ENSTA : cours IN204
Introduction à JAVA et UML

Olivier Sigaud
LIP6/ AnimatLab
olivier.sigaud@lip6.fr
01.44.27.88.53

Plan du cours 13

- Design Patterns
- Intérêt des patterns
- Quelques patterns
  - Factory Method
  - Singleton
  - AbstractFactory
  - Bridge
  - Adapter
  - Observer…

Design patterns

Le métier de concepteur

- 25 ans d'expérience de la POO
- Des projets de très grande taille
- Constitution progressive d'une véritable expertise dans la réalisation d'architectures de programmes orientés objets
- Les design patterns sont nés de la capitalisation de cette expertise

Généralités

- Les design patterns (patrons de conception) sont des solutions standardisées à des problèmes de conception classiques
- Proposés par des experts de la POO
- Utiles pour les applications de grande taille
- Il existe 23 patterns qui font l'unanimité
- Ouvrage de référence : << Design Patterns (catalogue de modèles de conception réutilisables), Gamma, Helm, Johnson et Vlissides >> (*gang of four*)

Nature des patterns

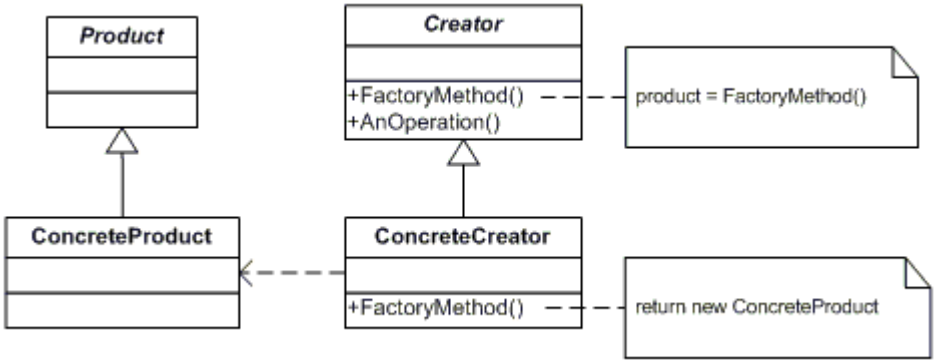
- Solutions indépendantes des langages
- Solutions abstraites, de haut niveau
- Souvent orientées vers le bon découpage en packages (modularité, flexibilité, réutilisabilité)
- Exprimées sous forme d'architecture reliant quelques classes très abstraites
- Reposent beaucoup sur des interfaces

Nouveaux patterns

- Pour devenir un pattern, un modèle de conception doit vérifier les contraintes suivantes :
  - Être présent dans au moins 3 très gros projets très largement utilisés
  - Se retrouver dans des programmes écrits dans des langages objets différents (c++, Eiffel, Smalltalk, Java, c#, O'CAML…)
  - Ne pas être décomposable en patterns plus simples
  - Être documenté complètement
- Seuls les 23 patterns initiaux ont franchi ce seuil, en 10 ans…

Patterns de création

Le pattern Factory Method



Justification

- Dans un Framework générique, il peut être nécessaire de créer des instances des objets manipulés
- Ces instances sont des objets de type spécifiques
- Donc leur type n'est pas connu par le concepteur du Framework
- Il ne peut pas écrire :

  MonInstanceGénérique = new MonTypeSpecifique ();
- Donc il écrit MaFactory.creeMonInstanceGénérique ();
- Méthode écrite par l'utilisateur, qui encapsulera la création de l'objet spécifique

Factory

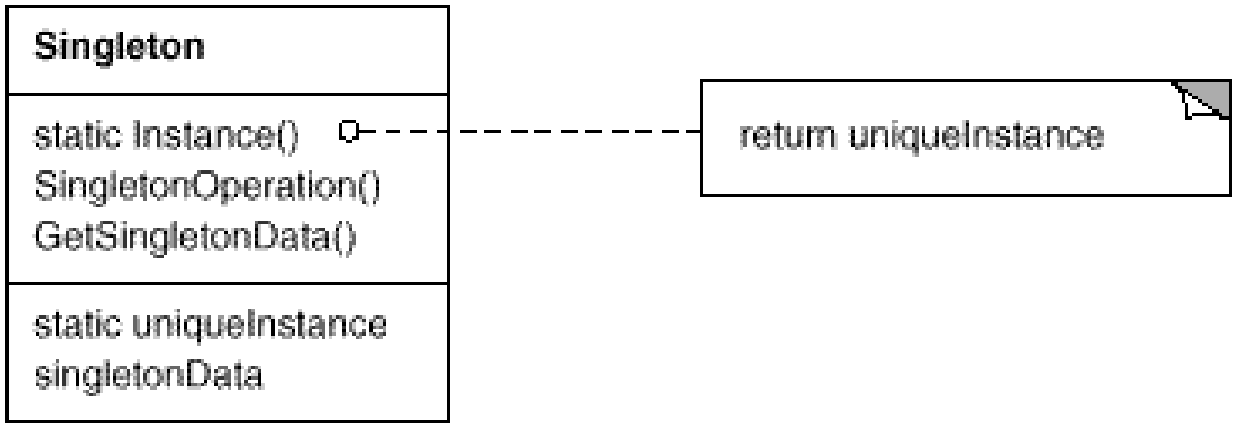
- Classe à laquelle est déléguée la création des objets spécifiques
- Définie dans le Framework comme une interface
- Implémentée par l'utilisateur

```
interface MaFactoryGenerique
{
        MonInstanceGénérique creeMonInstanceGénérique();
}

class MaFactoryConcrete implements MaFactoryGenerique
{
    MonInstanceGénérique creeMonInstanceGénérique()
        {
            return new MonInstanceConcrete();
        }
}
```

- Va correspondre à un besoin important dans votre projet in204

Le pattern Singleton

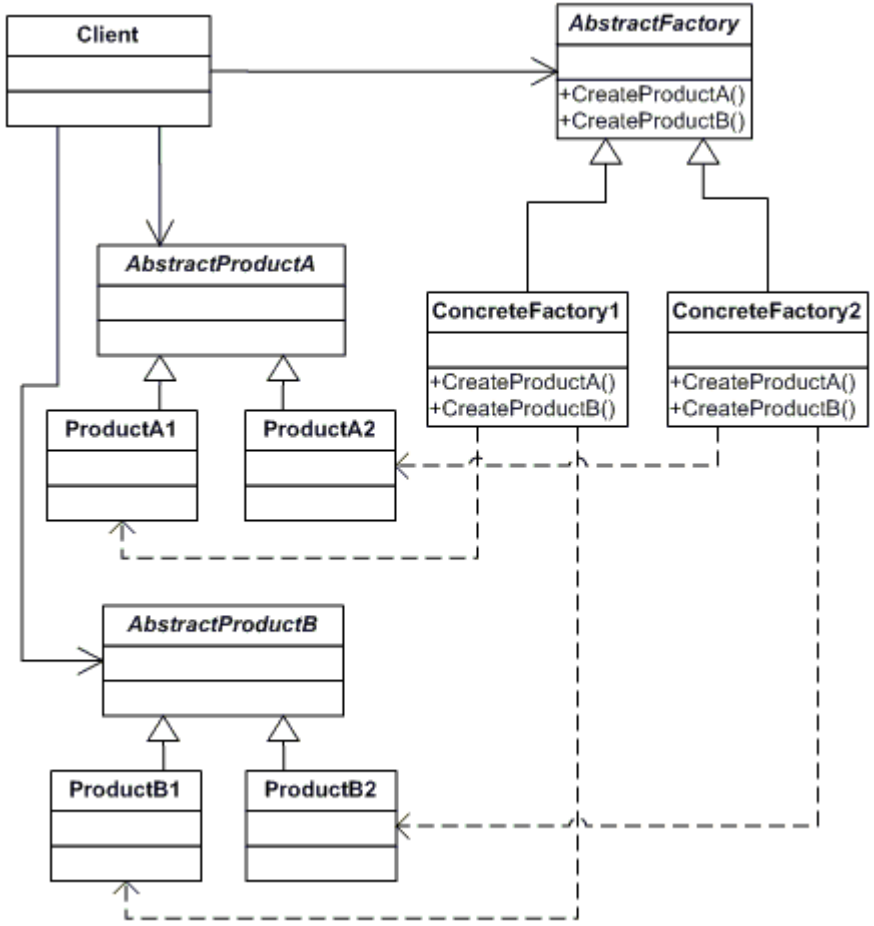


Il s'applique fréquemment à des objets de type « Manager »

Exemple

- Cas du générateur de nombres aléatoires
- Voir classe StaticRandom dans MACS
- Permet de garantir la répétition du comportement au moyen d'une seule graine
- Autres exemples :
  - La plupart des « managers »

Le pattern AbstractFactory



Justification

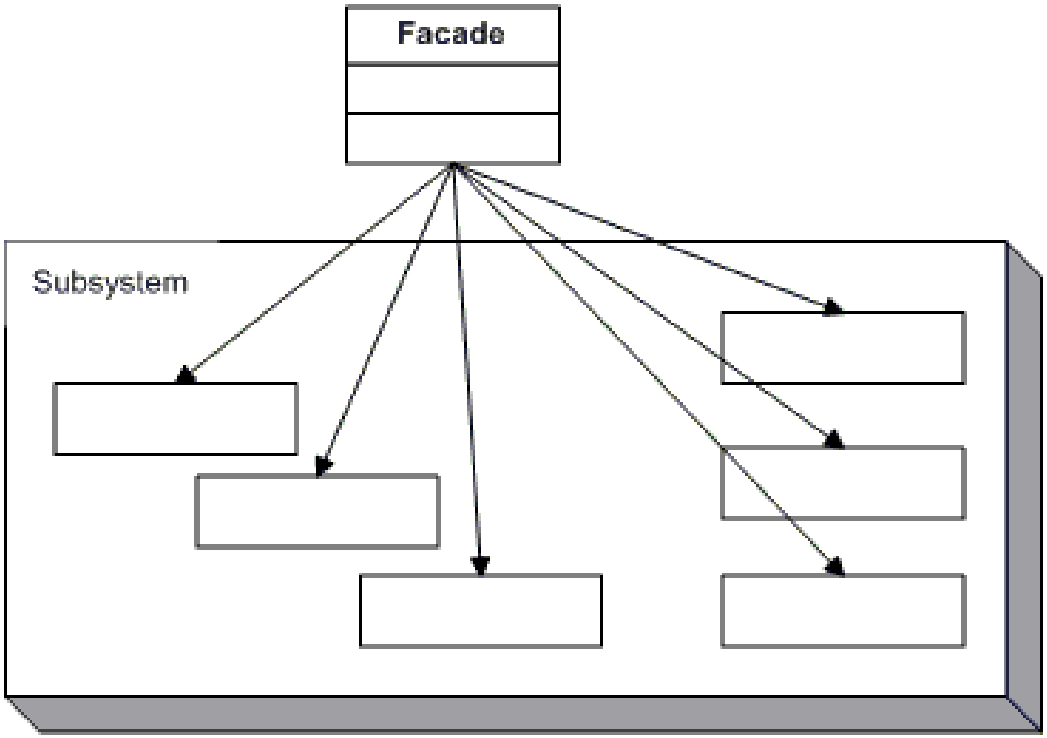
- Sorte de généralisation de l'idée de Factory dans le cas où on veut engendrer des groupes d'objets liés les uns aux autres

Autres patterns de création

- Builder : permet de créer des composants morceaux par morceau progressivement (chaîne de montage)
- Prototype : créer un prototype puis le dupliquer en créant des « clones », éventuellement de sous-classes différentes

Patterns de structure

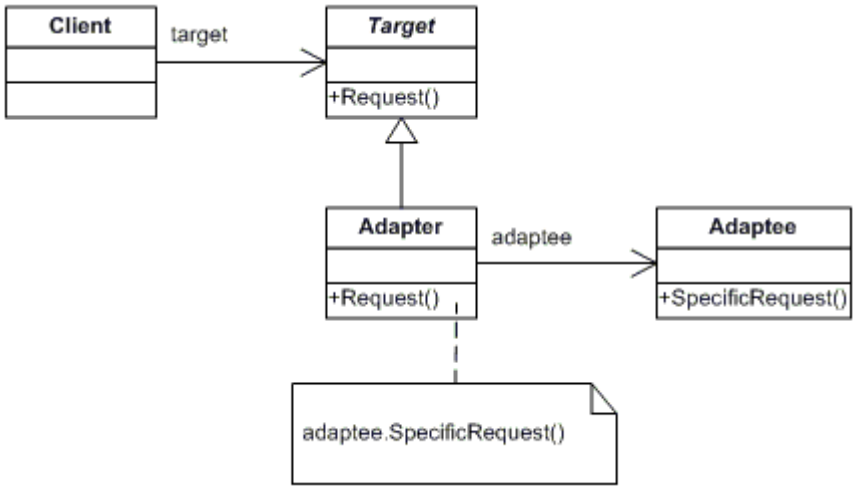
Le pattern Facade



Justification

- C’est le pattern de base de l’encapsulation
- On ajoute une classe unique qui redirige les méthodes visibles de l’extérieur vers des appels internes

Le pattern Adapter



Justification

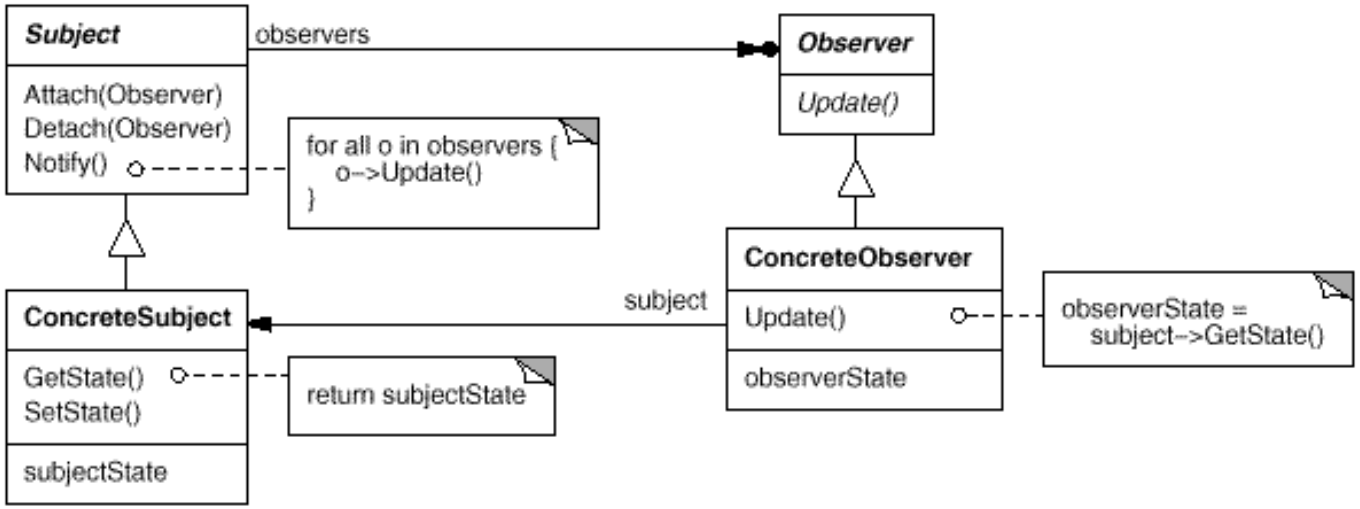
- Lorsque l'on veut se connecter à une classe dont on ne maîtrise pas l'interface (méthodes prédéfinies)
- Plutôt que de revoir sa propre conception, on glisse entre cette classe et la sienne une classe qui fait la traduction entre les deux interfaces

Autres patterns de structure

- Bridge : permet de découpler une interface de son implémentation
- Composite : composition d'objets avec imbrication récursive (ex : système de fichiers)
- Decorator : permet d'ajouter des attributs à des objets
- Flyweight : poids plume pour implanter des objets partagés
- Proxy (procuration) : utilisé pour le distribué : image locale d'un objet distant

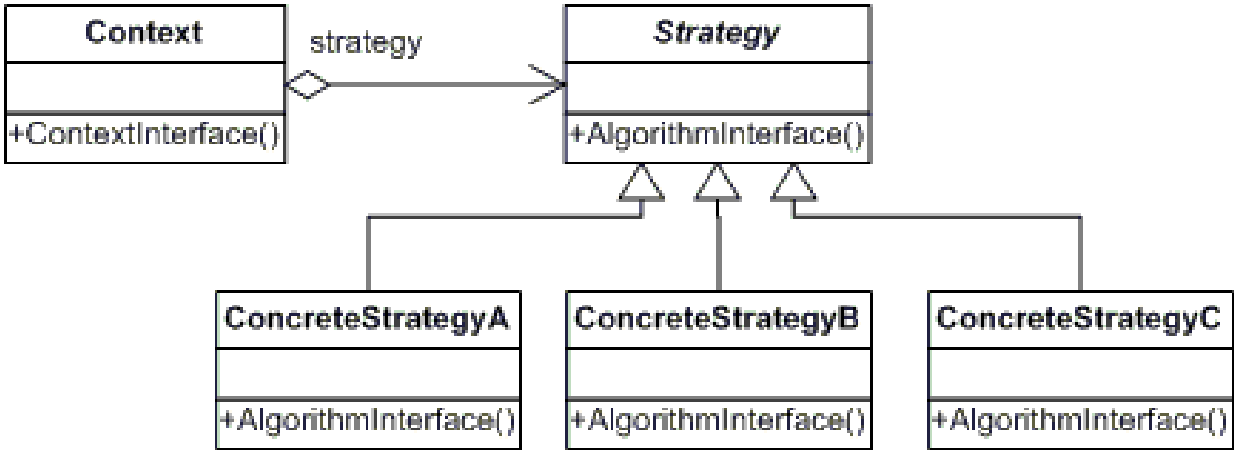
Patterns de comportement

Le pattern Observer



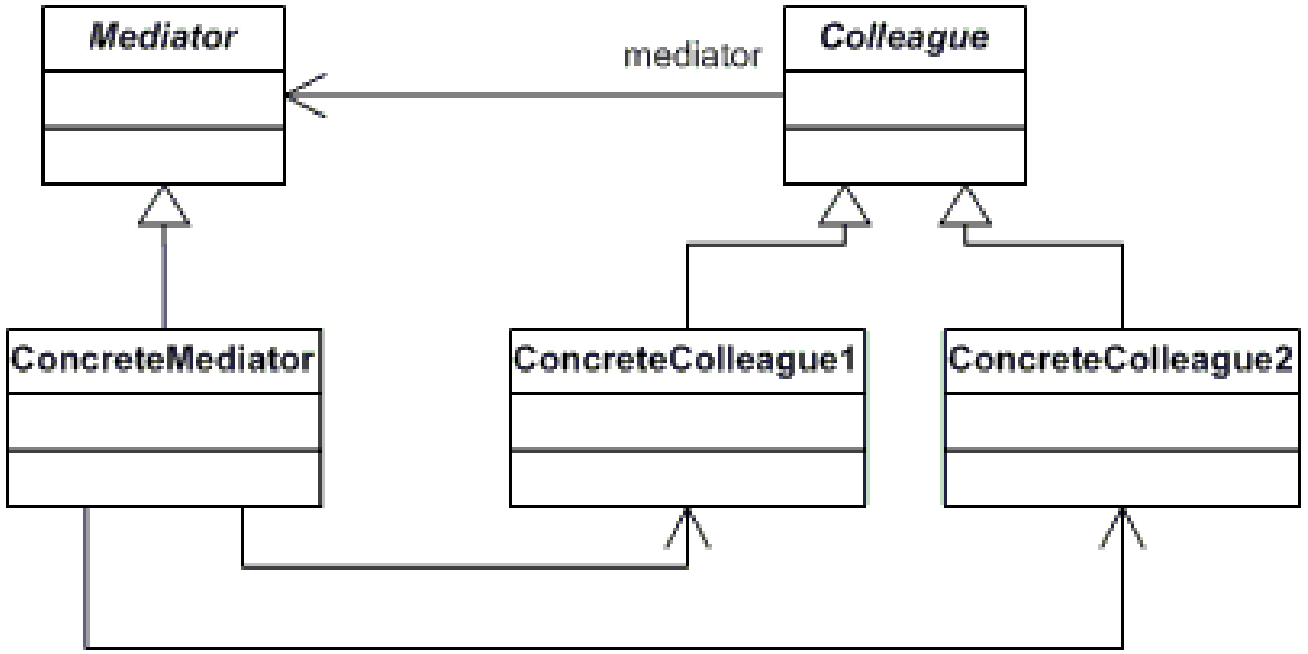
Déjà vu : facilite le découpage en packages
Sert à chaque fois qu'un objet dispose de plusieurs « listeners ». Proche de Visitor

Le pattern Strategy



Permet d'appliquer différentes stratégies (enchaînement de méthodes) aux mêmes données

Le pattern Mediator



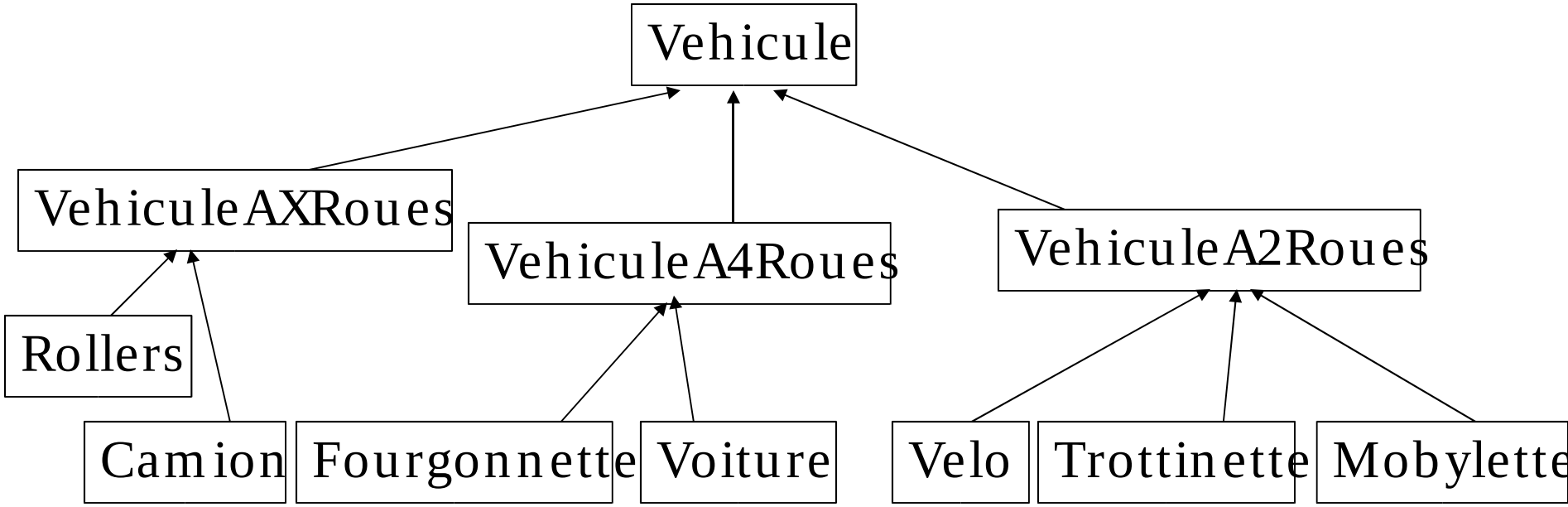
Assure un couplage faible entre objets qui doivent communiquer

Autres patterns de comportement

- Iterator : voir le cours sur les collections
- Command : commande encapsulée dans une classe
- Interpretor : interprête pour implanter un langage propre au logiciel
- Responsibility Chain : une commande qui passe par une succession d'objets
- Memento : pour sauver/restaurer l'état d'un objet (persistance)
- State : changement de comportement en fonction de l'état de l'objet
- Visitor : visiteur pour parcourir récursivement une structure composite (observer du récursif)
- Template Method : patron de méthode pour abstraire les étapes d'un algo (raffinement de Strategy)

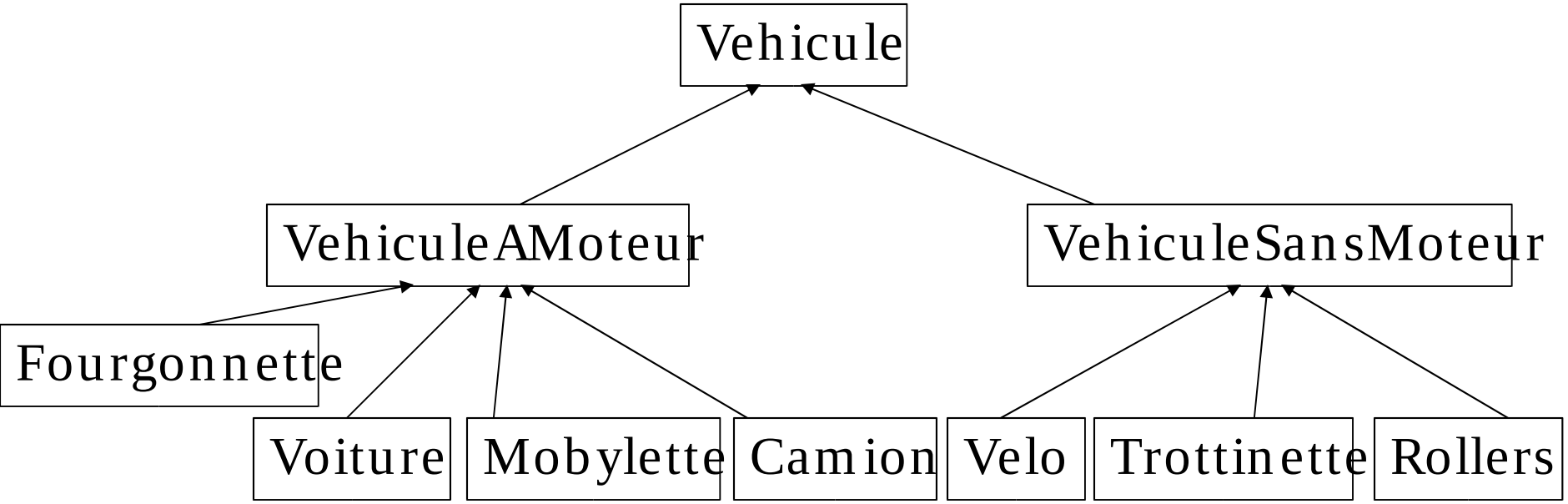
Héritage et délégation

Les hiérarchies complexes (1)



Exemple de mauvais critère de classification : le nombre de roues
Les descendants ont peu de chose en commun

Les hiérarchies complexes (2)



Cette fois, le critère est structurant

Les hiérarchies complexes (3)

Si l'on est amené à modifier une hiérarchie complexe cela implique des réorganisations majeures du code
En particulier, des attributs et des traitements doivent être partagés
Importance de la phase de conception préalable

Intérêt de l'héritage

```
class Sequenceur
{
    MonVecteur data;

    public void reverse()
    {
        data.reverse();
    }
}

class Sequenceur extends MonVecteur
{
    // pas besoin de public void reverse() : Sequenceur en hérite
}
```

Inconvénients de l'héritage

- Induit un couplage fort entre les classes de la hiérarchie : une modif de la classe mère induit des modifs de toutes les filles
- Ralentit l'exécution à cause du typage dynamique
- Les auteurs de « Design patterns » favorisent plutôt la délégation (composition) et le couplage faible

Héritage multiple : que faire ?

- Problème : pas d'héritage multiple : on ne peut pas hériter de tous les composants
- Eviter les hiérarchies trop importantes
- Choisir d'hériter du composant qui a le plus de méthodes ? Non
- Se demander si A est un B un peu spécial ou si A contient un B

Conclusion

- Dans un premier temps, ne pas avoir recours à des design patterns si ça ne répond pas à un besoin clairement identifié (ça a plutôt tendance à alourdir)
- Le métier de concepteur s'éloigne peu à peu du métier de programmeur
- Outils liés strictement à la conception, sans une ligne de code
- On va vers des conceptions « tout patterns » : permet aux concepteurs de se comprendre plus vite
- Les AGL les intègrent de plus en plus

Plus qu'un cours !