

---

# TDLOG séance 2 - Design patterns

Xavier Clerc – [xavier.clerc@enpc.fr](mailto:xavier.clerc@enpc.fr)

29 septembre 2025

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>UML (rappel 1PROG)</b>	<b>2</b>
2.1	Brève description . . . . .	2
2.2	Types de diagrammes . . . . .	2
<b>3</b>	<b>UML : diagrammes de classes (rappel 1PROG)</b>	<b>3</b>
3.1	Éléments de base . . . . .	3
3.2	Regroupement . . . . .	4
3.3	Agrégation et composition . . . . .	4
3.4	Diagramme de packages . . . . .	5
<b>4</b>	<b><i>Design Patterns</i></b>	<b>6</b>
4.1	Brève description . . . . .	6
4.2	Catégories de <i>patterns</i> . . . . .	6
<b>5</b>	<b><i>Design pattern : singleton (creational)</i></b>	<b>7</b>
5.1	Description . . . . .	7
5.2	Exemple . . . . .	7
5.3	Discussion . . . . .	7
<b>6</b>	<b><i>Design pattern : adaptateur (structural)</i></b>	<b>8</b>
6.1	Description . . . . .	8
6.2	Exemple . . . . .	8
6.3	Discussion . . . . .	9
<b>7</b>	<b><i>Design pattern : itérateur (behavioural)</i></b>	<b>10</b>
7.1	Description . . . . .	10
7.2	Exemple . . . . .	10
7.3	Discussion . . . . .	11
<b>8</b>	<b><i>Design pattern : visiteur (behavioural)</i></b>	<b>11</b>
8.1	Description . . . . .	11
8.2	Exemple . . . . .	11

# 1 Introduction

Après avoir, lors de la séance précédente, rappelé les bases de la programmation orientée objet objet, nous allons présenter lors de cette séance la notion de *design pattern* et l'illustrer à l'aide de quelques exemples. Nous allons commencer par quelques rappels sur UML (sections 2 et 3), avant de présenter les objectifs des *design patterns* (section 4), qui sont des "recettes" qui permettent de résoudre des problèmes communs. Les sections 5 à 8 présentent des *design patterns* communs : singleton, adaptateur, itérateur et visiteur.

## 2 UML (rappel 1PROG)

### 2.1 Brève description

UML (pour *Unified Modeling Language*) est, comme son nom l'indique, un langage de modélisation. Cependant, contrairement à ce que son nom suggère, *unified* signifie seulement qu'il prétend fournir un cadre unique pour les développements orientés objets et non pas qu'il permet d'exprimer de manière universelle comment s'architecture un système logiciel.

UML est un langage essentiellement graphique, que ses avocats qualifient de semi-formel et ses détracteurs d'informel. En tout état de cause, il ne s'agit pas d'un langage suffisamment riche et précis pour décrire complètement un logiciel. Il ne doit donc être envisagé que comme permettant de donner un aperçu rapide (et le plus souvent superficiel) de l'architecture et/ou du fonctionnement d'un logiciel.

Son principal intérêt est de pouvoir exprimer de manière claire les relations entre différents éléments d'un système :

- les parties du logiciel, selon différentes granularités et angles de vue ;
- les processus mis en œuvre lors de l'exécution du logiciel ;
- les interactions entre les différentes parties du logiciel ;
- les interactions entre le logiciel et ses utilisateurs.

UML est un standard de fait dans une large part de l'industrie informatique, en général dans les milieux où l'informatique est un outil et non une fin. Cela s'explique par le fait que la représentation graphique rend plus accessible l'architecture du logiciel à des non-spécialistes et peut être utilisée comme langage de *médiation* entre développeurs et non-développeurs.

### 2.2 Types de diagrammes

Les diagrammes UML se répartissent dans deux catégories :

- les diagrammes structurels ;
- les diagrammes comportementaux.

Les diagrammes structurels sont ceux qui définissent les éléments constitutifs de l'architecture d'un logiciel. S'agissant d'un langage de modélisation pour les systèmes objets, ils s'intéressent donc à la définition des différentes classes. La définition des classes va un peu au-delà de leur

seule signature : UML s'intéresse également aux relations entre les classes, ainsi qu'à leur répartition en paquetages (*i. e.* modules, dans le cas de Python).

Les diagrammes comportementaux sont ceux qui définissent les interactions qui se produisent lors de l'exécution d'un logiciel. Il s'agit à la fois des interactions se produisant à l'intérieur du logiciel ainsi que des interactions avec son environnement, qu'il soit composé d'autres logiciels ou d'utilisateurs.

### 3 UML : diagrammes de classes (rappel 1PROG)

Le type de diagramme UML probablement le plus utilisé est le diagramme de classes. Il permet de représenter graphiquement les signatures des classes d'un logiciel, ou plus vraisemblablement d'une sous-partie de ce dernier. En effet, dans la plupart des logiciels de taille *industrielle*, le nombre de classes est tel que le diagramme de classes nécessiterait plusieurs gigantesques affiches pour être complet.

#### 3.1 Éléments de base

Pour décrire graphiquement une classe, UML place sa définition dans un rectangle divisé en trois parties horizontales. De haut en bas, on trouve :

- un premier sous-rectangle contenant le nom de la classe ;
- un deuxième sous-rectangle contenant la liste des attributs de la classe ;
- un troisième sous-rectangle contenant la liste des méthodes de la classe.

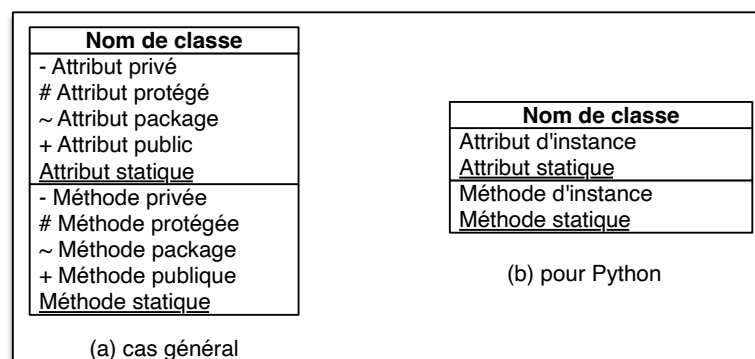


FIGURE 3.1 – Notation pour une classe.

La figure Figure 3.1 donne des exemples de telles représentations. On y voit que les attributs et méthodes sont préfixés par un caractère qui dépend de leur visibilité :

- + pour la visibilité "publique" (accessible partout) ;
- # pour la visibilité "protégée" (accessible seulement dans la classe et les sous-classes) ;
- - pour la visibilité "privée" (accessible seulement dans la classe) ;
- ~ pour la visibilité "package" (accessible seulement dans les classes du package).

En outre, un attribut (respectivement une méthode) est souligné pour indiquer qu'il s'agit d'un attribut (respectivement d'une méthode) statique.

Ainsi que le montre la figure Figure 3.1, la notion de visibilité n'ayant pas lieu d'être en Python, on peut raisonnablement décider de ne pas faire figurer les préfixes et se contenter d'utiliser le formatage souligné lorsque nécessaire.

Une fois que l'on dispose d'une notation pour les classes seules, il est nécessaire de définir une notation permettant de traduire graphiquement l'héritage. Cela se fait à l'aide de flèches pointant depuis la classe fille vers la ou les classes parentes. La figure [Figure 3.2](#) montre comment représenter les cas d'héritage simple et multiple dans un diagramme de classes.

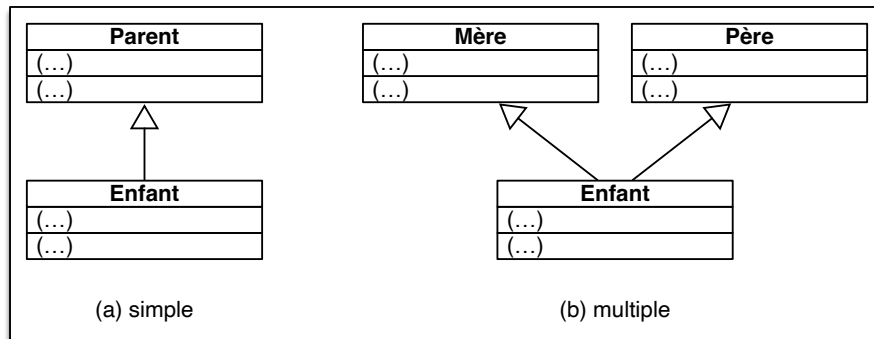


FIGURE 3.2 – Notation pour l'héritage.

## 3.2 Regroupement

À ce stade, on dispose d'une notation permettant de représenter sur un schéma différentes classes avec leurs lignées d'héritage. Comme la plupart des langages permettent de regrouper de telles classes en unités logiques (modules en Python, packages en Java, *namespaces* en C++, *etc*), cette information peut également être codée en UML. Cela se fait à l'aide d'un pictogramme représentant un dossier, sur lequel on fait figurer le nom du module dans le cas de Python, la terminologie UML étant package. La figure [Figure 3.3](#) montre l'usage d'un tel pictogramme.

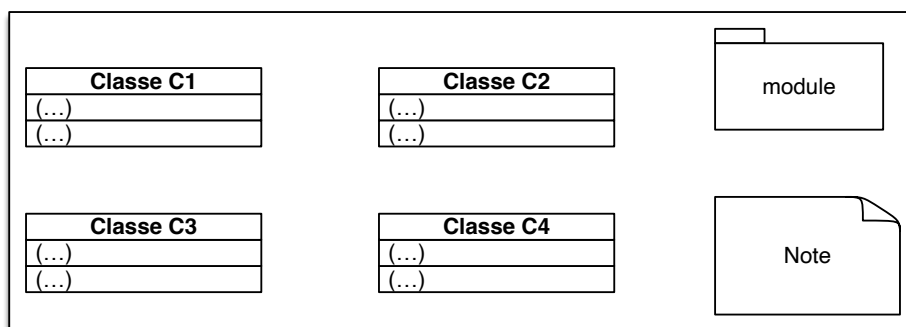


FIGURE 3.3 – Notation de package (ou module).

La figure [Figure 3.3](#) présente également un autre pictogramme que nous n'avons pas encore rencontré : celui ressemblant à une feuille de papier ou à un *post-it*. Ce pictogramme est simplement utilisé pour écrire dans le diagramme de classe un commentaire en langue naturelle.

## 3.3 Agrégation et composition

Outre la relation d'héritage, UML définit deux autres relations entre classes : l'agrégation et la composition. La figure [Figure 3.4](#) montre les notations correspondant à ces deux notations. Ces deux relations correspondent à l'utilisation d'une classe par une autre. Dans les deux cas, la classe du côté du losange est celle qui possède une référence sur une ou des instances de l'autre classe.

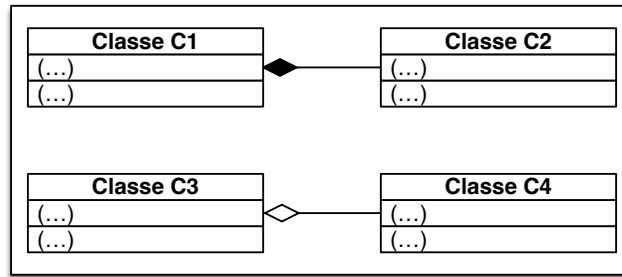


FIGURE 3.4 – Notations de composition (haut) et d'agrégation (bas).

La différence entre agrégation et composition est très ténue et tient essentiellement au *cycle de vie des objets*<sup>1</sup>. En pratique, dans un langage disposant d'un *garbage collector* pour la gestion de la mémoire, il n'y a pas de différence sérieuse entre les deux notions.

Enfin, il est d'usage de faire figurer des *cardinalités* de chaque côté de la relation entre deux classes. Les cardinalités permettent d'indiquer combien d'objets de l'autre classe peuvent être concernés par la relation. Dans l'exemple de la figure Figure 3.4, si *C1* est en relation avec une liste de *C2*, on fera figurer du côté de *C2* la cardinalité  $0..n$ <sup>2</sup> (cardinaux minimal et maximal). Du côté de *C1*, la cardinalité dépendra du nombre de listes d'instances de *C1* dans lesquelles une instance de *C2* peut se retrouver. On peut ainsi avoir communément les cardinalités suivantes :

- $0..n$  si ce nombre peut être quelconque ;
- $1..n$  si une instance de *C2* est en lien avec au minimum une instance de *C1* ;
- $0..1$  si une instance de *C2* est en lien avec au maximum une instance de *C1* ;
- $1..1$  si une instance de *C2* est en lien avec une et une seule instance de *C1*.

### 3.4 Diagramme de packages

Une fois que l'on a dessiné des diagrammes de classes pour les différents modules d'une application, il peut être utile de disposer d'une vue de plus haut niveau. Cela se fait typiquement à l'aide d'un diagramme de packages, tel que celui présenté à la figure Figure 3.5.

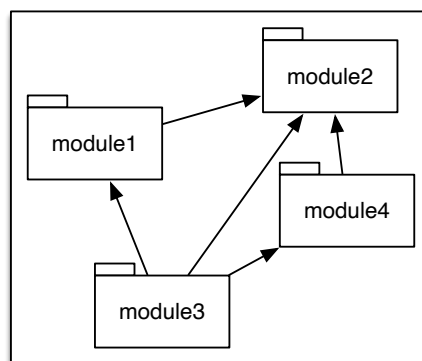


FIGURE 3.5 – Dépendances entre modules.

Dans un diagramme de packages, on fait figurer sous forme de dossiers les différents packages de l'application et l'on trace des flèches qui représentent les dépendances entre packages. Ces

1. *i. e.* quand ils sont alloués et désalloués.  
2. ou  $1..n$  si l'on veut signifier une liste qui ne peut être vide.

dépendances sont orientées, et une flèche d'un package  $p1$  vers un package  $p2$  signifie que  $p1$  dépend de  $p2$ . Typiquement, cela signifie qu'au moins une classe de  $p1$  utilise une classe de  $p2$ .

La topologie dessinée par le diagramme de packages peut permettre d'identifier des formes d'architectures que nous verrons lors d'une séance ultérieure. Par exemple, on peut voir émerger de cette représentation une architecture en couches (voir session sur l'architecture).

## 4 Design Patterns

### 4.1 Brève description

Les *design patterns* (ou patrons de conception) constituent un ensemble de "recettes" qui peuvent être appliquées pour résoudre des problèmes communs. Comme on le verra, il s'agit dans la plupart des cas de spécifier un agencement de classes et de décrire la dynamique entre elles. En particulier, il ne s'agit en général pas de solutions "algorithmiques", de telle sorte qu'elles ne peuvent pas faire partie d'une librairie, mais doivent être appliquées *mutatis mutandis* au problème que l'on souhaite résoudre.

Un *pattern* se définit typiquement par les éléments suivants :

- un nom ;
- la description d'un problème commun ;
- une solution à ce problème, sous la forme notamment d'une hiérarchie de classes ;
- les conséquences de l'application du *pattern*, notamment en termes de compromis selon diverses dimensions (*p. ex.* performances contre lisibilité).

S'ils se veulent génériques, les *patterns* s'incarnent dans un langage de programmation donné, de telle sorte que tel ou tel *pattern* est plus ou moins nécessaire dans tel ou tel langage. En particulier, un certain nombre de *patterns* sont moins importants dans un langage dont le système de type est fondé sur le *duck typing* comme Python. En effet, certains *patterns* visent à contourner les rigidités d'un système de types statique.

Les *patterns*, et leur recensement, ont pour objectifs explicites de promouvoir la flexibilité, la modularité et la réutilisation. Ils sont également très utiles, plus implicitement, en termes de documentation, communication et partage d'expérience entre développeurs. En effet, plutôt que de rédiger quelques phrases décrivant les différentes classes, il peut s'avérer tout aussi complet de simplement mentionner le ou les *patterns* appliqués dans le développement de telle ou telle partie d'une application.

### 4.2 Catégories de *patterns*

Les *design patterns* se répartissent dans trois catégories :

- *creational* : description de la création d'instances ;
- *structural* : description de la composition d'instances ;
- *behavioural* : description de l'interaction entre instances.

Les sections qui suivent proposent une brève présentation de quelques *patterns* de base : singleton (*creational*), adaptateur (*structural*), itérateur et visiteur (tous deux *behavioural*).

## 5 Design pattern : singleton (creational)

### 5.1 Description

L'objectif du *pattern* singleton est de garantir qu'il n'existe qu'une seule instance d'une classe donnée <sup>1</sup>. Cela permet de garantir la bonne utilisation d'une ressource unique, par exemple le microphone de l'ordinateur.

### 5.2 Exemple

Le code [Code 5.1](#) montre un exemple d'implémentation du *pattern* en Java. On voit que le constructeur est déclaré privé afin qu'il ne soit pas callable depuis l'extérieur de la classe et qu'une méthode statique, `getInstance` permet de renvoyer l'unique instance (en la créant à la demande). `getInstance` est ainsi le seul moyen d'obtenir une instance depuis l'extérieur de la classe, garantissant l'unicité.

Code 5.1 – Singleton en Java.

```
1 public class MySingleton {
2
3     // ... attribute declarations
4
5     private MySingleton() { // constructor
6         // ... attribute initializations
7     }
8
9     // Java's null is similar to Python's None
10    private static MySingleton instance = null;
11
12    public static MySingleton getInstance() {
13        if (instance == null) {
14            instance = new MySingleton(); // instance creation
15        }
16        return instance;
17    }
18
19 }
```

### 5.3 Discussion

Cet exemple est volontairement donné en *Java*, car il n'est pas possible le traduire "directement" en Python, notamment parce qu'il n'y pas véritablement de notion de constructeur privé. Par ailleurs, si l'objectif est de garantir l'existence d'une et une seule entité, on voit bien que le concept de module Python est un remplacement intéressant. Ainsi plutôt que de définir une classe pouvant être instanciée à de multiples reprises, on pourra déclarer un module avec des fonctions. Une approche fondée sur les classes est cependant plus flexible : il n'est par exemple

---

1. Une généralisation naturelle du *pattern* peut fixer une borne supérieure différente.

pas possible en Python de redéfinir l'affichage d'un module (méthode `__str__`), de maîtriser le moment de création de l'instance, *etc.*

## 6 Design pattern : adaptateur (*structural*)

### 6.1 Description

L'objectif du *pattern* adaptateur est de résoudre des incompatibilités de signatures entre des instances. On peut avoir à manipuler un même concept, par exemple un graphe, à l'aide de deux bibliothèques. Or, il est très possible que les deux bibliothèques utilisent chacune leurs propres classes. On appelle un tel problème un problème d'*impédance*, car il n'est pas possible de "brancher" les deux bibliothèques sans adaptation.

### 6.2 Exemple

Le code [Code 6.1](#) présente un adaptateur ; on y trouve trois éléments :

- le protocole ou interface (implicite en Python) `Target` avec les méthodes que l'on souhaite pouvoir utiliser sur une instance ;
- la classe `Adaptee` qui est celle des instances dont on dispose originellement ;
- la classe `Adapter` qui permet d'utiliser une instance d'`Adaptee` comme une instance de `Target`.

Comme l'indique l'implémentation de la méthode `Adapter.methi`, l'adaptation consiste essentiellement à "re-router" l'appel vers une autre méthode de la classe `Adaptee`.

Code 6.1 – Adaptateur en Python.

```
1 class Adaptee:
2     def alt_meth0(self, ...) -> ...:
3         # ...
4         return...
5
6     def alt_methm(self, ...) -> ...:
7         # ...
8         return...
9
10
11 class Target(Protocol):
12     def meth0(self, ...) -> ...: ...
13     def methn(self, ...) -> ...: ...
14
15
16 class Adapter: # implements Target
17     def __init__(self, adaptee: Adaptee) -> None:
18         self._adaptee = adaptee
19
20     def meth0(self, ...) -> ...:
21         return self._adaptee.alt_meth0(123)
```

(suite sur la page suivante)

```

22  def methn(self, ...) -> ...:
23      return self._adaptee.alt_methm(123)
24

```

La figure Figure 6.1 présente le *pattern* sous la forme d'un diagramme de classes UML.

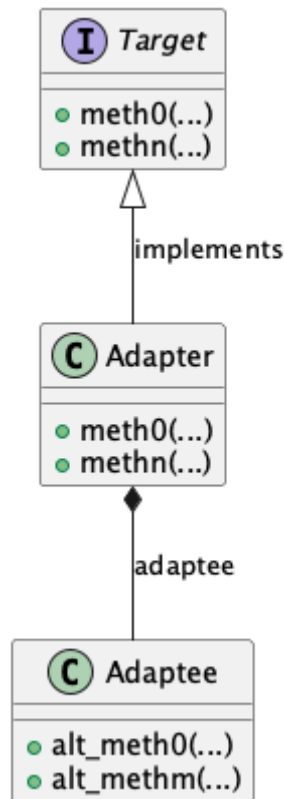


FIGURE 6.1 – Pattern adaptateur.

### 6.3 Discussion

La notion d'adaptateur est utile, car elle permet de centraliser le code de *conversion* entre `Adaptee` et `Target`, plutôt que de le voir réparti en différents points du code. En outre, on voit qu'il n'y a pas ici de *conversion* au sens strict du terme : on conserve l'instance originelle et on propose un moyen alternatif de la manipuler. Si cela n'a pas nécessairement de sens pour des types simples (entiers ou chaînes de caractères par exemple), on peut sans peine imaginer que cela est utile pour des types plus complexes et/ou mutables. Bien entendu, l'indirection introduite par les instances de `Adapter` représente un certain coût à l'exécution, en échange d'un gain en place (ne pas stocker deux versions d'une même donnée à la suite d'une véritable conversion). Il s'agit d'un compromis temps/espace classique en informatique.

## 7 Design pattern : itérateur (*behavioural*)

### 7.1 Description

L'objectif du *pattern* itérateur est de fournir une abstraction pour le parcours des éléments d'une collection telle qu'un tableau, une liste ou un ensemble. Il en existe de très nombreuses variantes, selon le mode exact de parcours, la possibilité de revenir en arrière, de supprimer des éléments pendant le parcours, *etc.*

### 7.2 Exemple

Le code [Code 7.1](#) présente un itérateur Python<sup>1</sup> qui permet de parcourir une suite d'entier, de *start* (inclus) à *stop* (exclu) par incrément *step*.

Code 7.1 – Itérateur en Python.

```
1 class xrange:
2     def __init__(self, start: int, stop: int, step: int = 1) -> None:
3         assert step > 0, f"invalid step ({step})"
4         self._start = start
5         self._stop = stop
6         self._step = step
7
8     def __iter__(self) -> Iterator[int]:
9         self._current = self._start
10        return self
11
12    def __next__(self) -> int:
13        if self._current < self._stop:
14            res = self._current
15            self._current += self._step
16            return res
17        else:
18            raise StopIteration
```

En toute généralité un itérateur Python met en jeu deux classes :

- la collection à parcourir, qui doit définir une méthode `__iter__` qui est chargée de renvoyer un itérateur sur les éléments de la collection ;
- l'itérateur lui-même qui doit définir une méthode `__next__` qui est chargée, au fil des appels successifs, de renvoyer les différents éléments de la collection (et de lever l'exception `StopIteration` lorsque tous les éléments ont déjà été renvoyés).

Le code [Code 7.1](#) est un cas simplifié où l'on utilise la même instance pour la collection (intervalle implicite d'entiers) et l'itérateur. Il permet par exemple d'écrire les appels suivants :

```
for i in xrange(0, 10):
    print(i)
```

(suite sur la page suivante)

1. <https://docs.python.org/3/library/stdtypes.html#iterator-types>

```
for i in xrange(0, 10, 3):
    print(i)
```

La figure Figure 7.1 présente le *pattern* sous la forme d'un diagramme de classes UML.



FIGURE 7.1 – Pattern itérateur.

## 7.3 Discussion

L'intérêt majeur de l'itérateur est qu'il rend le code *consommant* les valeurs de la collection indépendant de la représentation de celle-ci. Ainsi, on peut remplacer un ensemble par une liste sans avoir à modifier le code de parcours. Ce *pattern* est si important en pratique que de nombreux langages, dont Python, l'ont intégré au langage lui-même plutôt que de le reléguer dans une librairie ou extension. Lorsque Python exécute une boucle `for item in coll: body`, il appelle la méthode `__iter__` sur `coll` pour obtenir l'itérateur et avant chaque exécution de `body` appelle `__next__` sur l'itérateur pour obtenir la nouvelle valeur de `item`. La boucle prend fin lorsqu'un appel à `__next__` lève l'exception `StopIteration`.

# 8 Design pattern : visiteur (behavioural)

## 8.1 Description

Le *pattern* visiteur est essentiellement une généralisation du *pattern* itérateur où ce dernier peut être vu comme le parcours d'une collection "plate" où tous les éléments ont un même niveau dans la structure. Le *pattern* visiteur permet de parcourir une structure plus complexe, avec des éléments ayant différentes "formes" ou "niveau" dans la structure (typiquement un "arbre" plutôt qu'une "séquence").

## 8.2 Exemple

Le code Code 8.1 montre l'application du *pattern* à un arbre binaire dont les feuilles sont étiquetées. Les deux types d'éléments, feuille (classe `Leaf`) et nœud interne (classe `Node`) doivent implémenter l'interface `Element`, c'est-à-dire une méthode `accept` qui prend en paramètre une instance de `Visitor`. L'interface `Visitor` définit deux méthodes, `visit_leaf` et `visit_node`, qui doivent être respectivement appelées lorsque l'on visite une feuille et un nœud interne. C'est ce que font les implémentations de `accept` dans les classes `Leaf` et `Node`.

L'implémentation dans la classe Node est en outre responsable de l'exploration des sous-arbres gauche et droit.

Code 8.1 – Visiteur en Python.

```
1 class Visitor(Protocol):
2     def visit_node(self, node: Node) -> None: ...
3     def visit_leaf(self, leaf: Leaf) -> None: ...
4
5
6 class Element(Protocol):
7     def accept(self, visitor: Visitor) -> None: ...
8
9
10 class Leaf: # implements Element
11     def __init__(self, value: int) -> None:
12         self._value = value
13
14     @property
15     def value(self) -> int:
16         return self._value
17
18     def accept(self, visitor: Visitor) -> None:
19         visitor.visit_leaf(self)
20
21
22 class Node: # implements Element
23     def __init__(self, left: Element, right: Element) -> None:
24         self._left = left
25         self._right = right
26
27     def accept(self, visitor: Visitor) -> None:
28         self._left.accept(visitor)
29         self._right.accept(visitor)
30         visitor.visit_node(self)
```

On voit que ces définitions fournissent une abstraction pour une opération sur une hiérarchie d'objets, ici notre arbre binaire. Les différentes instanciations de l'interface du visiteur vont correspondre aux différentes opérations. Le code [Code 8.2](#) montre l'implémentation et l'utilisation de deux opérations. La première, `PrintVisitor`, consiste simplement à afficher les valeurs sur les feuilles tandis que la seconde, `SumVisitor` calcule la somme des valeurs en les accumulant dans un attribut du visiteur.

Code 8.2 – Utilisation de visiteurs en Python.

```
1 class PrintVisitor: # implements Visitor
2     def __init__(self) -> None:
3         pass
```

(suite sur la page suivante)

```

4
5  def visit_node(self, node: Node) -> None:
6      pass
7
8  def visit_leaf(self, leaf: Leaf) -> None:
9      print(leaf.value)
10
11
12 class SumVisitor: # implements Visitor
13     def __init__(self) -> None:
14         self._sum = 0
15
16     @property
17     def sum(self) -> int:
18         return self._sum
19
20     def visit_node(self, node: Node) -> None:
21         pass
22
23     def visit_leaf(self, leaf: Leaf) -> None:
24         self._sum += leaf.value
25
26
27 e = Node(Leaf(3), Leaf(4))
28
29 e.accept(PrintVisitor())
30
31 v = SumVisitor()
32 e.accept(v)
33 print(v.sum)

```

La figure Figure 8.1 présente le *pattern* sous la forme d'un diagramme de classes UML.

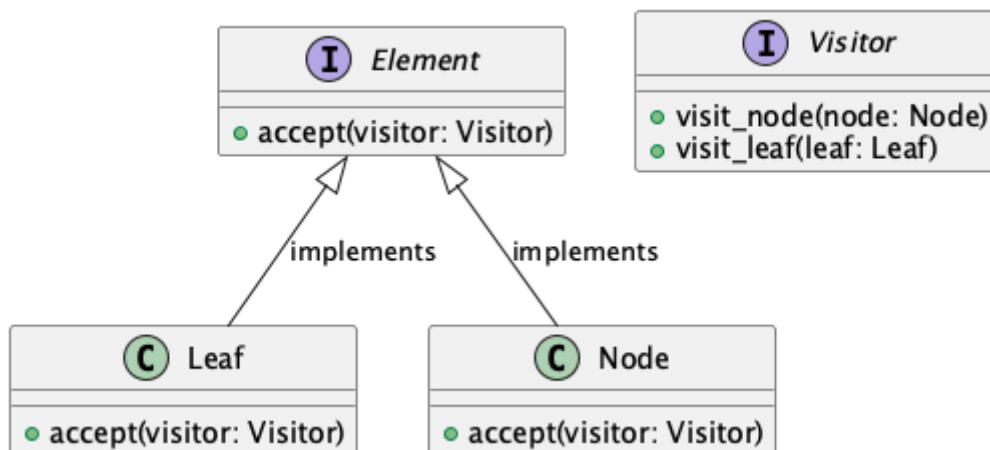


FIGURE 8.1 – Pattern visiteur.

## 8.3 Discussion

Le *pattern* visiteur est intéressant, car il permet de spécifier une nouvelle opération sans avoir à (re-)écrire le code de parcours de l'arbre<sup>1</sup>, seulement en décrivant la sémantique de l'opération sur chaque type d'élément possible (ici, une feuille ou un nœud interne). On voit qu'il est facile d'ajouter une nouvelle opération (*i. e.* une nouvelle classe implémentant `Visitor`) mais sensiblement plus coûteux d'ajouter un nouveau type d'élément puisque cela touche la définition de `Visitor` et donc invalide (c'est-à-dire nécessite une mise à jour) potentiellement les visiteurs déjà programmés. On notera en outre que notre implémentation fixe l'ordre de parcours des éléments de l'arbre (en profondeur et à gauche d'abord, suffixe) ce qui peut poser problème pour certaines opérations.

## Annexes

### À retenir

Le formalisme UML a une utilisation très fortement dépendante du milieu : quasi inexistant dans le monde de la recherche, il est en revanche largement utilisé dans les SSII<sup>1</sup> / ESN<sup>2</sup> où il apparaît dans les cahiers des charges. Il semble nécessaire de savoir au moins *lire* un diagramme de classes.

Les *design patterns* font partie de la culture générale d'un développeur. S'il est inutile de les connaître tous (ou même un grand nombre), il est important de comprendre ce qui justifie leur introduction et de connaître quelques "grands classiques" comme adaptateur et itérateur.

## Références

La documentation de Python n'utilise pas UML, mais présente (au moins implicitement) les *patterns* centraux (*p. ex.* itérateur).

- "Design Patterns : Elements of Reusable Object-Oriented Software" (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley) présente le travail sémi-nal sur le sujet, et encore très largement considéré comme une référence majeure du domaine. L'ouvrage est tellement connu, reconnu, et cité que ses auteurs sont surnommés le *Gang of Four (GoF)* et l'ouvrage le *Gang of Four Book*. L'ouvrage présente et analyse deux douzaines de *patterns*, et les illustre à l'aide de code C++ et *Smalltalk*.
- Description des itérateurs dans la documentation Python : <https://docs.python.org/3/library/stdtypes.html#typeiter>.

---

1. L'exemple de cette section utilise un arbre, mais il est possible d'utiliser un itérateur pour des structures plus complexes.

1. Société de Services en Ingénierie Informatique.

2. Entreprise de Services du Numérique.