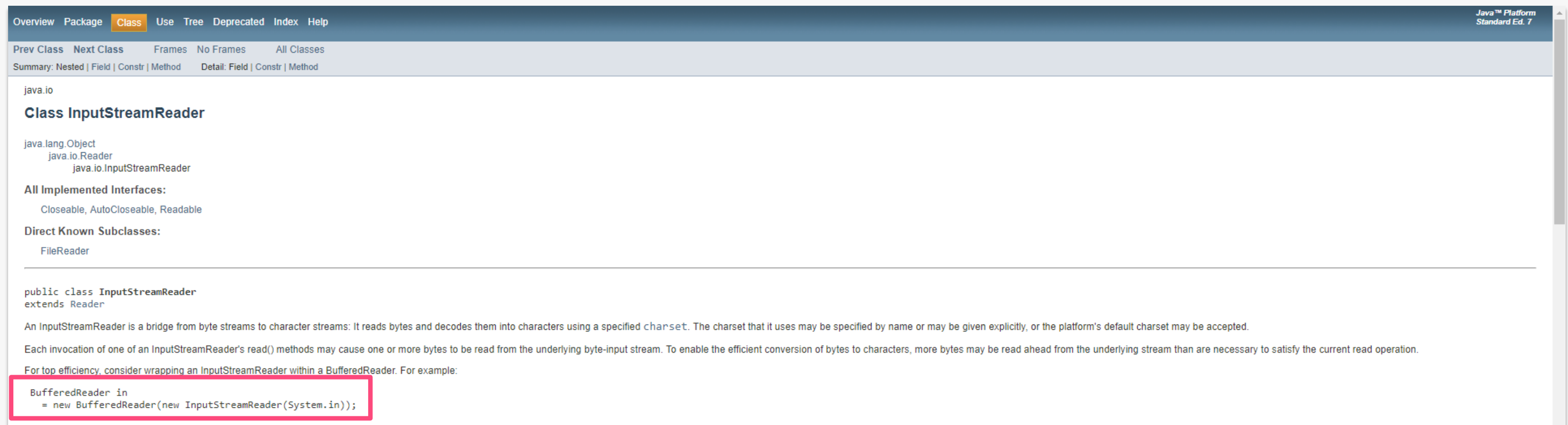
Les décorateurs peuvent même paramétrés dans les préférences d'Eclipse...dans les rubriques *Decorations*.



Conclusion

Dans le JDK, une série de classes assez importante est basée sur le concept de *Decorator*.

Mais pour l'étudiant débutant n'a pas souvent connaissance de ce concept au moment où il aborde ces classes ce qui les rend difficilement compréhensibles...



The screenshot shows the Java Platform Standard Ed. 7 documentation for the `java.io.InputStreamReader` class. The page includes navigation tabs (Overview, Package, Class, Use, Tree, Deprecated, Index, Help) and a sidebar with links to the previous class, next class, frames, no frames, and all classes. The main content area displays the class hierarchy, implemented interfaces, and direct known subclasses. A code snippet is highlighted with a red box, showing the creation of a `BufferedReader` instance wrapping an `InputStreamReader`.

```
public class InputStreamReader
    extends Reader

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

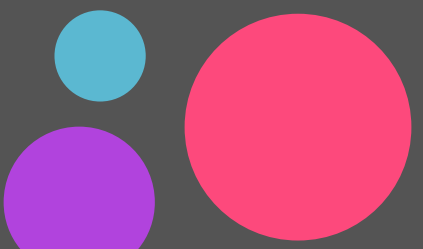
Each invocation of one of an InputStreamReader's read() methods may cause one or more bytes to be read from the underlying byte-input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation.

For top efficiency, consider wrapping an InputStreamReader within a BufferedReader. For example:

BufferedReader in
    = new BufferedReader(new InputStreamReader(System.in));
```



Visitor





Problème à résoudre

Différentier une structure et les algorithmes qui l'utilisent



Problème à résoudre

Nous avons :

- une structure composée de beaucoup d'éléments de types variés
- plusieurs traitements différents à effectuer sur cette structure

Exemple:

- Une **Académie** est composée d'**établissements scolaires** eux-mêmes composées de **classes** regroupant des **élèves**.
- Nous voulons, compter les élèves, lister les établissements, lister les classes en sur-effectif.



Limite de l'approche classique

En général les traitements sont au sein des objets de la structure.

Par exemple, pour le traitement *compter les élèves*, on va rajouter une méthode *compterEleves()* dans l'Académie qui va appeler *compterEleves()* dans chacun de ses Etablissements, qui va appeler *compterEleves()* dans chacune des ses Classes.



Limite de l'approche classique

Les inconvénients de cette approche sont les suivants:

- Pour chaque nouveau traitement il faut **ajouter des nouvelles méthodes dans un grand nombre d'objets**.
 - Certains objets ne sont pas directement concernés par les traitements mais se voient **alourdir par des méthodes *passe-plats***.
 - Par exemple Academie et Etablissement dans le comptage des élèves.
 - Ces traitements font, en général, transiter, d'appel en appel, des objets contenant les résultats intermédiaires
 - Par exemple le compte intermédiaire des élèves
- Avec cette approche, les développeurs hésitent à mettre en place de nouveaux traitements, préférant essayer de détourner des traitements existants.

L'objectif du Visitor est le suivant:

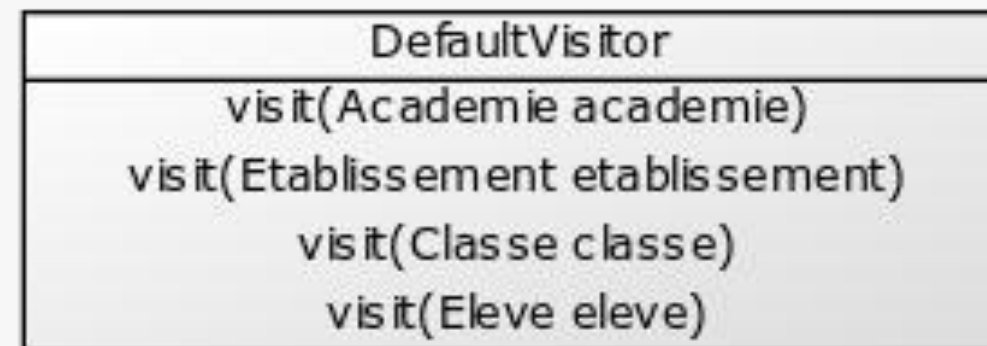
- Ne laisser dans les objets métiers uniquement la connaissance du parcours de la structure
→ Par exemple, une académie sait qu'être parcourue signifie parcourir tous ses établissements, un établissement sait qu'être parcourue signifie parcourir toutes ses classes...
- Mettre chaque traitement dans une classe unique.
→ Par exemple `VisiteurNombreEleves`, `VisiteurListerEleves`,...



Le Visitor

Le visiteur par défaut

Pour ce faire, on a défini un visiteur par défaut qui contient une méthode *visit()* pour **chaque type d'objet de la structure**.



Dans ce visiteur par défaut, chaque méthode ne fait rien.

Une méthode *visit()* correspondra à la visite d'un nœud de l'arborescence par le traitement.



Le Visitor

Un visiteur concret

Chaque traitement sera un visiteur qui héritera de ce visiteur par défaut. Il surchargera les seules méthodes qui le concernent:

```
public class VisiteurCompteEleve extends DefaultVisitor {  
  
    int nombreEleves = 0;  
  
    @Override  
    public void visit(final Eleve eleve) {  
        nombreEleves++;  
    }  
  
    public int getNombreEleves(final Academie academie) {  
        academie.accept(this);  
        return nombreEleves;  
    }  
}
```

Dans le cas du comptage des élèves, seule la visite des nœuds Eleve est pertinente.

On voit également que les variables stockant les résultats intermédiaires (nombreEleves) ne circulent plus d'objets en objets.



Le Visitor

Implémentation du parcours de la structure

Du côté des nœuds métiers, la seule méthode à ajouter est celle qui restreint le parcours de la structure. Par convention elle s'appelle *accept*. Elle va appeler la méthode *visit* pour le nœud courant et appeler *accept* sur ses enfants. Cette méthode ne sera modifiée que si la structure est modifiée

```
public class Etablissement {  
  
    ...  
  
    public void accept(final DefaultVisitor visiteur) {  
        visiteur.visit(this);  
        for (Classe classe : classes) {  
            classe.accept(visiteur);  
        }  
    }  
}
```

Merci

Des questions?

