2ème Ingéniorat en Informatique

Programmation Orientée Objet II

Chapitre I
Rappel sur la programmation orientée objet

2023/2024

Dr. HANNOUSSE Abdelhakim – hannousse.abdelhakim@univ-guelma.dz

Contenu de chapitre

- La philosophie derrière la POO
- Concepts de base
  - Classes et modificateurs d'accès
  - Objets
  - Relations entre objets
    - Association
    - Agrégation
    - Composition
    - Héritage
- Classes internes et classes anonymes
- Classes abstraites et les interfaces
- Patrons de conception
  - Principe et Origine
  - Quelques patrons utiles:
    - Singleton
    - Stratégie
    - Chaine de Responsabilité
    - Observateur
    - Adaptateur
    - Composite
    - Proxy
    - Décorateur

La philosophie derrière la POO

Evolution des paradigmes de programmation

1930

Programmation fonctionnelle

Principe: Repose sur le concept de fonctions mathématiques pures.

Impact: Émergé comme une approche pour résoudre certains problèmes de la logique mathématique.

Premier lanagage de programmation: Lisp en 1950.

1957

Programmation procedurale

Principe: Divise le programme en procédures, des blocs de code autonomes.

Impact: Améliorer la structuration du code.

Premier lanagage de programmation: Fortran en 1950.

1960
1970

Programmation par objets

Principe: Organise le code en objets qui encapsulent des données et des méthodes.

Impact: Améliorer la modularité et la réutilisabilité du code.

Premier lanagage de programmation: Simula en 1967.

1990

Programmation par aspects

Principe: Sépare les préoccupations transversales du code principal.

Impact: Résoudre les problèmes de la programmation modulaire traditionnelle.

Premier lanagage de programmation: AspectJ en 1990.

1990

Programmation par composants

Principe: Construire des logiciels à partir de composants logiciels existants autonomes et réutilisables.

Impact: Améliorer la modularité et la réutilisabilité du code.

Premier lanagage de programmation: Component Pascal en 1990.

2000

Programmation par services

Principe: Services indépendants, interagissant via des protocoles standardisés.

Impact: Implémenter les services web adoptés pour créer des applications distribuées.

Premier lanagage de programmation: Java, C#, et PHP.

2010

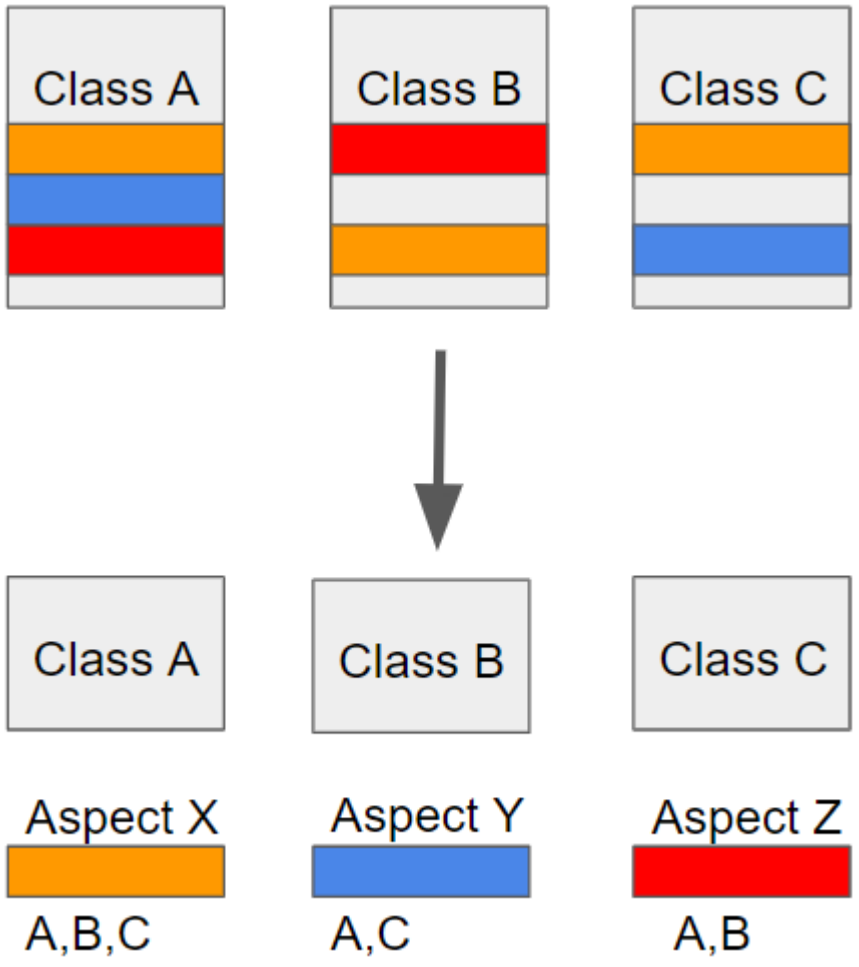
Programmation par micro-services

Principe: Découpent une application en petits services indépendants.

Impact: Réponse aux défis de l'évolution continue des applications à grande échelle.

Premier lanagage de programmation: Java, Python, Node.js, Go.

Evolution des paradigmes de programmation



1930 — Programmation fonctionnelle

Principe: Repose sur le concept de fonctions mathématiques pures.

Impact: Émergé comme une approche pour résoudre certains problèmes de la logique mathématique.

Premier lanagage de programmation: Lisp en 1950.

1957 — Programmation procedurale

Principe: Divise le programme en procédures, des blocs de code autonomes.

Impact: Améliorer la structuration du code.

Premier lanagage de programmation: Fortran en 1950.

1960 1970 — Programmation par objets

Principe: Organise le code en objets qui encapsulent des données et des méthodes.

Impact: Améliorer la modularité et la réutilisabilité du code.

Premier lanagage de programmation: Simula en 1967.

1990 — Programmation par aspects

Principe: Sépare les préoccupations transversales du code principal.

Impact: Résoudre les problèmes de la programmation modulaire traditionnelle.

Premier lanagage de programmation: AspectJ en 1990.

1990 — Programmation par composants

Principe: Construire des logiciels à partir de composants logiciels existants autonomes et réutilisables.

Impact: Améliorer la modularité et la réutilisabilité du code.

Premier lanagage de programmation: Component Pascal en 1990.

2000 — Programmation par services

Principe: Services indépendants, interagissant via des protocoles standardisés.

Impact: Implémenter les services web adoptés pour créer des applications distribuées.

Premier lanagage de programmation: Java, C#, et PHP.

2010 — Programmation par micro-services

Principe: Découpent une application en petits services indépendants.

Impact: Réponse aux défis de l'évolution continue des applications à grande échelle.

Premier lanagage de programmation: Java, Python, Node.js, Go.

Le paradigme objet

- La conception orientée objet implique la réflexion en termes d'objets et d'interactions.
  - Un système est décomposé en un ensemble d'objets interagissant entre eux.
- Les structures de données et les méthodes ne sont plus considérées comme séparées comme dans le cas de la programmation procédurale.
  - Les données nécessaires à un objet sont contenues à l'intérieur de l'objet lui-même.
- La conception orientée objet repose sur une correspondance isomorphe entre le monde réel et le monde numérique.
  - L'isomorphisme assure une cohérence entre la réalité d'un système et la façon dont elle est modélisée dans le logiciel, facilitant ainsi la compréhension et la maintenance du code.
  - Les principes tels que l'encapsulation, l'héritage et le polymorphisme sont utilisés pour capturer les similitudes et les relations entre les entités réelles dans la conception logicielle.

Processus de développement avec le paradigme objet (1/2)

- **Analyse des besoins :** Comprendre les exigences du système et identifier les différentes fonctionnalités requises.
- **Identification des objets :** Identifier et définir les objets du système, qui représentent des entités ou des concepts liés aux fonctionnalités requises.
- **Définition des méthodes de l'interface :** Élaborer les méthodes de l'interface des objets, décrivant les actions que chaque objet peut effectuer.
- **Conception de classes :** Déclarer les classes nécessaires pour mettre en œuvre les objets et leurs méthodes. Cette étape implique la définition des attributs et comportements de chaque classe.
- **Développement des méthodes :** Implémenter les méthodes définies dans chaque classe, en veillant à respecter les spécifications de l'interface.
- **Intégration des classes :** Combiner les différentes classes développées pour former des sous-systèmes ou des modules fonctionnels.

Processus de développement avec le paradigme objet (2/2)

- **Tests unitaires :** Effectuer des tests unitaires sur chaque classe pour garantir que les méthodes fonctionnent correctement et répondent aux spécifications.
- **Tests d'intégration :** Vérifier la cohérence et l'interaction entre les différentes classes et objets lorsqu'ils sont combinés.
- **Itération du processus :** Répéter le processus de développement itératif en fonction des retours des tests et des évaluations, en ajustant la conception au besoin.
- **Documentation :** Documenter les classes, les méthodes, et l'architecture globale du système pour une compréhension claire et une maintenance future.
- **Optimisation :** Optimiser le code et les structures pour améliorer les performances et l'efficacité du programme final.
- **Finalisation :** Finaliser le programme complet en assurant la stabilité, la fiabilité et la satisfaction des besoins initiaux.

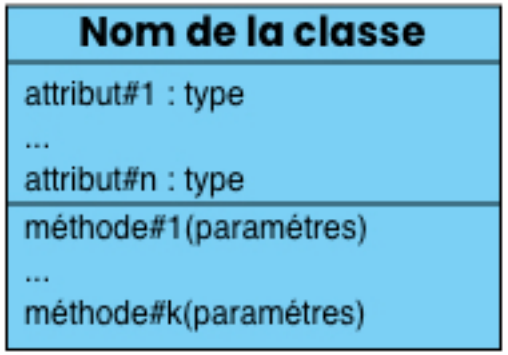
Concepts de base de la POO

Classes, Objets, Relations entre classes

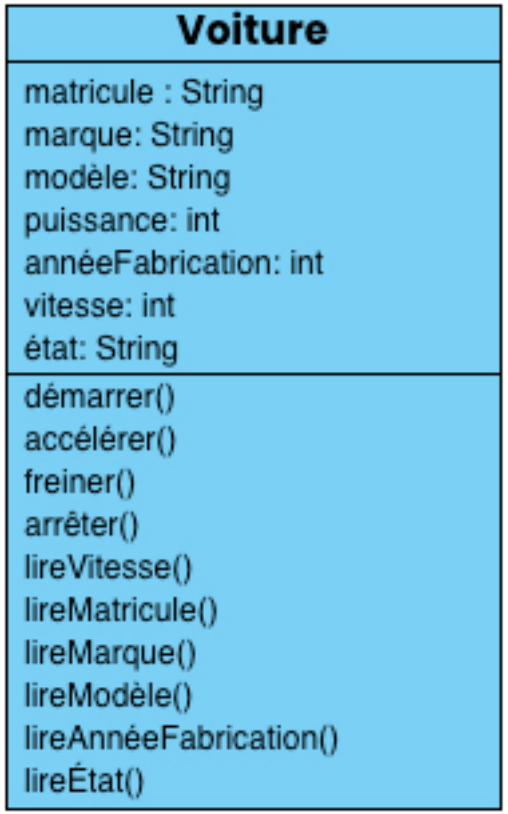
La notion de classes

- Une classe est un modèle ou un plan pour la création d'objets.
- Une classe est une entité regroupant les données (**attributs**) et les opérations (**méthodes**) associées à un ensemble ou groupe d'objets ayant la même structure.
  - Les attributs, également appelés propriétés, représentent les caractéristiques ou les données que chaque objet de cette classe possédera. Ces attributs décrivent l'état de l'objet.
  - Les méthodes sont des fonctions associées à la classe, spécifiant le comportement de ses objets. Elles définissent les actions que les objets peuvent effectuer.
- Une classe fournit une abstraction en définissant un modèle générique pour les objets. Elle permet aux développeurs de se concentrer sur les aspects pertinents sans se préoccuper des détails complexes.

Les classes en UML et en Java



Classe en UML



Exemple de classe en UML

La classe voiture en Java

```java
class voiture {
    // les attributs
    String matricule, marque, modèle;
    int puissance, annéeFabrication;
    int vitesse;
    String état;

    // les méthodes
    void démarrer() {
        if (état.equals("arrêté")) {
            état = "démarré";
            this.vitesse = 0;
        }
    }
    void accélérer() {
        if (état.equals("démarré")) {
            this.vitesse++;
        }
    }
    void freiner() {
        if (état.equals("démarré") && this.vitesse > 0) this.vitesse--;
    }
    void arrêter() {
        if (état.equals("démarré")) {
            this.vitesse = 0;
            état = "arrêté";
        }
    }
    int lireVitesse() { return this.vitesse; }
    String lireMatricule() { return this.matricule; }
    String lireMarque() { return this.marque; }
    String lireModéle() { return this.modéle; }
    int lirePuissance() { return this.puissance; }
    int lireAnnéeFabrication() { return this.annéeFabrication; }
    int lireÉtat() { return this.état; }
}
```

Les modificateurs d’accès (1/2)

- Les modificateurs d'accès sont des mots-clés qui définissent la visibilité (mode d’encapsulation) des membres d'une classe (méthodes, attributs, classes internes) dans d'autres parties du programme.
- Il existe quatre modificateurs d'accès principaux en Java :
  - **public:** Les membres déclarés comme public sont accessibles depuis n'importe quelle classe. Ils ont la portée la plus large.
  - **protected:** Les membres déclarés comme protégés sont accessibles dans la même classe, dans les classes du même paquet (package) et dans les sous-classes (héritage).
  - **private:** Les membres déclarés comme privés sont accessibles uniquement dans la même classe. Ils ont la portée la plus restrictive.
  - **default:** Si aucun modificateur n'est spécifié, le membre est accessible dans la même classe et dans les classes du même paquet.

Les modificateurs d'accès (2/2)

- Deux autres modificateurs sont aussi utilisés pour spécifier le comportement de certaines entités dans le code:
  - **final:**
    - **Attribut:** ne peut pas être modifiée une fois qu'elle a été initialisée (constante).
    - **Méthode:** ne peut pas être redéfinie (*override*) par les sous-classes.
    - **Classe:** ne peut pas être étendue (sous-classée).
  - **static:**
    - **Attribut:** l'attribut est partagée entre toutes les instances d'une classe.
    - **Méthode:** elle peut être appelée sans avoir besoin d'instancier la classe.
    - **Bloc:** un bloc statique (**static** {…}), au début d'une classe, est utilisé pour initialiser les variables statiques de la classe. Il est exécuté une seule fois lorsque la classe est chargée en mémoire. En cas de présence de plusieurs blocs, ils sont exécutés dans l'ordre de leur apparition dans le code.

La notion de constructeur

```
1 class c {
2     // déclaration de constructeur
3     modificateur_accès c(...) {
4         // corps du constructeur
5     }
6 }
```

- Le constructeur est une méthode particulière, portant le même nom que la classe, et qui est définie sans aucun retour.
- Lorsqu'un nouvel objet est créé à partir de la classe, le constructeur est appelé automatiquement.
  - Il n'est appelé que lors de la construction de l'objet
- Il est couramment utilisé pour initialiser les attributs de l'objet, leur donnant des valeurs de départ dès sa création.
- Une version par défaut est fournie par les langages de programmation.
  - Éviter de se reposer sur la version par défaut du constructeur.
- Une classe peut avoir plusieurs constructeurs, chacun pouvant prendre un ensemble différent de paramètres ➤ Il est souvent une des méthodes les plus surchargées.

La notion d'objets

- Un objet est une **instance** d'une classe, qui définit une structure et un comportement. La création d'un objet à partir d'une classe est appelée **instantiation**.
- Chaque objet a son propre état distinct, mais partage la même structure et le même comportement définis par la classe.
- Chaque objet a une identité unique qui le distingue des autres objets, même s'ils ont de la mêmes classe. L'identité permet de suivre et de référencer spécifiquement un objet dans le programme.
  - **Notation par la JVM :** NomPaquet.NomClasse@hashcode ➤ P1.C1@3d012ddd
- Les détails internes de l'objet sont cachés à l'extérieur, et l'accès aux données se fait généralement via des méthodes.
- Chaque objet peut être développé, testé et maintenu indépendamment des autres.
- Un objet est crée en Java en utilisant la commande **new** suivie par un appel au constructeur de la classe.

La destruction des objets en C++

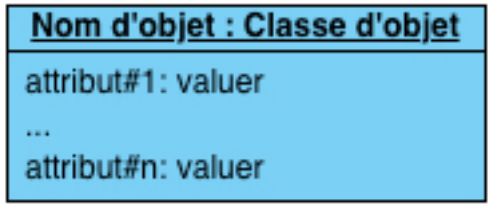
- Lorsque un objet n'est plus référencé nulle part, la zone mémoire qu'il occupe peut et doit être libérée.
- En C++, la destruction d'objets est gérée par des destructeurs - une méthode spéciale qui porte le même nom que la classe précédé par le caractère tilde ~.
  - Il est automatiquement appelé lorsque l'objet sort de la portée de son bloc d'exécution ou lorsqu'il est explicitement supprimé à l'aide de l'opérateur **delete**.

```
1  class MaClasse {
2  public:
3      // Constructeur
4      MaClasse() { /* Initialisation */ }
5
6      // Destructeur
7      ~MaClasse() {
8          /* Libération de ressources */
9      }
10 };
11
12 int main() {
13     {
14         // Le destructeur sera appelé lorsque
15         // 'objet' sortira de la portée ici
16         MaClasse objet;
17     }
18
19     MaClasse *objetDynamique = new MaClasse();
20     // Appel explicite au destructeur
21     delete objetDynamique;
22
23     return 0;
24 }
```

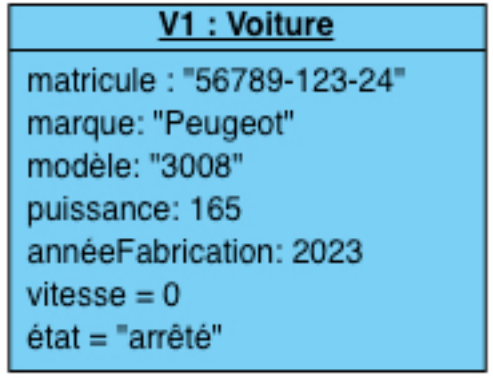
La destruction des objets en Java

- En Java, la gestion de la mémoire est effectuée par le ramasse-miettes (*garbage collector*). Lorsqu'un objet n'a plus de références le pointant, il devient éligible à la collecte par le ramasse-miettes.
- Le ramasse-miettes peut aussi être appelé explicitement par **System.gc()**
- La destruction d'objets en Java est donc automatisée, et les programmeurs n'ont généralement pas besoin de spécifier explicitement des destructeurs.
- En plus, la classe *Object* de Java a une méthode **finalize**(), ce qui signifie que toutes les autres classes aussi. Il est possible de surcharger la méthode **finalize**() pour implémenter un logique de nettoyage spécifique.
- Cependant, c'est la machine virtuelle Java qui décide d'appeler cette méthode ou non. Dans la majorité des cas, les objets créés à l'intérieur d'une méthode et déclarés inutiles lorsque la méthode se termine sont détruits immédiatement sans appel à **finalize()**. Cette méthode est une sorte d'assurance plutôt qu'une solution fiable.

Les objets en UML et en Java



Objet en UML



Exemple d'objet en UML

Instanciation d'un objet voiture en Java

```java
class voiture {
    // les attributs
    String matricule, marque, modèle;
    int puissance, annéeFabrication;
    int vitesse;
    String état;

    // le constructeur d'objets
    voiture (String matricule, String marque, String modèle, int puissance, int annéeFabrication) {
        this.matricule = matricule;
        this.marque = marque;
        this.modèle = modèle;
        this.puissance = puissance;
        this.annéeFabrication = annéeFabrication;
        this.vitesse = 0;
        this.état = "arrêté";
    }

    // les méthodes
    ...
}

class Test {
    public static void main(String[] args) {
        v1 = new voiture("56789-123-24", "peugeot", "3008", 165, 2023);
    }
}
```

Relations entre classes (1/2)

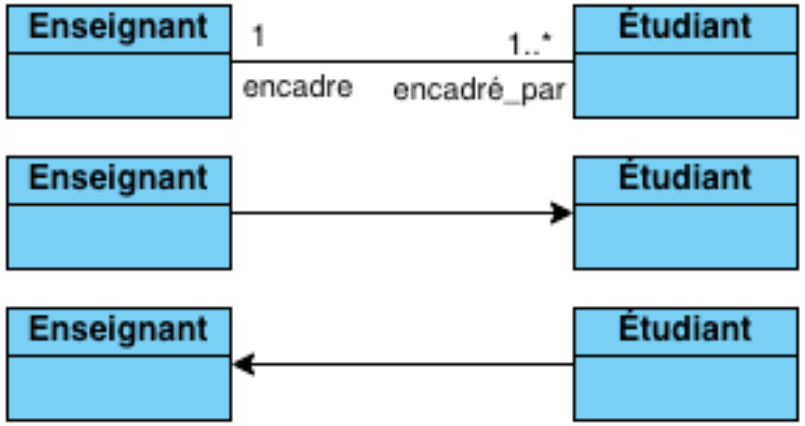
- Dans la programmation orientée objet, un objet communique avec d'autres objets pour utiliser leurs fonctionnalités associées et les services fournis par ces objets.
- Il existe différentes relations entre les objets d'un système, et ces relations sont généralisées et exprimées sous forme de liens entre les classes :
  - **Association :** Représente une connexion entre deux classes, indiquant qu'un objet d'une classe nécessite l'exécution des méthodes d'un objet. Elle n'implique pas de propriété de possession ou de durée de vie partagée entre les objets.
    - Un *enseignant en* enseigne une liste des *étudiants ets* ➤ Une association entre les deux classes *Enseignant* et *Etudiant*.
  - **Agrégation :** Forme spécifique d'association qui indique une relation *partie-tout* entre deux classes d'objets, mais les objets peuvent avoir une durée de vie indépendante.
    - Une classe *Département* peut être en agrégation avec la classe *Enseignant*, indiquant que le département comporte plusieurs enseignants.

Relations entre classes (2/2)

- **Composition :** Forme plus forte d'agrégation qui indique qu'un objet est responsable du cycle de vie d'autres objets.
  - Une classe *Université* peut être en composition avec la classe *Département*, indiquant que le département est une partie essentielle d'une université.
- **Héritage :** Représente une relation de type *est-un*, indiquant qu'une classe hérite des caractéristiques (des attributs et des méthodes) d'une autre classe.
  - Une classe *EnseignantPermanant* peut hériter de la classe *Enseignant*, car un enseignant permanant est un type spécifique d'enseignant.

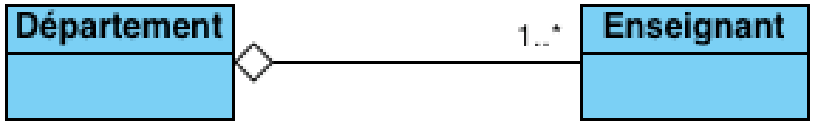
Les relations entre classes en UML et Java (1/2)

- **Association**




```java
class Enseignant {
    List<Etudiant> encadrement;
    ...
}
class Etudiant {
    Enseignant encadreur;
    ...
}
```

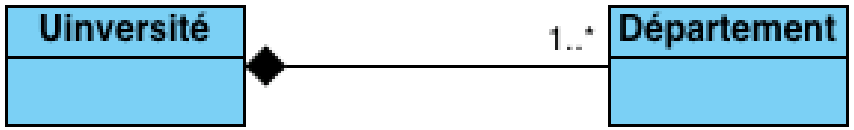
- **Agrégation**




```java
class Enseignant {
    ...
}
class Departement {
    List<Enseignant> enseignants;
    ...
}
```

Les relations entre classes en UML et Java (2/2)

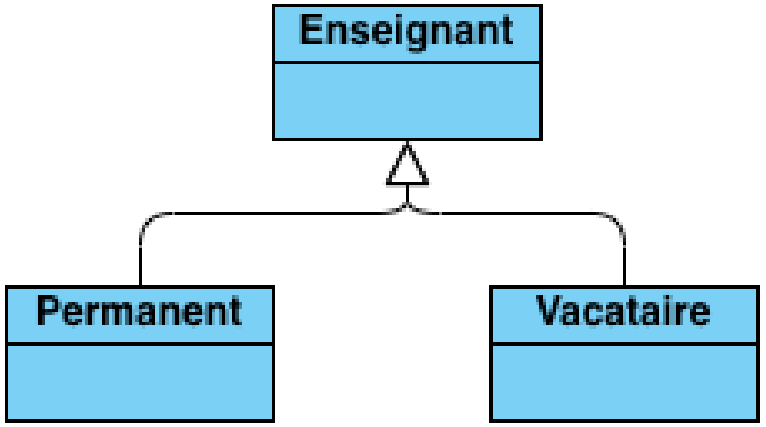
- **Composition**




```java
class Departement {
    public String nom;
    public Departement(String nom) {
        this.nom = nom;
    }
    ...
}
class Université {
    public List<Departement> departements;
    ...
    public Université() {
        departements.add(new Departement("Informatique"));
        departements.add(new Departement("Math"));
        departements.add(new Departement("Chimie"));
        ...
    }
}
```

Les relations entre classes en UML et Java (2/2)

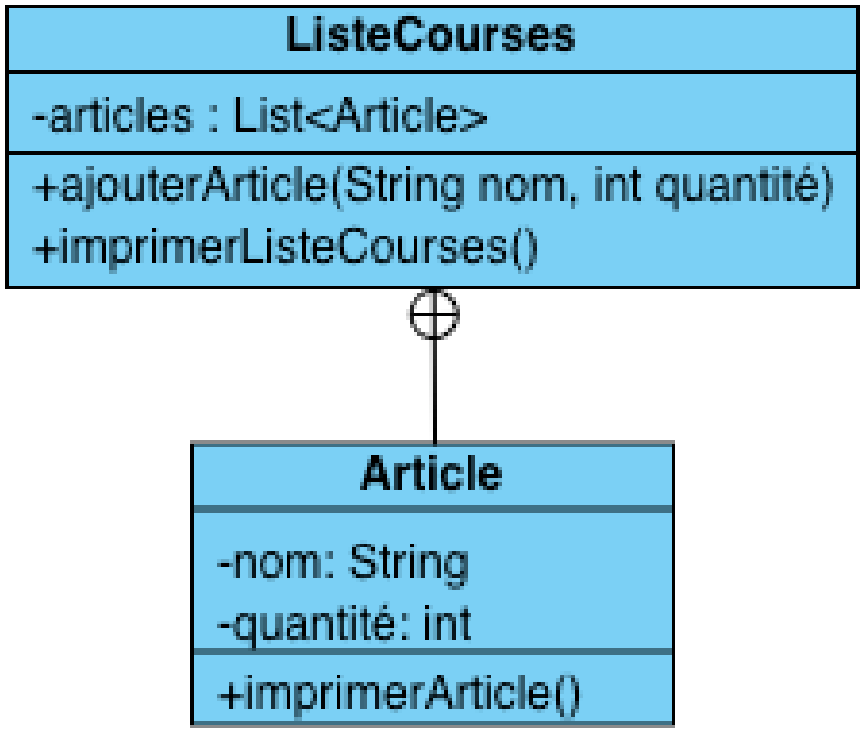
- **Héritage**



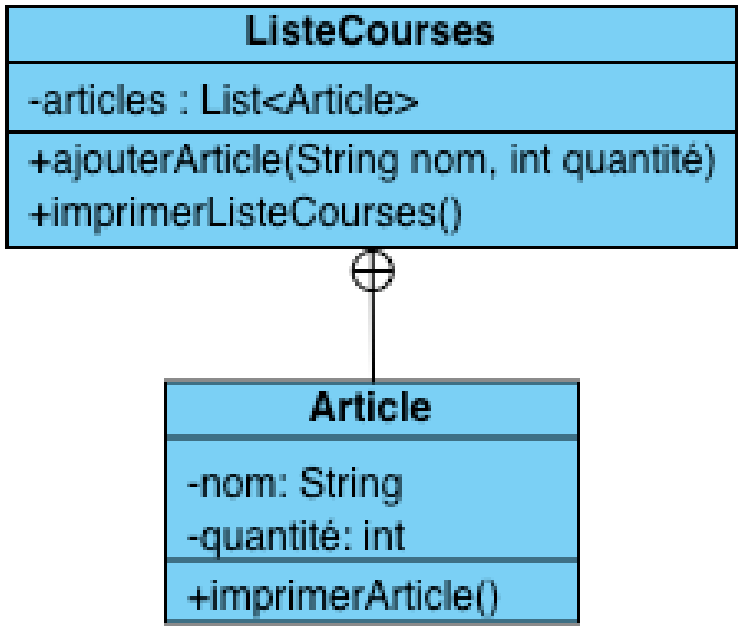

```java
class Enseignant {
    ...
}
class Permanant extends Enseignant {
    ...
}
class Vacataire extends Enseignant {
    ...
}
```

Classes Internes

- **Classes internes:**
  - Des classes définies à l'intérieur d'une autre classe et permet l'accès direct aux membres de la classe englobante.
  - Bénéfiques lorsqu'elles n'ont pas d'utilité en dehors de la classe englobante ➤ restreindre leur accès, renforçant ainsi leur encapsulation.
  - Simplifier la structure du code en regroupant les fonctionnalités liées au sein de la classe englobante.



Classes Internes Exemple




```java
import java.util.*;
public class ListeCourses {
    private List<Article> articles;

    public ListeCourses() {
        articles = new ArrayList();
    }

    public void ajouterArticle(String nom, int quantité) {
        articles.add(new Article(nom, quantité));
    }

    public void imprimerListeCourses() {
        for (Article a : articles) a.imprimerArticle();
    }

    class Article {
        private String nom; private int quantité;

        public Article(String nom, int quantité) {
            this.nom = nom; this.quantité = quantité;
        }

        public void imprimerArticle() {
            System.out.println("Article: " + this.nom + " Qt: " + this.quantité);
        }
    }
}
```

Classes Anonymes

- **Classes anonymes:**
  - Des classes sans nom, généralement utilisées pour implémenter une interface ou étendre une classe abstraite de manière concise.
  - Souvent utilisées pour créer des instances d'interfaces fonctionnelles directement dans le code, par exemple lors de l'utilisation de classes anonymes pour les gestionnaires d'événements.

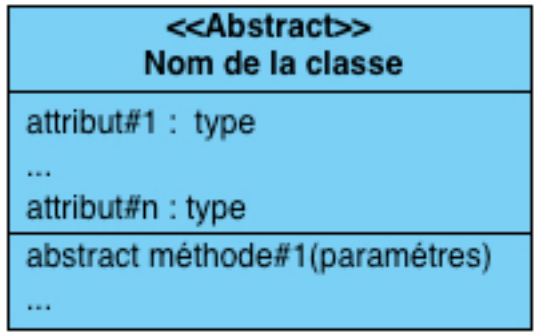
```
Runnable myRunnable = new Runnable() {
    public void run() {
        // Code de la méthode run
    }
};
```

Les classes abstraites et la notion d'interface

Les classes abstraites

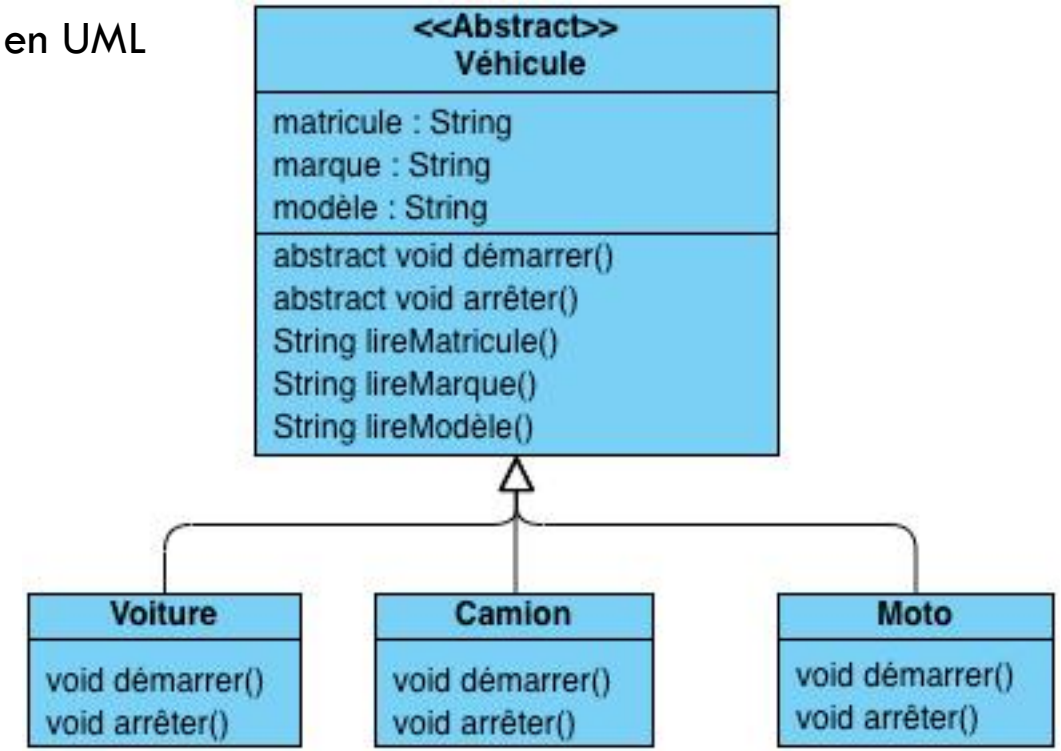
- Une classe abstraite sert de modèle pour d'autres classes en déclarant des méthodes abstraites. Les méthodes abstraites sont des méthodes qui n'ont pas de corps (pas d'implémentation), mais doivent être implémentées par les classes dérivées.
- Une classe abstraite est utile pour:
  - **Fournir une structure de base :** Les classes abstraites permettent de définir une structure de base commune pour un ensemble de classes dérivées.
  - **Forcer l'implémentation :** En déclarant des méthodes abstraites, une classe abstraite oblige les classes dérivées à implémenter ces méthodes, garantissant ainsi une certaine cohérence dans l'utilisation de l'héritage.
  - **Empêcher l'instanciation directe :** Une classe abstraite ne peut pas être instanciée directement, ce qui assure que seules les classes dérivées concrètes sont utilisées.
- Une classe abstraite peut aussi contenir des méthodes concrètes et des constructeurs et hérite des propriétés et des méthodes des classes concrètes.

Les classes abstraites en UML et Java



Classe abstraite en UML

Exemple d'une classe abstraite en UML




```java
abstract class véhicule {
    String matricule, marque, modèle;
    abstract void démarre();
    abstract void arrêter();
    String lireMatricule() { return this.matricule; }
    String lireMarque() { return this.marque; }
    String lireModèle() { return this.modèle; }
}

class voiture extends véhicule {
    void démarre() {
        ...
    }
    void arrêter() {
        ...
    }
}

class moto extends véhicule {
    void démarre() {
        ...
    }
    void arrêter() {
        ...
    }
}

class camion extends véhicule {
    void démarre() {
        ...
    }
    void arrêter() {
        ...
    }
}
```

Exemple d'une classe abstraite en Java

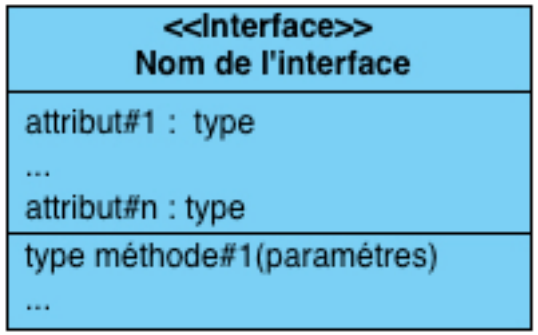
Les interfaces (1/2)

- Une interface est une collection de méthodes abstraites (sans implémentation) qui définissent un ensemble de comportements.
- Une classe implémentant une interface doit garantir qu'elle fournit une mise en œuvre concrète de chacune de ces méthodes.
- Une interface peut contenir des attributs mais sont implicitement **public, static** et **final**.
- Les interfaces servent à plusieurs fins essentielles, offrant une flexibilité et une abstraction qui améliorent la conception et la maintenance du code :
  - **Définition de contrats :** Les interfaces définissent des contrats qu'une classe doit suivre. Elles décrivent les méthodes qu'une classe concrète doit implémenter, créant ainsi une structure commune pour les différentes implémentations.
  - **Polymorphisme :** Les interfaces permettant à différentes classes de fournir leur propre implémentation pour une interface commune.

Les interfaces (2/2)

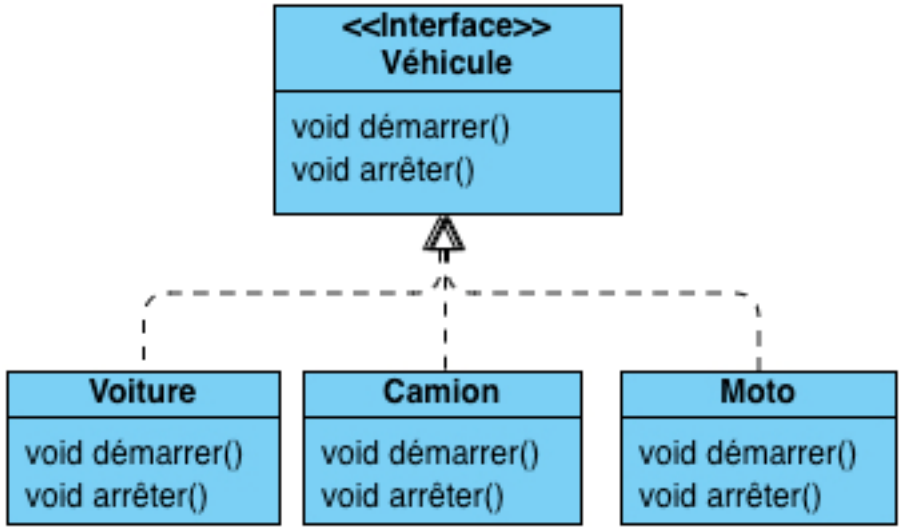
- **Héritage multiple :** Les classes en Java peuvent implémenter plusieurs interfaces en même temps, ce qui permet un héritage multiple virtuel.
- **Compatibilité :** Les interfaces permettent de définir des contrats communs dans des bibliothèques ou des frameworks. Les classes qui implémentent ces interfaces peuvent être utilisées de manière interchangeable, ce qui favorise l'interopérabilité.

Les interfaces en UML et Java



Interface en UML

Exemple d'une interface abstraite en UML




```java
interface véhicule {
    void démarre();
    void arrêter();
}

class voiture implements véhicule {
    void démarre() {
        ...
    }
    void arrêter() {
        ...
    }
}

class moto implements véhicule {
    void démarre() {
        ...
    }
    void arrêter() {
        ...
    }
}

class camion implements véhicule {
    void démarre() {
        ...
    }
    void arrêter() {
        ...
    }
}
```

Exemple d'une interface en Java

Classes abstraites vs. Interfaces

Caractéristiques	Classes abstraites	Interfaces
Déclaration des méthodes abstraites	Oui (pas obligatoire)	Oui
Déclaration des constructeurs	Oui	Non
Déclaration des attributs	Oui	Constantes uniquement
Héritage multiple	Non	Oui (virtuel)
Visibilité des méthodes	Oui	Public uniquement

Les interfaces en C++

- En C++, une interface n'est pas une entité distincte comme c'est le cas dans certains langages de programmation orientés objet comme Java.
- Le concept d'interface est souvent réalisé à l'aide de classes abstraites et de conventions de nommage dans le contexte des fichiers d'en-tête (.h) et des fichiers source (.cpp).

```
1  // véhicule.h
2  class véhicule {
3      public:
4          virtual void démarrer();
5          virtual void arrêter();
6  }
7
8  // voiture.cpp
9  #include "véhicule.h"
10 class voiture {
11     public:
12         void véhicule::démarrer() {
13             ...
14         }
15         void véhicule::arrêter() {
16             ...
17         }
18 }
```

Les patrons de conception

Design patterns ou Patrons de conception

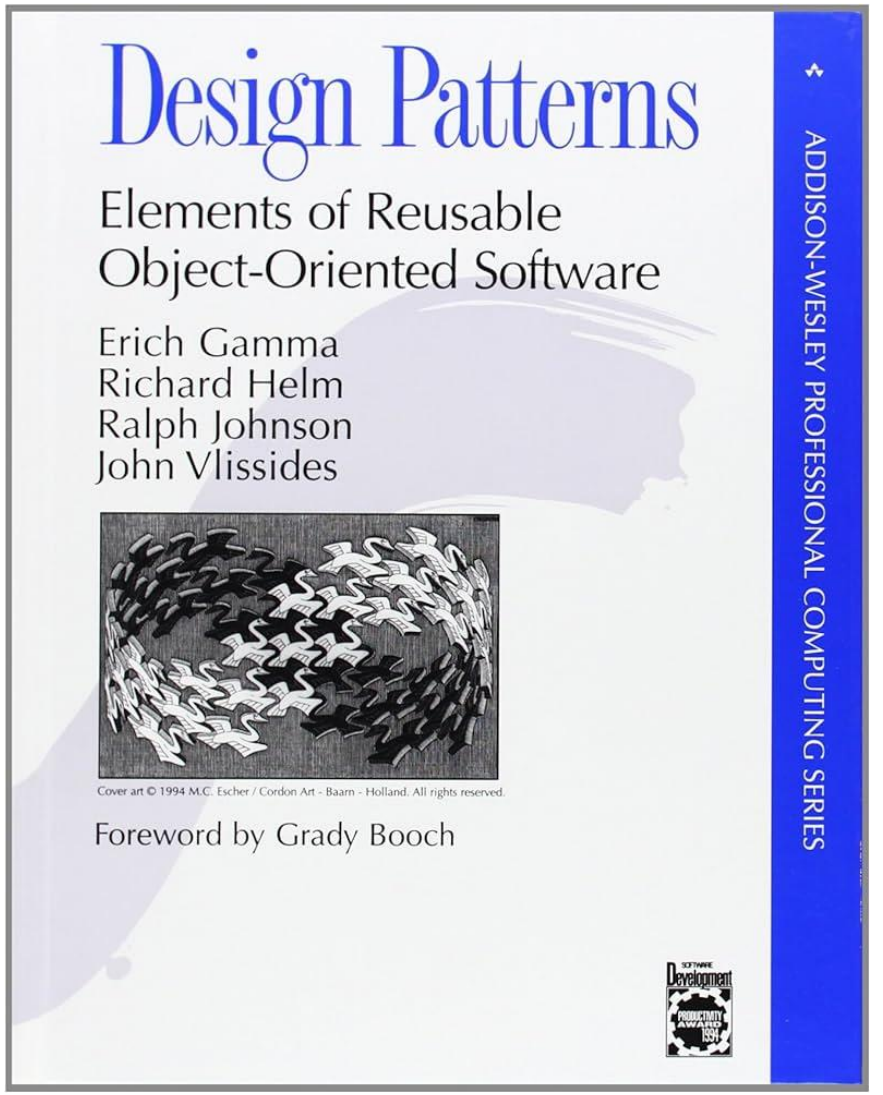
- Les patrons de conception, sont des solutions génériques réutilisables à des problèmes récurrents rencontrés dans le domaine de la conception logicielle. Ces patrons offrent des solutions éprouvées pour résoudre des problèmes communs tout en favorisant :
  - **Réutilisabilité :** Les patrons de conception fournissent des solutions prêtes à l'emploi pour des problèmes courants. Ils permettent aux développeurs de réutiliser des conceptions éprouvées plutôt que de concevoir des solutions à partir de zéro.
  - **Maintenabilité :** En utilisant des patrons de conception, les développeurs peuvent concevoir des systèmes plus modulaires et faciles à maintenir.
  - **Communication :** En utilisant des noms de patrons bien connus, les membres de l'équipe peuvent communiquer efficacement sur la conception du logiciel.

Patrons de conception vs Structures de données

Structures de données	Patrons de conception
Permettent de stocker, organiser et manipuler des données de manière efficace.	Fournissent des schémas éprouvés pour structurer, organiser et interagir avec le code.
Impliquant souvent des détails d'implémentation plus spécifiques.	Se concentrent sur des concepts de haut niveau et des relations entre les objets des systèmes.
Offrent des solutions à des problèmes de claculs: tri, recherche, etc.	Offrent des solutions à des problèmes architecturaux: comment inter-changer les algorithmes de tri sans perturber le reste du code.
Focalisent sur l'efficacité en terme du temps et d'espace.	Focalisent sur la réutilisabilité, la maintenabilité du code.

Origine et bible des Patrons de conception

- L'histoire des patrons de conception remonte aux travaux de l'architecte Christopher Alexander dans les années 1960. Christopher Alexander était un architecte qui a développé des principes de conception architecturale et urbaine. Ses idées ont été publié en 1977 dans un livre intitulé **A Pattern Language**.
- En 1994, quatre auteurs: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, souvent appelés le **Gang des Quatre (GoF)** ont publié un livre intitulé ***Design Patterns - Elements of Reusable Object-Oriented Software***, qui a initié le concept de patrons de conception dans le développement de logiciels.
- Dans ce livre 23 patrons de conception ont été décrit.

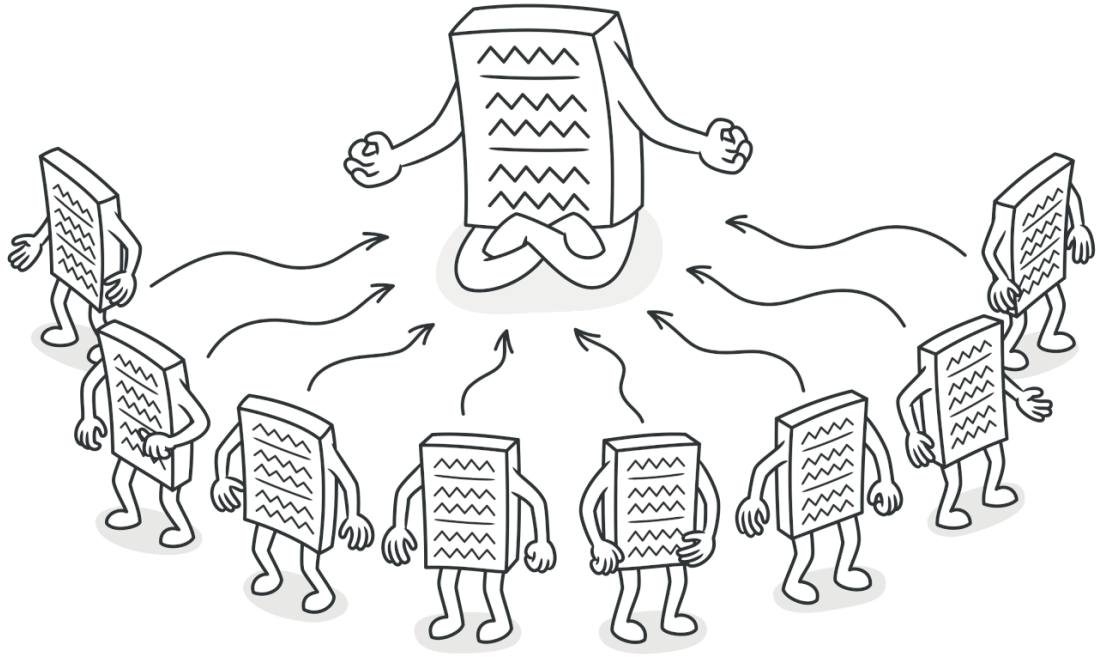


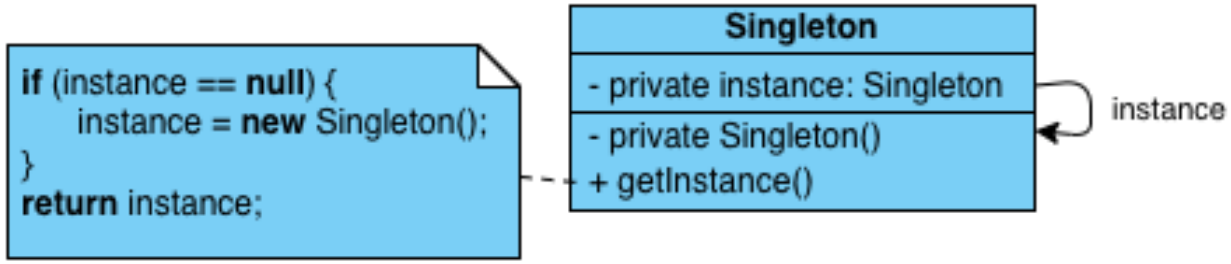
Catégories des patrons de conception

- Les patrons de conception, sont classés en trois catégories de base:
  - **Patrons de création :** Les patrons de conception de cette catégorie concentrent sur comment créer des objets dans un système. Ils fournissent des mécanismes flexibles et réutilisables pour instancier des objets tout en isolant les détails de leur création, de composition et de leur représentation.
    - **Exemple:** Singleton
  - **Patrons de comportement :** Ces patrons de conception concentrent sur la manière dont les objets interagissent et distribuent la responsabilité entre eux.
    - **Exemple:** Stratégie, Chaine de Responsabilité, Observateur.
  - **Patrons de structure :** Ces patrons de conception concentrent sur comment composer les classes ou les objets pour obtenir de structures plus complexes.
    - **Exemple:** Adaptateur, Composite, Proxy, Décorateur.

Patron Singleton

- **Objectif:** Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.
- **Exemple de cas d'utilisation:** Le système a besoin d'un seul gestionnaire de fenêtre, un seul spool d'impression, un seul point d'accès vers un moteur de base de données, etc.
- **Structure:**



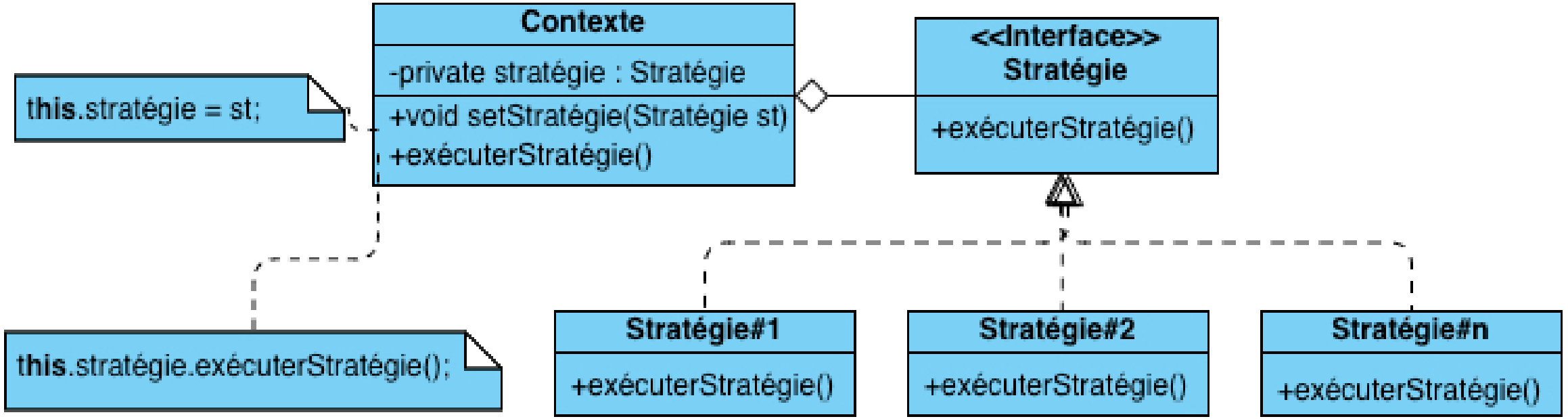


Patron Stratégie (1/2)

- **Objectif:** Permettre à une classe de choisir dynamiquement la manière de résoudre un problème parmi une famille de solutions possibles.
- **Exemple de cas d'utilisation:** Vous voulez avoir différentes variantes d'un algorithme à l'intérieur d'un objet à disposition (ex., algorithmes de tri), et pouvoir passer d'un algorithme à l'autre lors de l'exécution.

Patron Stratégie (2/2)

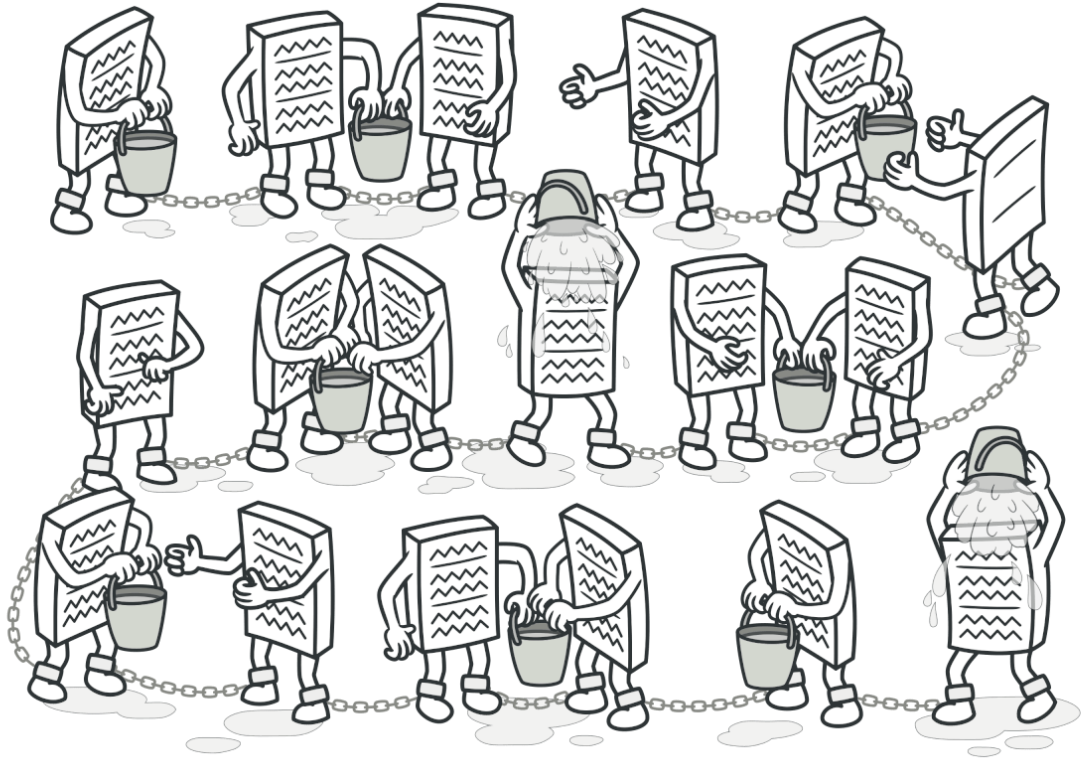
- **Structure:**



Une autre version utile consiste a utiliser une classe abstraite ***Stratégie*** et les classes ***Stratégie#i*** héritent cette classe.

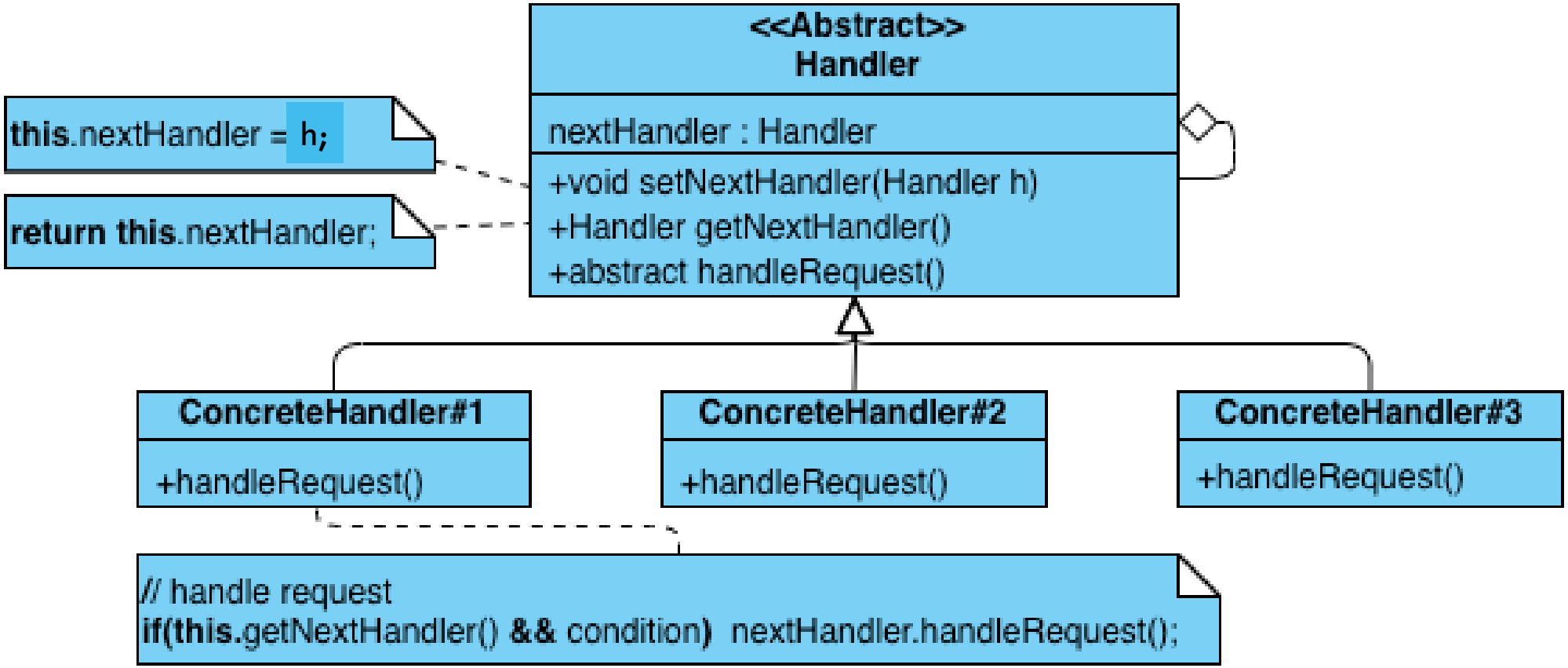
Patron Chaine de Responsabilité (1/3)

- **Objectif:** Facilite la circulation de demandes au sein d'une chaîne de gestionnaires.
- **Exemple de cas d'utilisation:** Les données d'un programme nécessitent être manipuler par plusieurs objets. Chaque objet applique son propre traitement avant de les transmettre au objet suivant (ex., passage des données par un séquence de filtres).



Patron Chaine de Responsabilité (2/3)

- **Structure:**

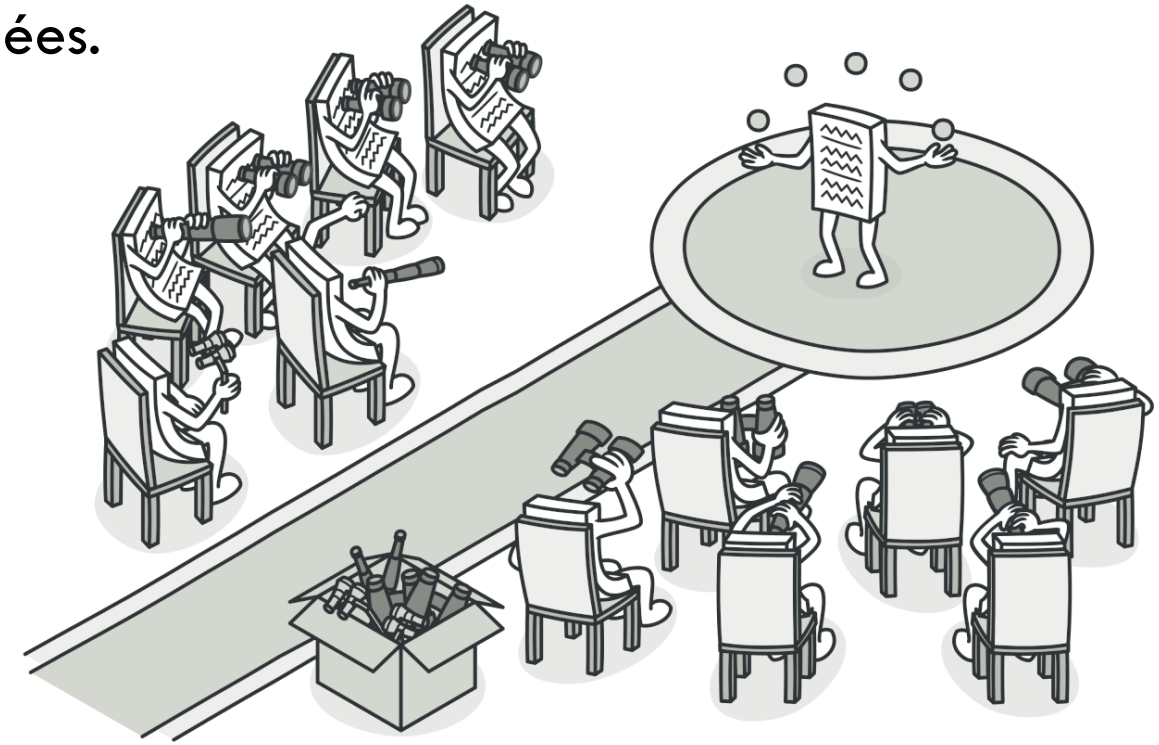


Patron Chaine de Responsabilité (3/3)

```
1  class ChainOfresponsibilityPatternTest {
2      public static void main(String[] args) {
3          Request r = new Request();
4
5          Handler h1 = new ConcreteHandler_1();
6          Handler h2 = new ConcreteHandler_2();
7          Handler h3 = new ConcreteHandler_3();
8
9          h1.setNextHandler(h2);
10         h2.setNestHandler(h3);
11
12         h1.handleRequest(r);
13     }
14 }
```

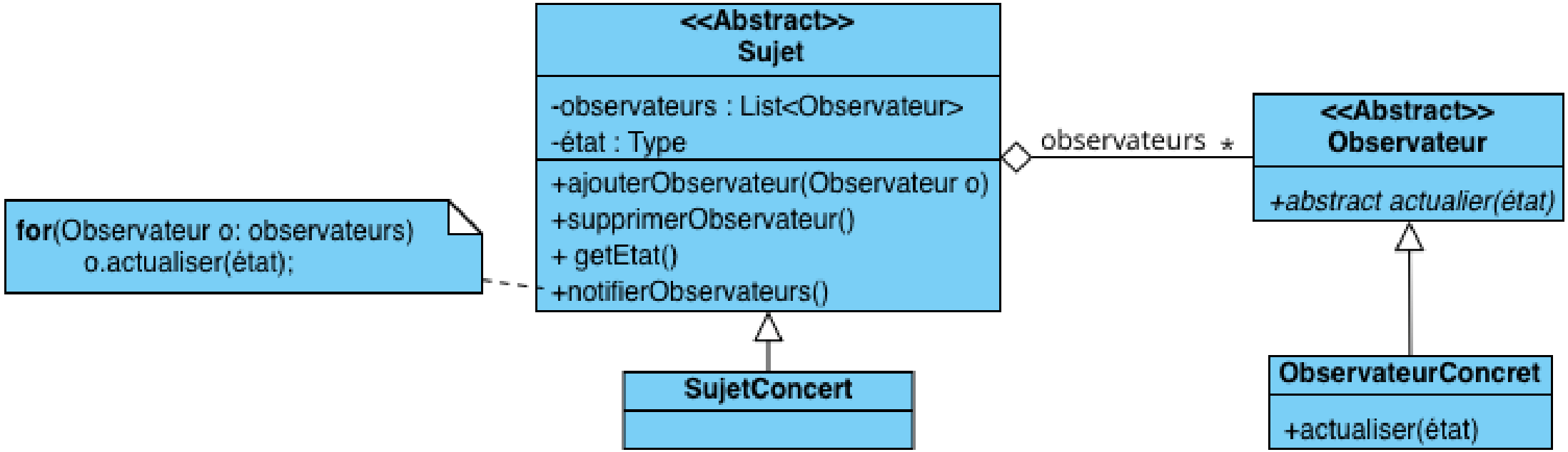
Patron Observateur (1/2)

- **Objectif:** Définit une relation de dépendance 1 à N entre les objets de telle manière que si un objet change d'état, tous les objets dépendants sont notifiés et mis à jour automatiquement.
- **Exemple de cas d'utilisation:** Les administrateurs doivent être notifiés à chaque changement dans la base de données.



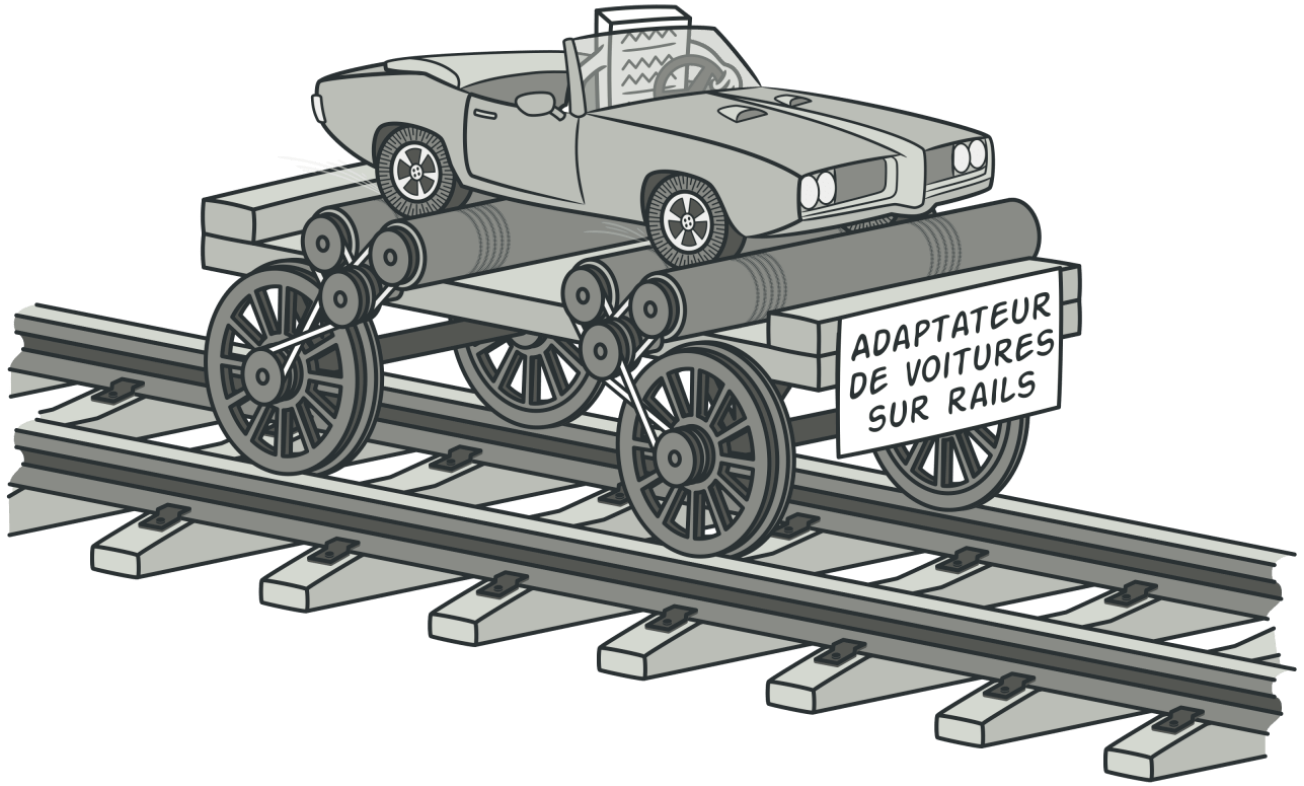
Patron Observateur (2/2)

- **Structure:**



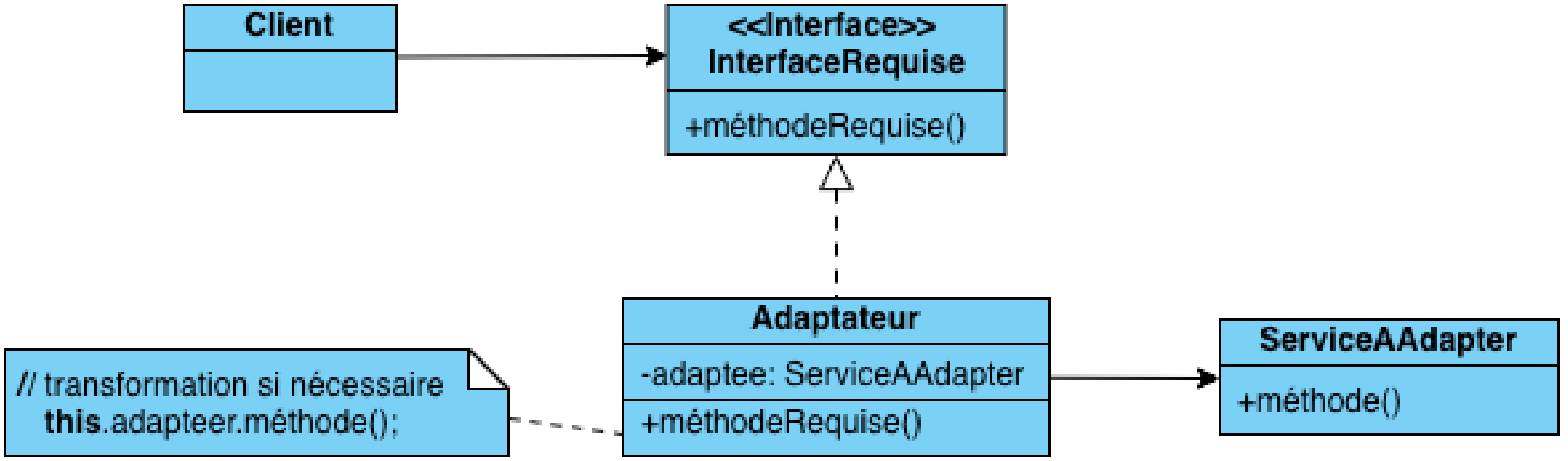
Patron Adaptateur (1/2)

- **Objectif:** Permet de faire collaborer des objets ayant des interfaces incompatibles.
- **Exemple de cas d'utilisation:** Utile dans les situations ou une classe existante offre un service, mais il y a incompatibilité entre l'interface offerte et celle que le client voudrais avoir.



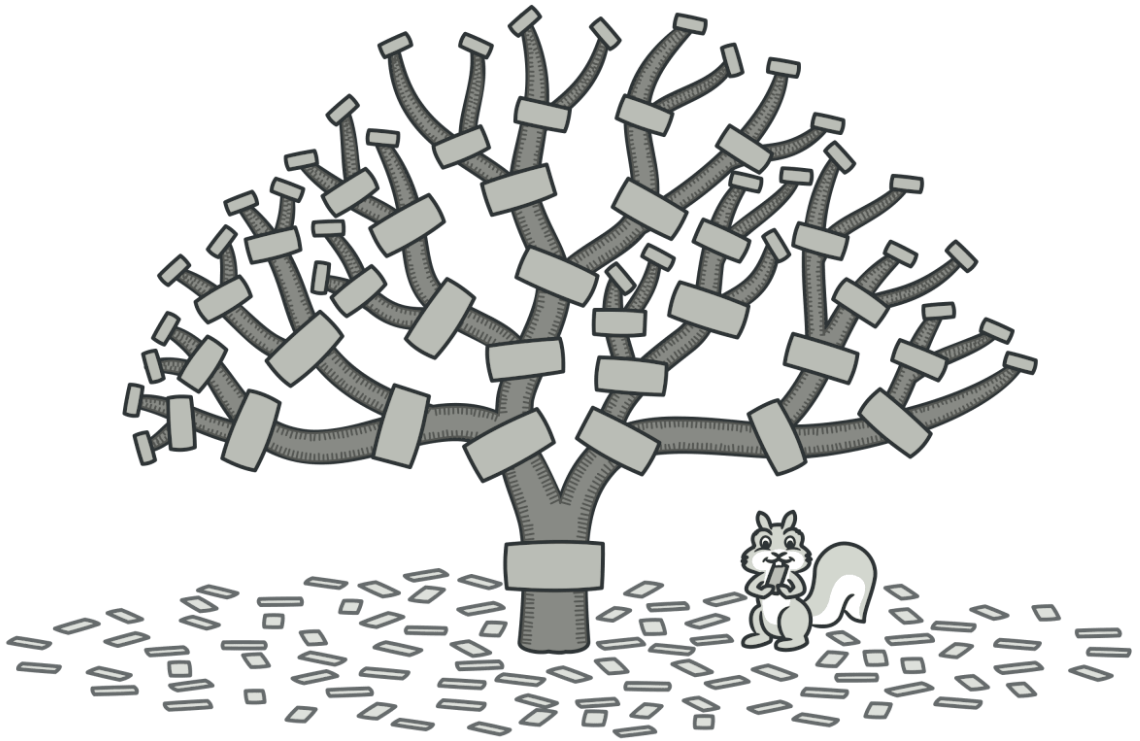
Patron Adaptateur (2/2)

- **Structure:**



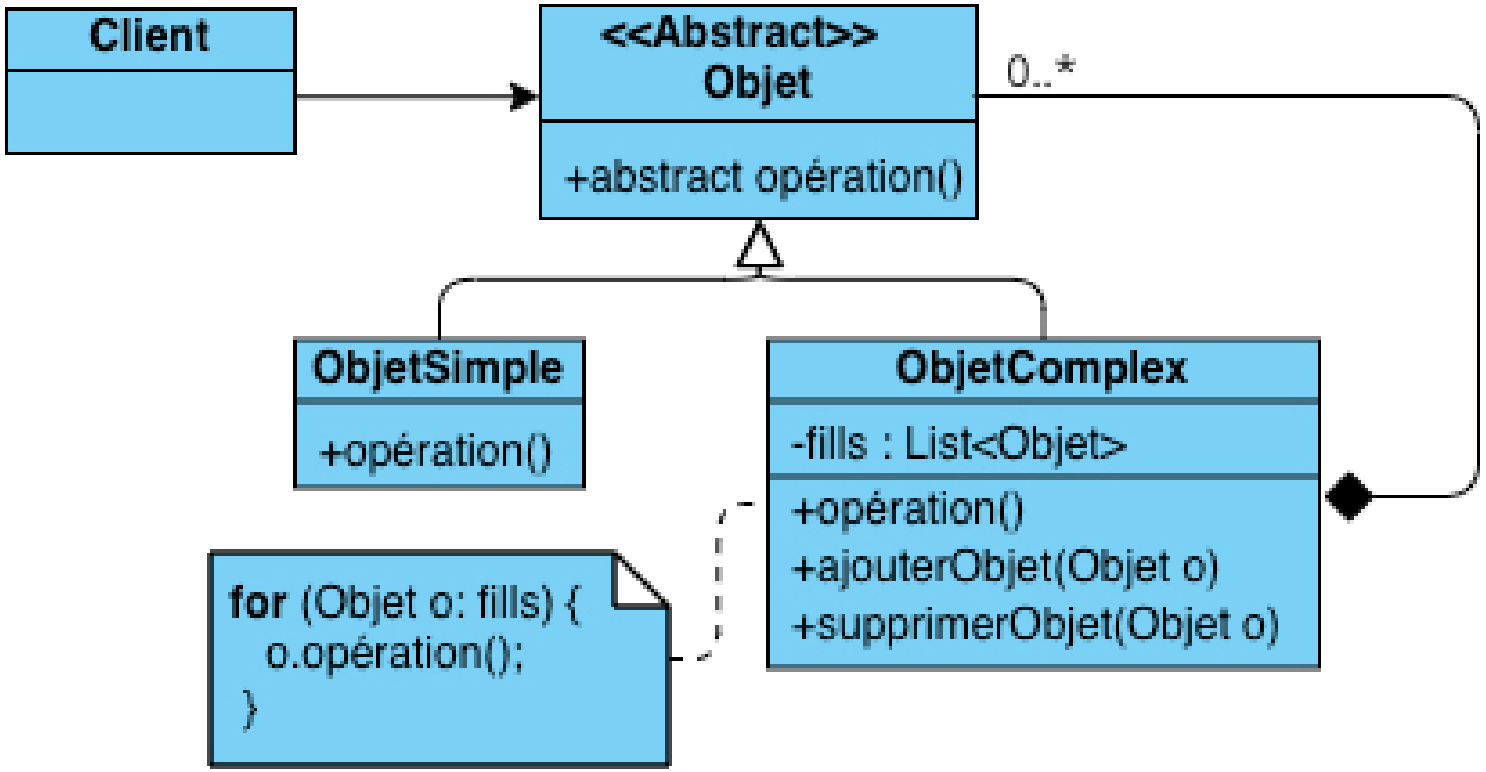
Patron Composite (1/2)

- **Objectif:** Composer des objets dans des structures d'arbre pour représenter les hiérarchies de partie-tout.
- **Exemple de cas d'utilisation:** Le programme traite les éléments simples aussi bien que complexes de façon uniforme (ex., manipulation des fichiers et des dossiers par un système de fichiers).



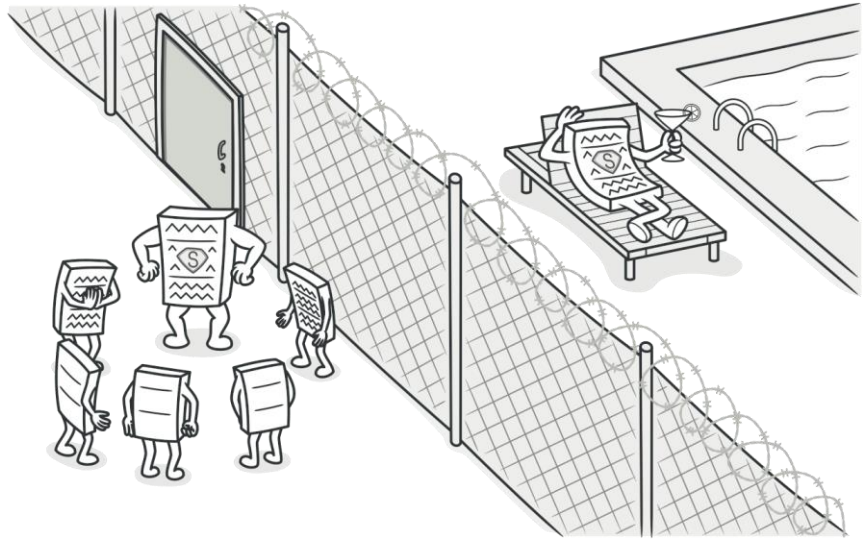
Patron Composite (2/2)

- **Structure:**



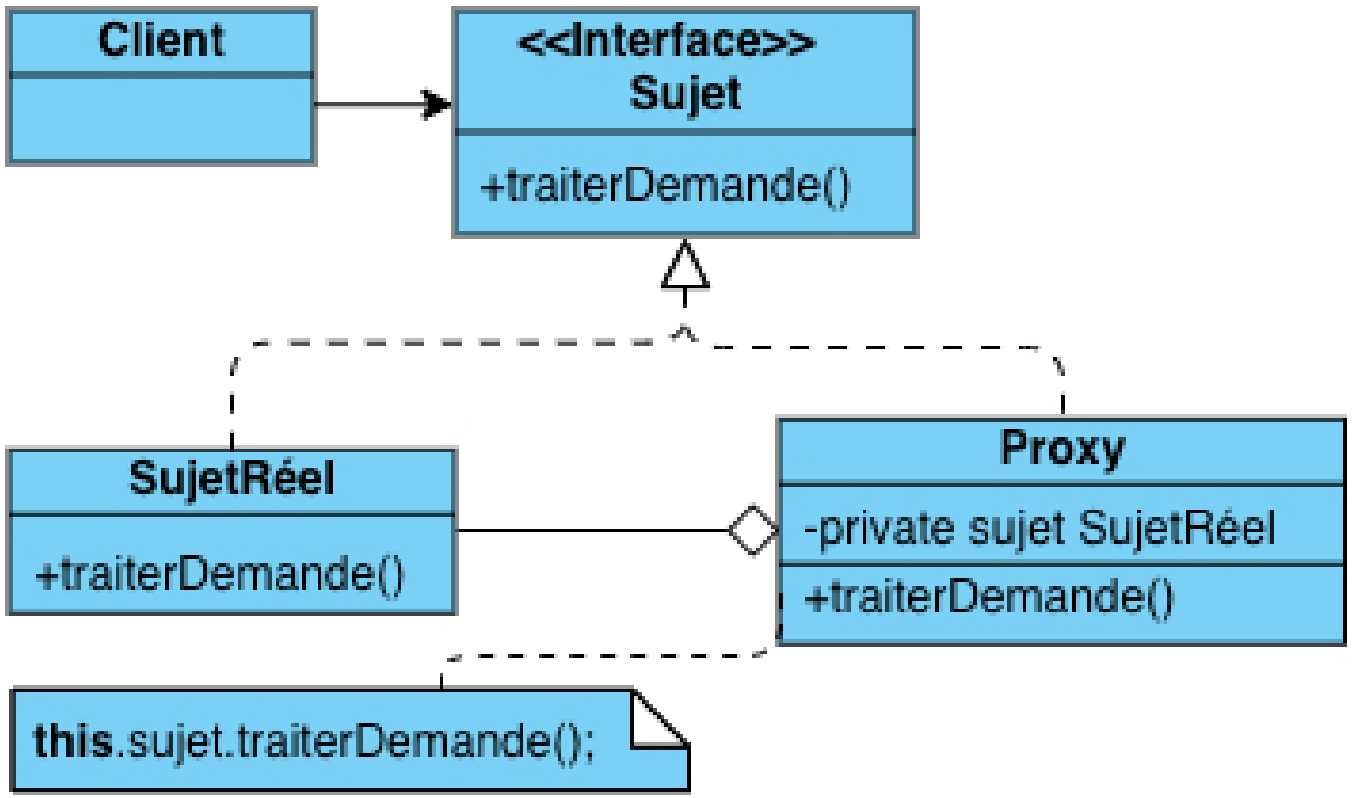
Patron Proxy – Procuration (1/2)

- **Objectif:** Permet d'utiliser un substitut pour un objet. Il donne le contrôle sur l'objet original, vous permettant d'effectuer des manipulations avant ou après que la demande ne lui parvienne.
- **Exemple de cas d'utilisation:**
  - Retarder l'allocation mémoire des ressources de l'objet jusqu'à son utilisation réelle.
  - Stocker le résultat d'opérations coûteuse en temps, afin de pouvoir les partager avec les différents objets utilisateurs (joue le rôle d'une mémoire cache).
  - Contrôler l'accès à un objet ou ajouter des fonctionnalités supplémentaires autour d'un objet existant sans modifier son code source. (ex. Protéger l'accès à l'objet par des objets malveillants).



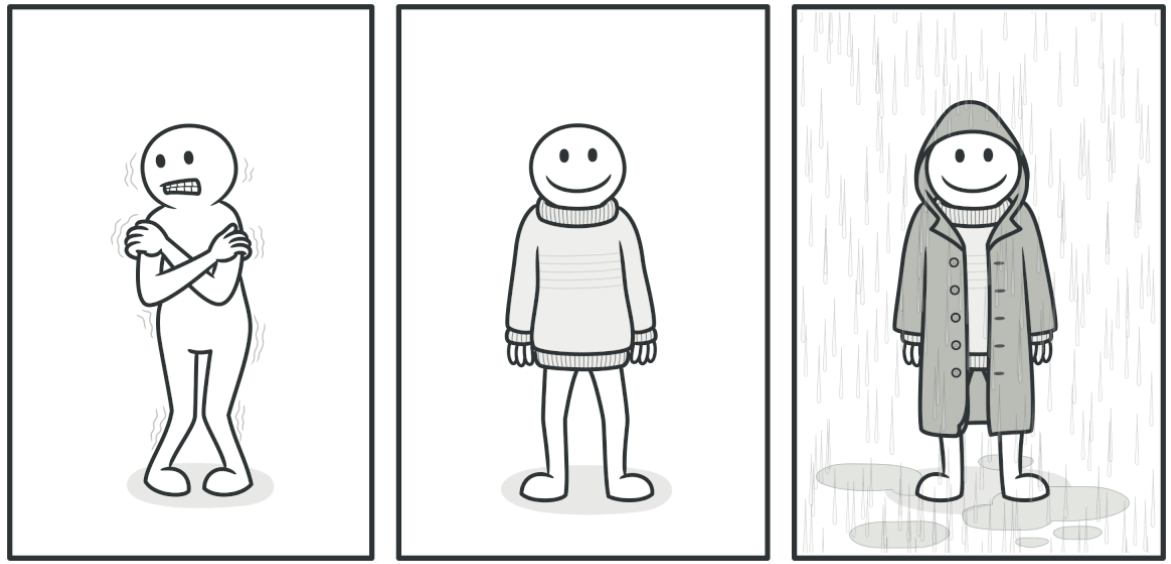
Patron Proxy – Procuration (2/2)

- **Structure:**



Patron Décorateur (1/2)

- **Objectif:** Attacher de manière flexible des responsabilités supplémentaires à un objet.
- **Exemple de cas d'utilisation:** Extension de fonctionnalités d'objets, ajout de comportements à des objets existants.



Patron Décorateur (2/2)

- **Structure:**
  - L'interface ***composant*** peut aussi être remplacer par une classe abstraite et les relations d'implémentations par des relations d'héritage.