

# Patrons de Conception *(Design Patterns)*

Nan MESSE  
[nan.messe@univ-tlse2.fr](mailto:nan.messe@univ-tlse2.fr)

# Rappel des concepts fondamentaux de l'Orienté objet

- Classe
- Objet
- Héritage
- Encapsulation
- Polymorphisme

# Pourquoi l'approche Orientée Objet a été créée ?

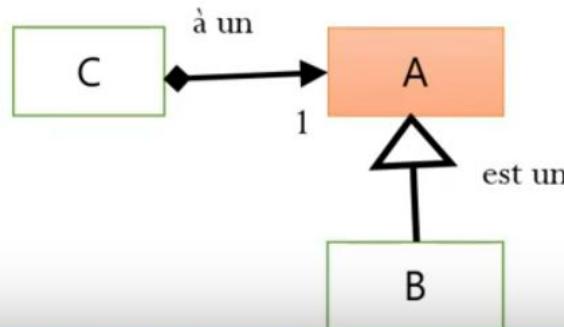
- Pour faciliter la **réutilisation** de l'expérience des autres :
  - Instancier des classes existantes (**Composition**)
  - Créer de nouvelles classes dérivées (**Héritage**)
  - Réutilisation des frameworks
- Créer des applications **faciles à maintenir** :
  - Applications fermées à la modification et ouvertes à l'extension
- Créer des applications **performantes**
- Créer des applications **sécurisées**
- Créer des applications **distribuées**
- **Séparer** les **différents aspects** d'une application :
  - Aspects Métiers (fonctionnels)
  - Aspect Présentation (Web, Mobile, Desktop, ...)
  - Aspect Techniques

# Héritage et Composition

Dans la programmation orientée objet, l'héritage et la composition sont deux moyens qui permettent la réutilisation des classes

- L'héritage traduit le terme « **Est un** » ou « Une sorte de »
- La composition traduit le terme « **A un** » ou « **A plusieurs** ».

Voyons à partir d'un exemple entre l'héritage et la composition de type « **A un** ».



Couplage faible grâce à l'interface

```
public class A {  
    int v1=2;  
    int v2=3;  
    public int meth1(){  
        return(v1+v2);  
    }  
}
```

Composition

```
public class C {  
    int v3=5;  
    A a=new A();  
    void meth2(){  
        System.out.print(a.meth1()*v3);  
    }  
}
```

Héritage

```
public class B extends A {  
    int v3=5;  
    void meth2(){  
        System.out.print(super.meth1()*v3);  
    }  
}
```

Couplage fort

# Couplage fort

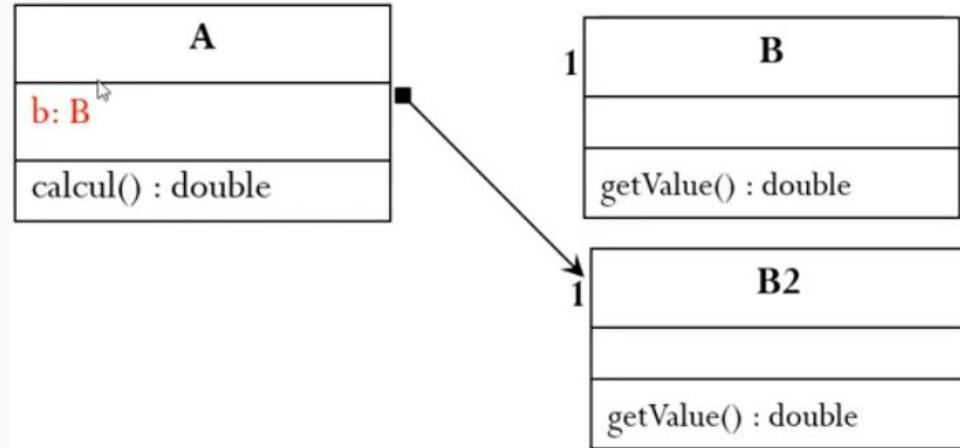
Quand une classe A est lié à une classe B, on dit que la classe A est fortement couplée à la classe B.

La classe A ne peut fonctionner qu'en présence de la classe B.

Si une nouvelle version de la classe B (soit B2), est créé, on est obligé de modifier les codes dans la classe A.

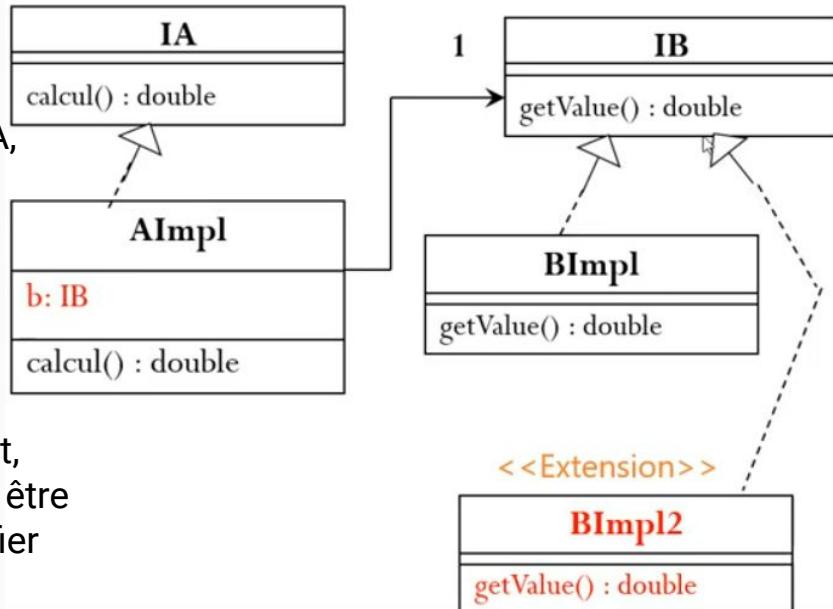
Modifier une classe implique :

- Il faut disposer du code source.
- Il faut recompiler, déployer et distribuer la nouvelle application aux clients.
- Ce qui engendre un cauchemar au niveau de la maintenance de l'application.



# Couplage faible

- Pour utiliser le couplage faible, nous devons utiliser les interfaces.
- Considérons une classe A qui implémente une interface IA, et une classe B qui implémente une interface IB.
- Si la classe A est liée à l'interface IB par une association, on dit que la classe A et la classe B sont liées par un couplage faible.
- Cela signifie que la classe A peut fonctionner avec n'importe quelle classe qui implémente l'interface IB.
- En effet, la classe A ne connaît que l'interface IB. De ce fait, n'importe quelle classe implementant cette interface peut être associée à la classe A, sans qu'il soit nécessaire de modifier quoi que ce soit dans la classe A.
- Avec le couplage faible, nous pourrons créer des applications fermées à la modification et ouvertes à l'extension.

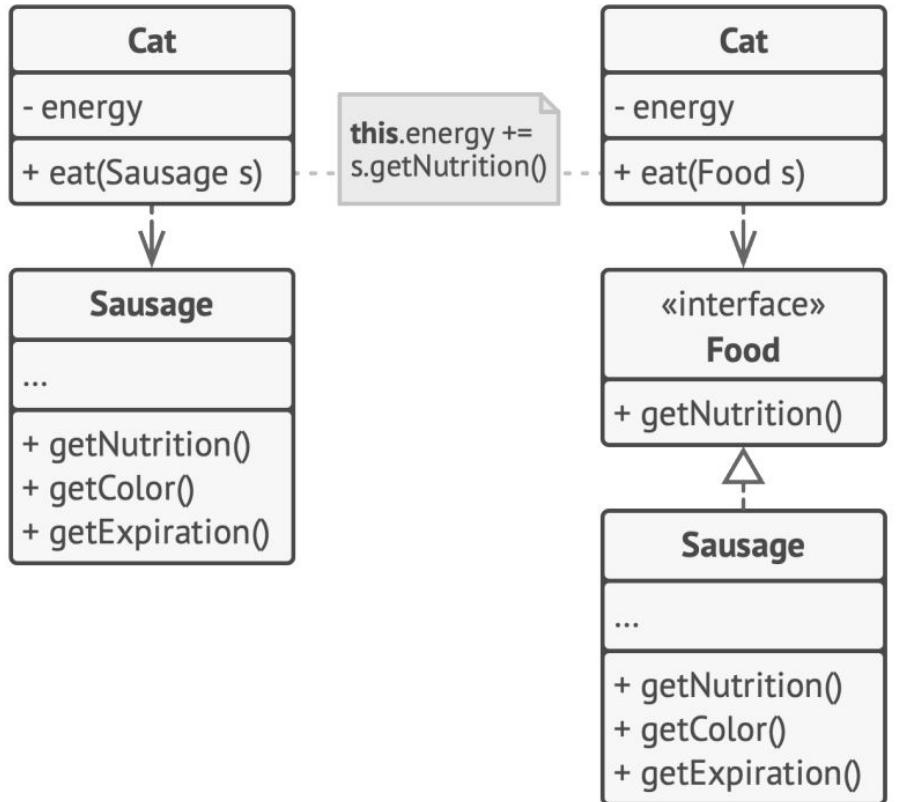


# Caractéristiques d'une bonne conception

- **Réutilisation du code**
  - + réduire le coût et le temps
  - adapter le code à un nouveau contexte demande un certain travail
- **Extensibilité**
  - + répondre aux changements

# Principes de conception

- **Encapsuler ce qui varie et le séparer de ce qui est statique**
  - + minimiser les effets causés par toute modification
  - + au niveau méthode et classe
- Programmer avec les interfaces, et non pas avec les implémentations
  - + Dépendre des abstractions, pas des classes concrètes



*Avant et après avoir extrait l'interface. Le code de droite est plus flexible que celui de gauche, mais également plus compliqué.*

- Encapsuler ce qui varie et le séparer de ce qui est statique

- + minimiser les effets causés par toute modification
- + au niveau méthode et classe

- **Programmer avec les interfaces**

- **Dépendre des abstractions, pas des classes concrètes**

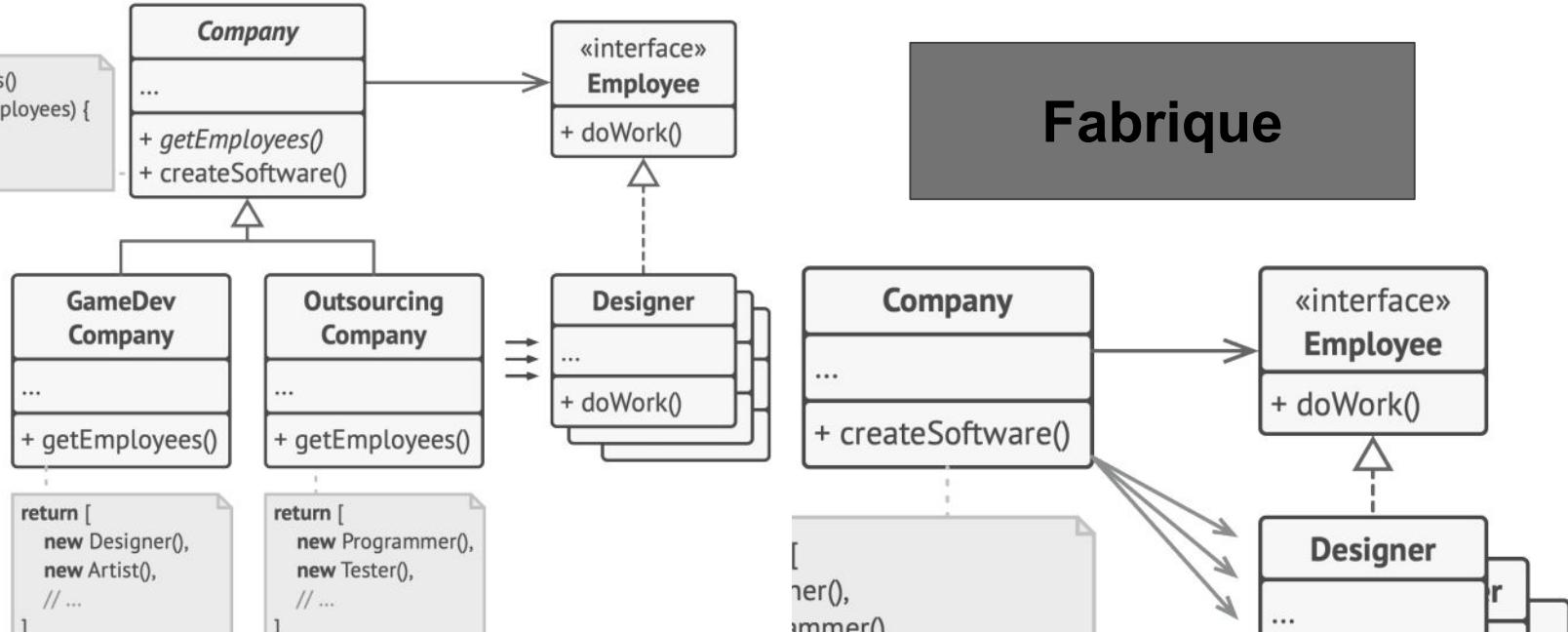
Extensibilité

Flexibilité

```

employees = getEmployees()
foreach (Employee e in employees) {
    e.doWork()
}

```



*APRÈS : la méthode principale de la classe Société est indépendante des classes concrètes des employés. Les objets employé sont créés dans les sous-classes concrètes de la société.*

## Fabrique

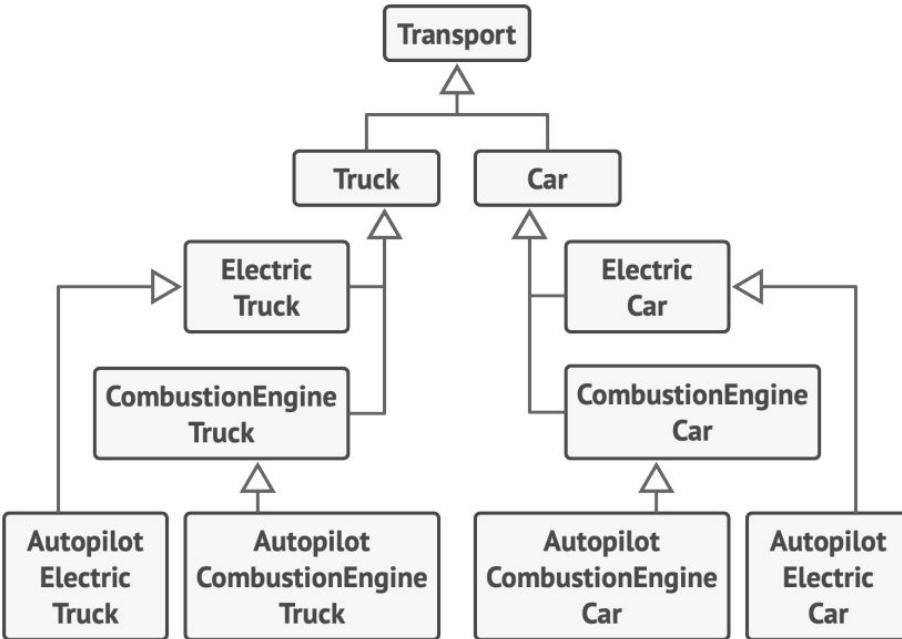
*MIEUX : le polymorphisme nous a aidés à simplifier le code, mais le reste de la classe Société dépend toujours des classes concrètes des employés.*

Conçoit l'architecture.

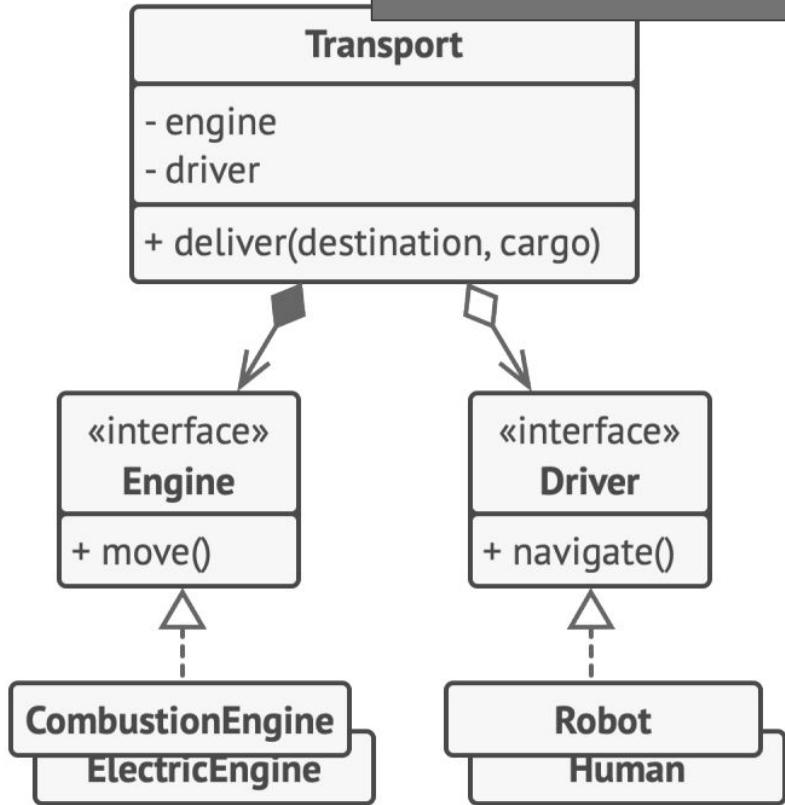
# Principes de conception

- Encapsuler ce qui varie et le séparer-les de ce qui est statique
  - + minimiser les effets causés par toute modification
  - + au niveau méthode et classe
- Programmer avec des interfaces, et non pas avec les implémentations
  - + Dépendre des abstractions, pas des classes concrètes
- **Préférer la composition à l'héritage**
  - Il faut assurer la compatibilité du comportement des méthodes redéfinies et celle de la classe de base
  - L'héritage détruit l'encapsulation de la classe mère
  - Les sous-classes sont fortement couplées aux classes mères
  - Peut avoir une explosion combinatoire de sous-classes

# Stratégie



**HÉRITAGE** : extension de la classe dans plusieurs dimensions (type de transport x type de moteur x type de conduite) qui peut mener à une explosion combinatoire de sous-classes.



**COMPOSITION** : les différentes « dimensions » de la fonctionnalité sont extraites dans leurs propres hiérarchies de classes.

# Histoire

- introduit par Christopher Alexander dans “A Pattern Language: Towns, Buildings, Construction”
- 1977
- repris par Erich Gamma, John Vlissides, Ralph Johnson, et Richard Helm
- dans “**Design Patterns: Elements of Reusable Object-Oriented Software**”
- 1994, 23 patrons
- remplacé par “the book by the Gang of Four”, puis “the GoF book”
- **Pattern-Oriented Software Architecture, Volume 1: A System of Patterns**
- 1996
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal
- POSA1 book



# Patron de Conception (*Design Pattern*) - Définition

Les **patrons de conception** sont des **solutions classiques** à des **problèmes récurrents** de la conception de logiciels [1].

Chaque patron est une sorte de **plan** ou de **schéma** que vous pouvez **personnaliser** afin de résoudre un problème récurrent dans votre code.

recopier



suivre le principe du patron  
et implémenter une solution



# Pourquoi apprendre les patrons ?

- Une **boîte à outils** de solutions fiables et éprouvées permettant de résoudre des **problèmes classiques** de la conception de logiciels.
- Une identification et spécification d'**abstractions** qui sont au dessus du niveau des simples classes et instances (s'applique aux **différents scénarios**).
- Un **vocabulaire commun** de se communiquer -> “Oh, tu n’as qu’à utiliser un singleton”.
- Un moyen de **documentation** de logiciels.
- Une **aide à la construction** de logiciels complexes et hétérogènes, répondant à des **propriétés précises**.

# Classification des *Design Patterns*

- **Patrons de création** : fournissent des mécanismes de création d'objets (ce qui augmente la flexibilité et la réutilisation du code) - Isolation du code
- **Patrons structurels** : expliquent comment assembler des objets et des classes en de plus grandes structures - Éviter des couplages forts
- **Patrons comportementaux** : mettent en place une interaction efficace et répartissent les responsabilités entre les objets.

## GoF Design Patterns Classified (F. Tip)

	Purpose		
	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton ✓	Adapter (object) Bridge Composite ✓ Decorator Facade Flyweight Proxy	Chain of Resp. Command Iterator Mediator Memento Observer ✓ State Strategy ✓ Visitor ✓

Exprime un schéma d'organisation structurel fondamental pour un système logiciel

Fournit un schéma pour affiner les composants d'un système logiciel ou les relations entre eux.

Architecture Patterns	Design Patterns
Layers	Whole-Part
Pipes & Filters	Master-Slave
Blackboard	Proxy
Broker	Command Processor
<b>Model-View-Controller</b> ✓	View handler
Presentation-Abstraction-Control	Forwarded-Receiver
Microkernel	Client-Dispatcher-Server
Reflection	Publisher-Subscriber

- **Nom du patron**
  - utilisé pour décrire le patron, ses solutions et les conséquences en un mot ou deux
- **Problème**
  - description des **conditions** d'applications. Explication du problème et de son **contexte**
- **Solution**
  - description des éléments (objets, relations, responsabilités, collaboration)
  - permettant de concevoir la solution au problème ; utilisation des diag. de classes, de séquences, ...
  - vision statique ET dynamique de la solution
- **Conséquences**
  - description des résultats (effets induits) de l'application du patron sur le système (positifs ET négatifs)

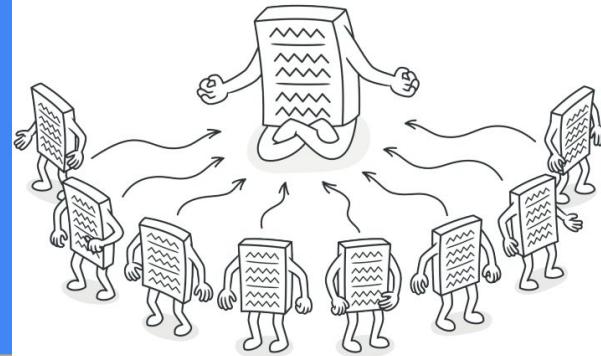
# GoF

## Patrons de création

fournissent des mécanismes de création d'objets (ce qui augmente la flexibilité et la réutilisation du code)

- Fabrique (Factory Method)
- Fabrique abstraite (Abstract Factory)
- Monteur (Builder)
- Prototype
- **Singleton**

# Singleton



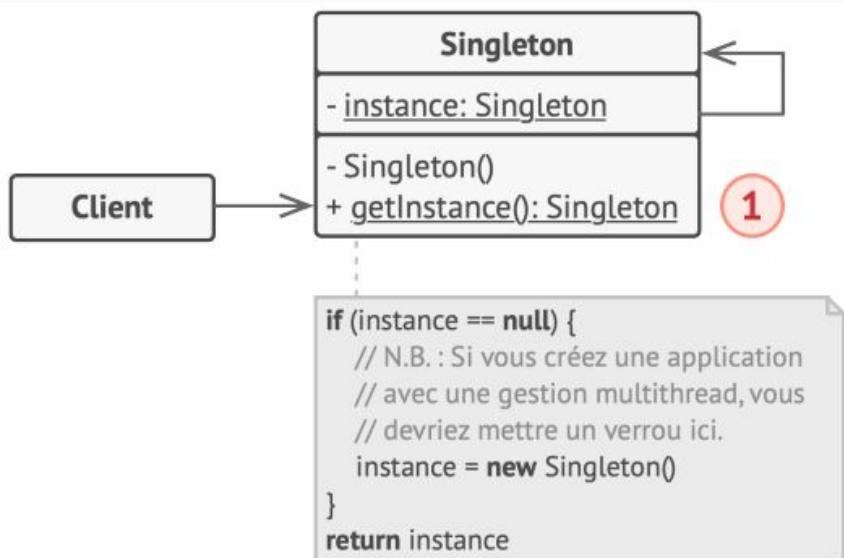
**Singleton** garantit la création d'une **instance unique** d'une classe durant toute la durée d'exécution d'une application, tout en fournissant un **point d'accès global** à cette instance.

**Objectifs** : Il garantit l'**unicité** d'une instance pour une classe et il fournit un **point d'accès global** (une **méthode**) à cette instance.

On l'utilise lorsqu'on veut **contrôler l'accès à une ressource partagée**, par ex.

## Singleton - Solution, Analogy et Structure

- Rendre le **constructeur** par défaut **privé** afin d'empêcher les autres objets d'utiliser l'opérateur ***new*** avec la classe du singleton
- Mettre en place une **méthode de création statique** qui se comporte comme un constructeur. Cette méthode appelle le constructeur privé pour créer un objet et le sauvegarde dans un **attribut statique**. Tous les appels ultérieurs à cette méthode retournent **l'objet en cache**.
- Exemple : gouvernement



## Singleton - Pseudo-code avec un exemple bdd

```
1 // La classe baseDeDonnées définit la méthode `getInstance` qui
2 // permet aux clients d'accéder à la même instance de la
3 // connexion à la base de données dans tout le programme.
4 class Database is
5     // L'attribut qui stocke l'instance du singleton doit être
6     // 'static'.
7     private static field instance: Database
8
9     // Le constructeur du singleton doit toujours être privé
10    // afin d'empêcher les appels à l'opérateur `new`.
11    private constructor Database() is
12        // Code d'initialisation (la connexion au serveur de la
13        // base de données par exemple).
14        // ...
15
```

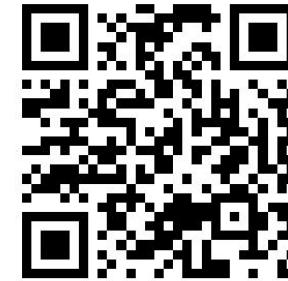
## Singleton - Pseudo-code avec un exemple bdd

```
16 // La méthode statique qui contrôle l'accès à l'instance du
17 // singleton.
18 public static method getInstance() is
19     if (Database.instance == null) then
20         acquireThreadLock() and then
21             // Ce thread attend la levée du verrou (lock) le
22             // temps de s'assurer que l'instance n'a pas
23             // déjà été initialisée dans un autre thread.
24         if (Database.instance == null) then
25             Database.instance = new Database()
26         return Database.instance
27
28 // Pour finir, tout singleton doit définir de la logique
29 // métier qui peut être exécutée dans sa propre instance.
30 public method query(sql) is
31     // Par exemple, toutes les requêtes sur la base de
32     // données d'une application passent par cette méthode.
33     // Par conséquent, vous pouvez définir le code des
34     // limitations ou de la mise en cache ici.
35     // ...
```

```
37 class Application is
38     method main() is
39         Database foo = Database.getInstance()
40         foo.query("SELECT ...")
41         // ...
42         Database bar = Database.getInstance()
43         bar.query("SELECT ...")
44         // La variable `bar` contiendra le même objet que la
45         // variable `foo`.
```

# Singleton - Possibilités d'application

- Utilisez le singleton lorsque l'une de vos classes ne doit fournir qu'**une seule instance** à tous ses clients.
  - Ex. une **base de données partagée** entre toutes les parties d'un programme.
- Utilisez le singleton lorsque vous voulez un **contrôle absolu** sur vos variables globales.
- Exercice : <https://www.wooclap.com/EPIRIZ>



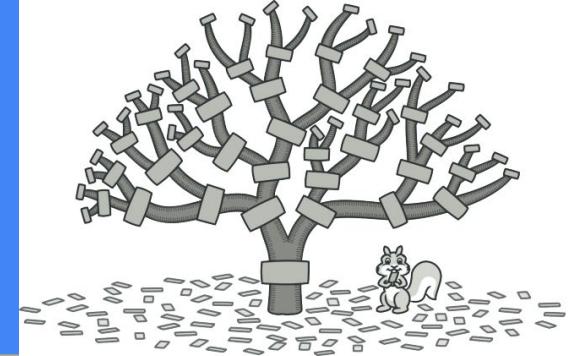
# GoF

## Patrons Structurels

expliquent comment assembler des objets et des classes en de plus grandes structures

- Adaptateur (Adapter)
- Pont (Bridge)
- **Composite**
- Décorateur (Decorator)
- Façade (Facade)
- Poids mouche (Flyweight)
- Procuration (Proxy)

# Composite

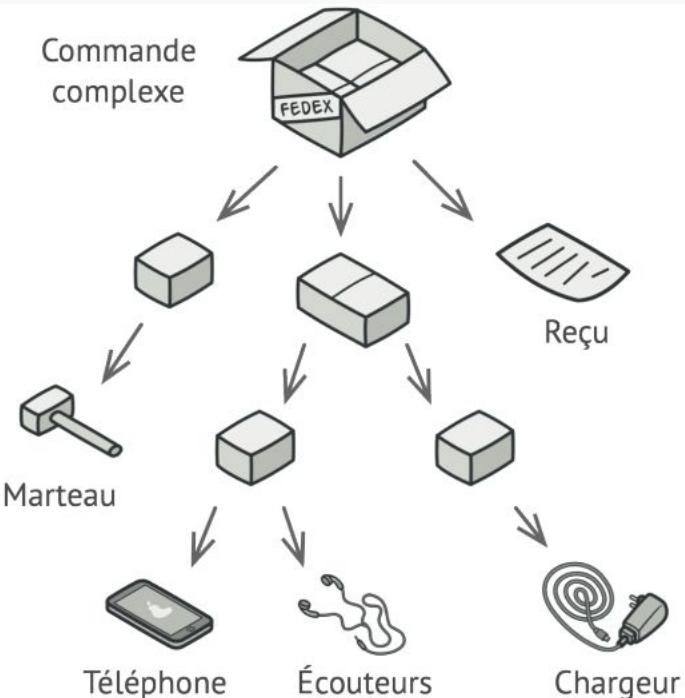


**Composite** permet d'organiser les objets en structure **arborescente** afin de représenter une **hiérarchie**.

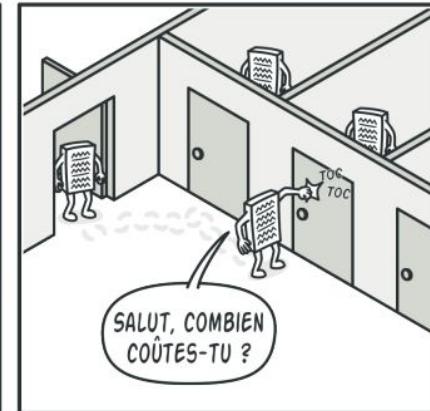
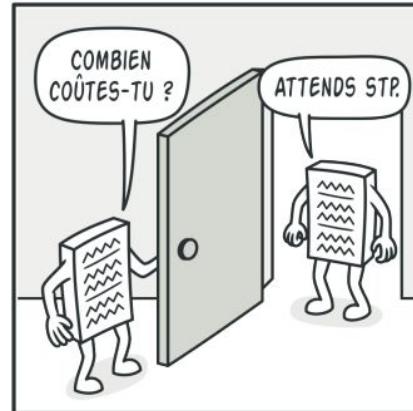
Ce patron permet à la partie cliente de manipuler un **objet unique** et un **objet composé de la même manière**.

L'utilisation de ce patron doit être réservée aux applications dont la structure principale peut être représentée sous la forme d'une **arborescence**.

# Composite - Calculer le coût total d'une commande

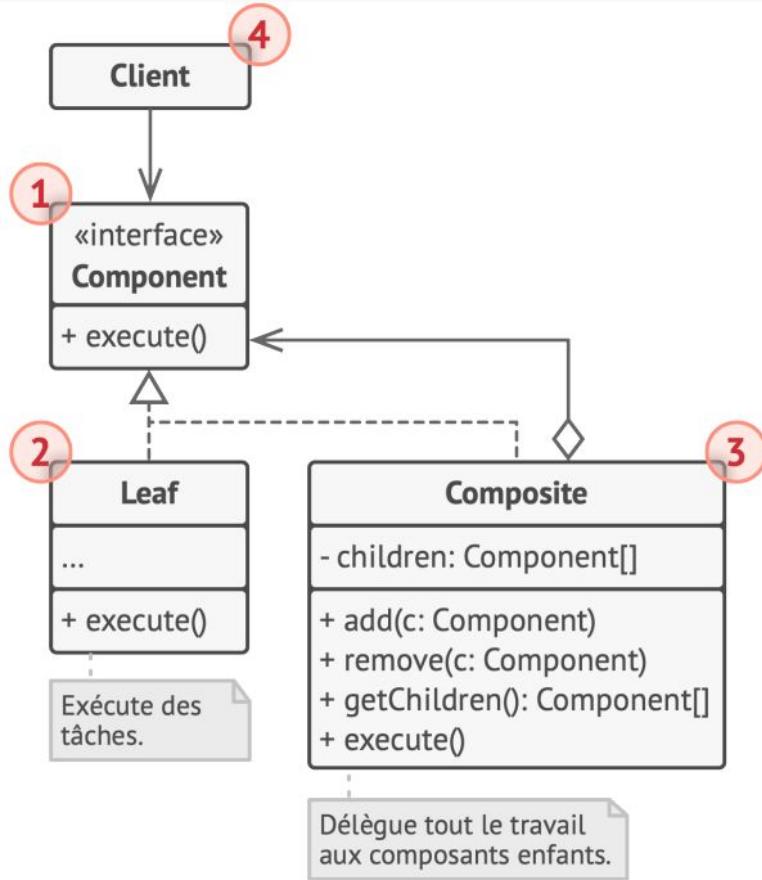


Une commande peut contenir divers produits empaquetés à l'intérieur de boîtes, elles-mêmes rangées dans de plus grosses boîtes, etc. La structure complète ressemble à un arbre inversé.



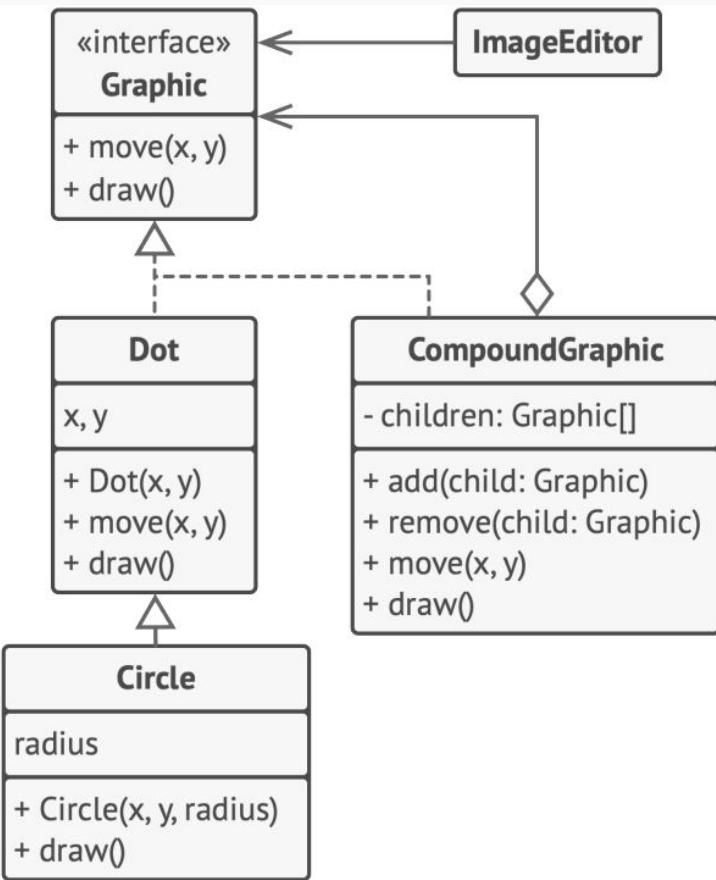
Le patron de conception composite emploie une méthode récursive afin de parcourir tous les composants d'une arborescence.

# Composite - Structure

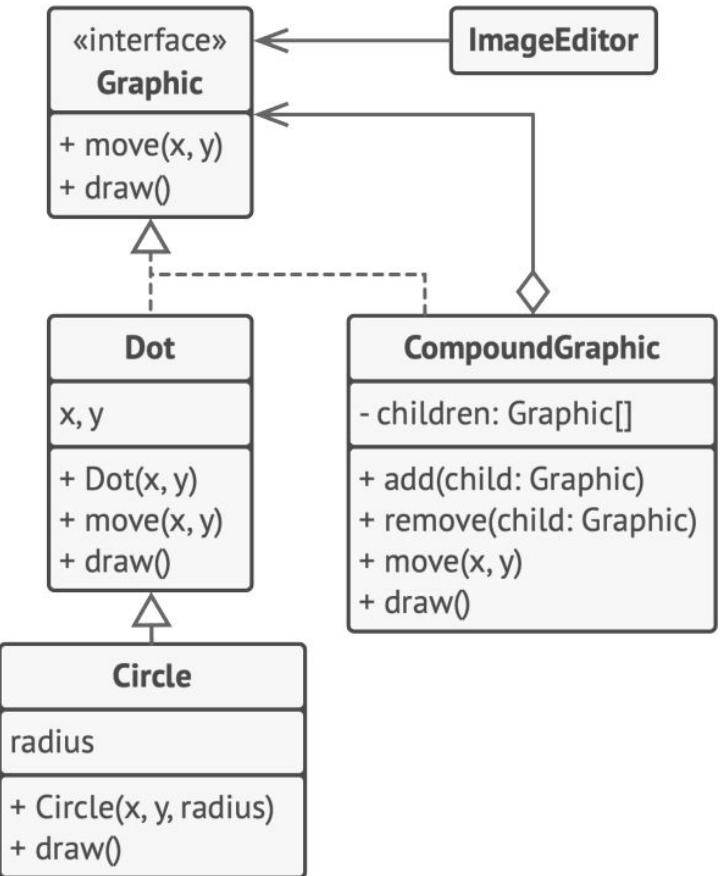


1. L'interface **Composant** décrit les opérations communes aux objets simples et complexes de l'arborescence.
2. Une **Feuille** est un élément de base d'une branche qui n'a pas de sous-élément.
3. Le **Conteneur** (alias **composite**) est un élément composé de sous-éléments : des feuilles ou d'autres conteneurs. Un conteneur ne connaît pas les classes de ses enfants. Il passe par l'interface composant pour interagir avec ses sous-éléments.
4. Le **Client** manipule les éléments depuis l'interface composant, ce qui lui permet de fonctionner **de la même manière** pour les éléments simples et complexes de l'arborescence.

# Composite - Exemple de l'éditeur des formes géométriques

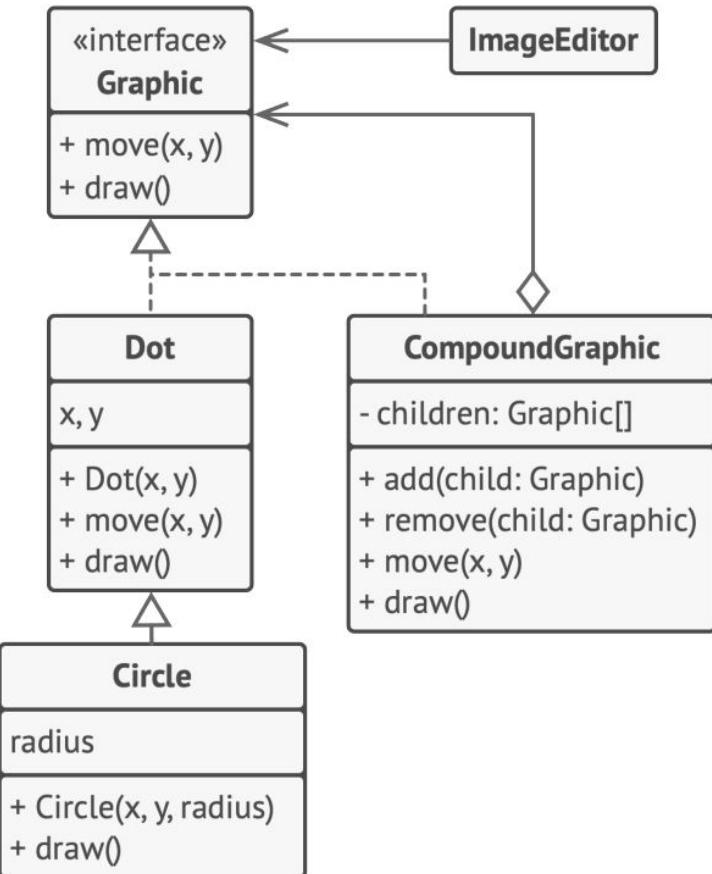


```
1 // L'interface du composant déclare des opérations communes pour
2 // les objets simples et complexes d'une composition.
3 interface Graphic is
4     method move(x, y)
5     method draw()
6
7 // La classe feuille représente les objets finaux d'une
8 // composition. Un objet feuille ne peut pas avoir de sous-
9 // objets. En général, ce sont les feuilles qui lancent les
10 // traitements. Les objets composite ne font que déléguer le
11 // travail à leurs sous-composants.
12 class Dot implements Graphic is
13     field x, y
14
15     constructor Dot(x, y) { ... }
16
17     method move(x, y) is
18         this.x += x, this.y += y
19
20     method draw() is
21         // Dessine un point aux coordonnées X et Y.
```



```

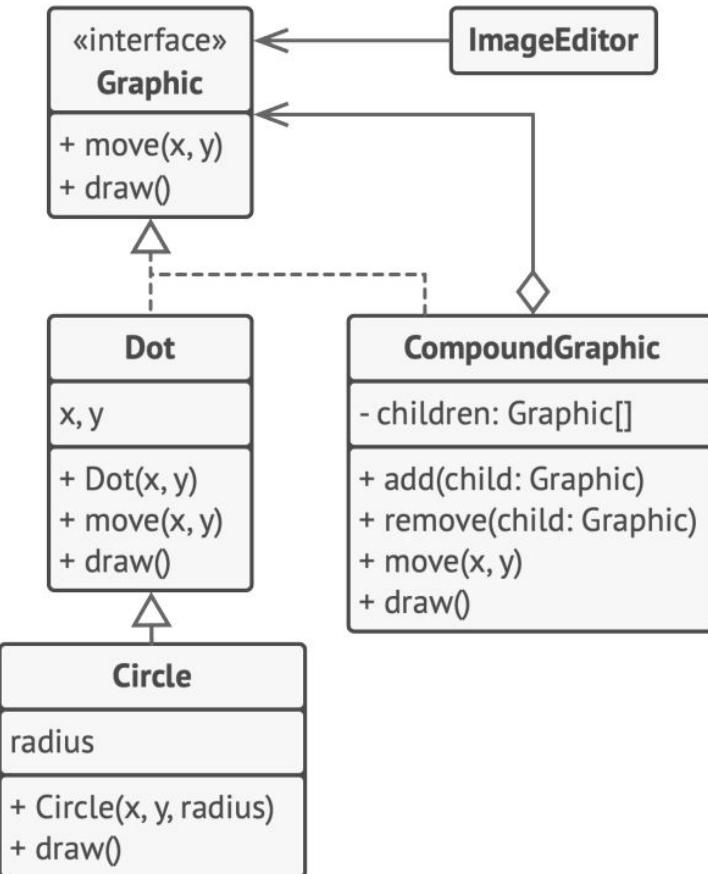
23 // Toutes les classes composant peuvent étendre d'autres
24 // composants.
25 class Circle extends Dot is
26   field radius
27   constructor Circle(x, y, radius) { ... }
28
29
30 method draw() is
31   // Trace un cercle de rayon R aux coordonnées X et Y.
32
33 // La classe composite représente les composants complexes qui
34 // peuvent avoir des enfants. Les objets composite délèguent
35 // généralement les tâches à leurs enfants et « additionnent »
36 // ensuite le résultat.
37 class CompoundGraphic implements Graphic is
38   field children: array of Graphic
39
40 // Un composite peut ajouter ou retirer d'autres composants
41 // (simples ou complexes) de la liste de ses enfants.
42 method add(child: Graphic) is
43   // Ajoute un enfant au tableau d'enfants.
44
45 method remove(child: Graphic) is
46   // Retire un enfant du tableau d'enfants.
47
48 method move(x, y) is
49   foreach (child in children) do
50     child.move(x, y)
  
```



```

52     // Un composite exécute sa logique principale d'une certaine
53     // manière : il parcourt récursivement tous ses enfants,
54     // puis récupère et additionne leurs résultats. L'objet est
55     // entièrement parcouru, car les enfants du composite
56     // passent ces appels à leurs propres enfants et ainsi de
57     // suite.
58
59     method draw() is
60         // 1. Pour chaque composant enfant :
61         //      – Dessine le composant.
62         //      – Met à jour le rectangle de délimitation.
63         // 2. Dessine un rectangle en pointillé en utilisant les
64         // coordonnées de la délimitation.

```



```

66 // Le code client manipule les composants grâce à leur interface
67 // de base. Ainsi, le code client peut aussi bien prendre en
68 // charge les composants simples que les complexes.
69 class ImageEditor is
70     field all: CompoundGraphic
71
72 method load() is
73     all = new CompoundGraphic()
74     all.add(new Dot(1, 2))
75     all.add(new Circle(5, 3, 10))
76     // ...
77
78 // Combine les composants sélectionnés en un seul composant
79 // complexe.
80 method groupSelected(components: array of Graphic) is
81     group = new CompoundGraphic()
82     foreach (component in components) do
83         group.add(component)
84         all.remove(component)
85     all.add(group)
86     // Tous les composants vont être dessinés.
87     all.draw()

```

# Composite - Possibilités d'application

- Utilisez le composite si vous devez gérer une structure d'objets qui ressemble à une **arborescence**.
- Utilisez ce patron si vous voulez que le client interagit avec les éléments simples aussi bien que complexes de **façon uniforme**.
- Exercice : <https://www.wooclap.com/EPIRIZ>



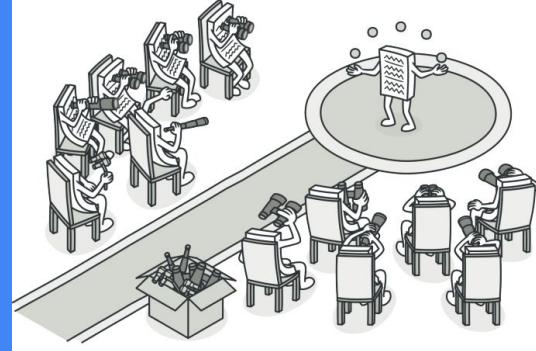
# GoF

## Patrons Comportementaux

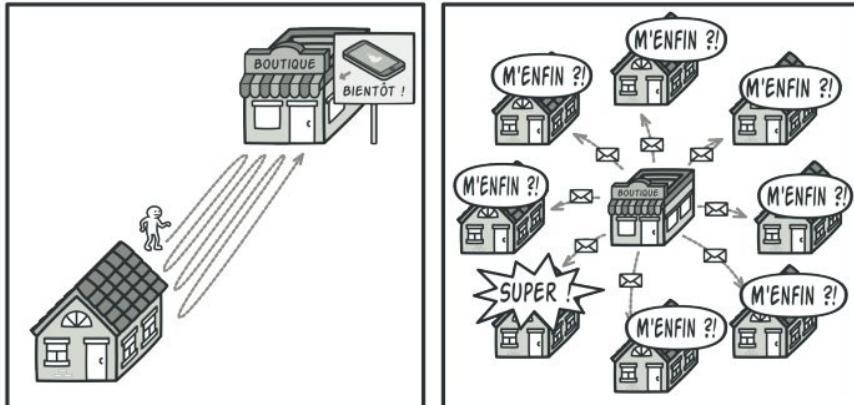
mettent en place une communication efficace et répartissent les responsabilités entre les objets

- Chaîne de Responsabilité (Chain of Responsibility)
- Commande (Command)
- Itérateur (Iterator)
- Médiateur (Mediator)
- Memento (Memento)
- Observateur (Observer)
- État (State)
- Stratégie (Strategy)
- Patron de Méthode (Template Method)
- Visiteur (Visitor)

# Observer



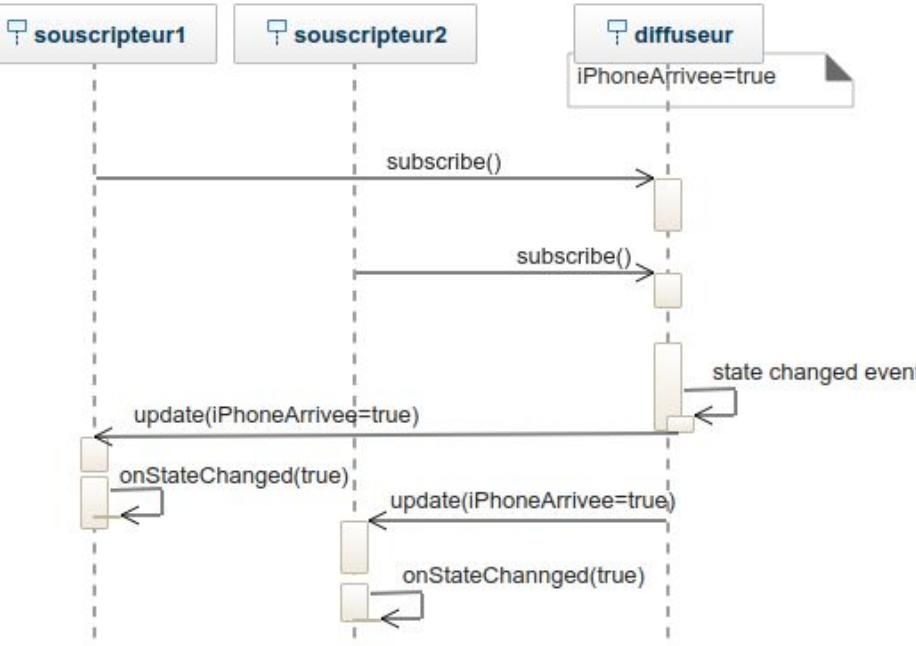
L'Observateur permet de mettre en place un mécanisme de **souscription** pour envoyer des **notifications** à plusieurs objets, au sujet d'**événements** concernant les objets qu'ils observent.



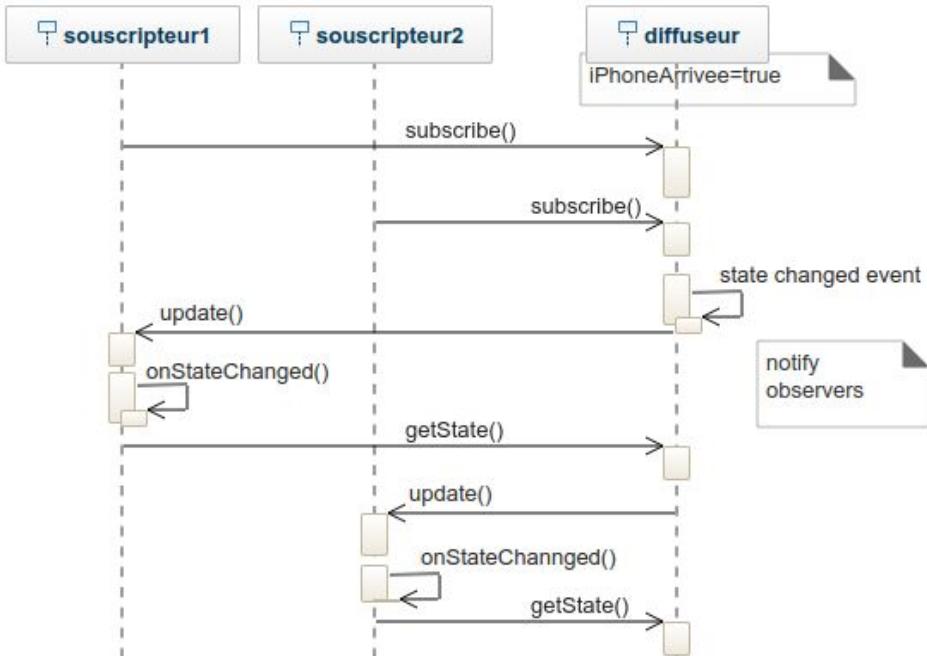
*Se rendre au magasin ou envoyer du spam.*

- L'objet que l'on veut suivre est en général appelé sujet, mais comme il va envoyer des notifications pour prévenir les autres objets dès qu'il est modifié, nous l'appellerons **diffuseur (publisher)**. Tous les objets qui veulent suivre les modifications apportées au diffuseur sont appelés des **souscripteurs (subscribers)** -> relation un à plusieurs.
- Le patron Observateur vous propose d'ajouter un **mécanisme de souscription** à la classe **diffuseur** pour permettre aux objets individuels de **s'inscrire ou se désinscrire** de ce diffuseur.
- À chaque fois que **l'état** de diffuseur change, tout ce qui en **dépendent** en soient **informés** et soient **mis à jour** automatiquement.

# Push and Pop : 2 moyens d'observer

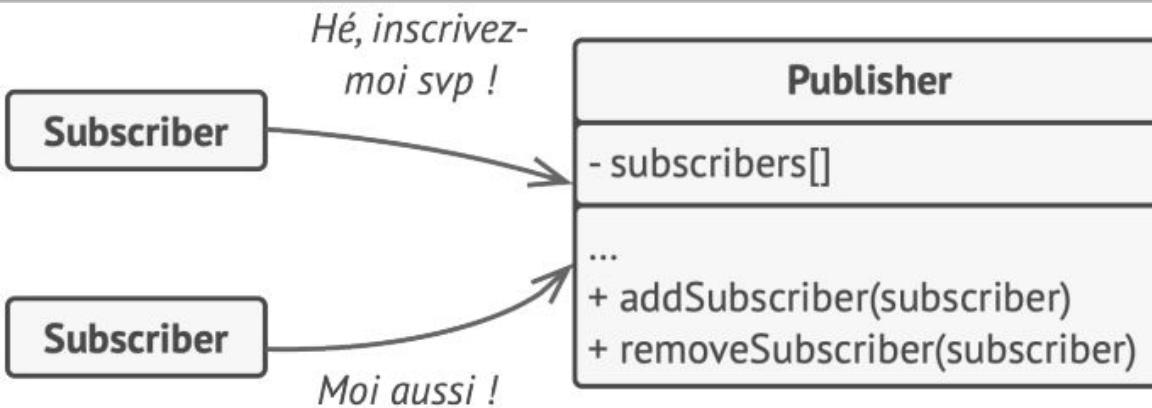


Push to observers model



Pop to observers model

# Observer



*Un mécanisme de souscription qui permet aux objets individuels de s'inscrire aux notifications des événements.*

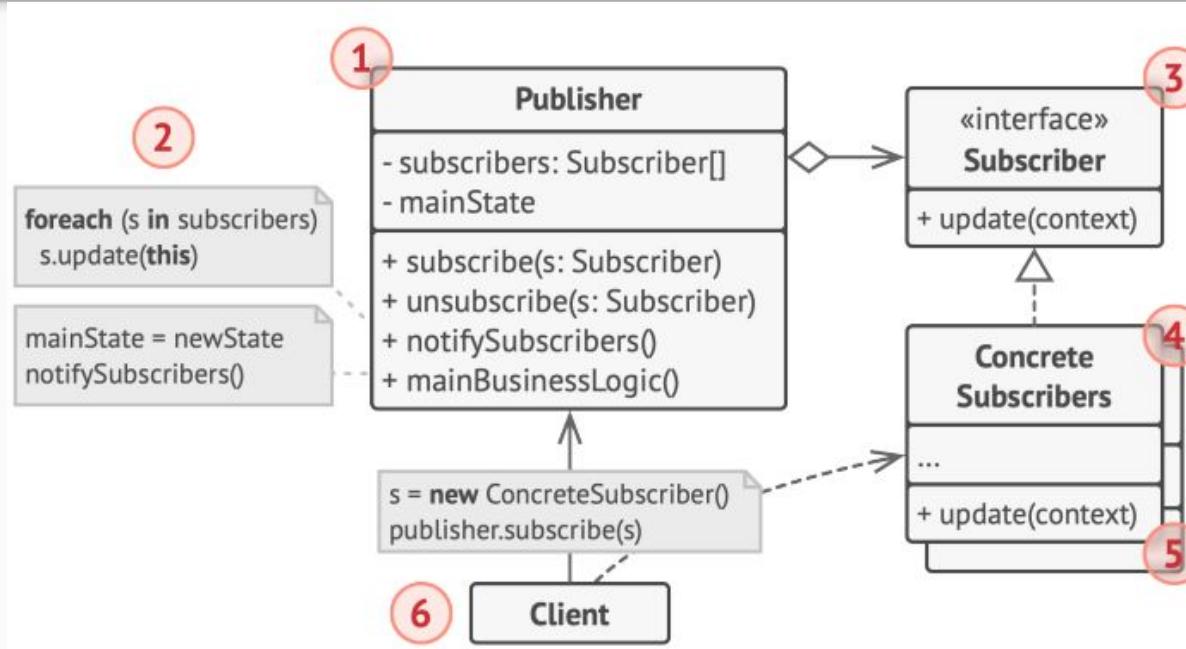
Ce mécanisme est composé :

- 1) d'un **tableau** d'attributs qui stocke une **liste de références** vers les objets **souscripteur**
- 2) de plusieurs méthodes publiques qui permettent d'**ajouter** ou de **supprimer** des souscripteurs de cette liste.

Quand un **événement** important arrive au **diffuseur**, il fait le tour de ses souscripteurs et appelle la **méthode** de **notification** sur leurs objets.

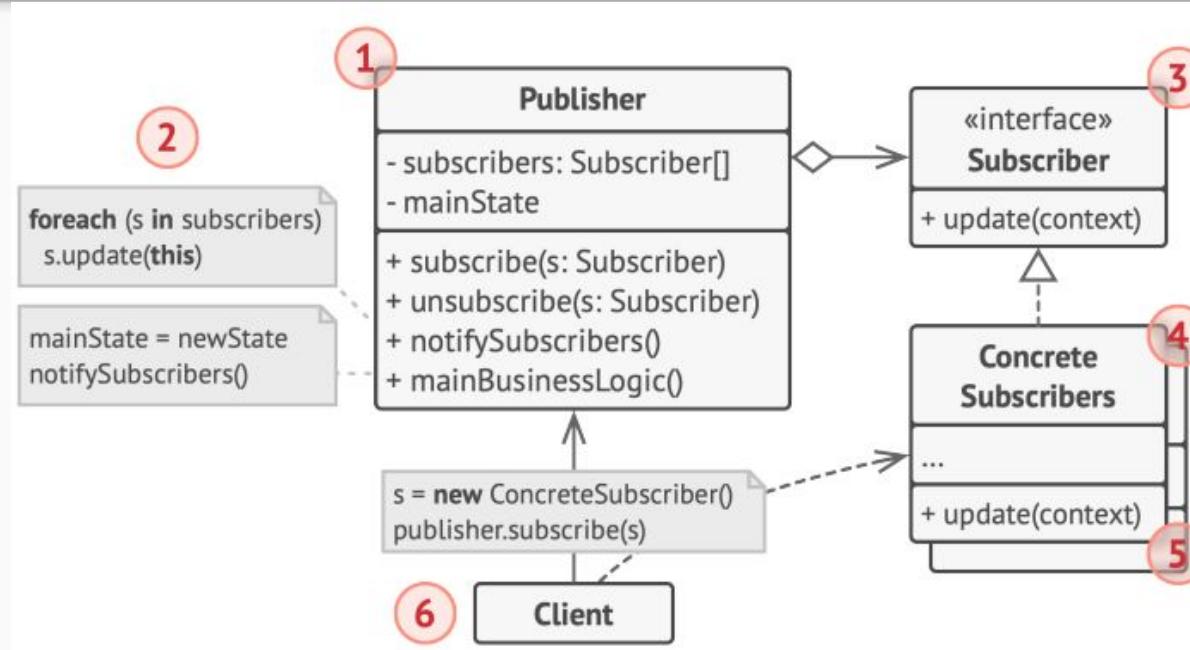
Analogie : abonnement aux magazines et aux journaux ; recherche d'emploi, ...

# Observer - Structure



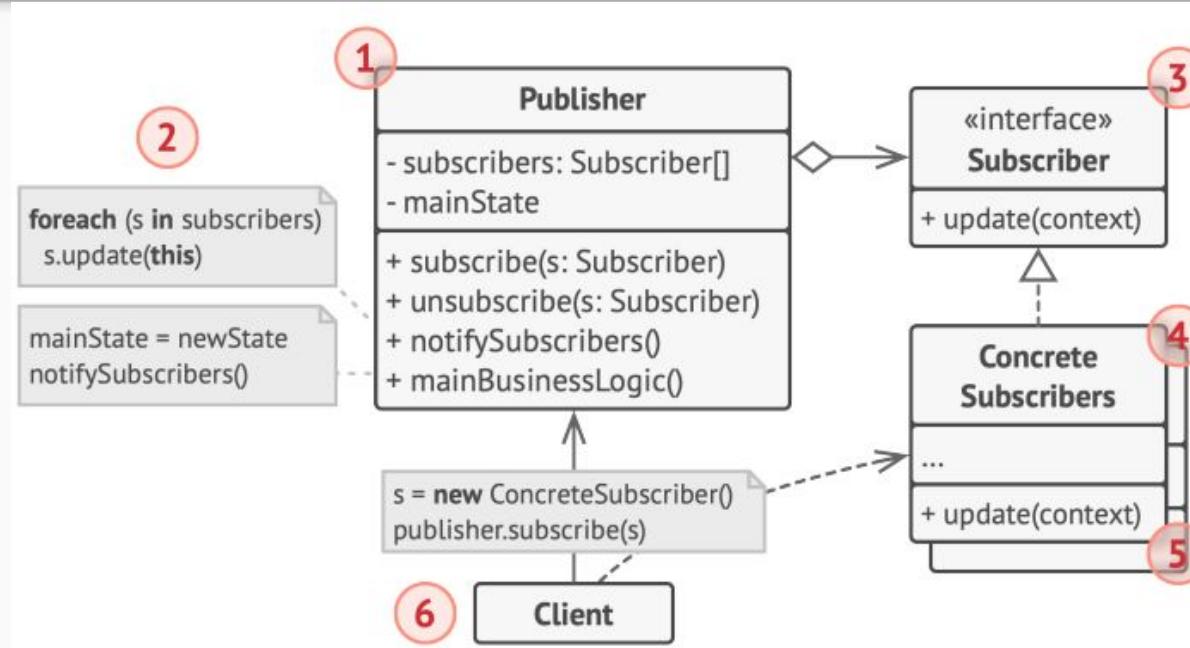
1. Le **Diffuseur** envoie des **événements** intéressants à d'autres objets. Ces événements se produisent quand le diffuseur **change d'état ou exécute certains comportements**. Le diffuseur possède une infrastructure **d'inscription** qui permet aux nouveaux souscripteurs de **rejoindre** la liste et aux souscripteurs actuels de la **quitter**.

# Observer - Structure



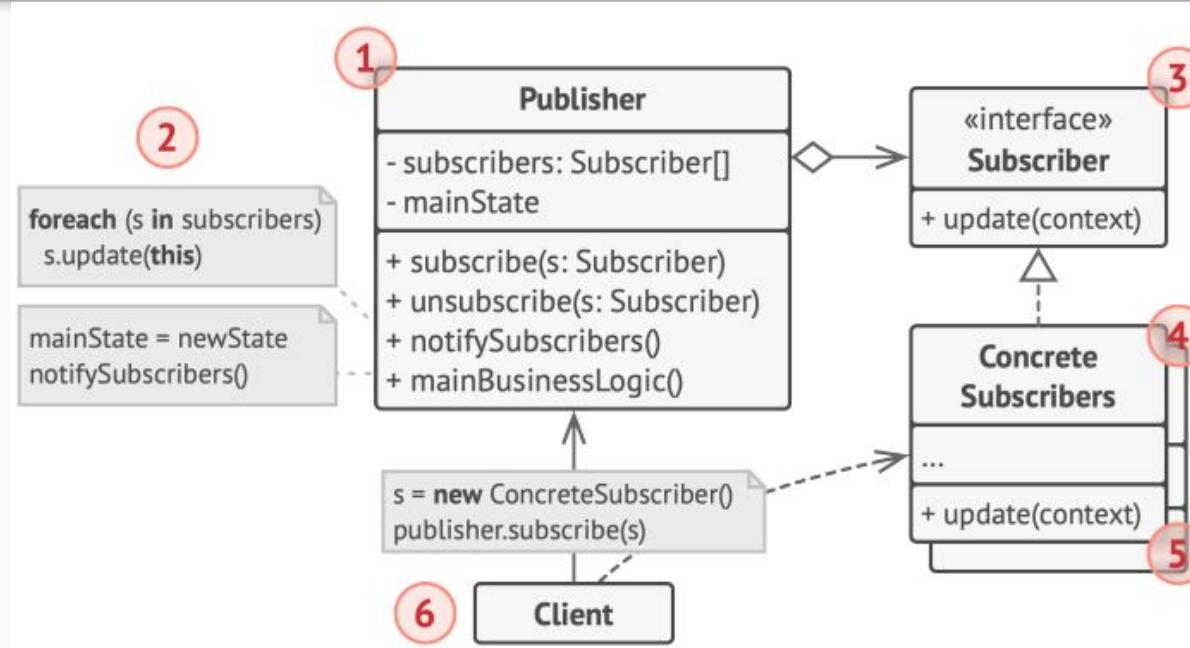
2. Quand un **nouvel événement** survient, le diffuseur parcourt la **liste d'inscriptions** et appelle la méthode de **notification** déclarée dans l'interface des souscripteurs sur chaque objet souscripteur.
3. L'**interface Souscripteur** déclare la méthode de **notification update()**. Elle peut prendre plusieurs paramètres pour que le diffuseur leur envoie plus de détails concernant la modification.

# Observer - Structure



4. Les **Souscripteurs Concrets** exécutent certaines actions en réponse aux notifications envoyées par le diffuseur. Toutes ces classes doivent **implémenter** la même **interface** pour ne pas coupler le diffuseur avec leurs classes concrètes.

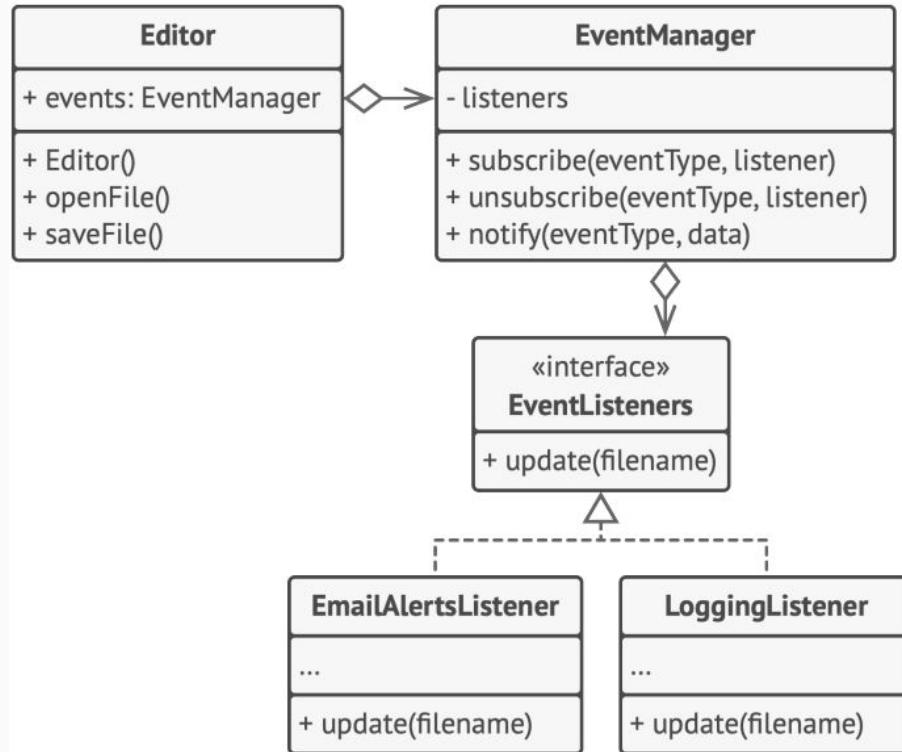
# Observer - Structure



5. En général, les souscripteurs ont besoin de détails à propos du **contexte** afin d'exécuter correctement la mise à jour. C'est pour cela que les diffuseurs passent souvent des données du contexte en paramètre de la méthode de notification.

6. Le **Client** crée des objets diffuseur et Souscripteur séparément et **inscrit** les souscripteurs aux mises à jour du diffuseur.

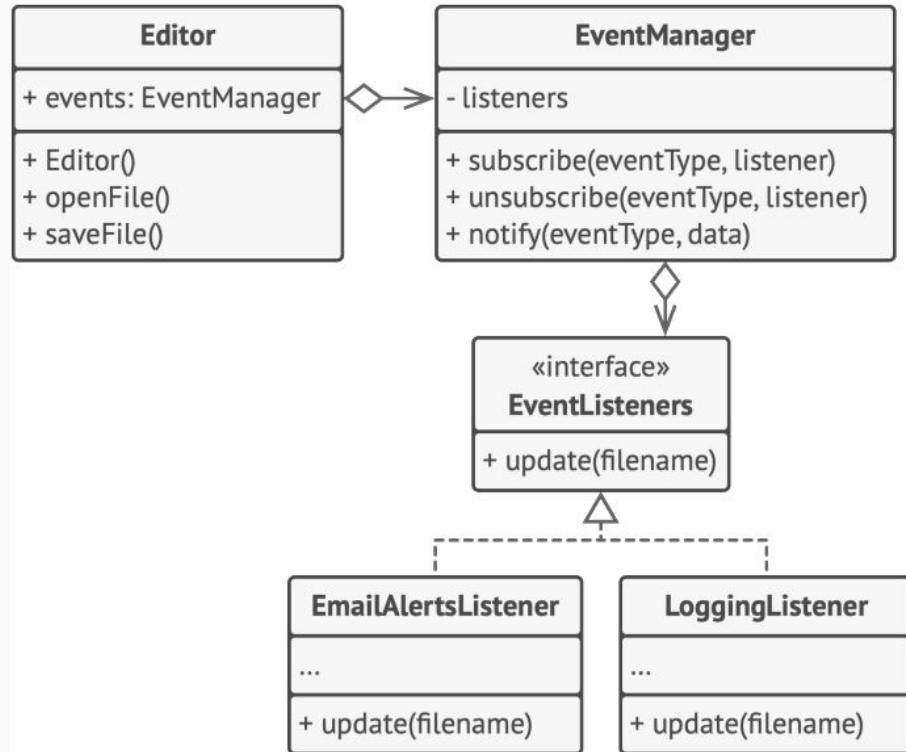
# Observer - Exemple avec l'éditeur de texte



*Envoyer des notifications aux objets au sujet d'événements qui affectent d'autres objets.*

```
1 // La classe de base diffuseur contient le code pour
2 // l'inscription et les méthodes de notification.
3 class EventManager is
4     private field listeners: hash map of event types and listeners
5
6     method subscribe(eventType, listener) is
7         listeners.add(eventType, listener)
8
9     method unsubscribe(eventType, listener) is
10        listeners.remove(eventType, listener)
11
12    method notify(eventType, data) is
13        foreach (listener in listeners.of(eventType)) do
14            listener.update(data)
15
16 // Le diffuseur concret abrite de la logique métier dédiée à
17 // certains souscripteurs. Nous pourrions dériver cette classe
18 // depuis le diffuseur de base, mais ce n'est pas toujours
19 // possible, car le diffuseur concret pourrait déjà être une
20 // sous-classe. Dans ce cas, vous pouvez utiliser la composition
21 // pour effectuer le lien avec la logique de souscription, comme
22 // effectué ci-dessous :
23 class Editor is
24     public field events: EventManager
25     private field file: File
```

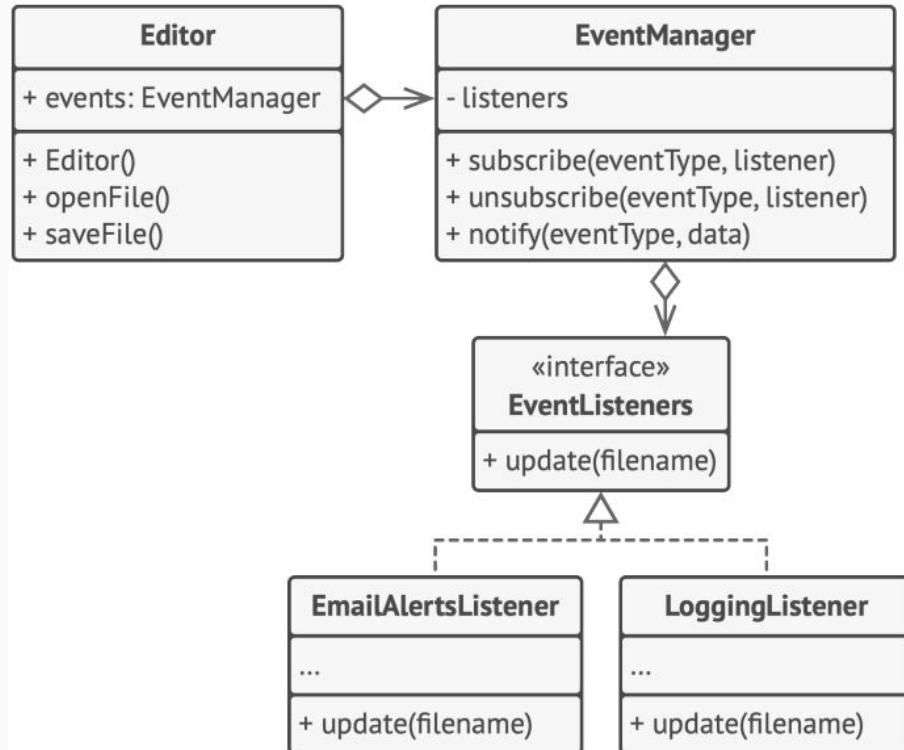
# Observer - Exemple avec l'éditeur de texte



*Envoyer des notifications aux objets au sujet d'événements qui affectent d'autres objets.*

```
27 constructor Editor() is
28     events = new EventManager()
29
30 // Les méthodes de la logique métier peuvent prévenir les
31 // souscripteurs de toute modification.
32 method openFile(path) is
33     this.file = new File(path)
34     events.notify("open", file.name)
35
36 method saveFile() is
37     file.write()
38     events.notify("save", file.name)
39
43 // Voici l'interface des souscripteurs. Si votre langage de
44 // programmation prend en charge les types fonctionnels, vous
45 // pouvez remplacer toute la hiérarchie des souscripteurs par un
46 // ensemble de fonctions.
47 interface EventListener is
48     method update(filename)
49
50 // Les souscripteurs concrets réagissent aux mises à jour de
51 // leur diffuseur.
52 class LoggingListener implements EventListener is
53     private field log: File
54     private field message
55
56 constructor LoggingListener(log_filename, message) is
57     this.log = new File(log_filename)
58     this.message = message
```

# Observer - Exemple avec l'éditeur de texte



Envoyer des notifications aux objets au sujet d'événements qui affectent d'autres objets.

```
60   method update(filename) is
61     log.write(replace('%s',filename,message))
62
63 class EmailAlertsListener implements EventListener is
64   private field email: string
65
66 constructor EmailAlertsListener(email, message) is
67   this.email = email
68   this.message = message
69
70   method update(filename) is
71     system.email(email, replace('%s',filename,message))
72
73
74 // Une application peut configurer des diffuseurs et des
75 // souscripteurs à l'exécution.
76 class Application is
77   method config() is
78     editor = new Editor()
79
80     logger = new LoggingListener(
81       "/path/to/log.txt",
82       "Someone has opened the file: %s")
83     editor.events.subscribe("open", logger)
84
85     emailAlerts = new EmailAlertsListener(
86       "admin@example.com",
87       "Someone has changed the file: %s")
88     editor.events.subscribe("save", emailAlerts)
```

# Observer - Possibilités d'application

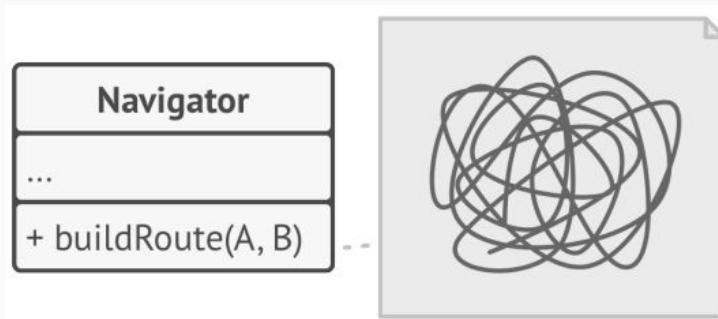
- Utilisez l'Observateur quand des **modifications** de l'état d'un objet peuvent en **impacter d'autres**, et que l'ensemble des objets n'est **pas connu à l'avance** ou qu'il **change dynamiquement**.
- Utilisez ce patron quand certains objets de votre application doivent en **suivre** d'autres, mais seulement pendant un certain temps ou dans des cas spécifiques.
- Exercice : <https://www.wooclap.com/EPIRIZ>



# Strategy

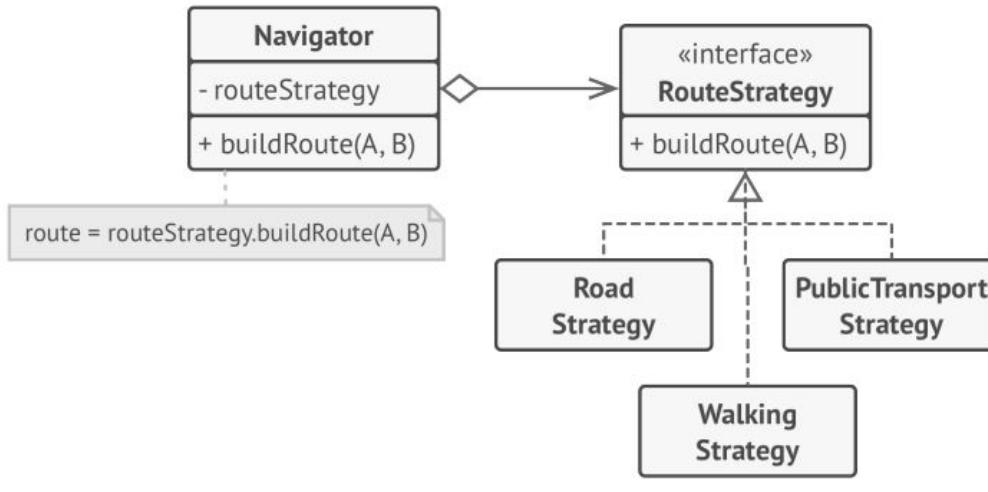


Objectif : Définir une famille d'algorithmes, et encapsuler chacun, et les rendre interchangeables, tout en assurant que chaque algorithme puisse évoluer indépendamment des clients qui l'utilisent.



*Le code du navigateur grossit à vue d'œil.*

# Strategy

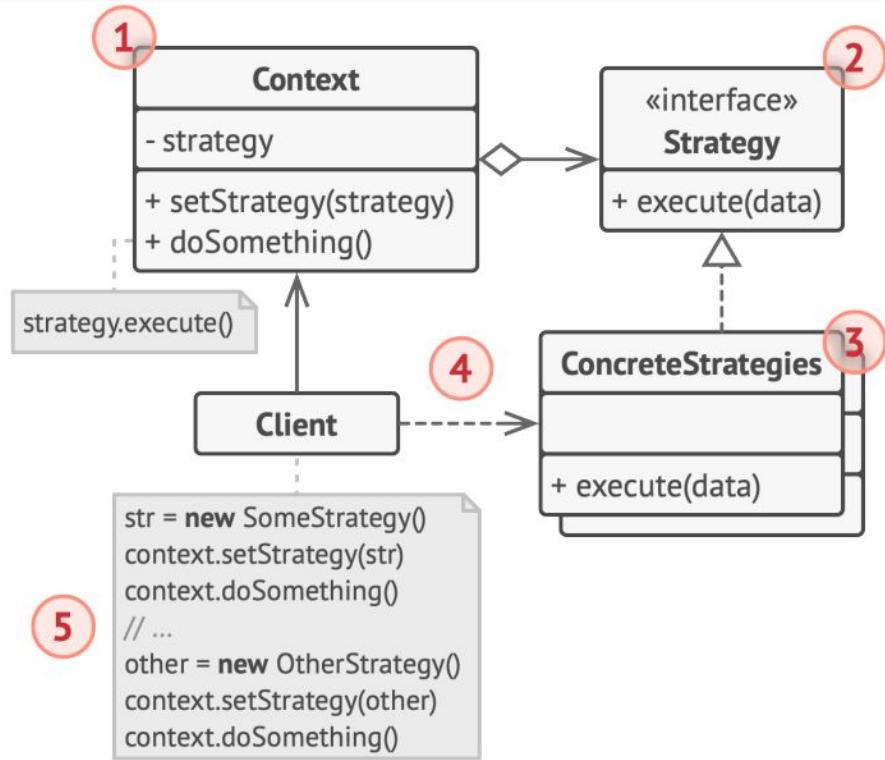


- La classe originale (le **contexte**) doit avoir un **attribut** qui garde une référence vers une des stratégies. Plutôt que de s'occuper de la tâche, le contexte la **délègue** à l'objet stratégie associé.
- Le contexte n'a pas la responsabilité de la sélection de l'algorithme adapté, c'est le client qui lui envoie la stratégie.
- Le contexte devient **indépendant** des stratégies concrètes. Vous pouvez ainsi modifier des algorithmes ou en ajouter de nouveaux sans toucher au code du contexte ou aux autres stratégies.

**Raison d'utilisation** : un objet doit pouvoir faire varier une partie de son algorithme dynamiquement.

**Résultat** : le patron stratégie permet d'isoler les algorithmes appartenant à une même famille d'algorithmes, mais vous créez beaucoup d'objets en mémoire

# Strategy - Structure

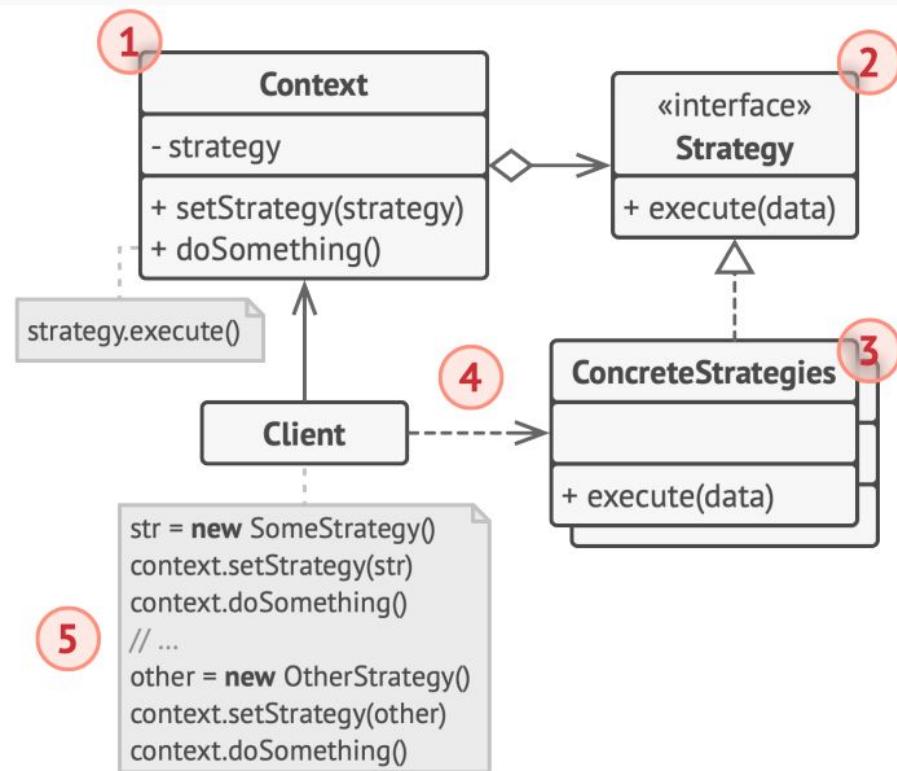


1. Le **Contexte** garde une référence vers une des stratégies concrètes et communique avec cet objet uniquement au travers de l'**interface stratégie**.
2. L'interface **Stratégie** est commune à toutes les **stratégies concrètes**. Elle déclare une méthode que le contexte utilise pour exécuter une stratégie.
3. Les **Stratégies Concrètes** implémentent différentes **variantes** d'algorithme utilisées par le contexte.
4. Chaque fois qu'il veut lancer un algorithme, le contexte appelle la méthode d'exécution de l'objet stratégie associé. Le contexte ne sait pas comment la stratégie fonctionne ni comment l'algorithme est lancé.
5. Le **Client** crée un **objet spécifique Stratégie** et le passe au contexte. Le contexte expose un **setter** qui permet aux clients de remplacer la stratégie associée au contexte lors de l'exécution.

# Strategy - Exemple avec les opérations arithmétiques

```
1 // L'interface stratégie déclare les traitements communs à
2 // toutes les versions supportées de l'algorithme. Le contexte
3 // utilise cette interface pour appeler l'algorithme défini par
4 // les stratégies concrètes.
5 interface Strategy is
6     method execute(a, b)
7
8     // Les stratégies concrètes implémentent l'interface de base
9     // Stratégie et hébergent l'algorithme. L'interface les rend
10    // interchangeables dans le contexte.
11 class ConcreteStrategyAdd implements Strategy is
12     method execute(a, b) is
13         return a + b
14
15 class ConcreteStrategySubtract implements Strategy is
16     method execute(a, b) is
17         return a - b
18
19 class ConcreteStrategyMultiply implements Strategy is
20     method execute(a, b) is
21         return a * b
22
23 // Le contexte définit l'interface dont les clients ont besoin.
24 class Context is
25     // Le contexte maintient une référence à l'un des objets
26     // Stratégie. Le contexte ne connaît pas la classe concrète
27     // de la stratégie. Il doit manipuler toutes les stratégies
28     // via l'interface stratégie.
29     private strategy: Strategy
30
31     // En général, le contexte accepte une stratégie en
32     // paramètre du constructeur et fournit un setter pour
33     // permettre de changer de stratégie lors de l'exécution.
34     method setStrategy(Strategy strategy) is
35         this.strategy = strategy
36
37     // Le contexte délègue certaines tâches à l'objet stratégie,
38     // plutôt que d'implémenter plusieurs versions de
39     // l'algorithme.
40     method executeStrategy(int a, int b) is
41         return strategy.execute(a, b)
```

# Strategy - Exemple avec les opérations arithmétiques



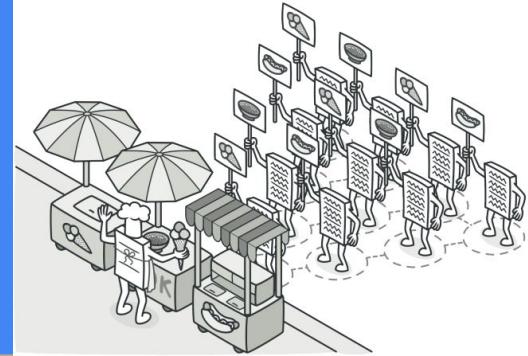
```
44 // Le code client choisit une stratégie concrète et la passe au
45 // contexte. Le client doit connaitre les différences entre les
46 // stratégies afin de faire le bon choix.
47 class ExampleApplication is
48   method main() is
49     Create context object.
50
51     Read first number.
52     Read last number.
53     Read the desired action from user input.
54
55     if (action == addition) then
56       context.setStrategy(new ConcreteStrategyAdd())
57
58     if (action == subtraction) then
59       context.setStrategy(new ConcreteStrategySubtract())
60
61     if (action == multiplication) then
62       context.setStrategy(new ConcreteStrategyMultiply())
63
64     result = context.executeStrategy(First number, Second number)
65
66     Print result.
```

# Strategy - Possibilités d'application

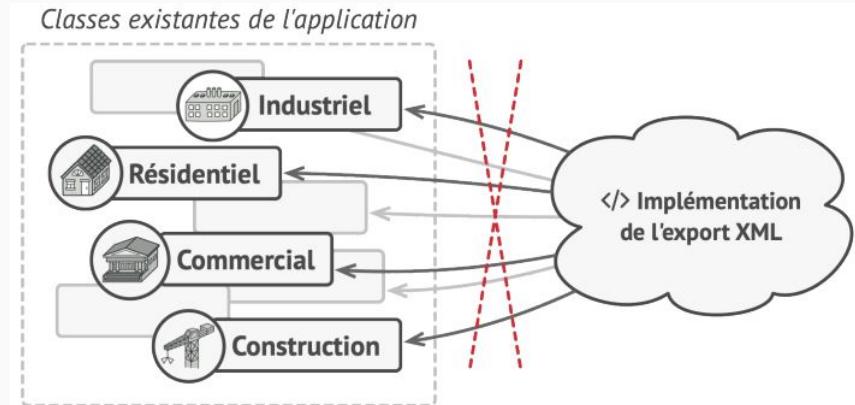
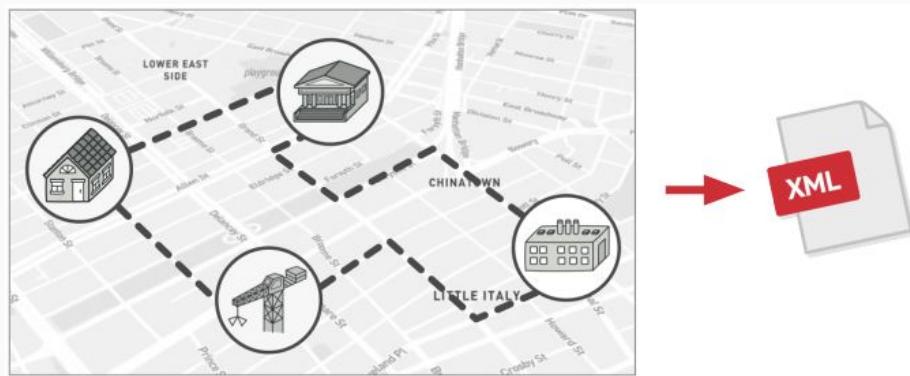
- Utilisez le Stratégie si vous voulez avoir **différentes variantes** d'un algorithme à l'intérieur d'un objet à disposition, et pouvoir **passer d'un algorithme à l'autre** lors de l'**exécution**.
- Utilisez la stratégie pour **isoler la logique métier** d'une classe, **de l'implémentation des algorithmes** dont les détails ne sont pas forcément importants pour le contexte.
- Utilisez ce patron si votre classe possède **un gros bloc conditionnel** qui choisit entre différentes variantes du même algorithme.
- Exercice : <https://www.wooclap.com/EPIRIZ>



# Visitor



Visiteur vous permet de séparer les algorithmes et les objets sur lesquels ils opèrent.



La méthode d'export XML devait être ajoutée dans toutes les classes noeud, ce qui risquait de compromettre l'intégrité du code de l'application.

# Visitor

- Le patron de conception visiteur vous propose de placer ce nouveau comportement dans une classe séparée que l'on appelle **visiteur**, plutôt que de l'intégrer dans des classes existantes.
- L'objet qui devait lancer ce traitement à l'origine est maintenant passé en **paramètre** des méthodes du visiteur, ce qui permet à la méthode d'avoir accès à toutes les données nécessaires qui se trouvent à l'intérieur de l'objet.

Comment fait-on pour que ce comportement puisse être exécuté sur des objets de différentes classes ?

# Visitor

La classe visiteur va donc avoir besoin d'un **ensemble de méthodes** et chacune d'entre elles pourra prendre des **paramètres de différents types**, comme ce qui suit :

```
1  class ExportVisitor implements Visitor is
2      method doForCity(City c) { ... }
3      method doForIndustry(Industry f) { ... }
4      method doForSightSeeing(SightSeeing ss) { ... }
5      // ...
```

Mais comment allons-nous appeler ces méthodes, surtout celles qui gèrent le graphe complet ?

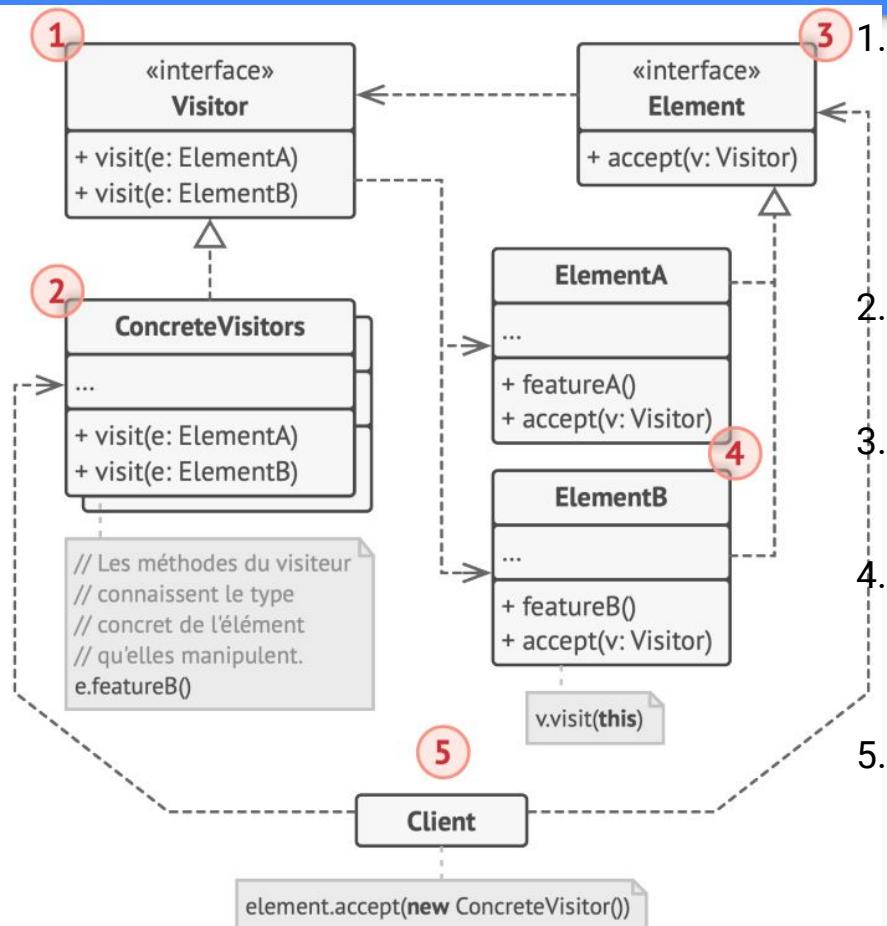
```
1  foreach (Node node in graph)
2      if (node instanceof City)
3          exportVisitor.doForCity((City) node)
4      if (node instanceof Industry)
5          exportVisitor.doForIndustry((Industry) node)
6      // ...
7 }
```

# Visitor

Le patron Visiteur utilise une technique appelée **double répartition** (double dispatch), qui aide à lancer la bonne méthode sans s'encombrer avec des blocs conditionnels.

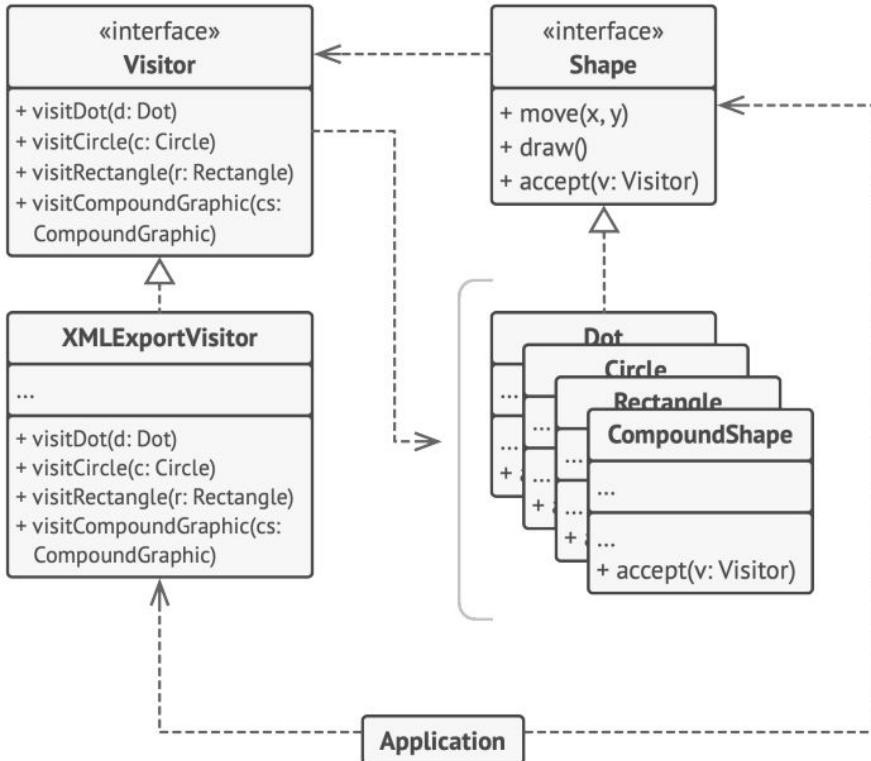
```
1 // Code client
2 foreach (Node node in graph)
3     node.accept(exportVisitor)
4
5 // City
6 class City is
7     method accept(Visitor v) is
8         v.doForCity(this)
9         // ...
10
11 // Industry
12 class Industry is
13     method accept(Visitor v) is
14         v.doForIndustry(this)
```

# Visitor - Structure



1. L'interface **Visiteur** déclare un **ensemble de méthodes** de parcours qui peuvent prendre les éléments concrets d'une structure d'objets en paramètre. Ces méthodes peuvent avoir le même nom si le programme est écrit dans un langage qui gère la surcharge, mais le type de ses **paramètres** sera **different**.
2. Chaque **Visiteur Concret** implémente plusieurs versions des mêmes comportements, en fonction des classes des éléments concrets.
3. L'interface **Élément** déclare une méthode qui « **accepte** » les visiteurs. Cette méthode déclare un **paramètre** du type de l'interface **visiteur**.
4. Chaque **Élément Concret** doit implémenter une méthode d'acceptation. Le but de cette méthode est de rediriger l'appel vers la méthode appropriée du visiteur en fonction de la **classe de l'élément actuel**.
5. Le **Client** représente en général une collection ou tout autre objet complexe (par exemple un arbre Composite). En général, les clients n'ont pas de visibilité sur les classes des éléments concrets, car ils manipulent les objets de cette collection via une interface abstraite.

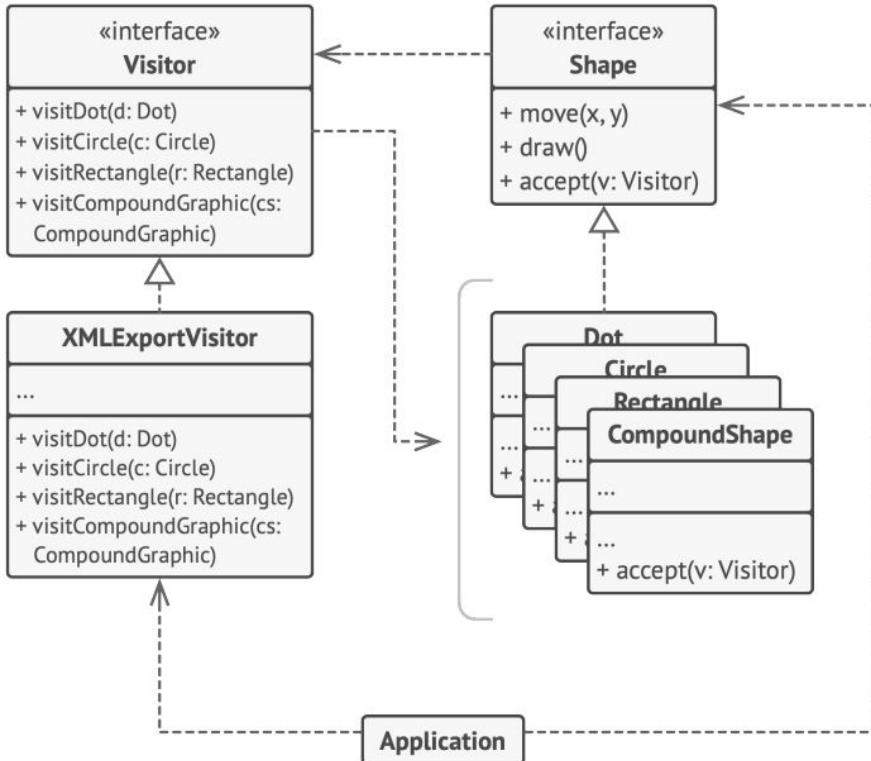
# Visitor - Exemple avec l'export XML



Exporter différents types d'objets vers le format XML à l'aide d'un objet visiteur.

```
1 // L'interface élément déclare une méthode `accepter` qui prend
2 // l'interface de base visiteur en argument.
3 interface Shape is
4     method move(x, y)
5     method draw()
6     method accept(v: Visitor)
7
8 // Chaque classe concrète Élément doit implémenter la méthode
9 // `accepter` et la faire appeler la méthode du visiteur qui
10 // correspond à la classe de l'élément.
11 class Dot implements Shape is
```

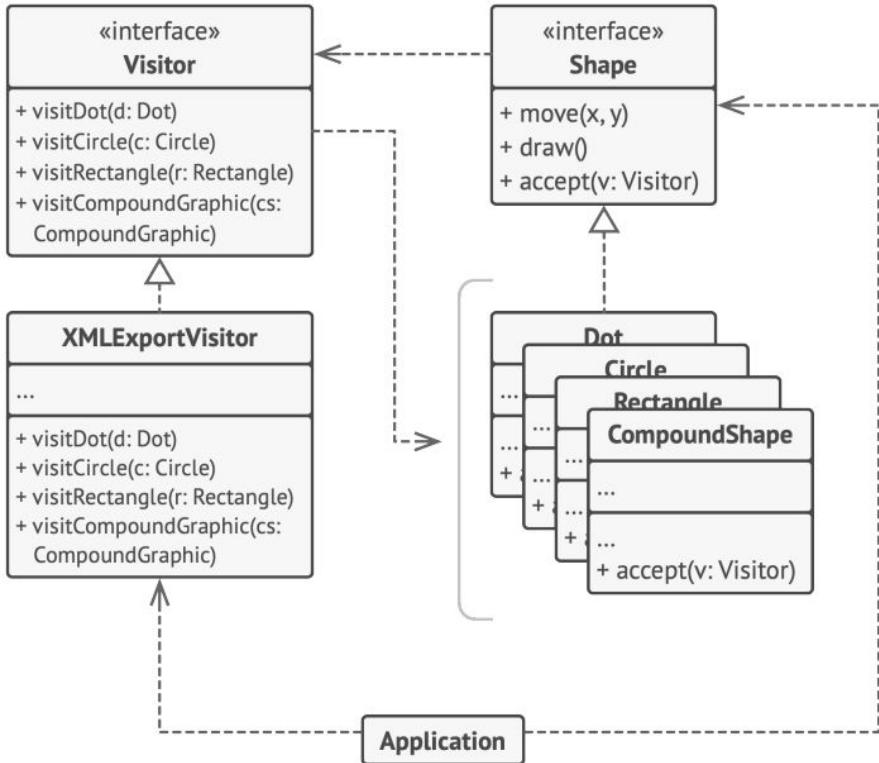
# Visitor - Exemple avec l'export XML



Exporter différents types d'objets vers le format XML à l'aide d'un objet visiteur.

```
14 // Vous remarquerez que nous appelons `visiterPoint`, ce qui
15 // correspond au nom de la classe actuelle. Ainsi, nous
16 // fournissons le nom de la classe de l'élément au visiteur
17 // qui le manipule.
18 method accept(v: Visitor) is
19     v.visitDot(this)
20
21 class Circle implements Shape is
22     // ...
23     method accept(v: Visitor) is
24         v.visitCircle(this)
25
26 class Rectangle implements Shape is
27     // ...
28     method accept(v: Visitor) is
29         v.visitRectangle(this)
30
31 class CompoundShape implements Shape is
32     // ...
33     method accept(v: Visitor) is
34         v.visitCompoundShape(this)
```

# Visitor - Exemple avec l'export XML



Exporter différents types d'objets vers le format XML à l'aide d'un objet visiteur.

```
37 // L'interface visiteur déclare un ensemble de méthodes visiteur
38 // qui correspondent aux classes élément. La signature de la
39 // méthode visiter permet au visiteur d'identifier la classe
40 // exacte de l'élément qu'il manipule.

41 interface Visitor is
42     method visitDot(d: Dot)
43     method visitCircle(c: Circle)
44     method visitRectangle(r: Rectangle)
45     method visitCompoundShape(cs: CompoundShape)

46

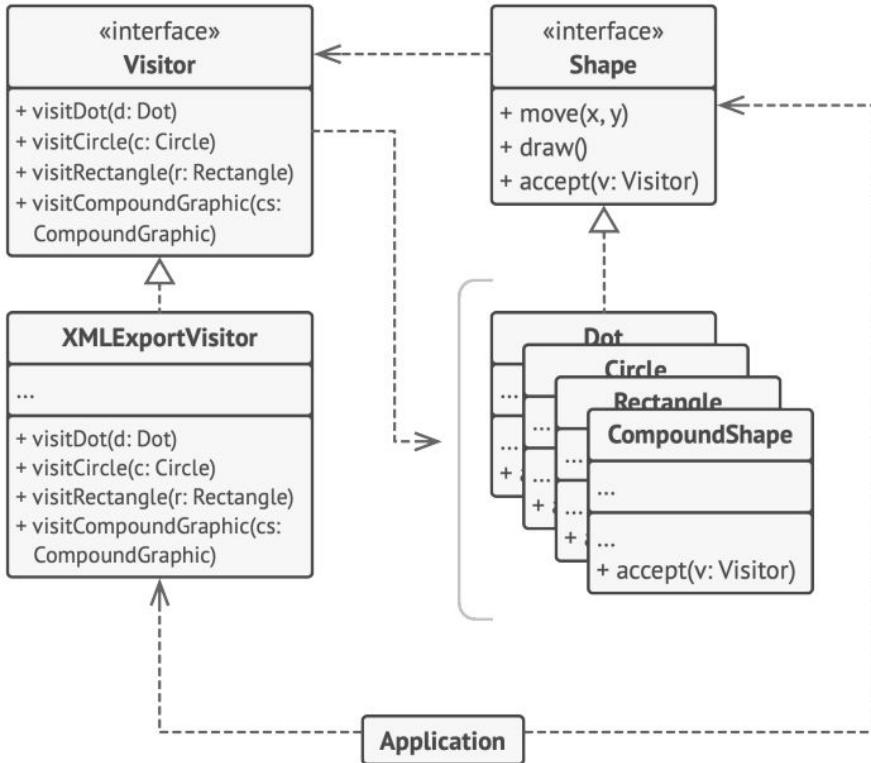
47 // Les visiteurs concrets implémentent plusieurs versions du
48 // même algorithme. Ce dernier fonctionne avec toutes les
49 // classes concrètes Élément.
50 //
51 // Vous tirez tous les bénéfices du patron de conception
52 // visiteur lorsque vous l'utilisez avec une structure complexe
53 // d'objets, comme une arborescence. Dans ce cas, il peut être
54 // pratique de stocker certains états intermédiaires de
55 // l'algorithme tout en exécutant les méthodes du visiteur sur
56 // les différents objets de la structure.

57 class XMLExportVisitor implements Visitor is
58     method visitDot(d: Dot) is
59         // Exporte l'ID du point et ses coordonnées.

60

61     method visitCircle(c: Circle) is
62         // Exporte l'ID du cercle, les coordonnées de son centre
63         // et son rayon.
```

# Visitor - Exemple avec l'export XML



Exporter différents types d'objets vers le format XML à l'aide d'un objet visiteur.

```
65  method visitRectangle(r: Rectangle) is
66      // Exporte l'ID du rectangle, les coordonnées du point
67      // supérieur gauche, ainsi que sa largeur et sa
68      // longueur.
69
70  method visitCompoundShape(cs: CompoundShape) is
71      // Exporte l'ID de la forme, ainsi qu'une liste des ID
72      // de ses enfants.
73
74
75  // Le code client peut lancer des traitements du visiteur sur
76  // n'importe quel ensemble d'éléments sans connaître leurs
77  // classes concrètes. La méthode accepter envoie un appel au
78  // traitement approprié de l'objet visiteur.
79 class Application is
80     field allShapes: array of Shapes
81
82     method export() is
83         exportVisitor = new XMLExportVisitor()
84
85         foreach (shape in allShapes) do
86             shape.accept(exportVisitor)
```

## Visitor - Possibilités d'application

- Utilisez le visiteur lorsque vous voulez lancer des traitements sur les éléments d'un objet ayant une **structure complexe** (une arborescence par exemple).
- Visiteur vous permet de lancer des traitements sur un ensemble d'objets de différentes classes à l'aide d'un objet visiteur qui implémente **une variante d'un même traitement** pour chaque classe visée.
- Utilisez le visiteur si un comportement n'est adapté que pour certaines classes d'une hiérarchie de classes, mais pas pour les autres.
- Exercice : <https://www.wooclap.com/EPIRIZ>



## GoF Design Patterns Classified (F. Tip)

	Purpose		
	<i>Creational</i>	<i>Structural</i>	<i>Behavioral</i>
<i>Class</i>	<i>Factory Method</i>	<i>Adapter (class)</i>	<i>Interpreter</i> <i>Template Method</i>
<i>Object</i>	<i>Abstract Factory</i> <i>Builder</i> <i>Prototype</i> <i>Singleton</i> ✓	<i>Adapter (object)</i> <i>Bridge</i> <i>Composite</i> ✓ <i>Decorator</i> <i>Facade</i> <i>Flyweight</i> <i>Proxy</i>	<i>Chain of Resp.</i> <i>Command</i> <i>Iterator</i> <i>Mediator</i> <i>Memento</i> <i>Observer</i> ✓ <i>State</i> <i>Strategy</i> ✓ <i>Visitor</i> ✓

Exprime un schéma d'organisation structurel fondamental pour un système logiciel

Fournit un schéma pour affiner les composants d'un système logiciel ou les relations entre eux.

Architecture Patterns	Design Patterns
Layers	Whole-Part
Pipes & Filters	Master-Slave
Blackboard	Proxy
Broker	Command Processor
<b>Model-View-Controller</b> ✓	View handler
Presentation-Abstraction-Control	Forwarded-Receiver
Microkernel	Client-Dispatcher-Server
Reflection	Publisher-Subscriber

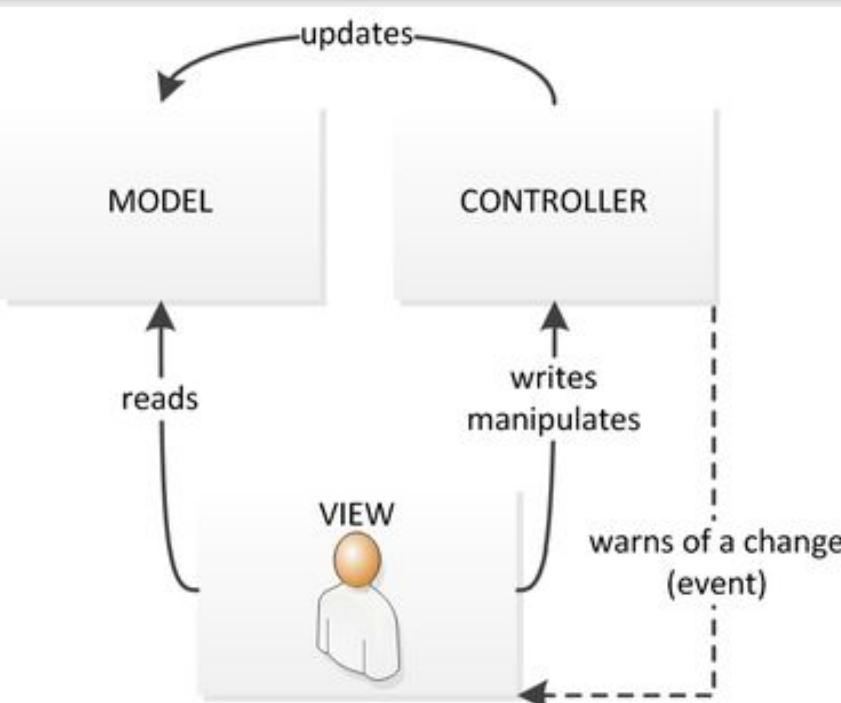
# POSA1

## Patrons d'architecture

exprime un schéma d'organisation  
structurel fondamental pour un système  
logiciel

- Layers
- Pipes & Filters
- Blackboard
- Broker
- Model-View-Controller
- Presentation-Abstraction-Control
- Microkernel
- Reflection

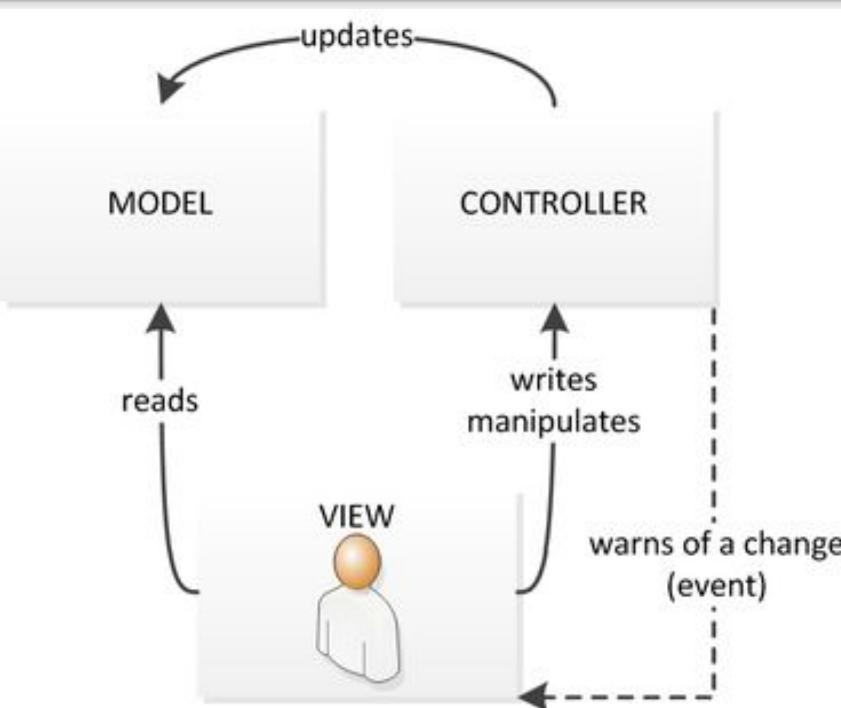
# Model-View-Controller (MVC)



- Un **modèle** (Model) contient les **données** à afficher.
- Une **vue** (View) contient la **présentation** de l'interface graphique.
- Un **contrôleur** (Controller) contient la **logique** concernant les **actions** effectuées par l'utilisateur et **coordonne** les **interactions** entre la vue et le modèle.

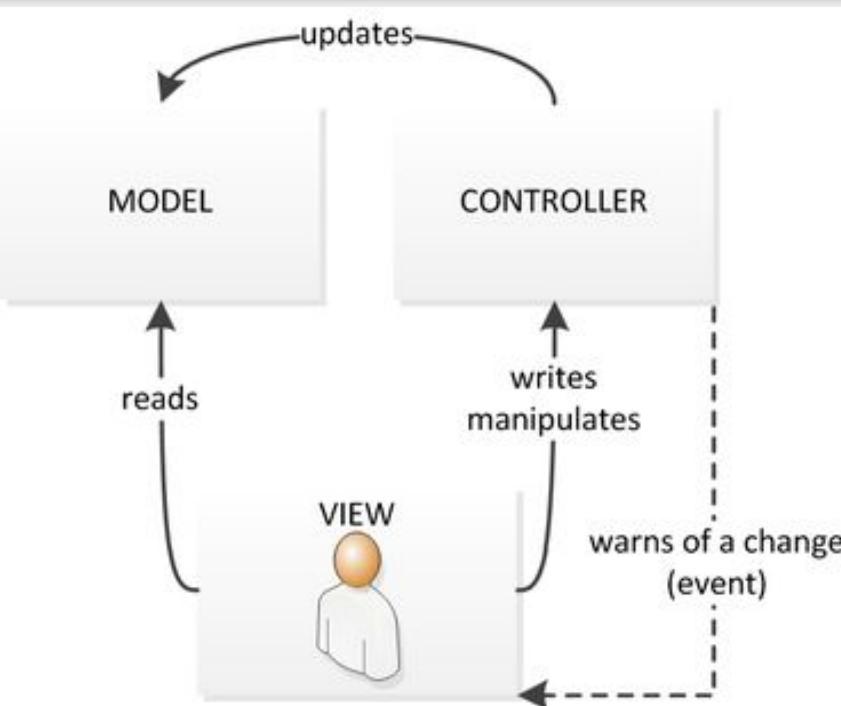
MVC a été mise au point en 1979 par Trygve Reenskaug, qui travaillait alors sur Smalltalk dans les laboratoires de recherche Xerox PARC

# Model-View-Controller (MVC)



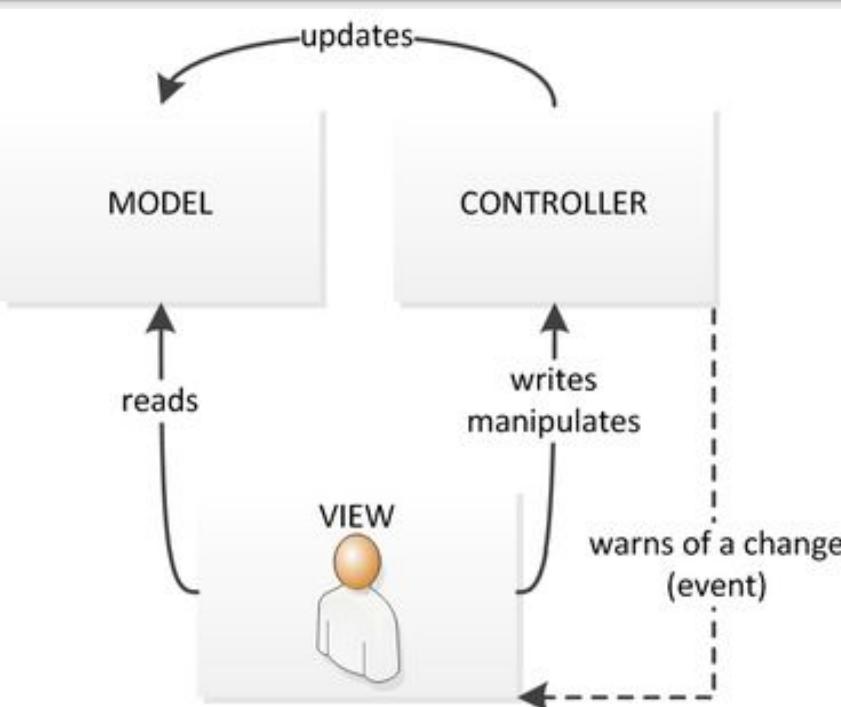
La **vue** correspond à l'interface avec laquelle l'utilisateur interagit. Sa première tâche est de **présenter les résultats** renvoyés par le modèle. Sa seconde tâche est de **recevoir toutes les actions de l'utilisateur** (clic de souris, sélection d'une entrée, boutons...). Ces différents événements sont **envoyés au contrôleur**. La vue n'effectue aucun **traitement**, elle se contente d'**afficher** les résultats des traitements effectués par le modèle. Plusieurs vues, partielles ou non, peuvent afficher des informations d'un même modèle.

# Model-View-Controller (MVC)



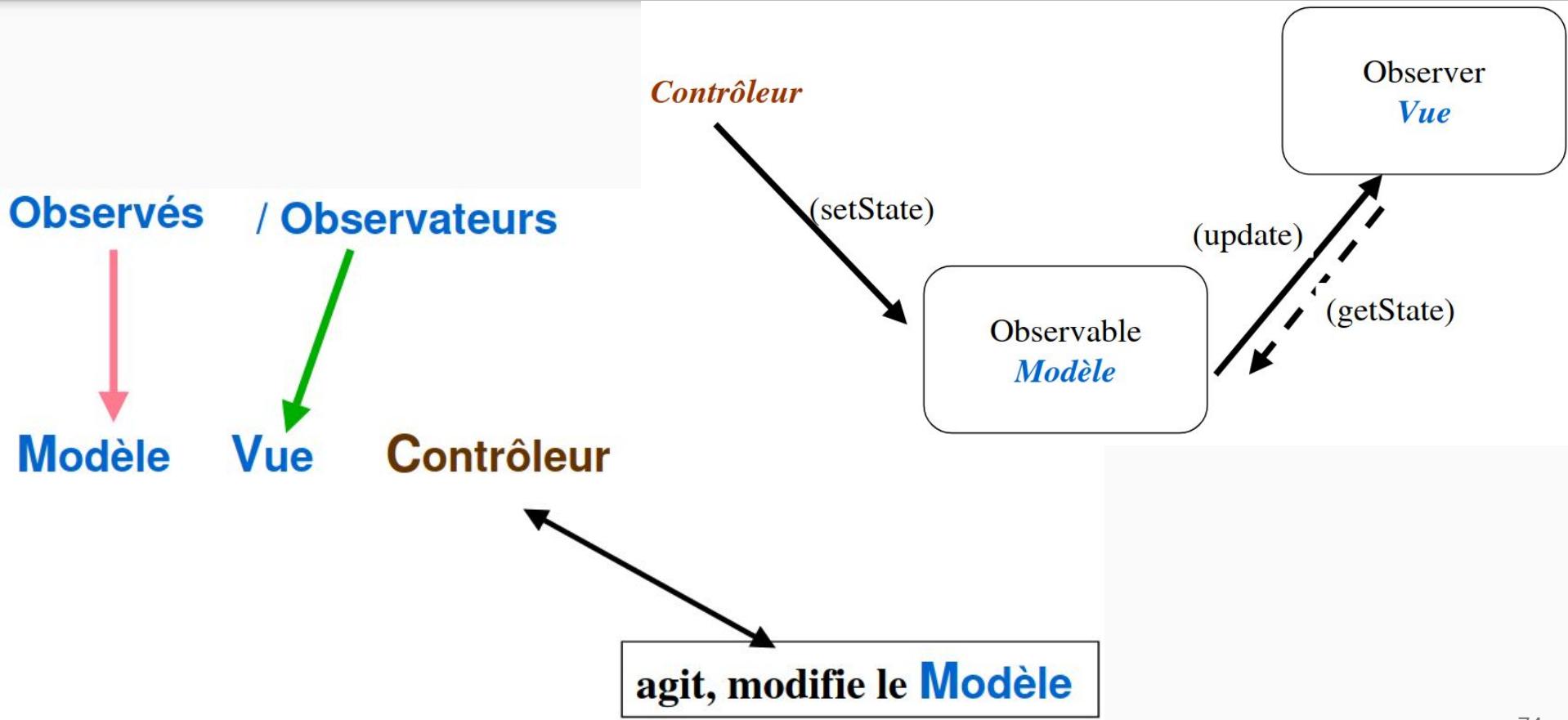
Le **modèle** représente le comportement de l'application : traitements des **données**, **interactions** avec la base de données, etc. Il décrit ou contient les **données** manipulées par l'application. Il assure la **gestion** de ces données et garantit leur **intégrité**. Dans le cas typique d'une base de données, c'est le modèle qui la contient. Le modèle offre des **méthodes** pour **mettre à jour** ces données (insertion, suppression, changement de valeur). Il offre aussi des méthodes pour **récupérer** ces données. Les résultats renvoyés par le modèle sont dénués de toute présentation.

# Model-View-Controller (MVC)



Le **contrôleur** prend en charge la **gestion des événements de synchronisation** pour mettre à jour la vue ou le modèle et les synchroniser. Il reçoit tous les événements de l'**utilisateur** et enclenche les actions à effectuer. Si une action nécessite un changement des données, le contrôleur demande la modification des données au modèle et ensuite avertit la vue que les données ont changé pour qu'elle se mette à jour. Certains événements de l'utilisateur ne concernent pas les données mais la vue. Dans ce cas, le contrôleur demande à la vue de se modifier. Le contrôleur n'effectue aucun traitement, ne modifie aucune donnée.

MVC : Observer est inclus



# Principles SOLID

1. **Single** Responsibility Principle
2. **Open/Closed** Principle
3. **Liskov** Substitution Principle
4. **Interface** Segregation Principle
5. **Dependency** Inversion Principle

## Single Responsibility Principle

Une classe ne devrait être modifiée que pour **une seule raison**.

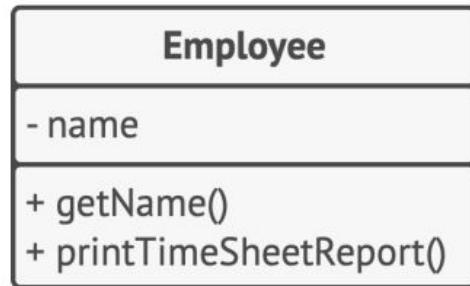
Essayez de faire en sorte qu'une classe ne soit **responsable** que d'**une partie d'une fonctionnalité** de votre logiciel, et **encapsuler** (vous pouvez aussi dire cachez à l'intérieur) cette responsabilité entièrement dans la classe.

**Objectif** : diminuer la complexité

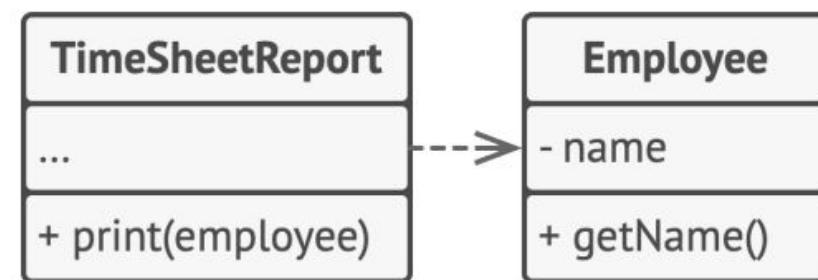
Si une classe s'occupe de trop de choses à la fois, vous devrez la **modifier à la moindre évolution -> provoquer des bugs**.

Si vous éprouvez des **difficultés** à vous **concentrer** sur des **aspects spécifiques** du programme, souvenez-vous de ce principe et vérifiez s'il est grand temps de **découper certaines classes en plusieurs parties**.

# Single Responsibility Principle - Example



AVANT : la classe contient plusieurs comportements différents.



APRÈS : le comportement supplémentaire se retrouve dans sa propre classe.

## Open/Closed Principle

Les classes doivent être **ouvertes à l'extension**, mais **fermées à la modification** (sauf s'il y a un bug).

L'idée est d'**éviter** de créer des **bugs** dans du code existant lorsque vous ajoutez de nouvelles fonctionnalités.

Une classe est **ouverte** si vous pouvez l'**étendre**, créer une sous-classe ou faire n'importe quoi d'autre avec ; Une classe est **fermée** si elle est **prête** à 100 % à **être utilisée par d'autres** classes (son **interface** est clairement définie et **ne sera plus modifiée** dans le futur).

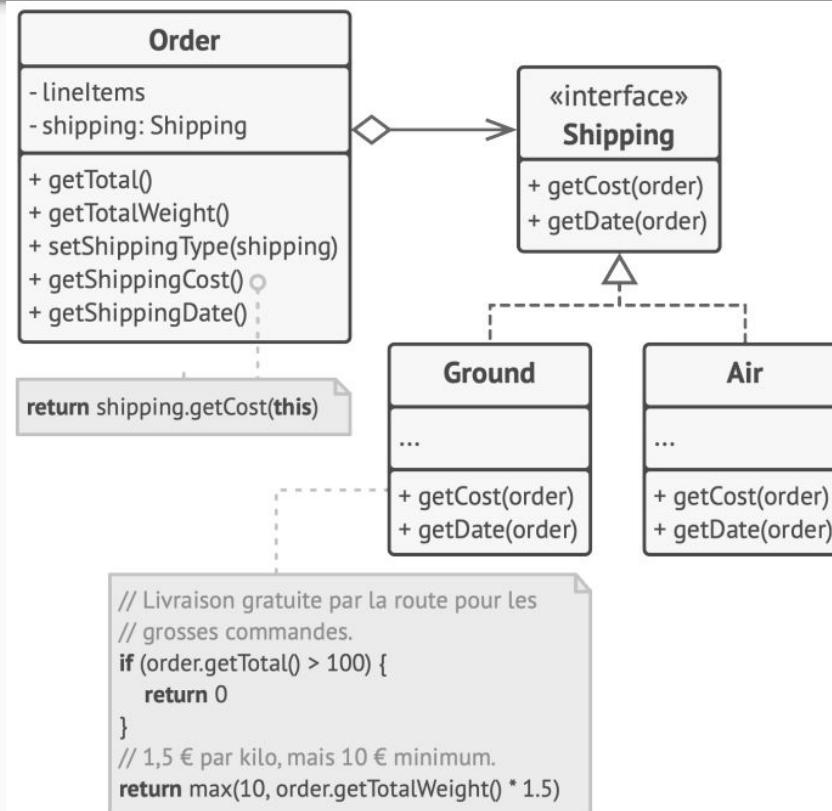
## Open/Closed Principle - Example



```
if (shipping == "ground") {  
    // Livraison gratuite par la route pour les  
    // grosses commandes.  
    if (getTotal() > 100) {  
        return 0  
    }  
    // 1,5 € par kilo, mais 10 € minimum.  
    return max(10, getTotalWeight() * 1.5)  
}  
  
if (shipping == "air") {  
    // 3 € par kilo, mais 20 € minimum.  
    return max(20, getTotalWeight() * 3)  
}
```

*AVANT : vous avez modifié la classe **Commande** chaque fois que vous avez ajouté une méthode d'expédition à l'application.*

# Open/Closed Principle - Example



Stratégie

En concordance avec le **principe de responsabilité unique**, cette solution vous permet de déplacer le calcul du délai de livraison dans des **classes plus pertinentes**.

*APRÈS : ajouter une nouvelle méthode d'expédition ne vous oblige pas à effectuer des modifications dans les classes existantes.*

## Liskov Substitution Principle

Lorsque vous **étendez une classe**, rappelez-vous que vous devez être en mesure de **passer des objets de la sous-classe à la place des objets de la classe mère sans faire planter le code.**

Cela signifie que les sous-classes restent **compatibles** avec le comportement de la classe mère. Lorsque vous redéfinissez une méthode, **étendez le comportement** de base plutôt que de le remplacer complètement avec autre chose.

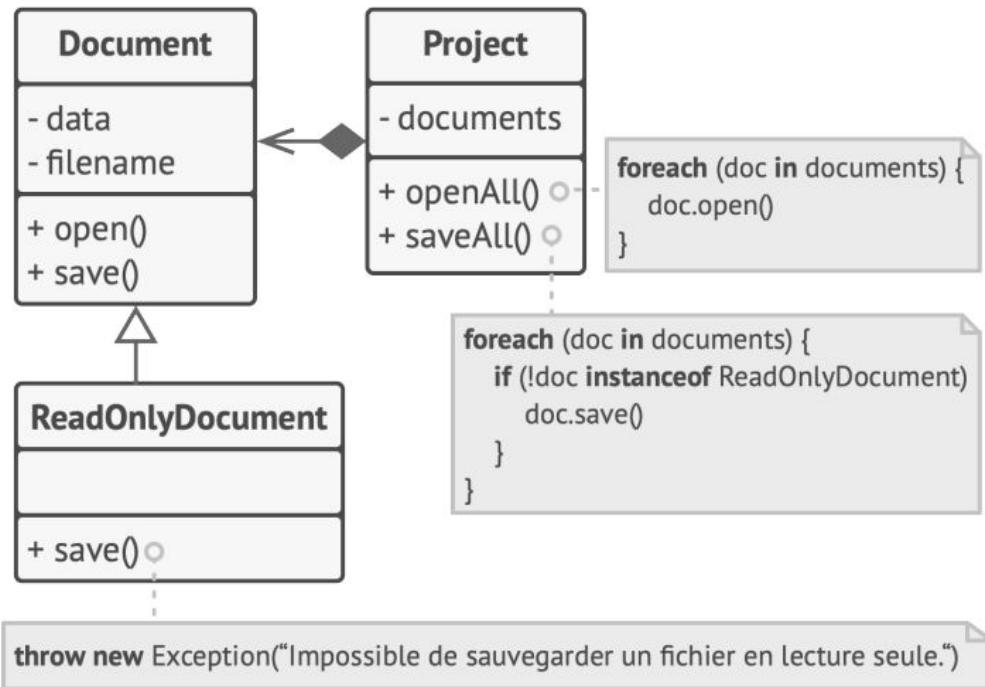
## Liskov Substitution Principle : prérequis des sous-classes

- Les **types des paramètres** de la **méthode d'une sous-classe** doivent correspondre ou être **plus abstraits** que les **types des paramètres** dans la **méthode de la classe mère** :
  - classe mère : nourrir(Chat c)
  - sous-classe : nourrir(Animal c) ? ou nourrir(ChatBengal c) ?
- Le **type de retour** dans une **méthode d'une sous-classe** doit correspondre ou être un **sous-type du type de retour de la méthode de la classe mère** :
  - classe mère : acheterChat(): Chat
  - sous-classe : acheterChat(): ChatBengal ? ou acheterChat(): Animal ?
- Une **méthode** dans une **sous-classe** ne devrait **pas lever les types d'exceptions** que la méthode de base n'est pas censée lever.

## Liskov Substitution Principle : prérequis des sous-classes

- Une **sous-classe** ne devrait **pas renforcer des préconditions**.
  - classe mère : int somme(int nbre1, int nbre2)
  - sous-classe : int somme(int nbre1, int nbre2) à condition que nbre1 et nbre2 soient positives (en levant une exception si la valeur est négative)
- Une **sous-classe** ne devrait **pas affaiblir des postconditions**.
  - classe mère : une méthode pour connecter à une bdd. Cette méthode est censée toujours fermer les connexions ouvertes à la bdd lorsqu'elle renvoie une valeur.
  - sous-classe : laisser les connexions ouvertes, alors que le client s'attend à ce que toutes les méthodes ferment les connexions, alors il pourrait très bien décider de fermer le programme juste après avoir appelé la méthode, polluant le système avec une connexion fantôme à la base de données.
- Les **invariants** d'une classe mère doivent être **préservés**.
- Une sous-classe ne doit **jamais modifier les valeurs des attributs privés** d'une classe mère.

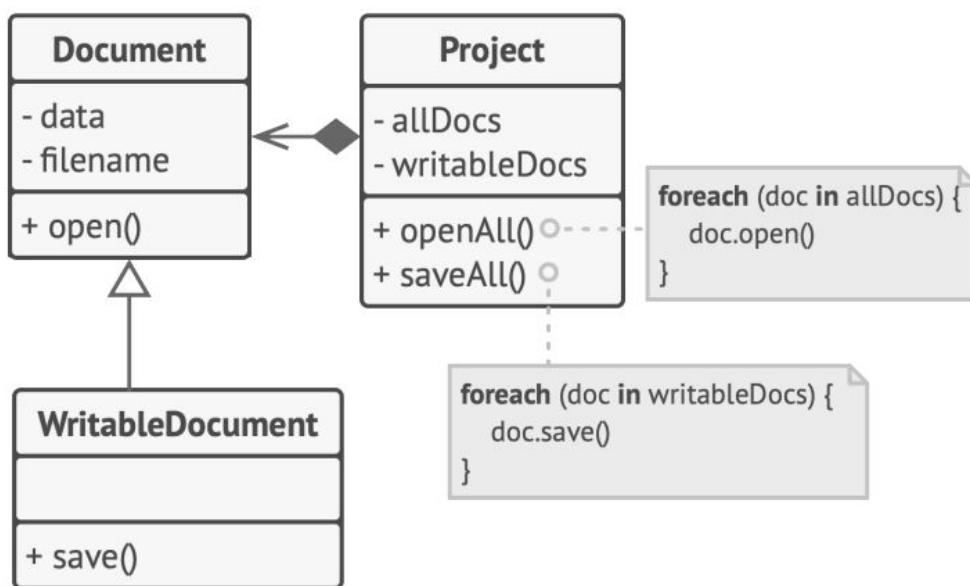
# Liskov Substitution Principle - Exemple



La méthode de la sous-classe **DocumentEnLectureSeule** lève une exception si elle est appelée. La méthode de base ne possède pas cette restriction.

*AVANT : cela n'a aucun sens de vouloir sauvegarder un document en lecture seule, la sous-classe tente donc de résoudre cette situation en réinitialisant le comportement de base de la méthode redéfinie.*

# Liskov Substitution Principle - Exemple



Une sous-classe peut étendre le comportement d'une classe mère. Le document en lecture seule devient donc la classe de base de la hiérarchie. Le document ouvert en écriture est maintenant une sous-classe qui étend la classe de base et ajoute le comportement de sauvegarde.

*APRÈS : le problème est résolu en faisant de la classe documentEnLectureSeule, la classe de base de la hiérarchie.*

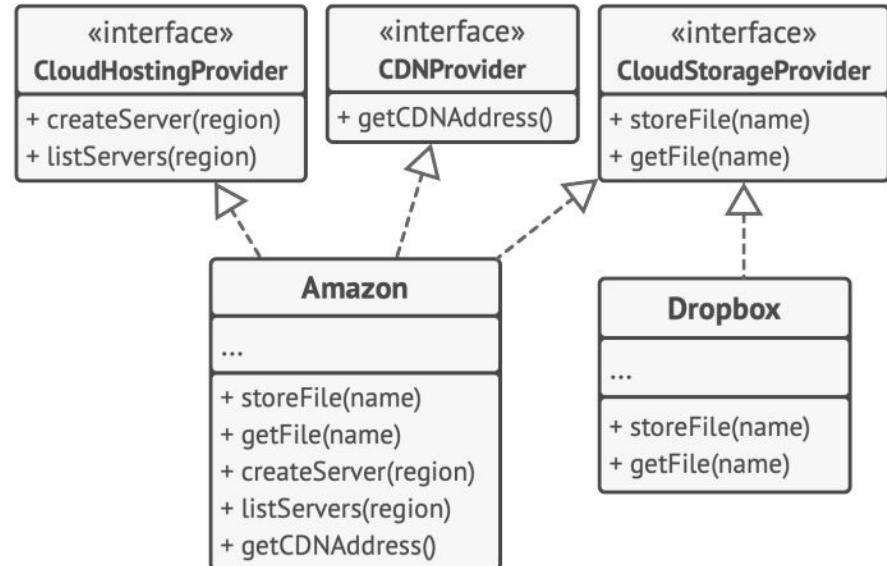
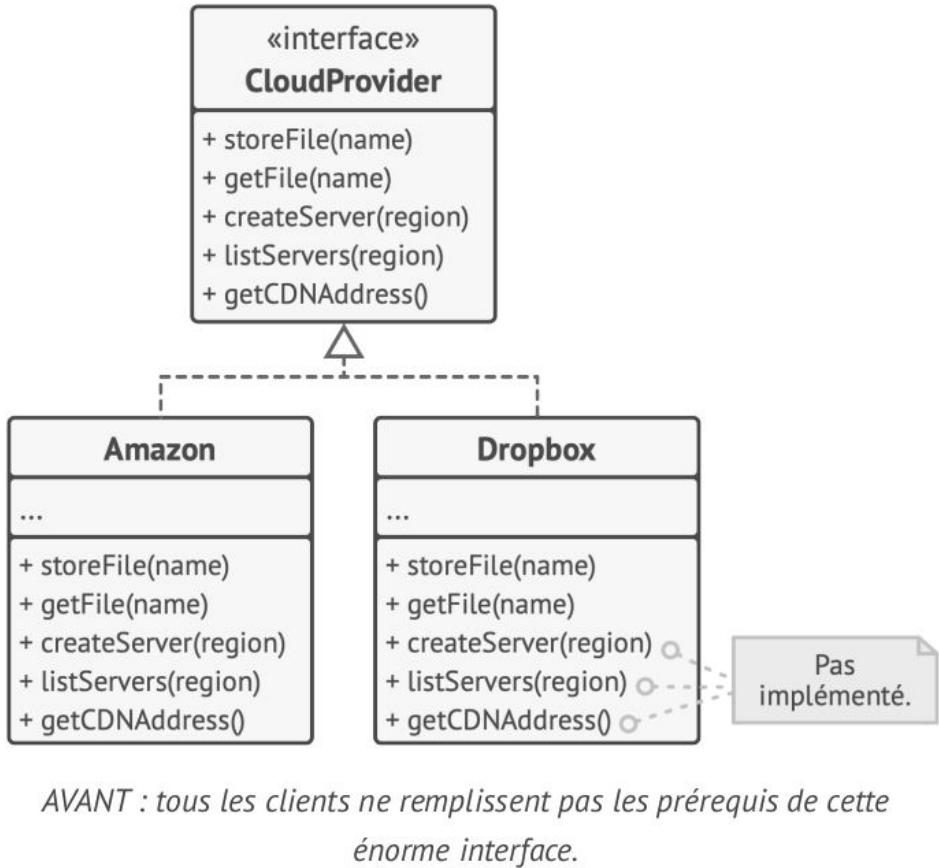
## Interface Segregation Principle

Les clients ne devraient pas être forcés à dépendre de méthodes qu'ils n'utilisent pas.

Essayez de rendre vos interfaces aussi **étroites** que possible afin de ne pas obliger les classes des clients à implémenter des comportements dont elles n'ont pas besoin -> découper vos « grosses » interfaces pour les rendre plus **précises et spécifiques**.

Les clients ne doivent implémenter que les méthodes dont ils ont **vraiment besoin**.  
Sinon, une modification apportée dans une « grosse » interface va même faire planter les clients qui n'utilisent pas les méthodes modifiées.

# Interface Segregation Principle - Exemple



*APRÈS : une interface géante est découpée en petites interfaces plus spécifiques.*

*Gardez un certain équilibre.*

## Dependency Inversion Principle

Les classes de **haut niveau** ne devraient **pas dépendre** des classes de **bas niveau**. Elles devraient dépendre toutes les deux d'**abstractions**. **Une abstraction ne doit pas dépendre des détails**. Les détails doivent dépendre de l'abstraction.

Les **classes de bas niveau** implémentent des **traitements basiques** comme utiliser le disque, transférer des données sur un réseau, se connecter à une base de données, etc.

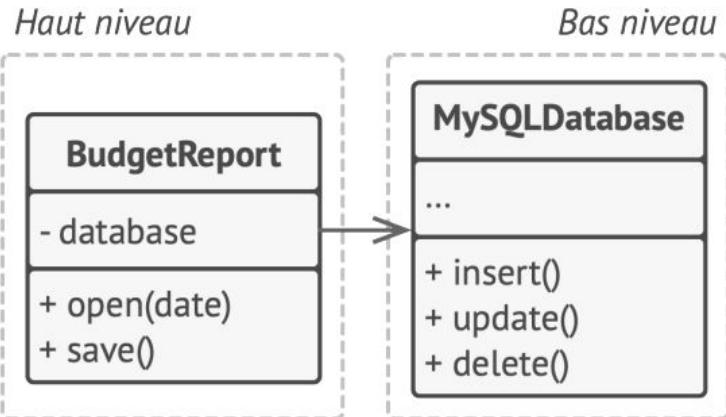
Les **classes de haut niveau** contiennent la **logique métier** qui indique aux classes de bas niveau ce qu'elles doivent faire.

Le principe d'inversion des dépendances est souvent utilisé en **corrélation** avec le **principe ouvert/fermé** : vous pouvez **étendre** les classes de **bas niveau** pour utiliser différentes classes de la logique métier sans avoir à modifier les classes existantes.

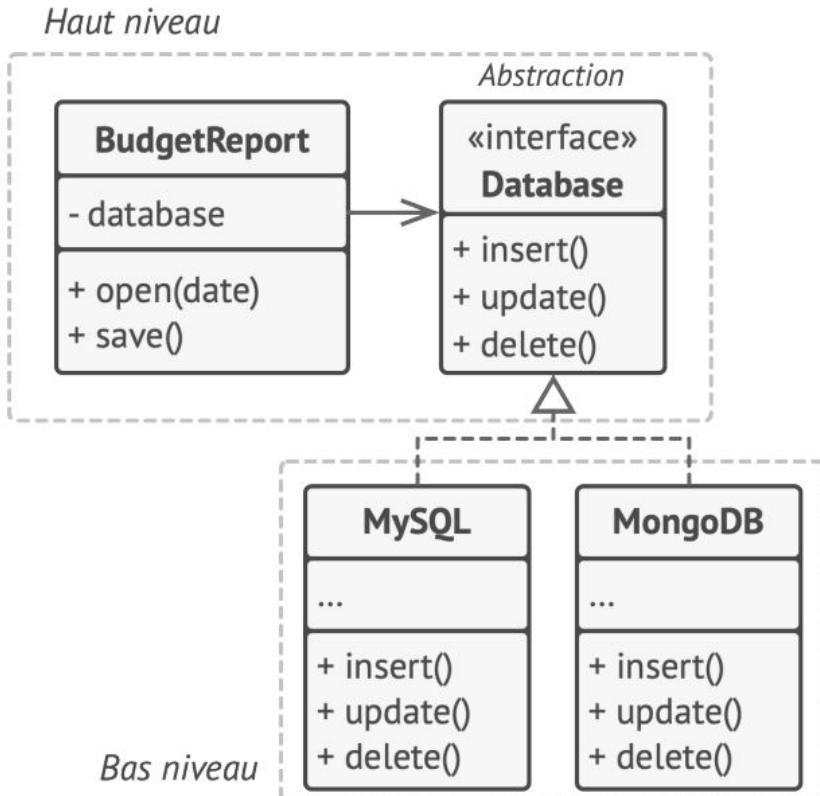
## Dependency Inversion Principle

1. Pour commencer, vous devez décrire les **interfaces** pour les traitements de bas niveau dont les classes de haut niveau vont avoir besoin, de préférence en utilisant **les termes de la logique métier**.
  - a. Par exemple, la logique métier doit appeler une méthode ouvrirRapport(fichier) plutôt qu'une suite de méthodes ouvrirFichier(x) , lireOctets(n) ,fermerFichier(x) . Ces interfaces font partie du haut niveau.
2. Maintenant, vous pouvez rendre les classes de **haut niveau dépendantes** de ces **interfaces**, plutôt que de les rendre dépendantes des classes concrètes de bas niveau. Cette dépendance sera bien plus **faible** que celle d'origine.
3. Une fois que les classes de bas niveau implémentent ces interfaces, elles deviennent **dépendantes de la logique métier**, inversant la direction de la dépendance originale.

# Dependency Inversion Principle - Exemple



AVANT : une classe de haut niveau dépend d'une classe de bas niveau.



APRÈS : les classes de bas niveau dépendent d'une abstraction de haut niveau.

# Références

1. <https://refactoring.guru/fr/design-patterns>
2. <https://springframework.guru/gang-of-four-design-patterns/>
3. <http://fr.slideshare.net/mohamedyoussfi9>
4. <http://www.newthinktank.com/videos/design-patterns-tutorial/>
5. Alexander Shvets. "Plongée au cœur des patrons de conception.", v2021-1.6.