UNIVERSITÉ DE MONTPELLIER

I.U.T MONTPELLIER - SETE

Design Patterns

Qualité de Développement R3.04 (cours 1)

Nadjib Lazaar (nadjib.lazaar@umontpellier.fr)

Présentation

Conception et Programmation Objet Avancées

- Production d'une conception détaillée **(StarUML)** en appliquant des patrons de conception **(design patterns)**

- Réalisation **(Java)** et application des bonnes pratiques POO

Organisation

EDT et évaluation

- **08 séances de cours/TD/TP (1h + 2h + 1h)**
  - Du 07/11/2022 au 13/01/2023
  - TP sur machine avec StartUML / Java
  - Dépôts Github :github.com/IUTInfoMontpellierSete-R304
- **Note TD/TP :** 25% de la note finale
- **Contrôle Amphi :** 5% de la note finale
- **Contrôle final sur table :** 70% de la note finale
- **Cours et TD/TP disponibles sur MOODLE** (R3.04 - Qualité de Développement)

Conception Orientée Objet

Concepts de base

- **Classes/Objets :** Système est un ensemble d'objets produits par des classes, des objets qui communiquent entre eux par appels de méthodes

- **Encapsulation :** accès privée de la structure de l'objet, accès publique des services (méthodes)

- **Héritage :** Spécialisation/Généralisation de classes organisées en arborescence

- **Substitution :** Une sous-classe qui prend le rôle d'une super-classe

- **Surcharge :** Différentes versions d'une même méthode selon le nombre et le type des paramètres fournis

- **Polymorphisme :** Une méthode d'une sous-classe peut modifier le comportement de la même méthode de la super-classe.

Conception Orientée Objet

Why COO?

- **Sécurité :** accès privé d'une partie d'un objet

- **Souplesse :** Les méthodes polymorphes permettent de modifier le comportement des sous-classes sans modifier le comportement des super-classes

- **Factorisation :** Réutilisation du code des super-classes

- **Réutilisation :** Faire appel aux services des objets sans avoir à comprendre comment le service est réalisé

Conception Orientée Objet

Maintenance et évolutivité

- **Rigidité :** Effet avalanche suite à une petite modification dans la conception / code

- **Fragilité :** Conception / code en cristal sensible aux modifications

- **Immobilisme :** Conception / code impossible à réutiliser

- **Viscosité :** Conception / code à réviser au lieu de le réutiliser

- **Opacité :** Conception / code difficile à comprendre

Conception Orientée Objet

Maintenance et évolutivité

- **Rigidité :** Effet avalanche suite à une petite modification dans la conception / code

- **Fragilité :** Conception / code en cristal sensible aux modifications

- **Immobilisme :** Conception / code impossible à réutiliser

- **Viscosité :** Conception / code à réviser au lieu de le réutiliser

- **Opacité :** Conception / code difficile à comprendre

Principes SOLID !

Conception Orientée Objet

Principes SOLID

- **Agile Software Development, Principles, Patterns and Practices.**
  Robert C. Martin, 2002

- **S**ingle responsibility principle - Responsabilité unique
- **O**pen close principle - Overt/fermé
- **L**iskov principle - Substitution de Liskov
- **I**nterface segregation principle - Ségrégation des interfaces
- **D**ependency inversion principle - Inversion des dépendances

Principes SOLID

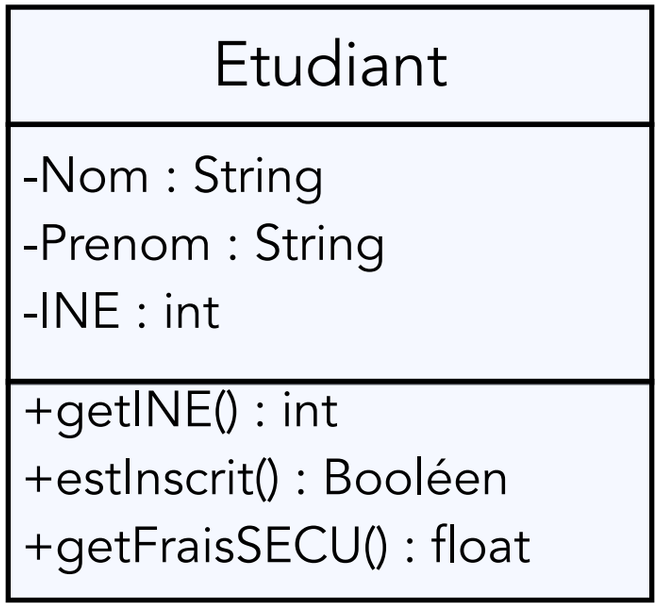
Single responsibility principle

Principe : Si une classe a plus d'une responsabilité, ces dernières seront couplées. Les modifications apportées à une responsabilité impacteront les autres, augmentant la **rigidité** et la **fragilité** de la conception / du code.

Principes SOLID

Single responsibility principle

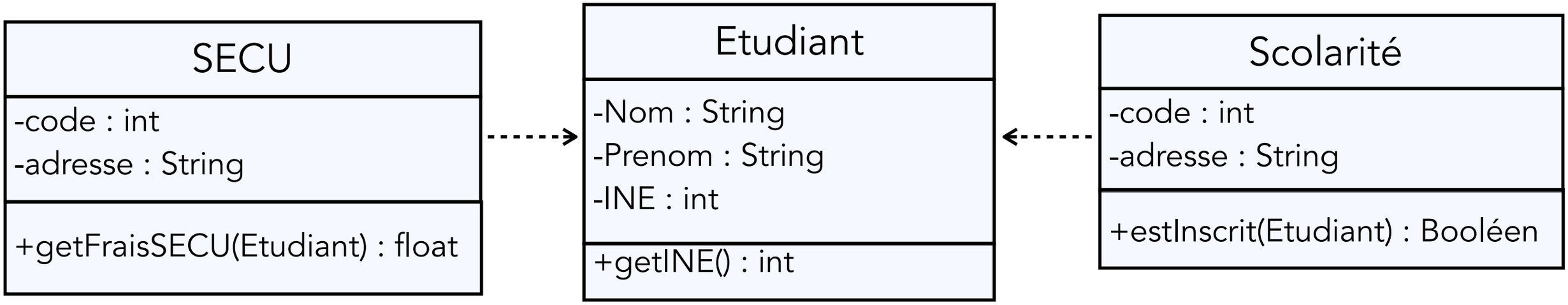
Principe : Si une classe a plus d'une responsabilité, ces dernières seront couplées. Les modifications apportées à une responsabilité impacteront les autres, augmentant la **rigidité** et la **fragilité** de la conception / du code.



Principes SOLID

Single responsibility principle

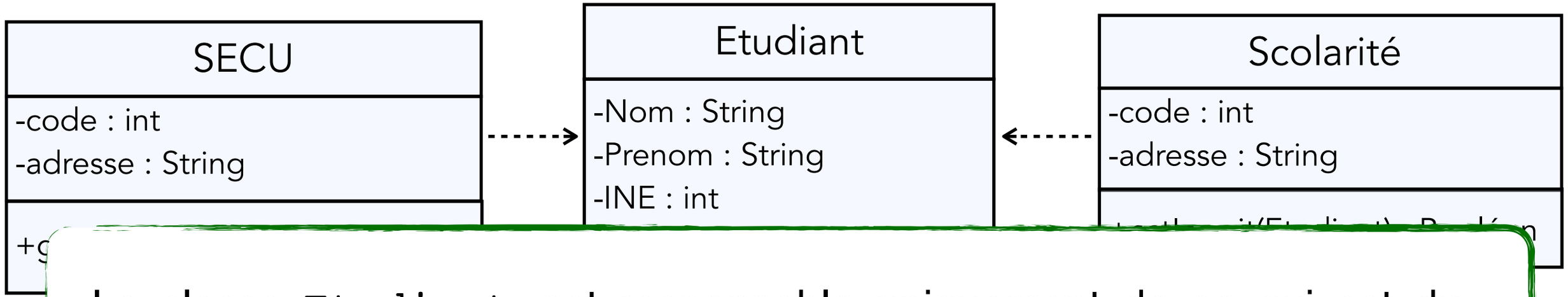
Principe : Si une classe a plus d'une responsabilité, ces dernières seront couplées. Les modifications apportées à une responsabilité impacteront les autres, augmentant la **rigidité** et la **fragilité** de la conception / du code.



Principes SOLID

Single responsibility principle

Principe : Si une classe a plus d'une responsabilité, ces dernières seront couplées. Les modifications apportées à une responsabilité impacteront les autres, augmentant la **rigidité** et la **fragilité** de la conception / du code.



Principes SOLID

Open close principle

Principe : Ouvert aux extensions, fermé aux modifications. Une classe doit être extensible sans être modifiée.

(Conception et programmation orientées objet, 2000, B. Meyer)

Principes SOLID

Open close principle

Principe : Ouvert aux extensions, fermé aux modifications. Une classe doit être extensible sans être modifiée.

(Conception et programmation orientées objet, 2000, B. Meyer)

```
public Shape(ShapeTypeEnum ShapeType) {
     if (ShapeType == ShapeTypeEnum.CIRCLE) {
          Shape = new Circle() ;
     } else if (…) {
          …
     }
}
```

✘

Principes SOLID

Open close principle

Principe : Ouvert aux extensions, fermé aux modifications. Une classe doit être extensible sans être modifiée.

(Conception et programmation orientées objet, 2000, B. Meyer)

```
public Shape (ShapeType shapeType) {
         shape = ShapeFactory.getShape(shapeType) ;
          }
```

Principes SOLID

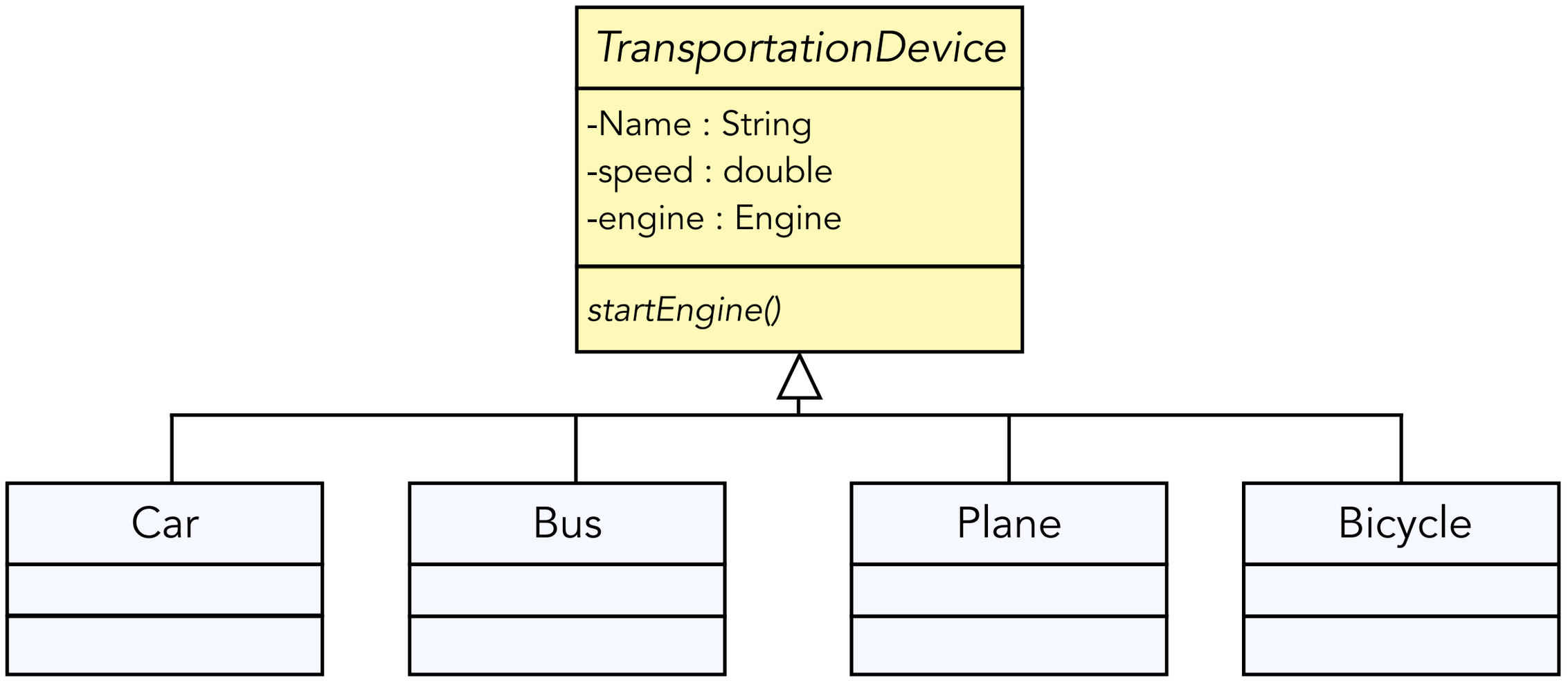
Liskov substitution principle

Principe : Les sous-classes doivent pouvoir jouer le rôle de leur super-classe sans aucun problème

Principes SOLID

Liskov substitution principle

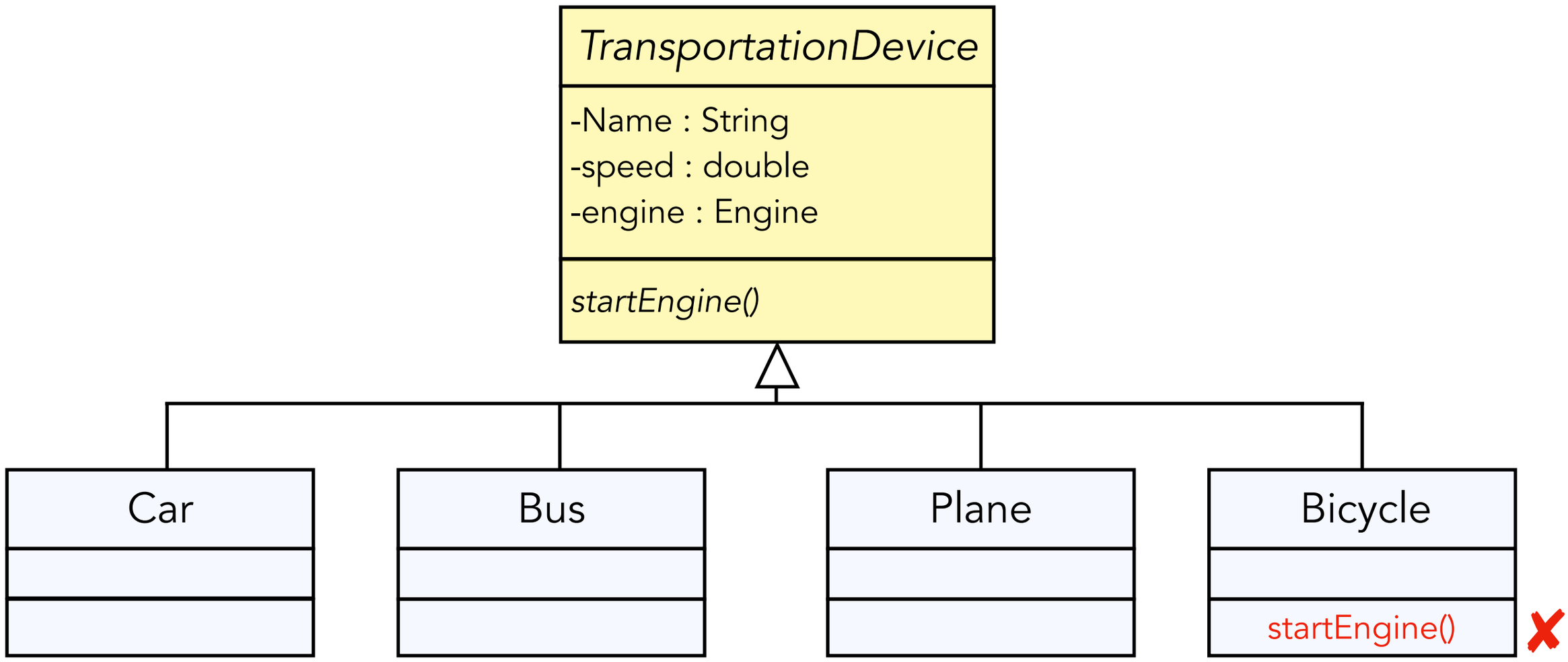
Principe : Les sous-classes doivent pouvoir jouer le rôle de leur super-classe sans aucun problème



Principes SOLID

Liskov substitution principle

Principe : Les sous-classes doivent pouvoir jouer le rôle de leur super-classe sans aucun problème



Principes SOLID

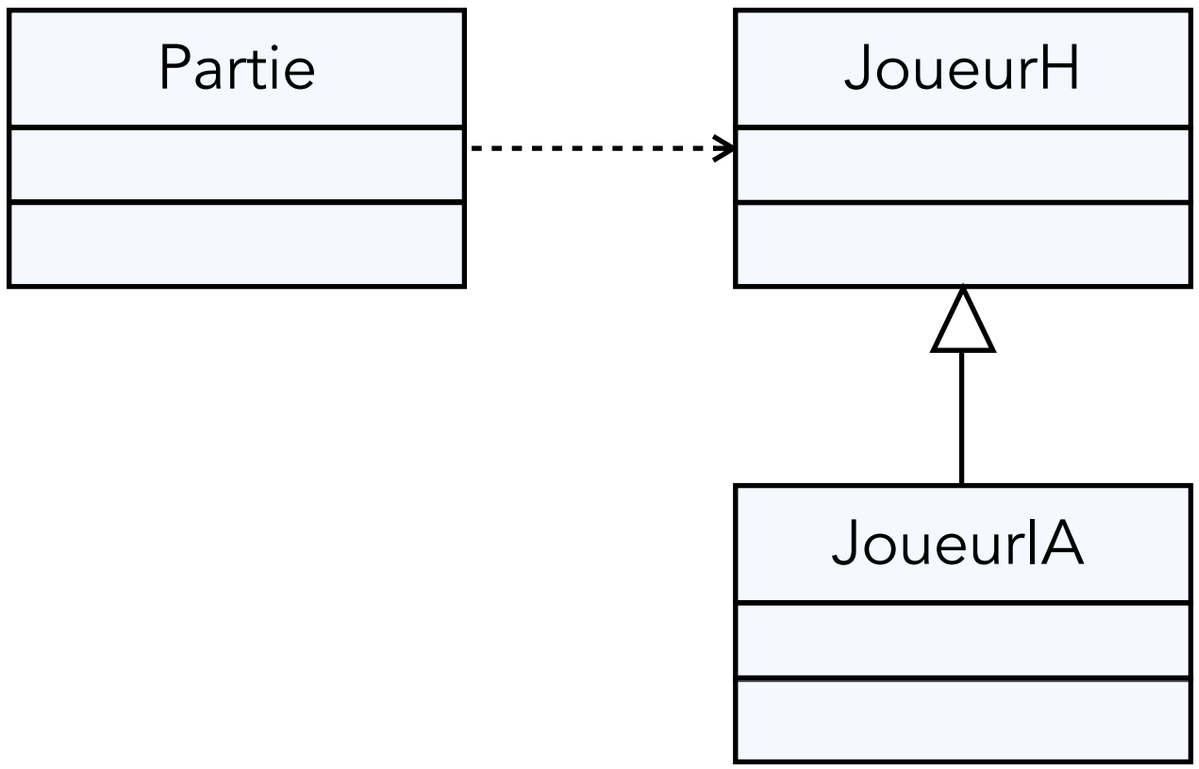
Interface segregation principle

Principe : Utiliser les interfaces pour définir les contrats

Principes SOLID

Interface segregation principle

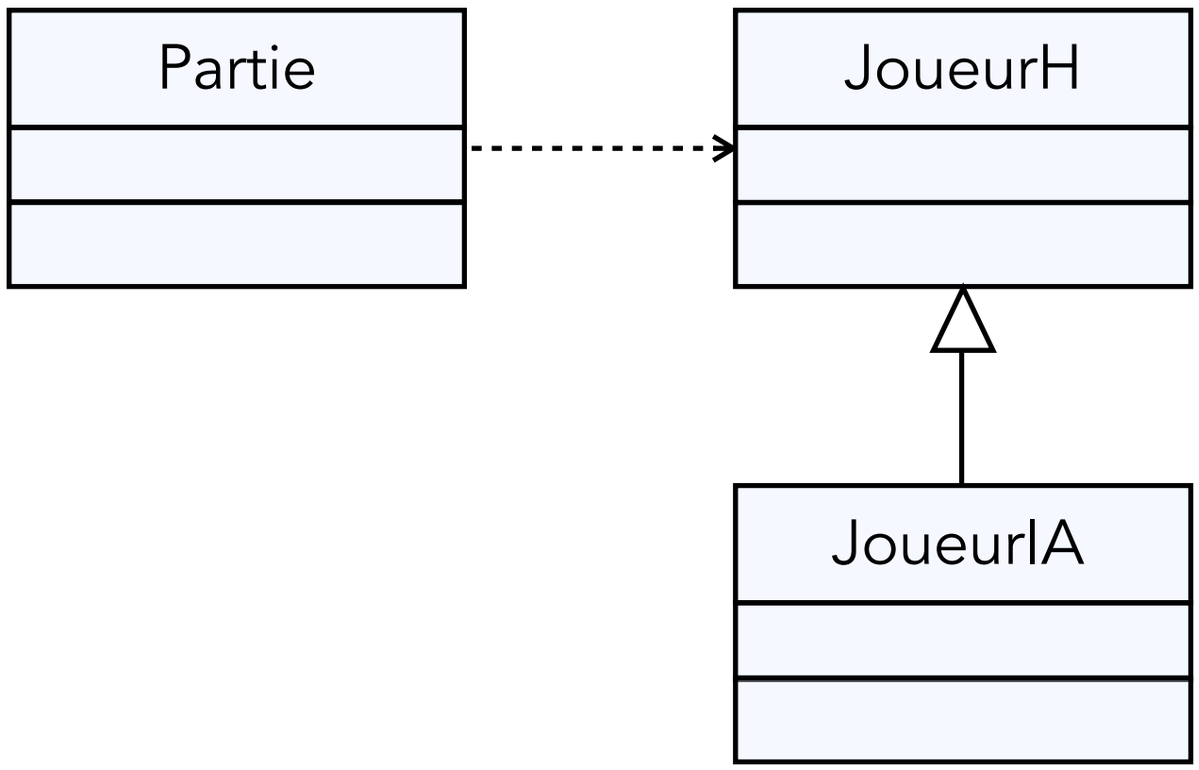
Principe : Utiliser les interfaces pour définir les contrats



Principes SOLID

Interface segregation principle

Principe : Utiliser les interfaces pour définir les contrats

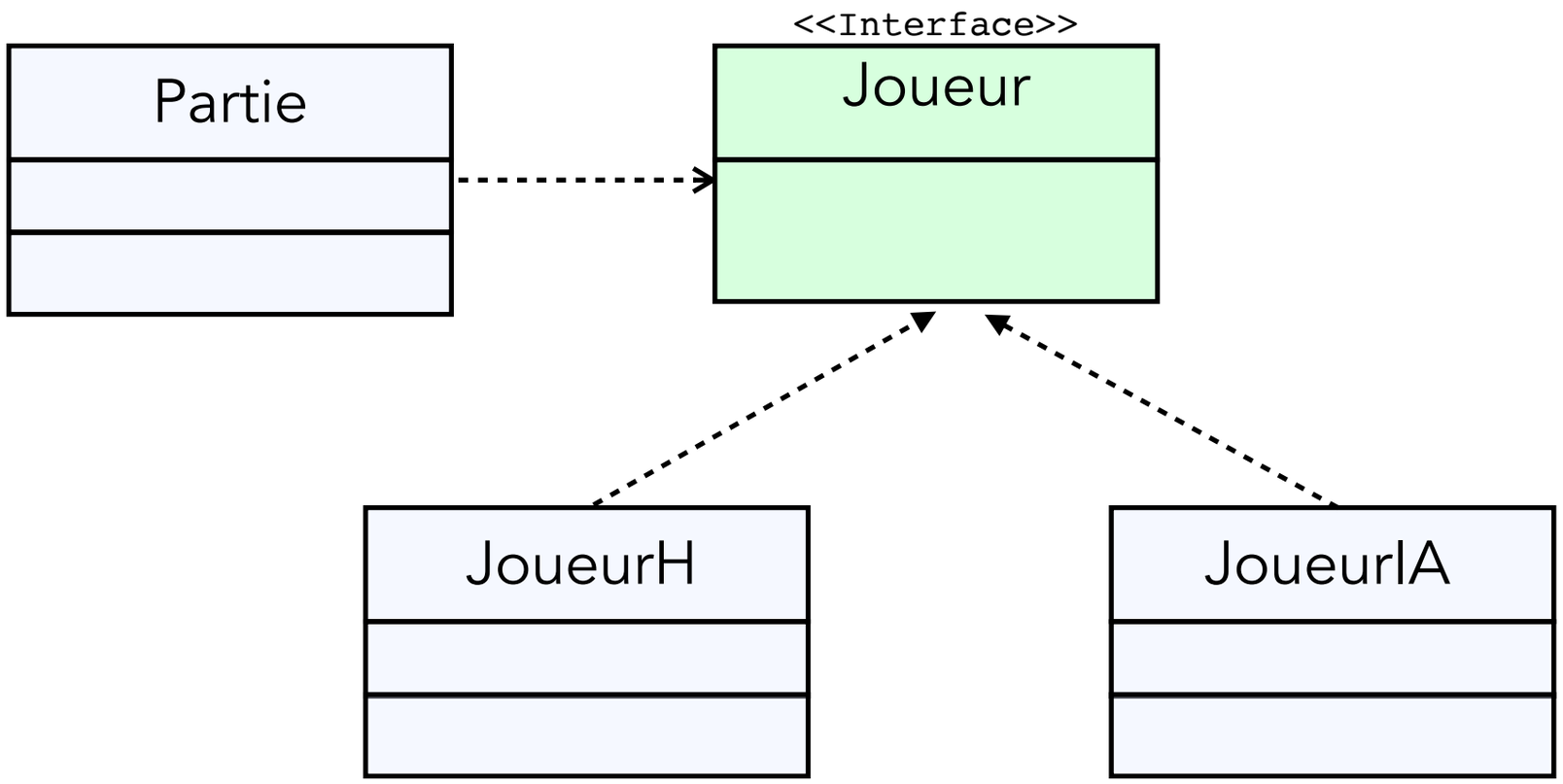


Le joueur IA hérite de tout le code du joueur humain

Principes SOLID

Interface segregation principle

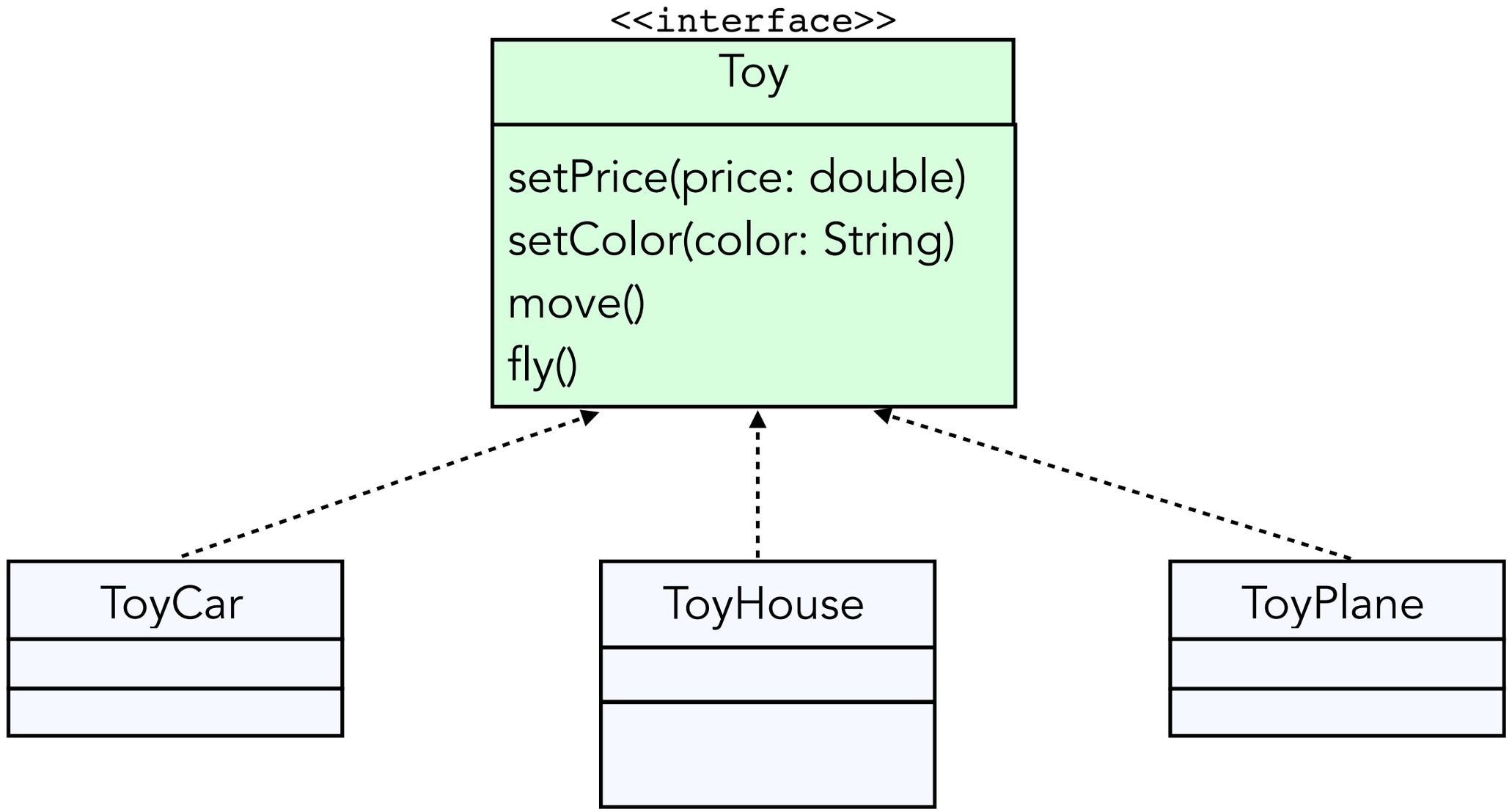
Principe : Utiliser les interfaces pour définir les contrats



Principes SOLID

Interface segregation principle

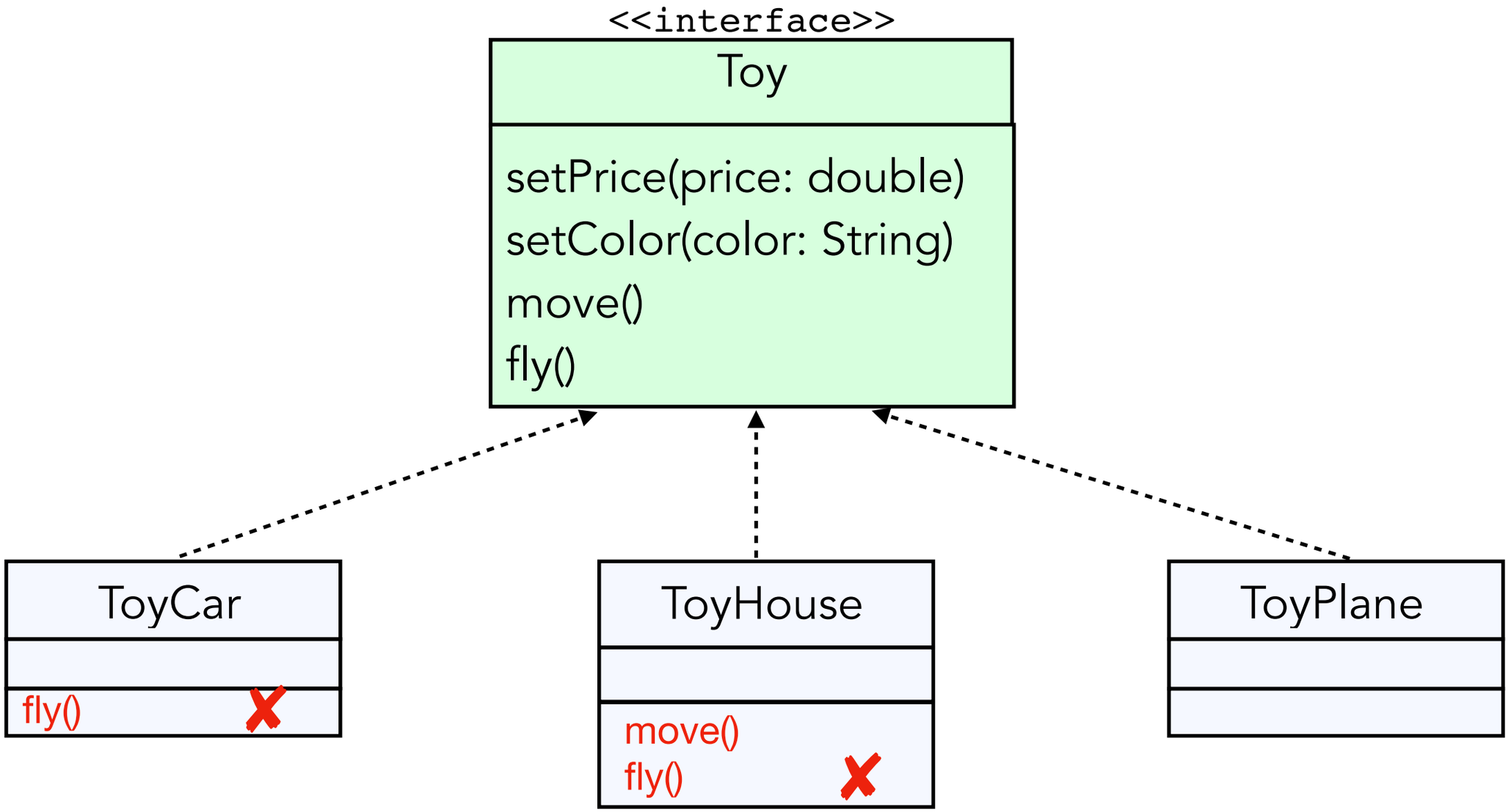
Principe : Utiliser les interfaces pour définir les contrats



Principes SOLID

Interface segregation principle

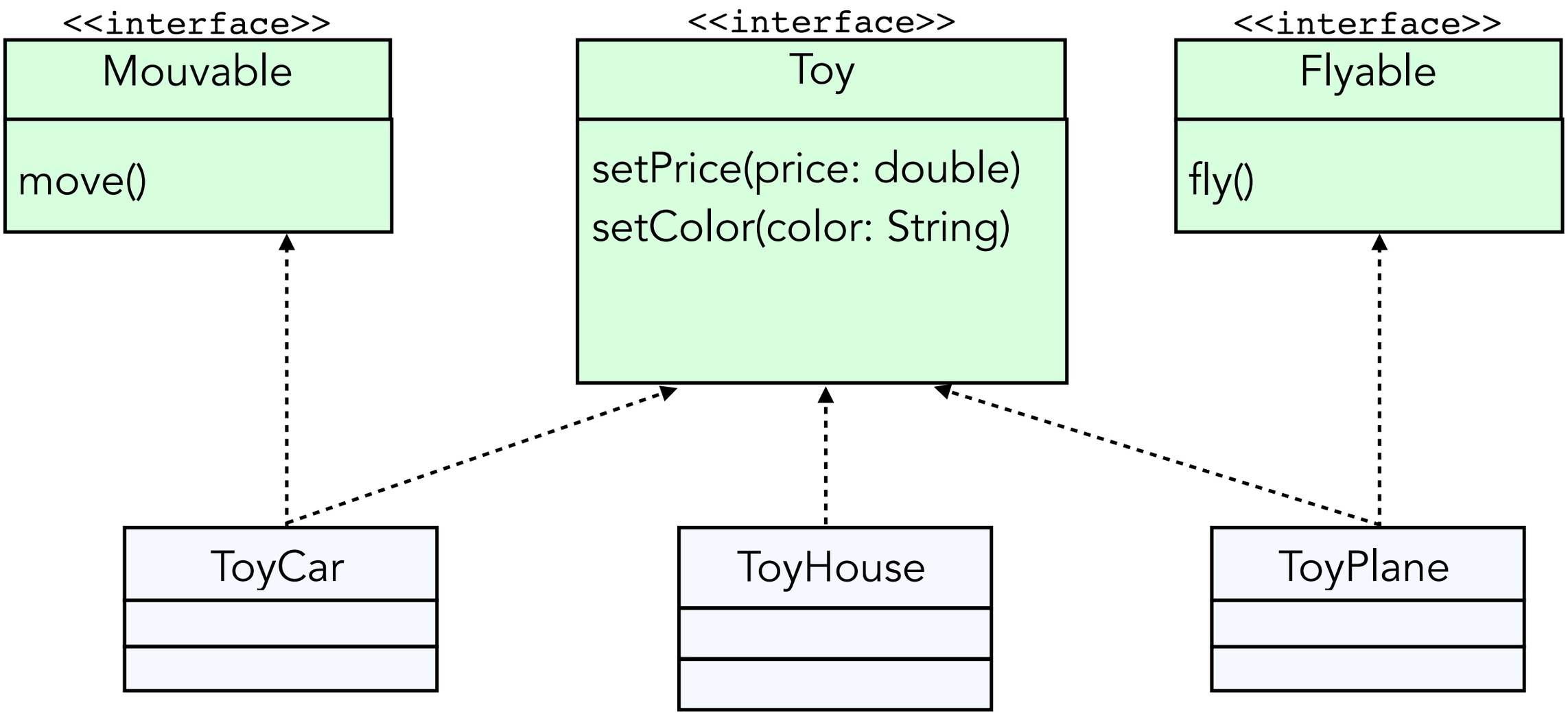
Principe : Utiliser les interfaces pour définir les contrats



Principes SOLID

Interface segregation principle

Principe : Utiliser les interfaces pour définir les contrats



Principes SOLID

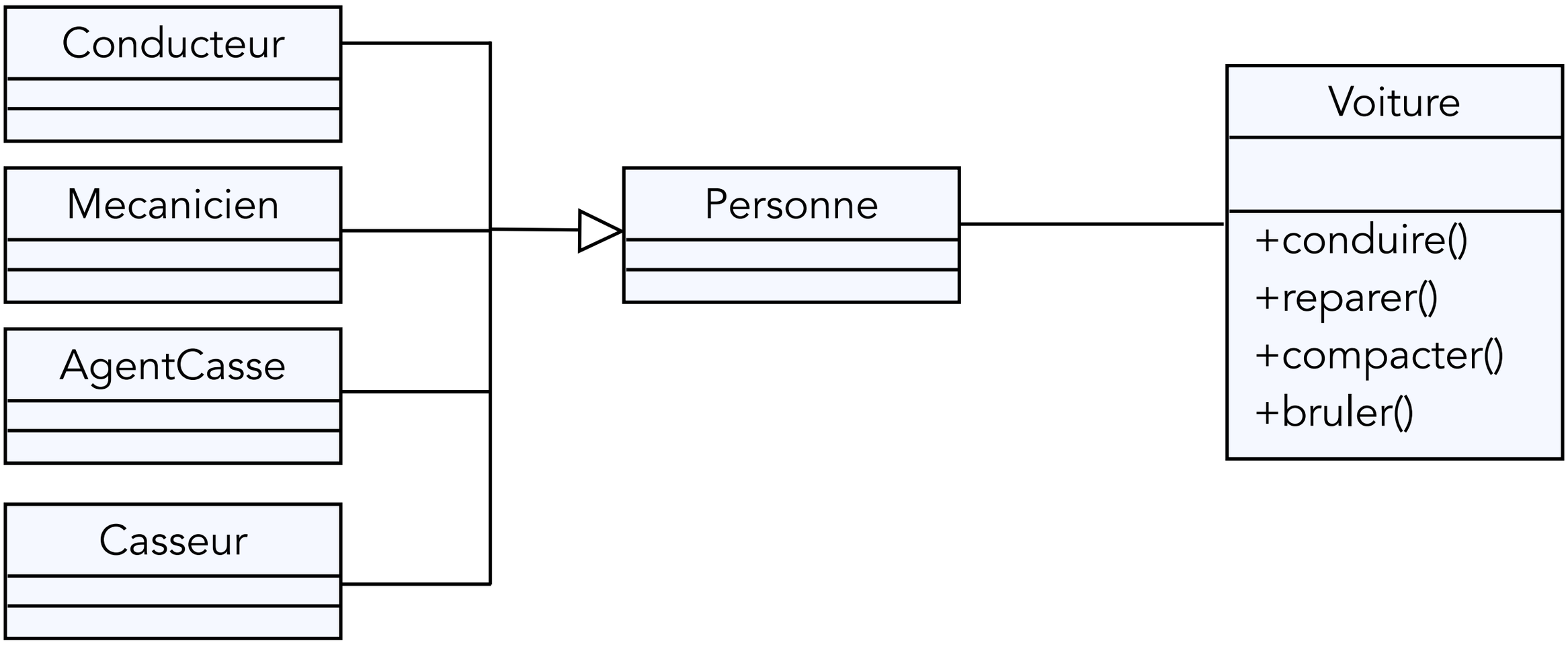
Dependency inversion principle

Principe : Non-dépendant à l'inutile => Multiplier les interfaces, plus simples, thématiquement cohérentes

Principes SOLID

Dependency inversion principle

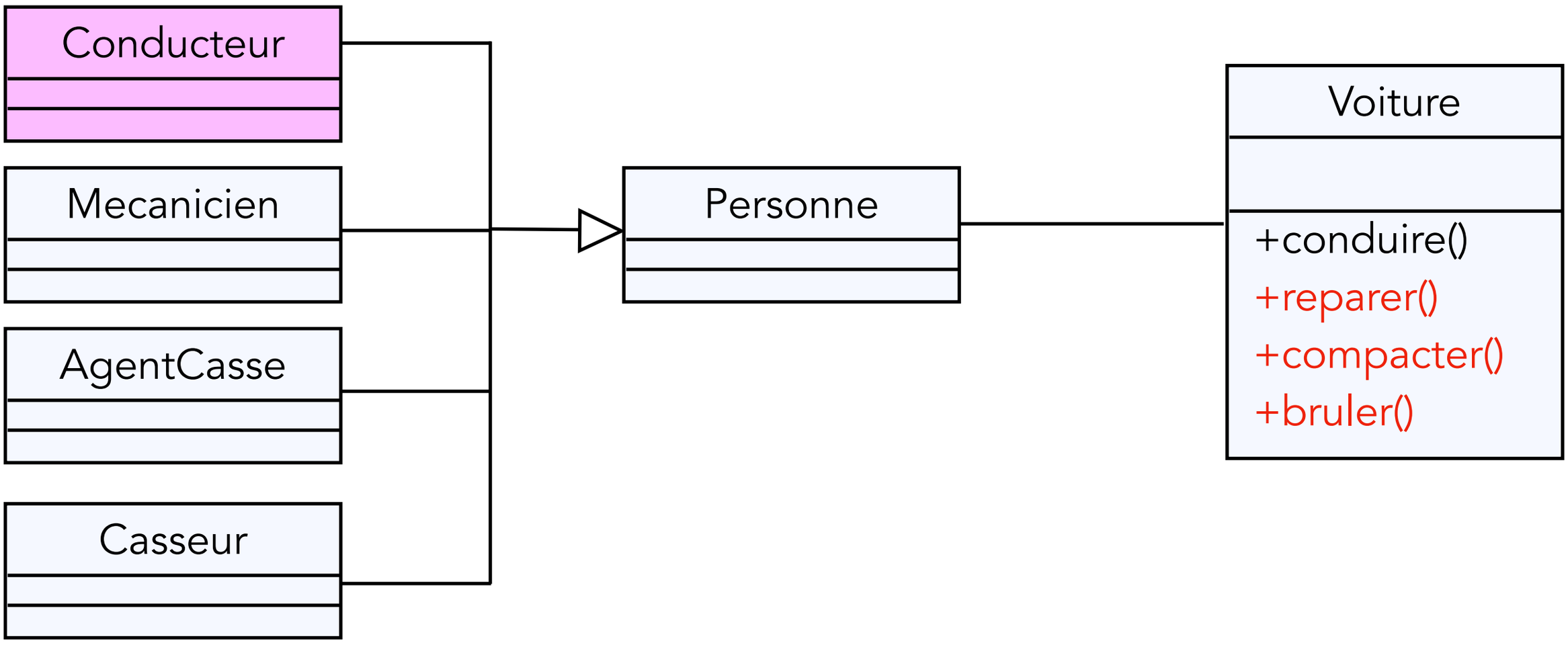
Principe : Non-dépendant à l'inutile => Multiplier les interfaces, plus simples, thématiquement cohérentes



Principes SOLID

Dependency inversion principle

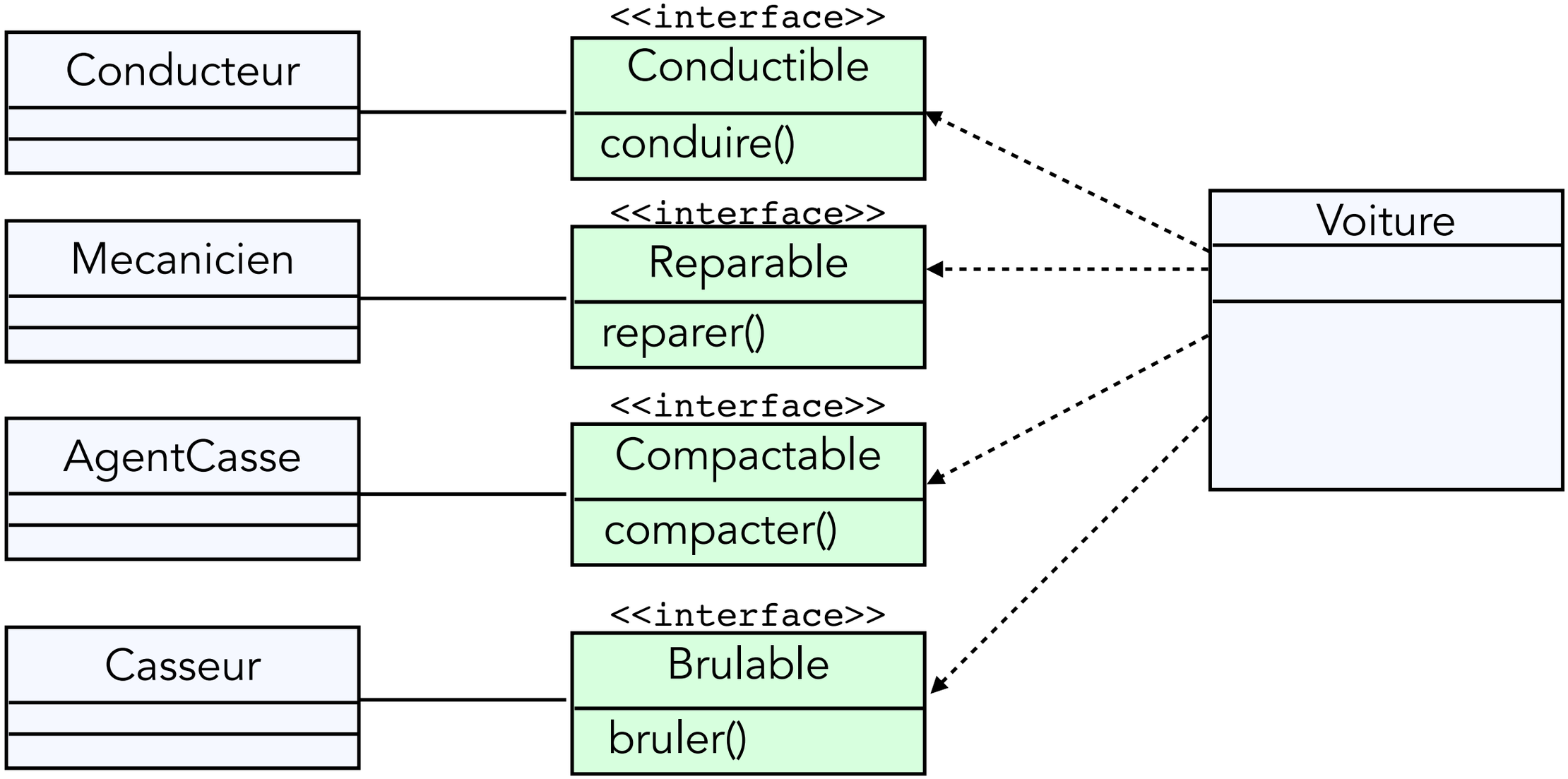
Principe : Non-dépendant à l'inutile => Multiplier les interfaces, plus simples, thématiquement cohérentes



Principes SOLID

Dependency inversion principle

Principe : Non-dépendant à l'inutile => Multiplier les interfaces, plus simples, thématiquement cohérentes



Principes SOLID

Dependency inversion principle

Principe : Non-dépendant à l'inutile => Multiplier les interfaces, plus simples, thématiquement cohérentes

```java
/** Une voiture */
public class Voiture implements
        Conductible,Reparable,Compactable,Brulable{ ... }

/** Un conducteur lambda */
public class Conducteur {
   private Conductible vehicule;
   /** Crée un nouveau conducteur */
   public Conducteur(Conductible v) {
      vehicule = v;
      vehicule.conduire(); // Fait un tour d'essai
   }
}
```

Pattern

Patron ou Modèle

Un pattern décrit à la fois un problème qui se produit très fréquemment dans l'environnement et l'architecture de la solution à ce problème de telle façon que l'on puisse utiliser cette solution des milliers de fois sans jamais l'adapter deux fois de la même manière.

C. Alexander 1977

Pattern

Patron ou Modèle

Un pattern décrit à la fois un problème qui se produit très fréquemment dans l'environnement et l'architecture de la solution à ce problème de telle façon que l'on puisse utiliser cette solution des milliers de fois sans jamais l'adapter deux fois de la même manière.

C. Alexander 1977

- Pattern Language : Towns, Buildings Construction (Alexander, Ishikouwa, et Silverstein 1977)

Pattern

Patron ou Modèle

- **Coad [Coad92] –** Une abstraction d'un doublet, triplet ou d'un ensemble de classes qui peut être réutilisé encore et encore pour le développement d'applications

- **Appleton[Appleton97] –** Une règle tripartite exprimant une relation entre un certain contexte, un certain problème qui apparaît répétitivement dans ce contexte et une certaine configuration logicielle qui permet la résolution de ce problème

- **Aarsten [Aarsten96] –** Un groupe d'objets coopérants liés par des relations et des règles qui expriment les liens entre un contexte, un problème de conception et sa solution

Pattern

Patron ou Modèle

- **Coad [Coad92] –** Une abstraction d'un doublet, triplet ou d'un ensemble de classes qui peut être réutilisé encore et encore pour le développement d'applications

- **Appleton[Appleton97] –** Une règle tripartite exprimant une relation entre un certain contexte, un certain problème qui apparaît répétitivement dans ce contexte et une certaine configuration logicielle qui permet la résolution de ce problème

- **Aarsten [Aarsten96] –** Un groupe d'objets coopérants liés par des relations et des règles qui expriment les liens entre un contexte, un problème de conception et sa solution

Pattern : Une solution standard, utilisable dans la conception de différents logiciels

Design Patterns

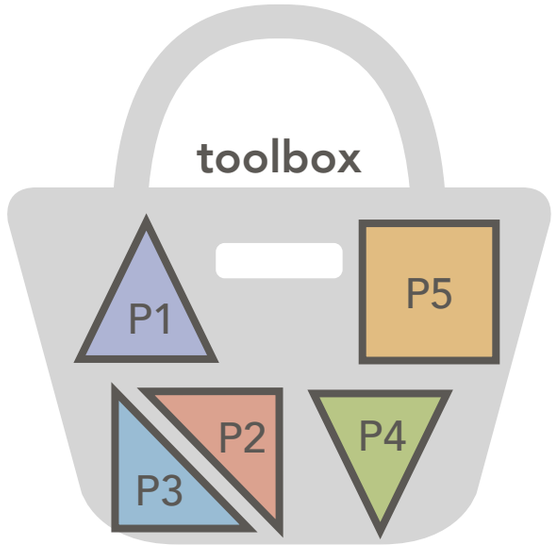
Modèles ou Patrons de Conception

Design Patterns

Modèles ou Patrons de Conception

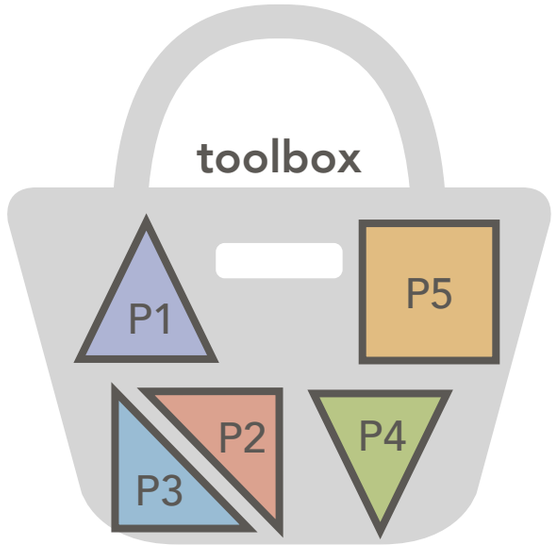
Design Patterns

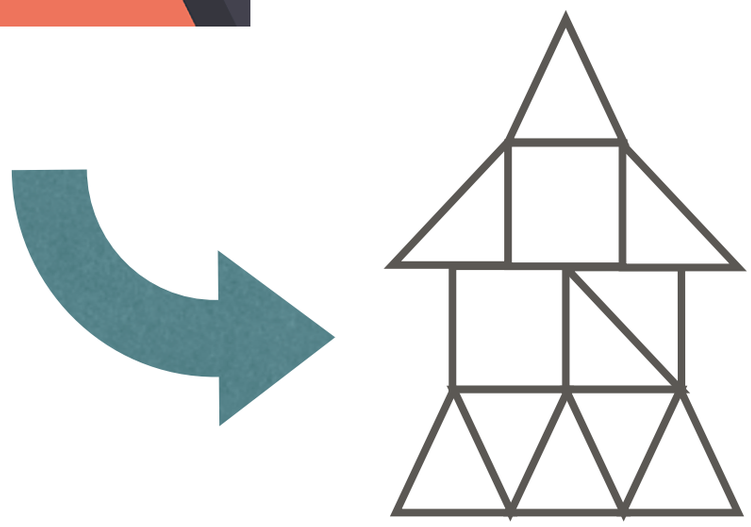
Modèles ou Patrons de Conception



Design Patterns

Modèles ou Patrons de Conception

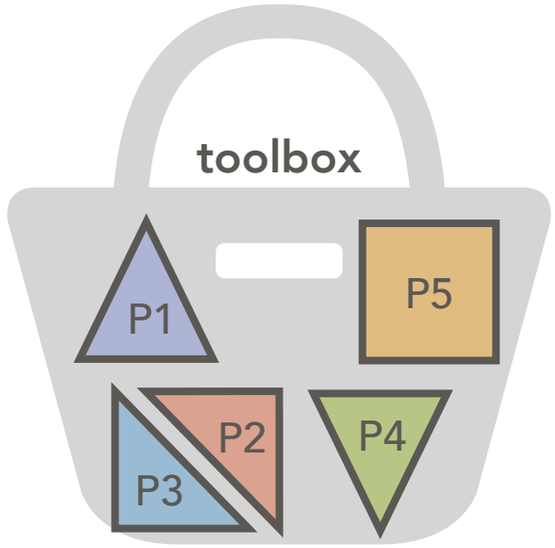


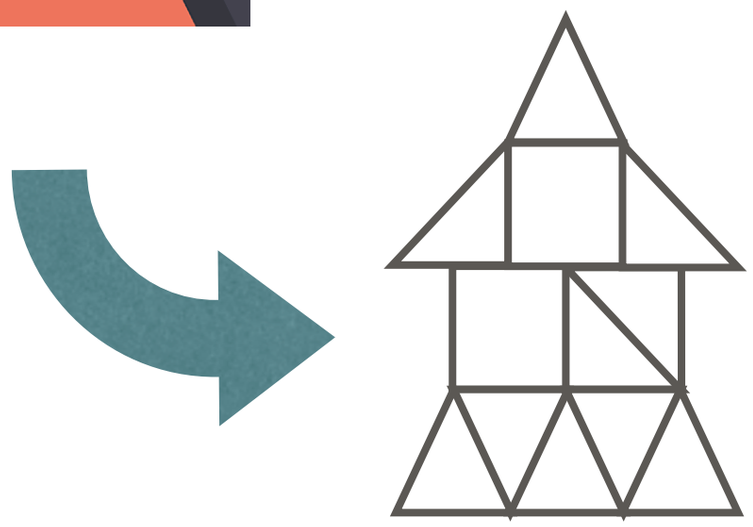


Règles de découpage

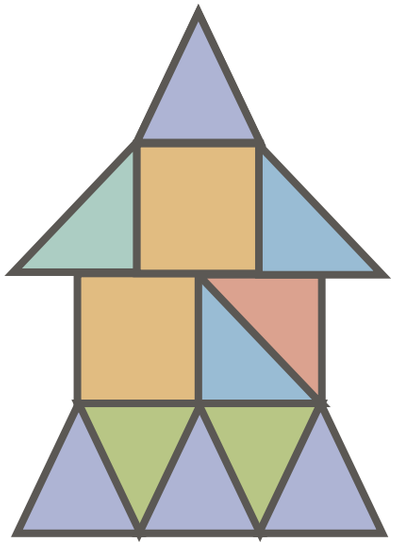
Design Patterns

Modèles ou Patrons de Conception





Règles de découpage



Règles de composition

Patterns Categories

- **Patterns de construction :** Ces patterns sont très courants pour déléguer à d'autres classes la construction des objets

- **Patterns de structure :** Ces patterns tendent à concevoir des agglomérations de classes avec des macro-composants

- **Patterns de comportement :** Ces patterns tentent de répartir les responsabilités entre chaque classe (l'usage est plutôt dynamique)

Patterns Categories

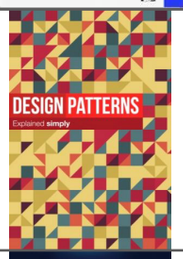
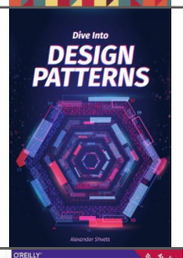
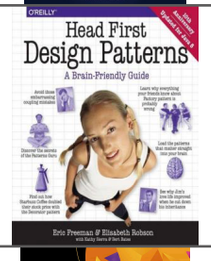
- **Patterns de construction :** Ces patterns sont très courants pour déléguer à d'autres classes la construction des objets

- **Patterns de structure :** Ces patterns tendent à concevoir des agglomérations de classes avec des macro-composants

- **Patterns de comportement :** Ces patterns tentent de répartir les responsabilités entre chaque classe (l'usage est plutôt dynamique)

Books

Design Patterns

• **Design Patterns: Elements of Reusable Object-Oriented Software.** Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (GoF: Gang of Four).	
• **DESIGN PATTERNS Explained simply.** Alexander Shvets. 2013	
• **Dive Into Design Patterns.** Alexander Shvets. 2019	
• **Head First Design Patterns.** Freeman et al. 2014	
• **Java Design Patterns.** Vaskaran Sarcar. 2019