Conception objet avancée

Principes, pratiques et mise en œuvre

DUT Informatique
Semestre 3

Mourad Ouziri
mourad.ouziri@parisdescartes.fr

Partie 2

Design pattern

Etude de quelques bonnes pratiques de conception objet

Références bibliogtaphiques

- *Design Patterns: Elements of Reusable Object-Oriented Software,* Gamma, Helm, Johnson and Vlissides (*GoF*), Addison-Wesley, 1995
- *Pattern Oriented Software Architecture : A System of Patterns,* Frank Buschmann (Editor), Wiley, 1996

Design pattern

- Recettes d'experts (donc éprouvées) permettant de résoudre des problèmes de conception bien identifiés récurrents
- Architectures de classes « standardisées »
- Reposent sur les principes de conception objet déjà vus en cours
- Idée inspirée par l'architecte *Christopher Alexander* et ses collègues

  par l'ouvrage « A Pattern Language : Towns, Buildings Construction » (1977)
- Adapté à la conception du logiciel par GoF (Gang of Four) : *Gamma, Helm, Johnson et Vlissides*

  Livre de référence « Design Patterns : Elements of reusable objected-oriented softwares » (1995)

Design pattern

- Description d'un pattern
  - Nom : identification du pattern
  - Problématique : description du problème traité
  - Solution : description de l'architecture de classes proposée
  - Conséquences : effets positifs et négatifs du pattern
  - Autres:
    - Exemples d'implantation
    - Conseil d'implantation

Catégories de design pattern

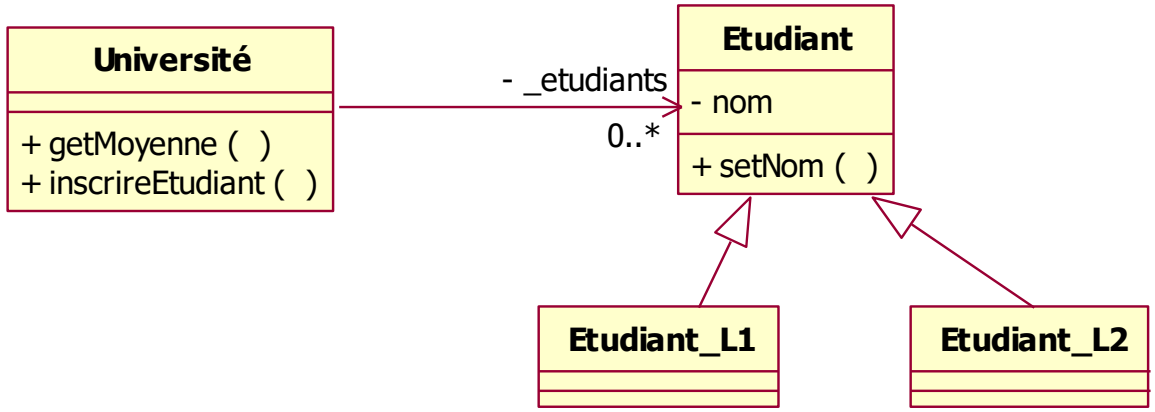
- Création
  - Description de la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés et configurés
  - ***Factory, Factory Method, Abstract Factory, Singleton****, Builder, Prototype*
- Structure
  - Organisation des classes de manière à produire des applications facilement évolutives
  - ***Decorator, Facade, Bridge****, Adapter, Composite, Flyweight, Proxy*
- Comportement
  - Interaction entre objets tout en les maintenant le plus indépendants possible
  - ***State, Command, Strategy, Visitor,*** *Iterator, Observer, Chain of Responsability, Interpreter, Mediator, Memento, Template Method*

Design pattern de création

« Les fabriques »

Pattern de création

- Objectif
  - Indépendance de la manière dont les objets sont créés et assemblés
  - Instancier des objets sans connaissance de leur classe « concrète/instables »
- Dépendance d'instanciation
  - Les objets sont instanciés à partir de classes concrètes, souvent instables
  - Exemple : comment inscrire un nouvel étudiant ?

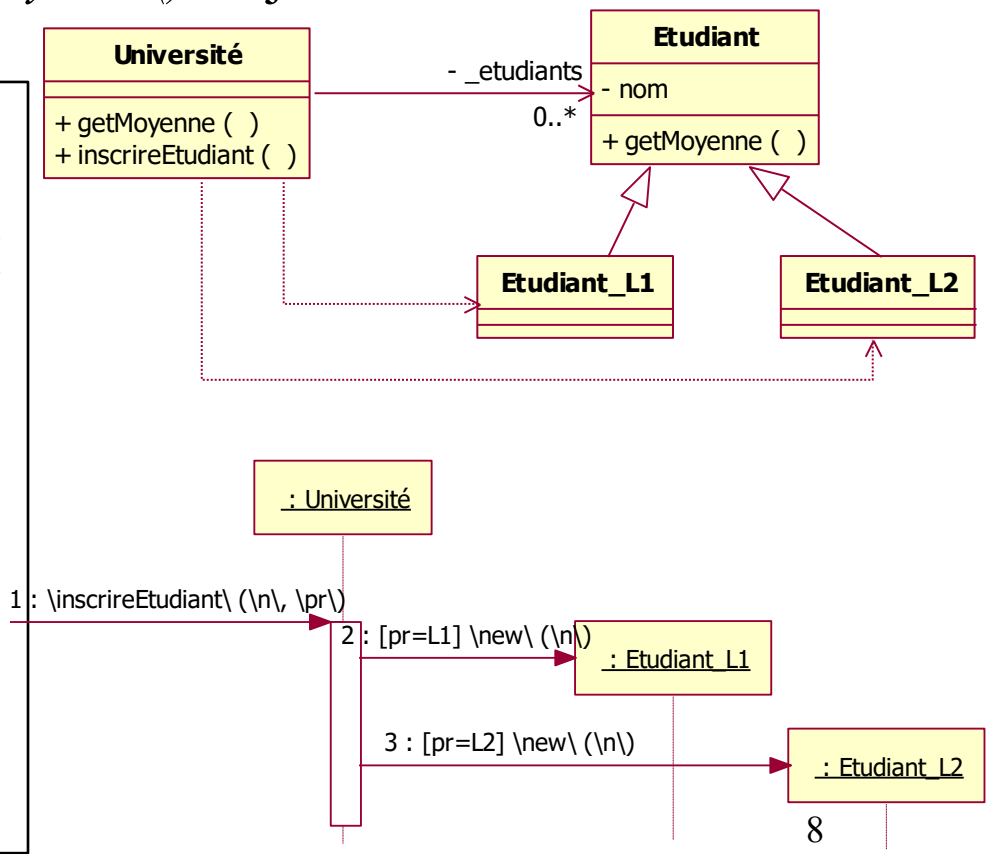


Pattern de création

☞ Problème : dépendance de classes instables à l'instanciation !

- La création d'objets rélève de la responsabilité de la classe qui stocke les objets (*Université*)
- **new** est non délégable comme *getMoyenne()*, objet *Etudiant* inexistant encore !

```
public class Université {
    private List<Etudiant> _etudiants;
    public void inscrireEtudiant (String  nom, String annee) {
        Etudiant etu;
        if (annee = "LI") {
            etu = new Etudiant_LI (nom);
        else { if (annee = "L2")
            etu = new Etudiant_L2 (nom);
         }
         _etudiants.add(etu);
    }
}
```



Pattern de création

Factory

☞ Principe et solution

- Objectif : Créer des objets d'une même famille sans se soucier des classes concrètes/instables utilisées
- Solution : déléguer la création des objets de classes isntables à des classes « *Factory* » spécialisées

```
public class Université {
   private List<Etudiant> _etudiants;
   public EtudiantFactory factory = new EtudiantFactory();
   public void inscrireEtudiant (String  nom, String annee) {
      Etudiant  etu = factory.creerEtudiant (nom, annee);
     _etudiants.add(etu);
   }
}
```

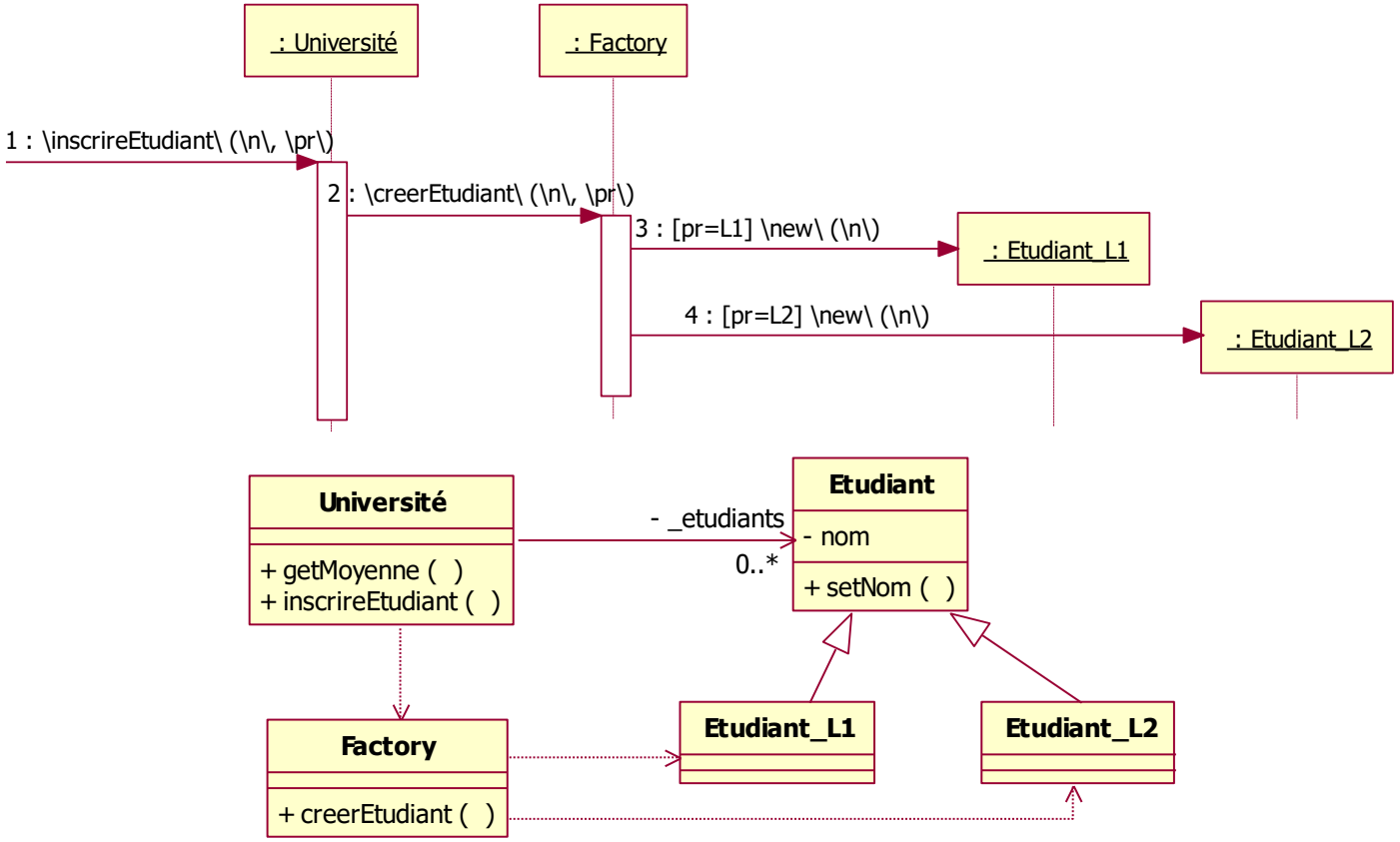
```
public class EtudiantFactory {
     Etudiant creerEtudiant (Str nom , Str annee) {
          Etudiant  etu;
          if (annee= "L1")
                    etu = new Etudiant_L1 (nom);
          else if (annee= "L2")
                     etu = new Etudiant_L2 (nom);
          else throw new IllegalArgumentException(annee);
          return etu;
     }
}
```

Pattern de création

Factory

☞ Principe et solution : le pattern « Factory » (solution intermédiaire)

– Les évolutions seront amorties et **localisées** dans la classe « Factory »



– Impacts en cas d'ajout d'une nouvelle sous-classe de « Etudiant » ? OCP !

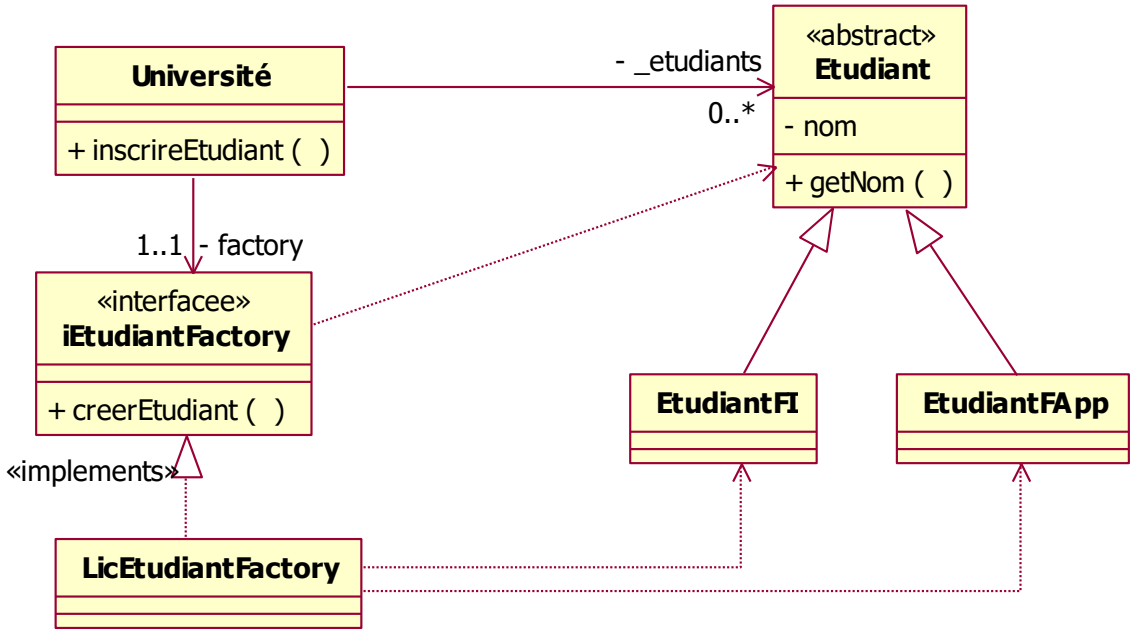
– Classe « Factory » **instable** !

Pattern de création

Factory

☞ Pour ce pas dépendre de l'implémentation de la *factory*

- (1) La généraliser par son interface (*IEtudiantFactory*)
- (2) Injecter ses objets via un constructeur ou une méthode du client (*Université*)



Pattern de création

Factory

☞ Pour ce pas dépendre de l'implémentation de la *factory*

– (1) La généraliser par son interface (*IEtudiantFactory*)

```
public class Université {
  private iEtudiantFactory factory = new LicEtudiantFactory
();

   void inscrireEtudiant (String  nom, String annee) {
      Etudiant  etu = factory.creerEtudiant (nom, annee);
     _etudiants.add(etu);
   }
}
```

```
public interface iEtudiantFactory {
     Etudiant creerEtudiant (Str nom , Str promo) ;
}
public lass LicEtudiantFactory implements iEtudiantFactory {
 public Etudiant creerEtudiant (Str nom , Str annee) {
       Etudiant  etu = null;
       if (annee= "L1")  etu = new Etudiant_L1 (nom);
       elseif (annee= "L2")  etu = new Etudiant_L2 (nom);
       return etu;
 }}
```

– Problème d'instanciation de la « Factory », à partir de la classe instable *LicEtudiantFactory* !

– Solution : injection de dépendance

Pattern de création

Factory : injection de dépendance

☞ Pour ce pas dépendre de l'implémentation de la *factory*

– (2) Injecter ses objets via un constructeur ou une méthode du client (*Université*)

```
public class Université {
   private iEtudiantFactory factory;
   public Université (iEtudiantFactory factInjectee) {
     this.factory = factInjectee;
   }
   public void useFactory (iEtudiantFactory factInjectee) {
     this.factory = factInjectee;
   }
   void inscrireEtudiant (String  nom, String annee) {
      Etudiant  etu = factory.creerEtudiant (nom, annee);
     _etudiants.add(etu);
   }
}
```

```
public interface iEtudiantFactory {
     Etudiant creerEtudiant (Str nom , Str promo) ;
}
public class LicEtudiantFactory implements iEtudiantFactory {
  public Etudiant creerEtudiant (Str nom , Str annee) {
       Etudiant  etu = null;
       if (annee= "L1")  etu = new Etudiant_L1 (nom);
       elseif (annee= "L2")  etu = new Etudiant_L2 (nom);
       return etu;
 }}
```

– Evolution sur *Etudiant* : coder la nouvelle sous-classe *Etudiant* et la « Factory » sous-jacente (+). *Université* n'est pas impactée (++)

Pattern de création

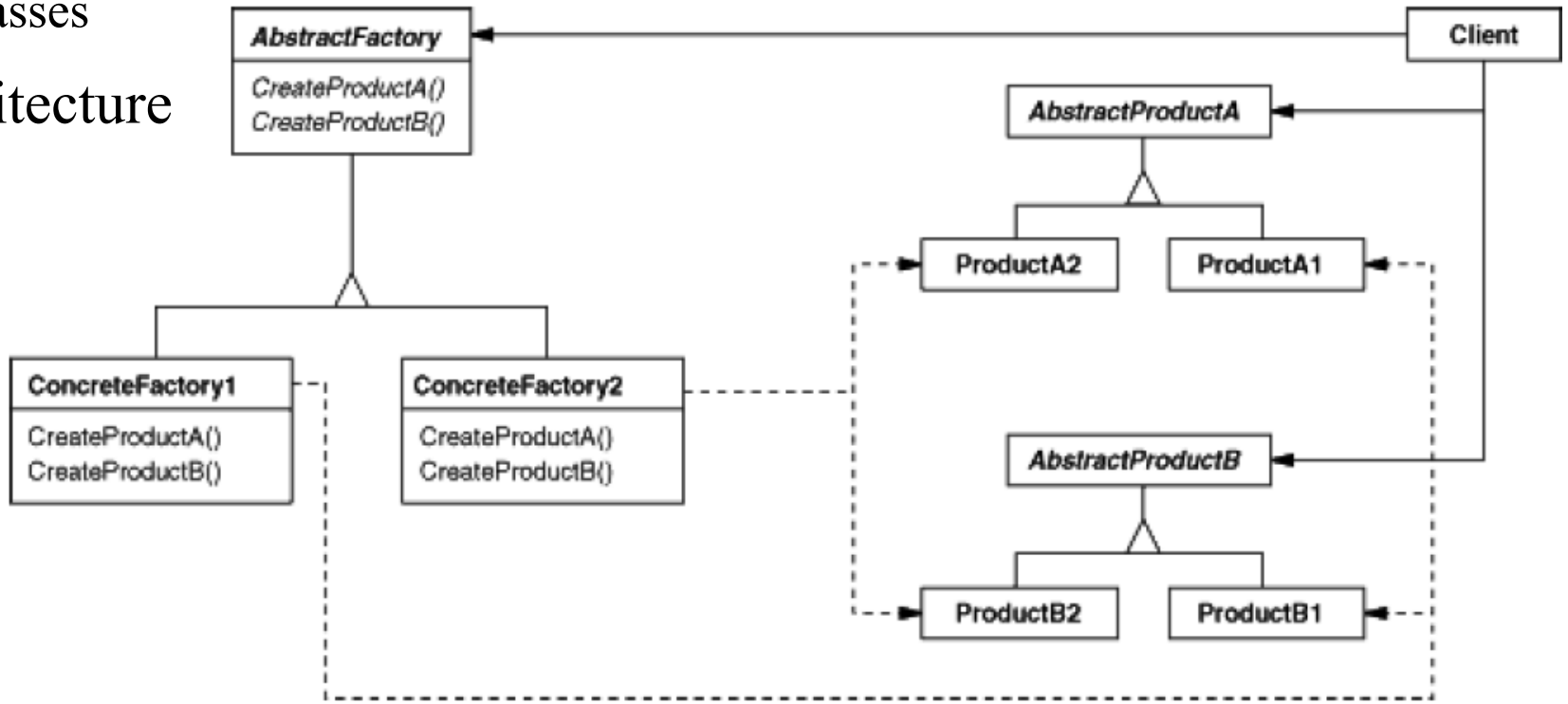
☞ Pattern de création (définitions tirées du livre « GoF »)

- Abstract Factory : « *provides an interface for creating families of related or dependent objects without specifying their concrete classes* »
- Factory Method : « *defines an interface for creating an object, but let subclasses decide which class to instantiate* »
- Builder : « *separates the construction of a complex object from its representation so that the same construction process can create different representations* »
- Prototype : « *specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype* »
- Singleton : « *ensures a class only has one instance, and provide a global point of access to it* »

Pattern de création

Abstract Factory

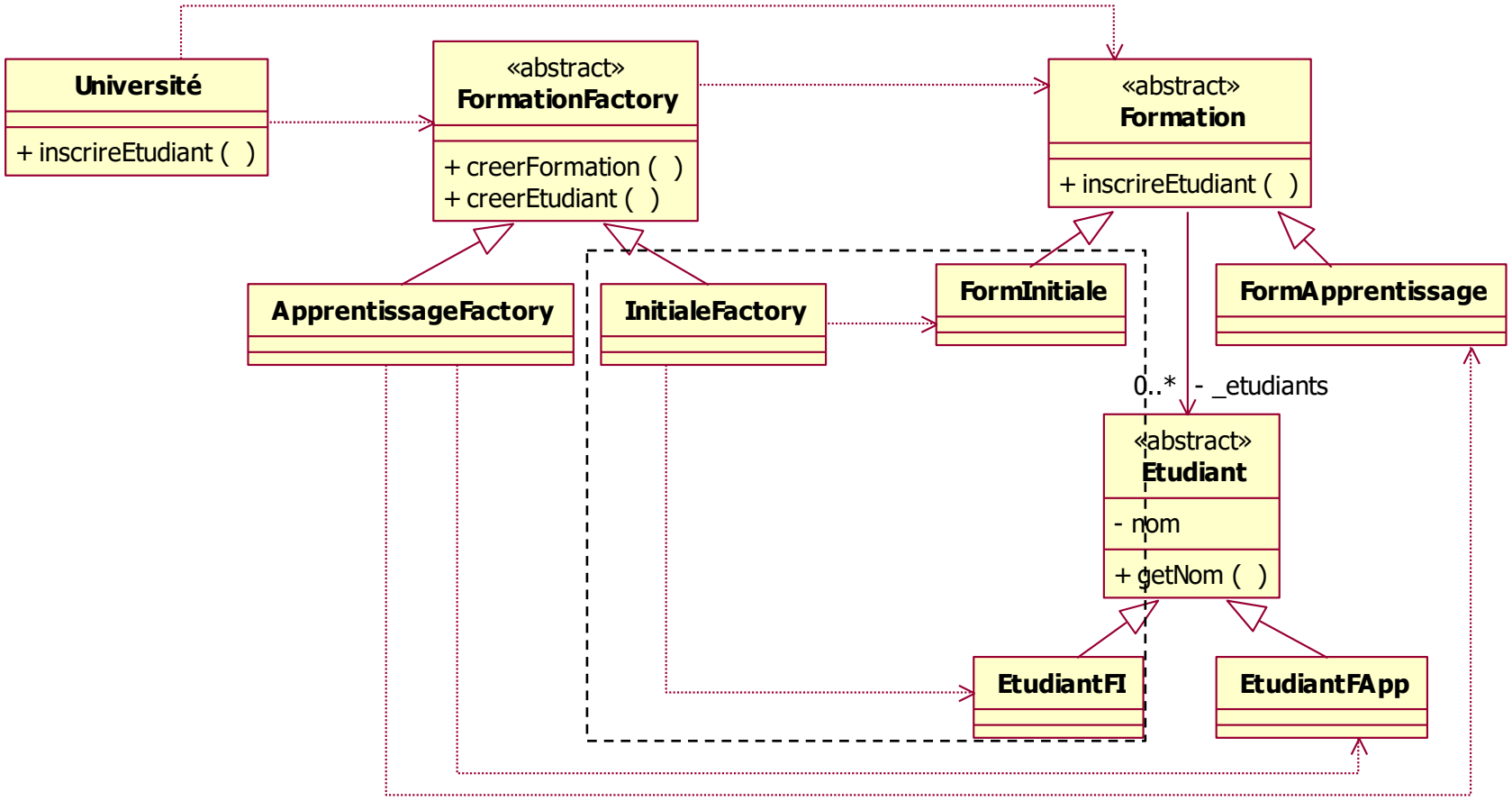
- Principe
  - Une interface (classe abstraite) pour créer des familles de classes assorties (compatibles)
  - Contraintes d'assortiment respectées à la création des objets
  - Des classes « Factory » dédiées à des classes assorties des deux familles de classes
- Architecture



Pattern de création

Abstract Factory

☞ Exemple



Pattern de création

Abstract Factory

☞ Exemple (solution intermédiaire)

```
public class Université {
    private FormationFactory factory;
    public Formation nouvelleFormation (Str nForm, Str  type) {
       if (type = "Initiale") factory = new InitialeFactory ();
       else factory = new ApprentissageFactory ();
       Formation  form = factory.creerFormation (nForm) ;
       return form ;
    }
}
```

```
class FormInitiale {
    Formation inscrireEtudiant (Str nom) {
        FormationFactory factory = new InitialeFactory ();
        Etudiant etu =  factory.creerEtudiant (nom) ;
        _etudiants.add (etu);
       return etu ;
    }
}
```

```
public abstract class FormationFactory {
     Formation creerFormation (String  nom) ;
     Etudiant creerEtudiant (String  nom);
}
public class InitialeFactory  extends FormationFactory {
    public Formation creerFormation (String nom) {
        return new FormInitiale (nom);
     }
     public Etudiant creerEtudiant (String nom) {
         return new EtudiantFI (nom);
}}
public slass ApprentissageFactory extends FormationFactory  {
    public Formation creerFormation (String nom) {
        return new FormApprentissage (nom);
     }
     public Etudiant creerEtudiant (String nom) {
         return new EtudiantFApp (nom);
}}
```

☞ Pb : dépendance de Factory concrètes !

Pattern de création

Abstract Factory avec injection de dépendance

```
public class Université {
    private static Map<String, FormationFactory> factories;
    public Université (Map<String, FormationFactory> factories) {
        this.factories = factories;
    }
    public Formation nouvelleFormation (String nForm, String  type) {
        FormationFactory factory = factories.get (type);
        Formation  form = factory.creerFormation (nForm) ;
        return form ;
    }
    public void addFactory (String type, FormationFactory factory) {
        factories.put (type, factory);
    }
}
```

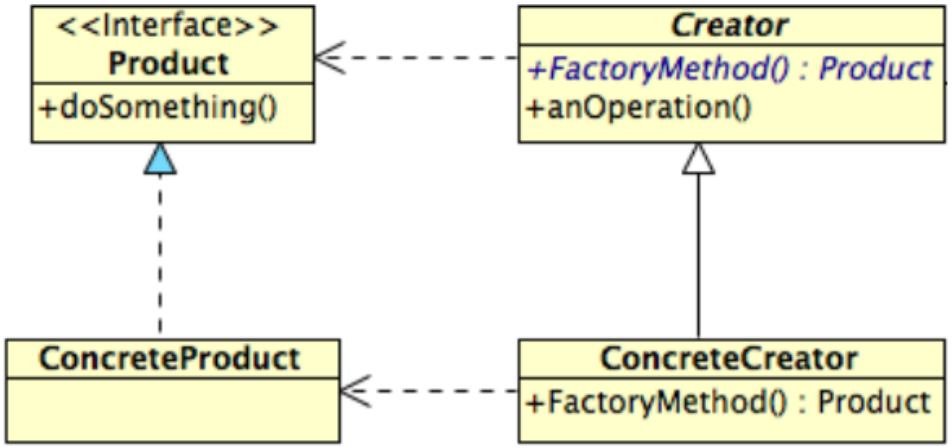
```
public class Appli {
    public Appli () {
        Map<String, FormationFactory> factories = new TreeMap<>();
        factories.put ("Initiale", new InitialeFactory ()) ;
        factories.put ("Apprentissage", new ApprentissageFactory ()) ;
        Université u = new Université (factories);
        start ();
}}
```

```
public interface FormationFactory {
    Formation creerFormation (String  nom) ;
    Etudiant creerEtudiant (String  nom);
}
public class InitialeFactory implements FormationFactory {
    public Formation creerFormation (String nom) {
        return new FormInitiale (nom);
    }
    public Etudiant creerEtudiant (String nom) {
        return new EtudiantFI (nom);
    }
}
public class ApprentissageFactory implements FormationFactory  {
    public Formation creerFormation (String nom) {
        return new FormApprentissage (nom);
    }
    public Etudiant creerEtudiant (String nom) {
        return new EtudiantFApp (nom);
    }
}
```

Pattern de création

Factory Method

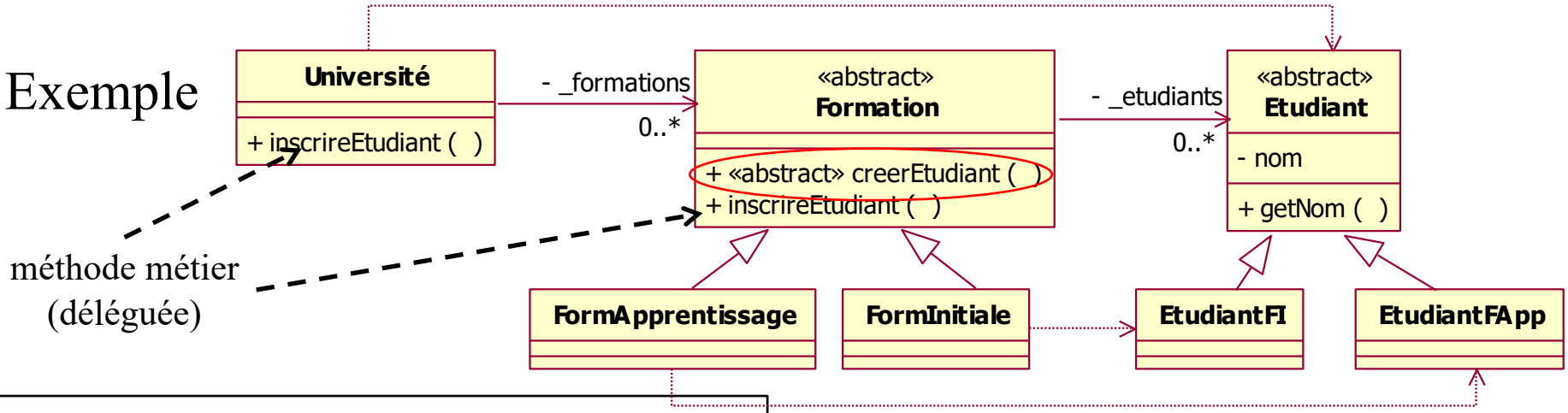
- Principe
  - Instanciation d'objets réalisée par une méthode polymorphe (et non par une classe), non implémentée !
  - La méthode « Factory » est implémentée (création d'objets) dans les classes concrètes d'assortiment
- Architecture



Pattern de création

Factory Method

☞ Exemple




```
public class Université {
    public Etudiant inscrireEtudiant (Formation form, Str  nom){
        Etudiant  etu = form.inscrireEtudiant (nom) ;
        return etu ;
}}
public abstract class Formation {
   private List <Etudiant> _etudiants ;
   public Etudiant inscrireEtudiant (String  nom){
     Etudiant  etu = this.creerEtudiant (nom);
     this._etudiants.add(etud) ;
      return etu;
   }
 public abstract Etudiant creerEtudiant (String  nom);
}
```

```
public class FormInitiale extends Formation {
    public Etudiant creerEtudiant (String nom) {
        return new EtudiantFI (nom);
     }
}
public class FormApprentissage extends Formation {
    public Etudiant creerEtudiant (String nom) {
        return new EtudiantFApp (nom);
     }
}
```

Pattern de création

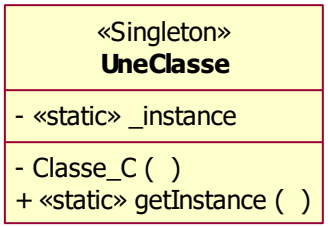
Quelle « Fabrique » utiliser ?

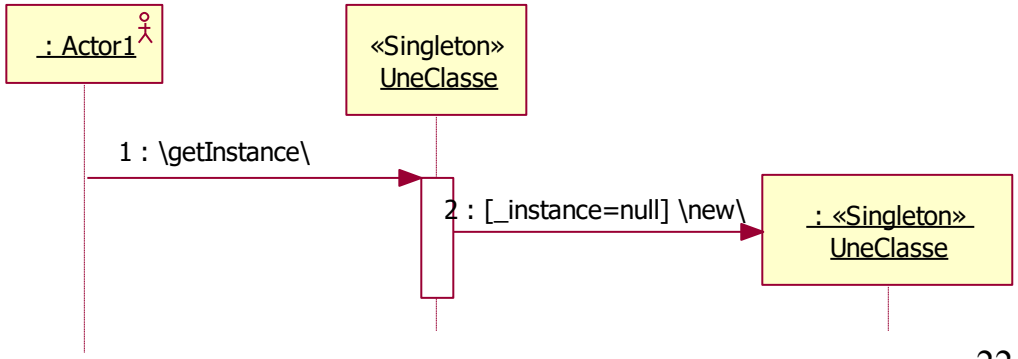
- Pattern « Factory »
  - Lorsque la création concerne une seule famille d'objets
  - La création est déléguée à une classe « Factory »
- Pattern « Factory Method »
  - Lorsque la création concerne une seule famille d'objets
  - Les objets créés doivent être assortis à une autre famille d'objets
  - La création est déléguée aux sous-classes (héritage)
- Pattern « Abstract Factory »
  - Lorsque la création concerne des familles de classes liées (assorties)
  - La création est déléguée à une « Factory »
  - La « Factory » peut être elle même créée par une autre « fabrique » (fabrique de fabrique)

Pattern de création

Singleton

- Principe
  - La classe n'admet qu'une seule instance
  - L'instanciation de la classe est contrôlée par une méthode statique de la classe
- Architecture
  - Constructeurs privés
  - Un attribut statique référençant l'instance unique de la classe
  - Une méthode permettant de retourner l'instance unique de la classe





Pattern de création

Singleton

☞ Implémentation du « Singleton » : instanciation *tardive* (au premier appel de *getInstance()* !)

```
class final class Université {
    private static Université _instance = null;
    private Université() {}
    public static synchronized Université getInstance () {
        if (_instance == null)
                _instance = new Université ();
        return _instance;
    }

    public void inscrire (Etudiant e) {
        ….
    }
}
```

```
public class Appli {
    public static void main () {
      Université uniqueInstanceUniv = Université.getInstance ();
       uniqueInstanceUniv.inscrire (…);
     }
}
```

Pattern de création

Singleton

- Instanciation (*immédiate*) au chargement de la classe
  - L'instance du singleton est créée au chargement (*ClassLoader*) des classes *Appli* puis *Université*

```
class final class Université {
    private static Université _instance = new Université ();
    private Université() {}
    public static synchronized Université getInstance () {
            return _instance;
    }

    public void inscrire (Etudiant e) {
        ….
    }
}
```

```
public class Appli {
    private Université uniqueInstanceUniv;
    public static void main () {
        uniqueInstanceUniv = Université.getInstance
();
        uniqueInstanceUniv.inscrire (…);
    }
}
```

- Chargement coûteux !

Pattern de création

Singleton

☞ Instanciation *tardive* (à la demande) avec une classe interne !

```
public final class Université {
    private static class SingletonLoader {
        private static Université _instance = new Université ();
    }

    public static Singleton getInstance() {
        return SingletonLoader.instance;
    }
    private Université () {}
   …
}
```

☞ Chargement coûteux !

Design pattern de structure

Patterns de structure

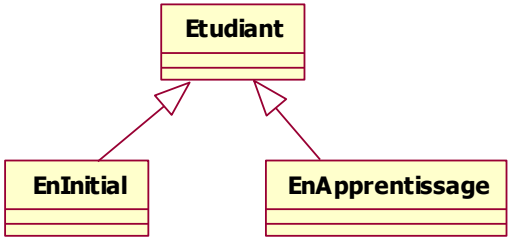
- Objectif
  - Organisation (de la structure) des classes de manière à pouvoir les faire évoluer à moindre effort
  - ***Decorator***, ***Facade***, ***Bridge***, *Adapter, Composite, Flyweight, Proxy*
- Principe
  - Utilisation intensive des principes de conception vus en première partie
  - *Dépendances, délégation, SRP, OCP*
  - Et le principe de « *préférer la composition à l'héritage* »

Composition *vs.* Héritage

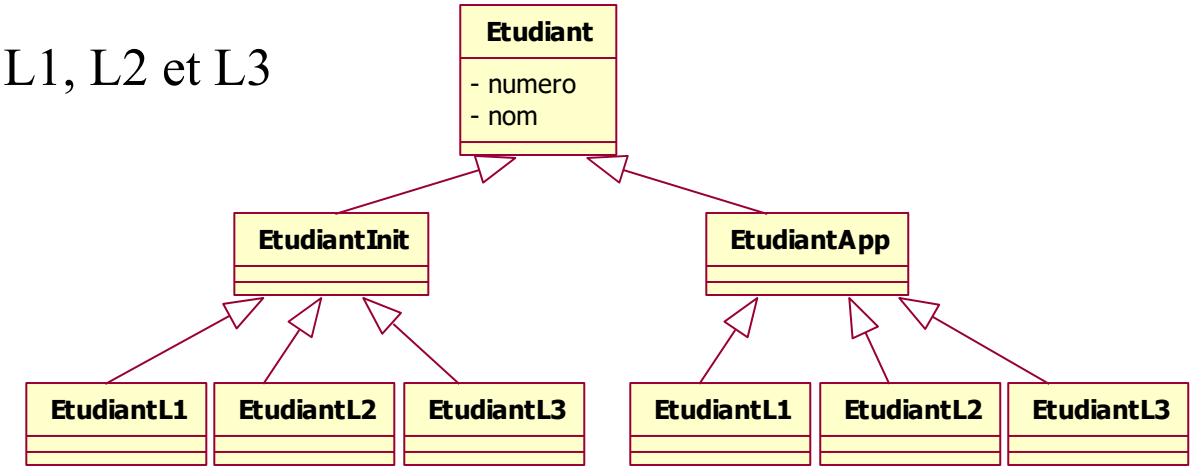
Organisation de classes

☞ Gestion des étudiants : solution avec héritage

– Deux filières : initial et apprentissage



– et trois années d'études : L1, L2 et L3



Composition *vs.* Héritage

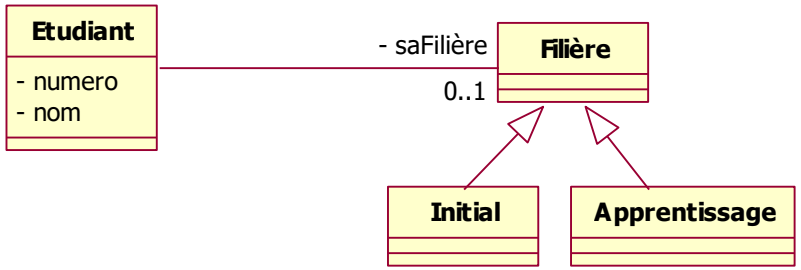
Organisation de classes

- ☞ Explosion combinatoire du nombre de classes
  - Les évolutions risquent d'engendrer un nombre excessif de classes
  - Exemple : faire l'évolution suivante :
    - Gérer différemment les étudiants boursiers et non boursiers
    - Nombre de classes ajoutées ?
- ☞ Redondance de code
  - Exemple : calcul de moyenne !
- ☞ Changement d'état lourd à gérer
  - Exemple : passage de L1 à L2
  - Nécessite des suppressions, recréations d'objets et …

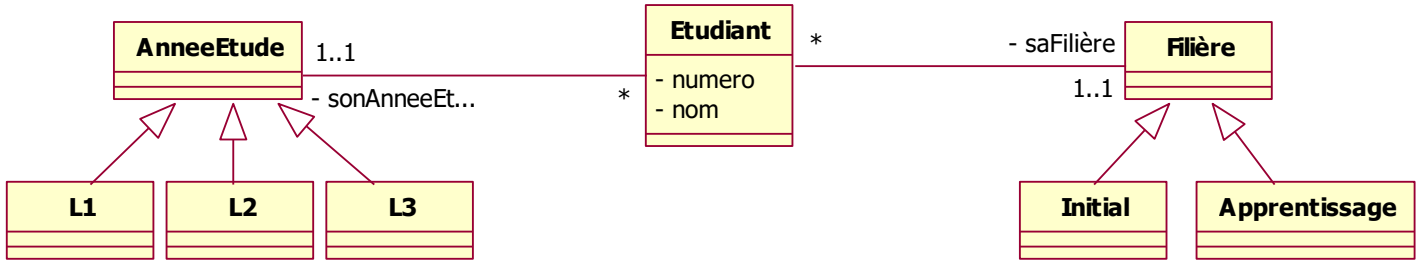
Composition *vs.* Héritage

Organisation de classes

- Gestion des étudiants : solution avec la composition
  - Deux filières : initial et apprentissage



  - Et trois années d'études : L1, L2 et L3



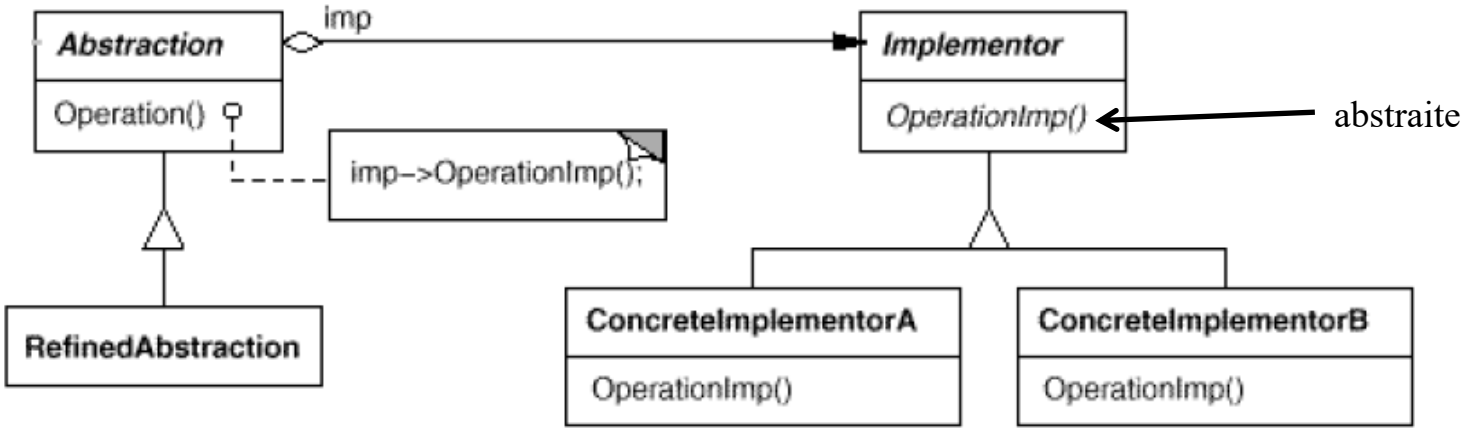
- Implémentations par délégation (de *Etudiant* à *AnnneeEtude* et *Filiere*)
- Inconvénient : nécessite un plus grand nombre d'objets

Pattern « Bridge »

Principe

- Objectif
  - Découpler l'interface d'une classe de son implémentation
  - Se préparer à changer l'implémentation des opérations d'une classes
- Principe
  - Déléguer l'implémentation des opérations d'une classe à une autre classe (concrète)
  - Utilise le principe de « *préférer la composition à l'héritage* »
- Architecture de classes

La classe traitée doit être stable ou relativement stable

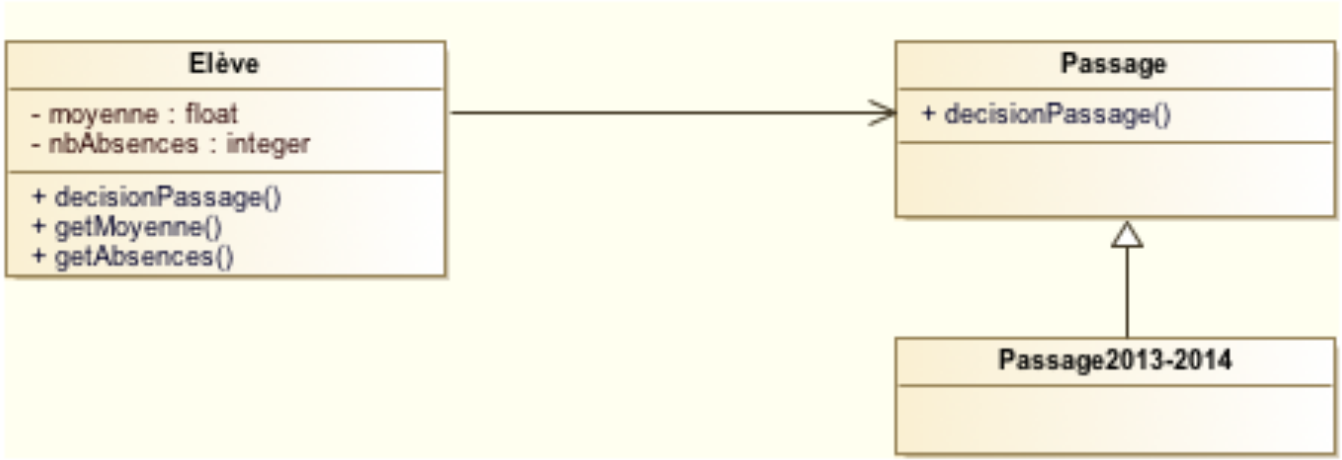


Pattern « Bridge »

Exemple

☞ Gestion des étudiants (suite)

– La décision de passage est déclarée dans *Elève* et implémentée dans une sous-classe de *Passage*




```
public class Elève {
   public boolean decisionPassage (Passage regle) {
       return regle.decisionPassage (this);
}}
```

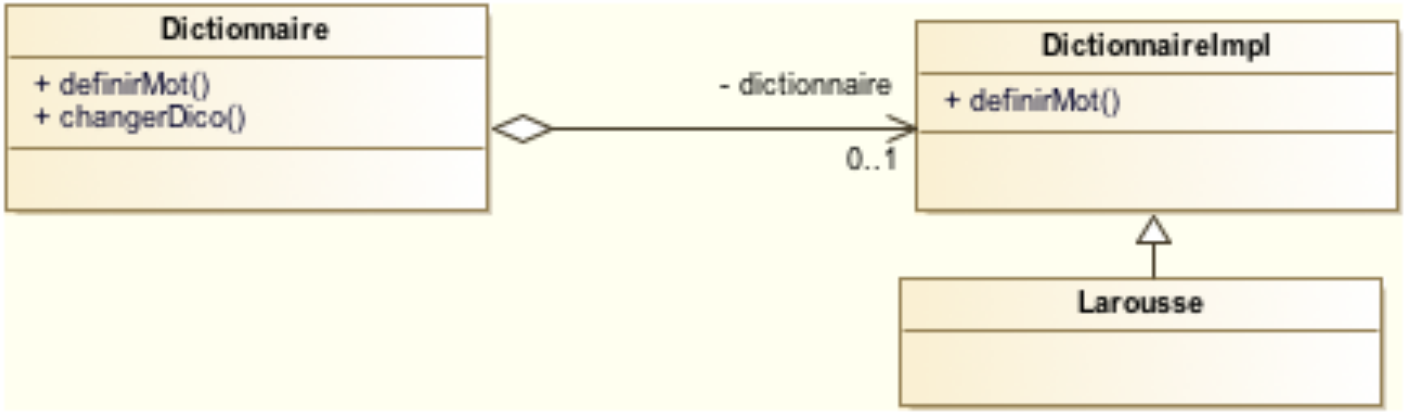
```
public class Passage2013-2014 implements Passage {
    public boolean decisionPassage (Eleve e) {
        if (e.getMoyenne() >= 10) return true;
        else return false;
}}
```

Pattern « Bridge »

Exemple

☞ Dictionnaire

– Définition d'un mot dans le Larousse




```
class Dictionnaire{
    private DictionnaireImp implementation;
    public String definirMot (String  mot) {
       return implementation.definirMot (mot);
}}
```

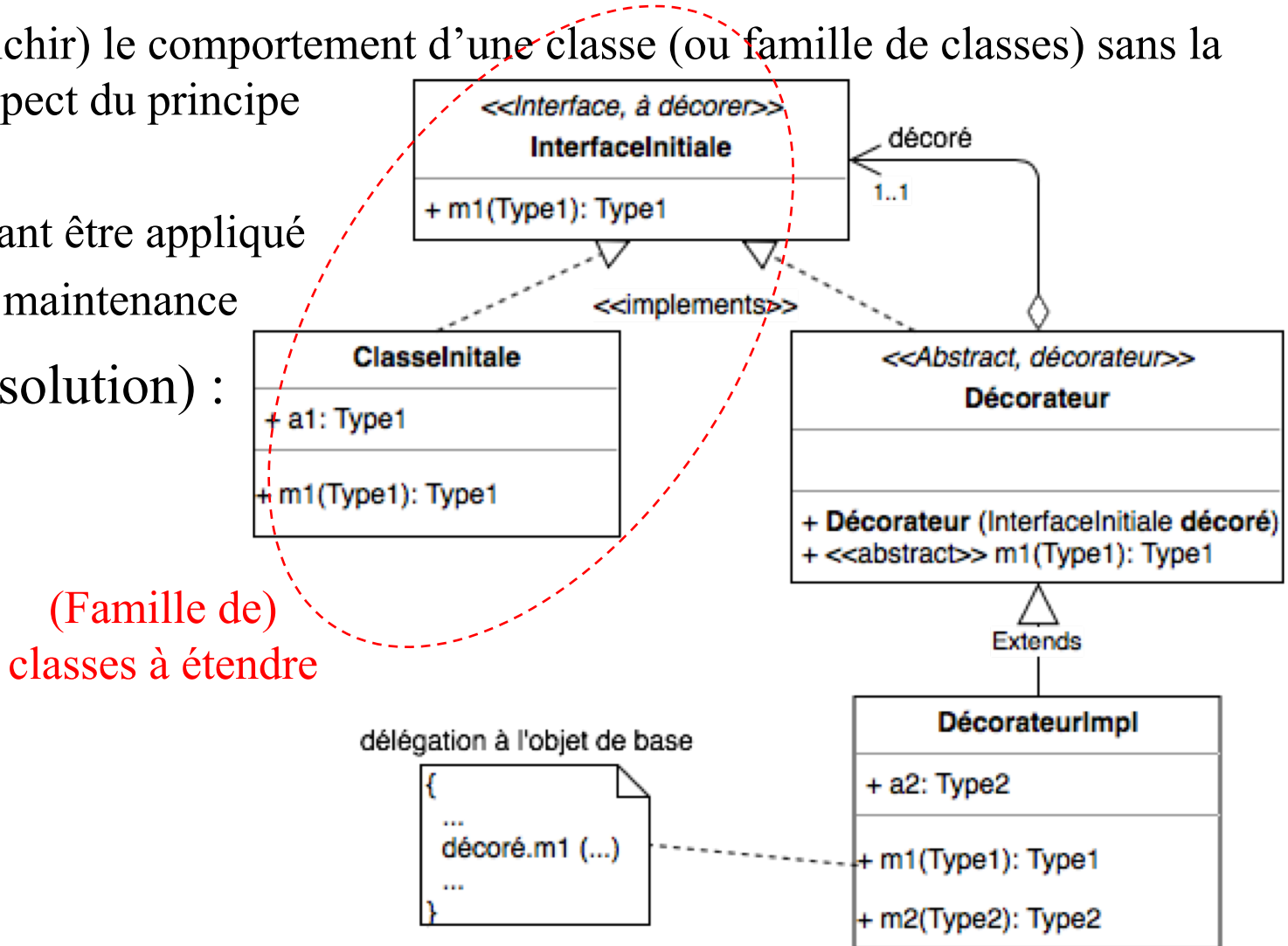
```
class Larousse extends DictionnaireImpl {
    public String definirMot (String  mot) {
        if (mot="Personne") return "n.f. être humain";
        …
}}
```

– OCP : on peut facilement changer de dictionnaire pour un nouveau (LeRobert)

Pattern « Décorateur »

Principe

- Objectif
  - Etendre (enrichir) le comportement d'une classe (ou famille de classes) sans la modifier (respect du principe de l'OCP)
  - Pattern pouvant être appliqué pendant une maintenance
- Architecture (solution) :

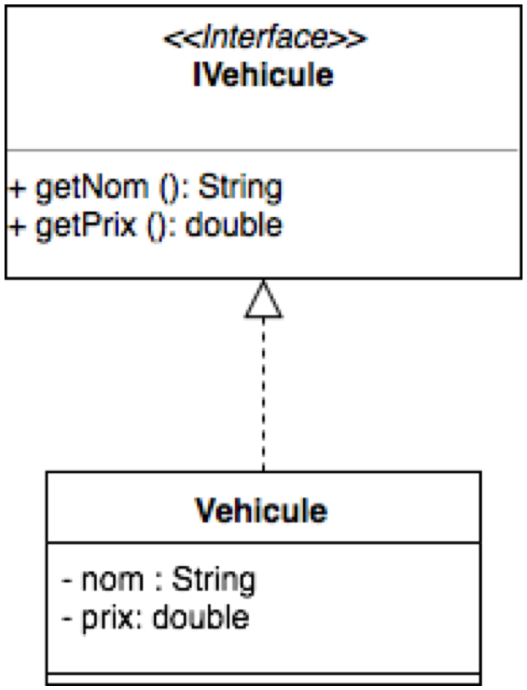


Pattern « Décorateur »

Exemple

☞ Exemple : ajouter/décorer les véhicules avec de nouveaux équipements

- Equipements (décorations) :
  GPS, Caméra de recul, etc.
- Chaque équipement à un surcoût, devant additionné au prix du véhicule



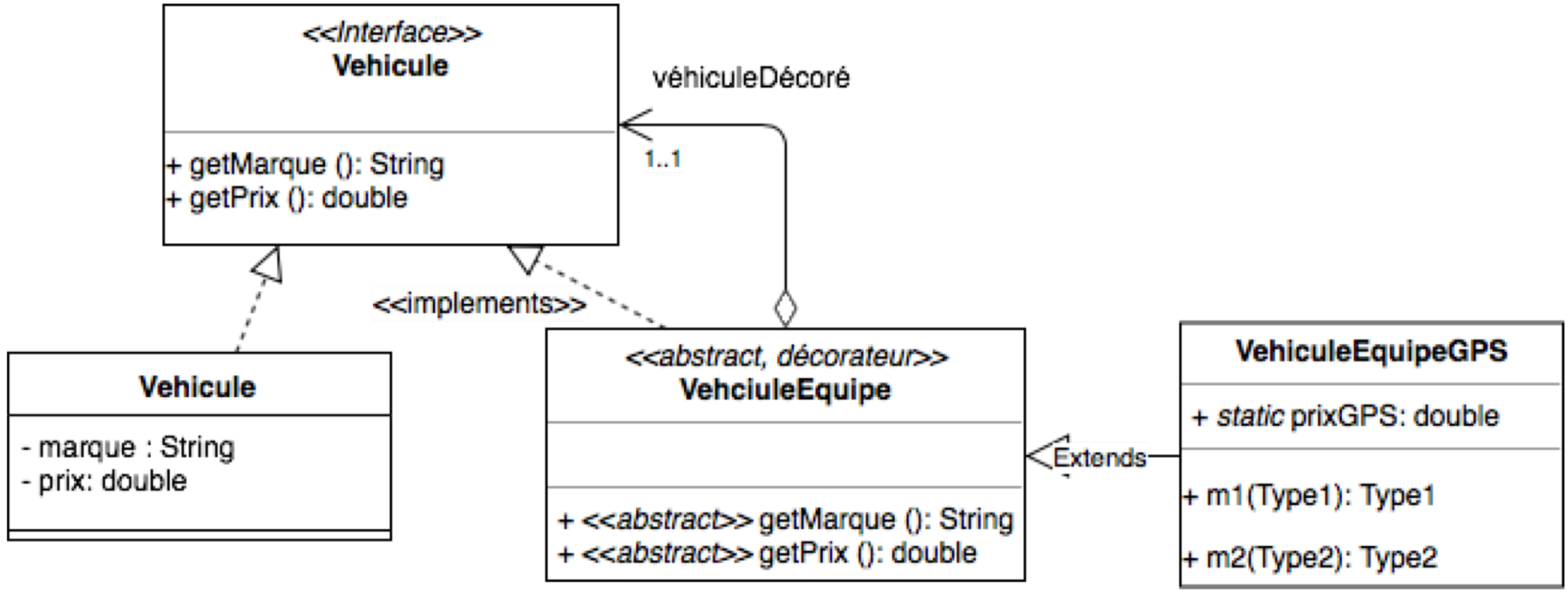
☞ Solutions possibles (respectant l'OCP !)

- Héritage : sous-classer les classes de la hiérarchie. Pb : nombre de classes ajoutées
- Composition : une classe suffit pour étendre la hiérarchie, délégation des fonctions existantes. Limite : impossibilité de décorer le décorateur
- **Composition combinée à l'héritage (++) : c'est le pattern « décorateur » !**

Pattern « Décorateur »

Solution

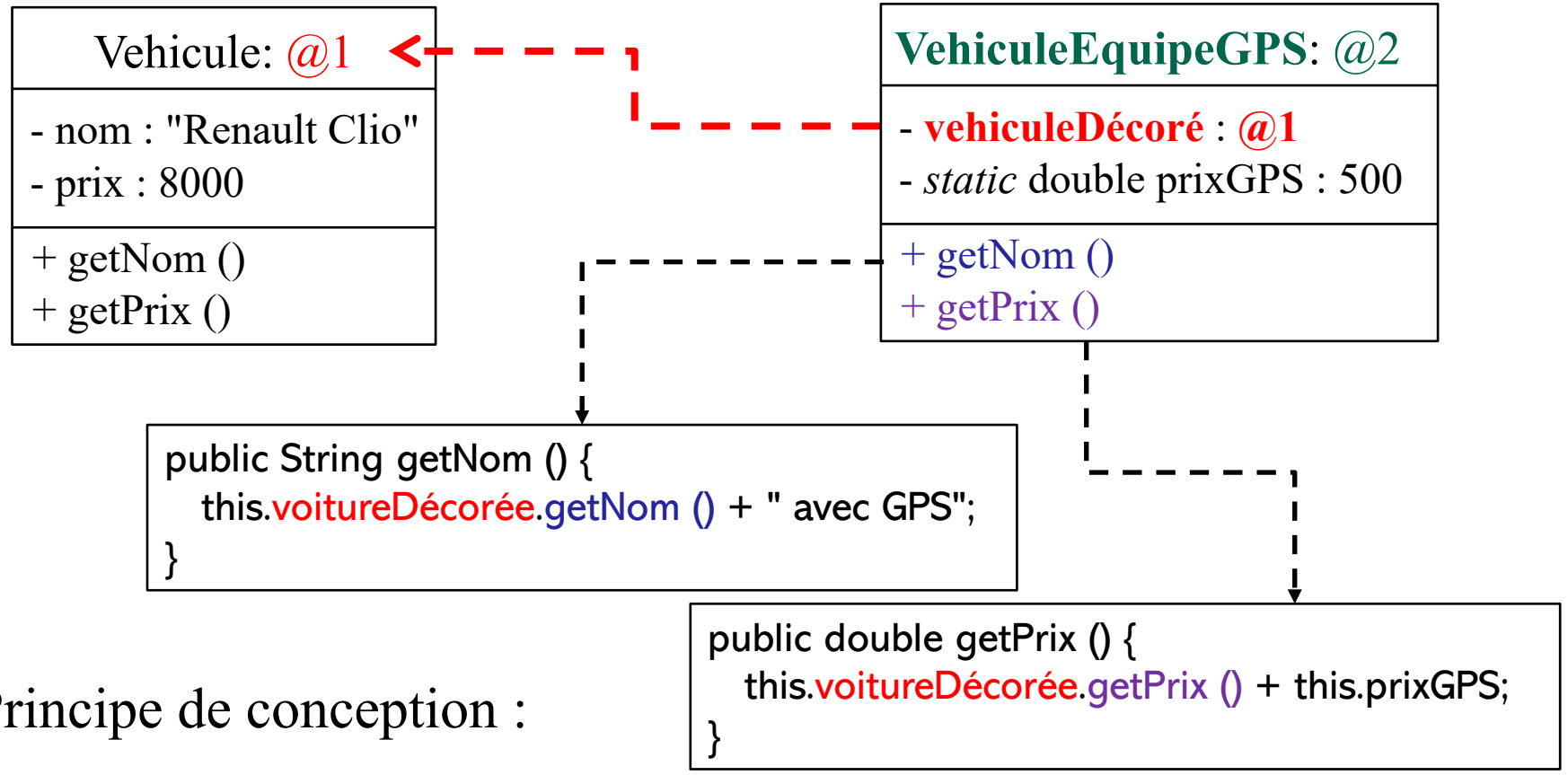
- Principe du « décorateur »
  - Composition/association : implémentation des comportements existants par délégation à l'objet décoré (ici *Etudiant*)
  - Héritage : pouvoir enrichir/décorer le décorateur à son tour
  - Classe abstraite : décorateur supposé instable !



Pattern « Décorateur »

Solution

☞ Décorateur : un objet qui encapsule l'objet qu'il décore...



☞ Principe de conception :

- ☞ **Composition** : encapsuler l'objet décoré dans son objet décorateur
- ☞ **Délégation** : le décorateur délègue les traitements existants à l'objet qu'il décore

Pattern « Décorateur »

Utilisation

☞ Décorateur : un objet qui encapsule l'objet décoré

```
// Objet décoré : véhicule de base (sans équipements)
IVehicule clio75 = new Vehicule ("Renault Clio", 8000);
// Décorer le véhicule clio75
IVehicule clio75Gps = new VehiculeEquipeGPS (clio75);
assertEquals ("Renault Clio avec GPS", clio75Gps.getNom ()) ;
assertEquals (8000+500, clio75Gps.getPrix ()) ;
```

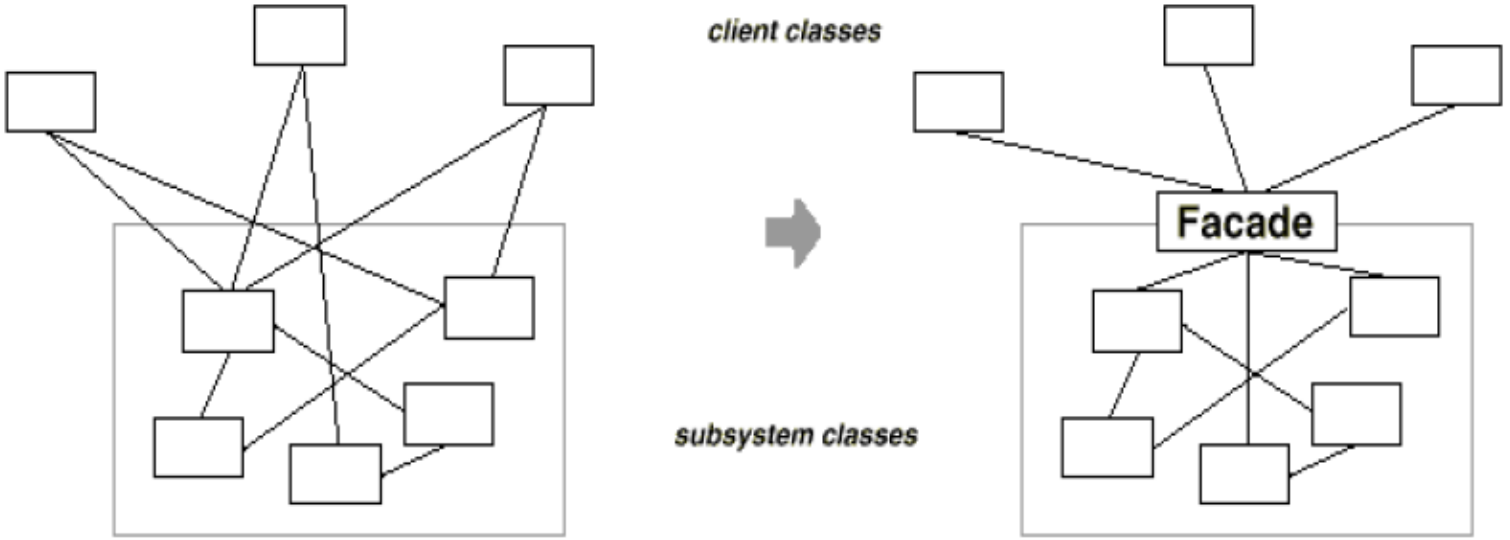
☞ Décorer de manière récursive le décorateur

```
// Décorer le décorateur : le décorateur est un IVehicule !
IVehicule clio75DoubleGps = new VehiculeEquipeGPS (clio75Gps);
assertEquals (8000+500+500, clio75DoubleGps.getPrix ()) ;
```

Pattern « Façade »

Principe

- Objectif
  - Découpler un module complexe (sous-système, ensembles de classes, paquage) de ses classes clientes
- Principe
  - Produire une interface unique (façade) qui cachera la complexité du module
  - La façade amortira l'impact des changements des classes du module
- Architecture de classes



Pattern « Façade »

Exemple sans le pattern

☞ Inscription d'étudiants sans le pattern *Facade*

```
class class InterfaceUtilisateur {
   private Universite univ;
   public void inscrireInit (String nomE, String nomDpt) {
      Departement d = univ.searchDept (nomDept);
      EtudiantInit e = new EtudiantInit (nomE);
      d.inscrire (e);
   }
   public void inscrireApp (String nomE, String nomDpt) {
      Departement d = univ.searchDept (nomDept);
      EtudiantApp e = new EtudiantApp (nomE);
      d.inscrire (e);
   }
}
```

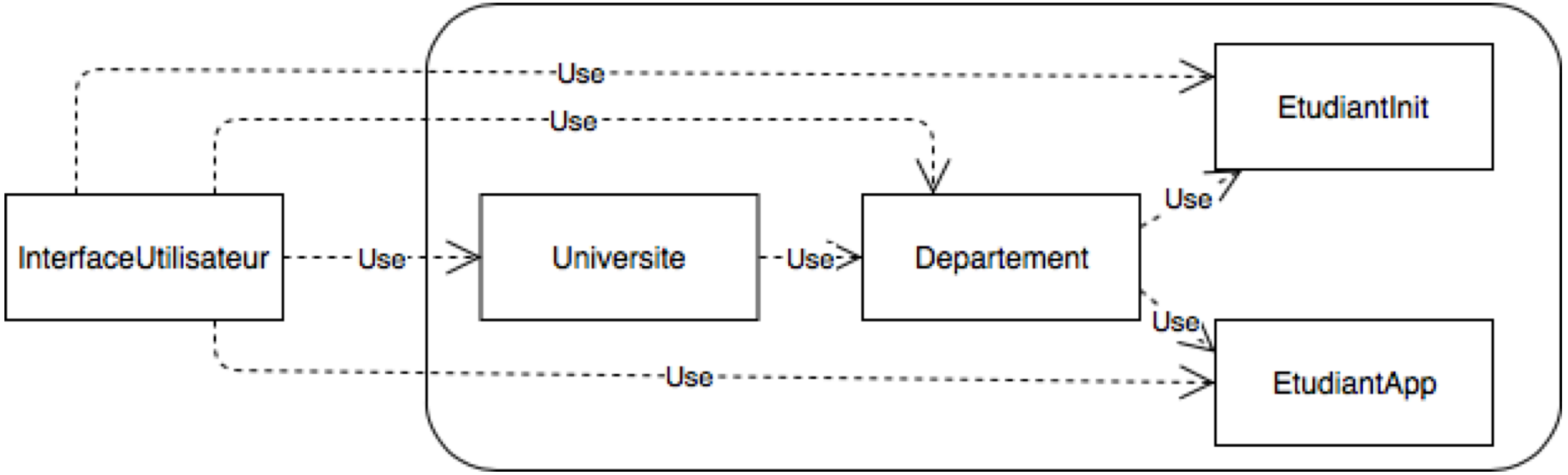
```
class class Université {
   private Map<String, Departement> depts;
   public Departement searchDept (String nomDpt) {
      return depsts.get (nomDpt);
}}
```

```
public class Departement {
   private List<EtudiantIni> etuInit;
   private List<EtudiantIni> etuApp;
   public void inscrire (EtudiantIni e) {
     etuInit.add (e);
   }
  public void inscrire (EtudiantApp e) {
     etuApp.add (e);
}}
```

Pattern « Façade »

Architecture sans le pattern

- Problèmes (de maintenabilité)
  - Dépendance de l'interface utilisateur (classe externe) de classes internes
  - Problème de non délégation de l'interface utilisateur aux classes de l'application



Pattern « Façade »

Exemple avec le pattern

☞ La Façade permettra l'ensemble des services/fonctions aux clients

- Elle doit donc déléguer pour limiter ses responsabilités !
- Le client (interafec u.) est allégé !

```
class class InterfaceUtilisateur {
   private Universite univ;
   public void inscrireEtud (String nomE,, String type,
                                          String nomDpt) {
      univ.inscrireEtud (nomE, String typeE, nomDept);
      //inscription déléguée à la façade (univ) !
  }
}
```

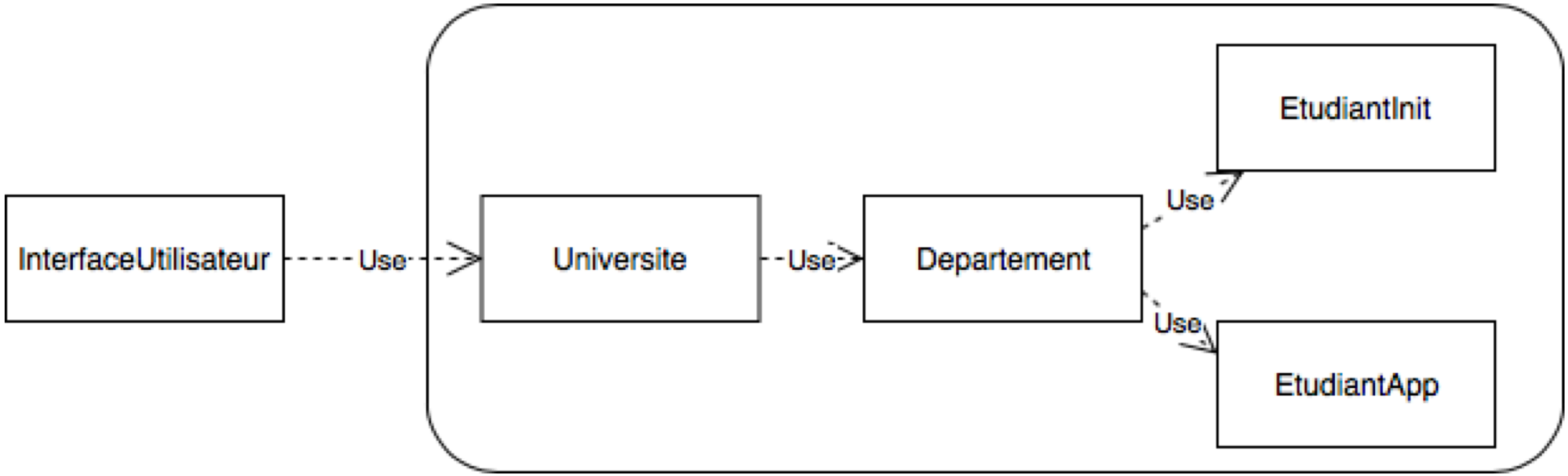
```
class class Université {
   private Map<String, Departement> depts;
  public void inscrireEtud (String nomE, String typeE,
                                          String nomDpt) {
      Departement dpt = univ.get (nomDept);
      dpt.inscrireEtud (nomE, String typeE);   //délégation
}}
```

```
public class Departement {
  private List<EtudiantIni> etuInit;
  private List<EtudiantIni> etuApp;
  public void inscrireEtud (String nomE,, String typeE) {
    if (typeE.equals("Initial"))
        etuInit.add (new EtudiantInit(nomE));
    else if (typeE.equals("Apprentissage"))
        etuApp.add (new EtudiantApp(nomE));
    }
}
```

Pattern « Façade »

Architecture avec le pattern

- ☞ Les clients (*interface utilisateur*) ne dépendent plus que de la *façade* !
- ☞ Architecture linéaire !



Design pattern de comportement

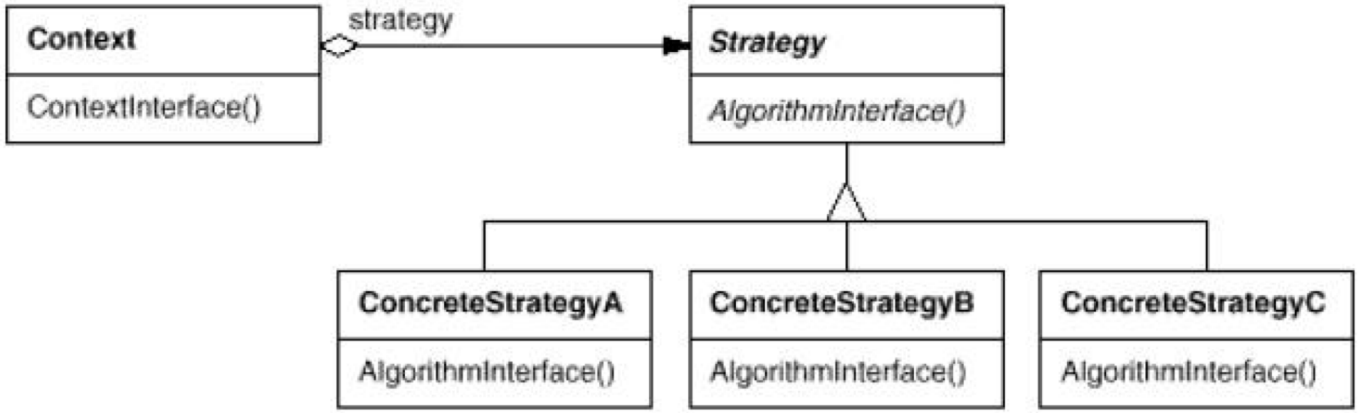
Patterns de comportement

- Objectif
  - Maintenabilité et flexibilité du comportement (algorithme, implémentation) des objets
  - ***Strategy, State, Visito****r, Command****,*** *Observer, Iterator, Chain of Responsability, Interpreter, Mediator, Memento, Template Method*
- Principe
  - Abstraction du comportement des classes
  - Implémenter le comportement dans des objets de manière à le changer facilement
  - Définir des objets de type comportement (et non structurel)

Pattern « Strategy »

Principe

- Objectif
  - Définir plusieurs implémentations interchangeables d'une même opération
  - Changer le comportement d'un objet pendant l'exécution
  - Définir une famille d'algorithmes pour une même opération et pouvoir en choisir un et le remplacer pendant l'exécution
- Principe
  - Encapsuler le comportement (implémentation des méthodes, algorithme) dans des objets dédiés
- Architecture de classes

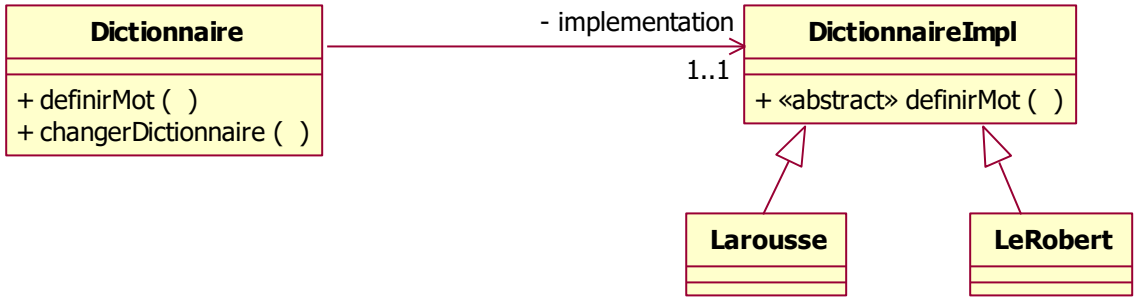


Pattern « Strategy »

Exemple

☞ Exemple : dictionnaire

– Définir les dictionnaires avec possibilité d'en changer pendant l'exécution



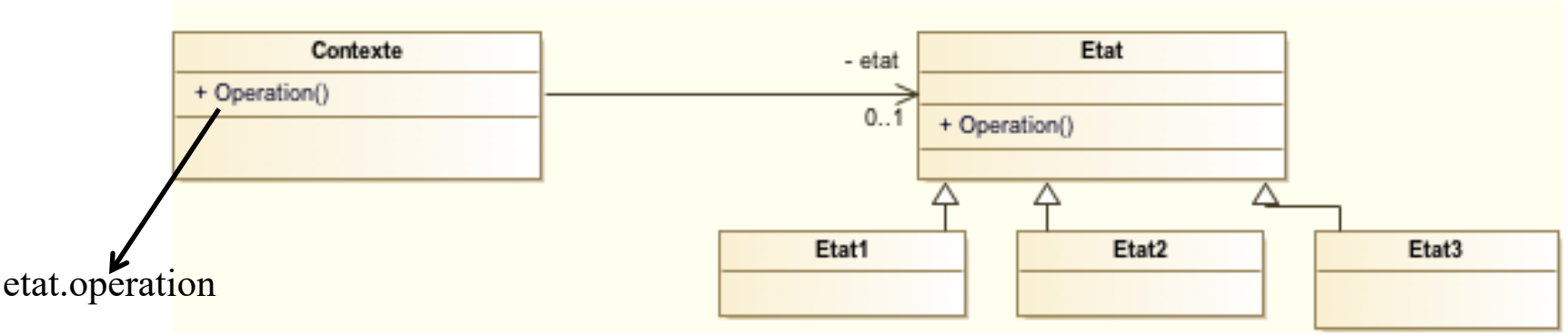

```
public class Dictionnaire{
    private DictionnaireImp implementation;
    public String definirMot (String  mot) {
       return implementation.definirMot (mot);
    public String changerDico (DictionnaireImpl d) {
       implementation = d;
    }
}
```

```
public class Larousse extends DictionnaireImpl {
   public String definirMot (String  mot) {
      return  "[Larousse] : " + mot;
}}
public class LeRobert extends DictionnaireImpl {
   public String definirMot (String  mot) {
      return  "[LeRobert] : " + mot;
}}
```

Pattern « State »

Principe

- Objectif
  - Changer dynamiquement le comportement d'un objet en fonction de son état interne
- Principe
  - Externaliser l'état à des objets-états
  - Déléguer les comportements dépendant de l'état à ces objets-états
  - Le comportement change automatiquement avec le changement d'état
- Architecture de classes

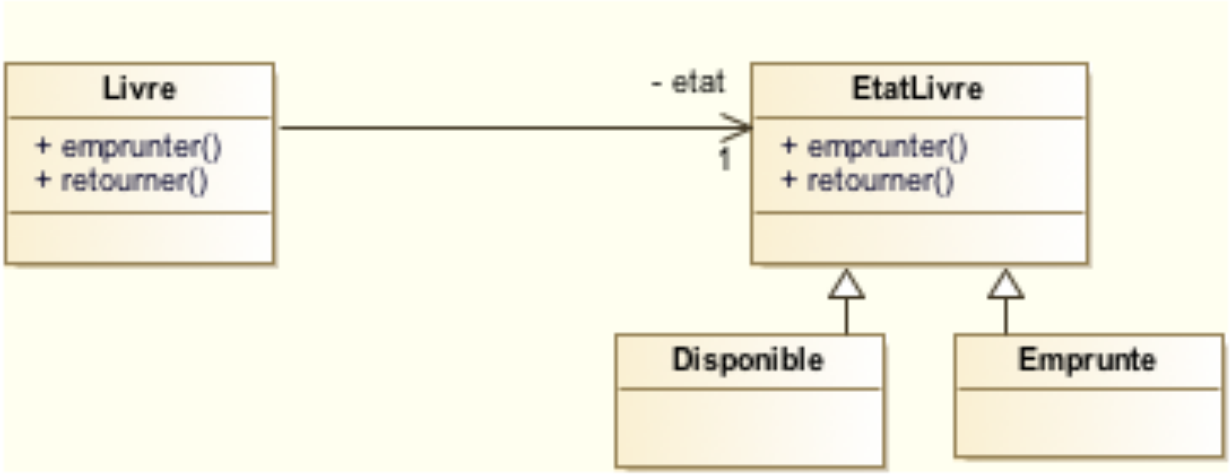


Pattern « State »

Exemple

☞ Exemple : Emprunt de livre

- Etats possibles : disponible, emprunté (puis prise en compte de l'état « abîmé »)
- Les états sont supposés évolutifs : les découpler de la classe Livre
- Traitements dépendant de l'état :
  - Emprunt d'un livre : le livre devient « Emprunté » s'il est « Disponible » et Exception sinon
  - Retour d'un livre : le livre passe à l'état « Disponible » s'il est « Emprunté » et Exception sinon



Pattern « State »

Code Java

☞ Exemple : Emprunt de livre

```
public class Livre {
    private EtatLivre etat;
    public void emprunter () {
        this.etat.emprunter (this);
    }
    public void retourner () {
        this.etat.retourner (this);
    }
    public void setEtat (EtatLivre e) {
        this.etat = e;
}}
```

```
public interface EtatLivre {
    public void emprunter (Livre livre) ;
    public void retourner (Livre livre) ;
}
```

```
public class Disponible implements EtatLivre {
    public void emprunter (Livre livre) {
        livre.setEtat (new Emprunte() );
    }
    public void retourner (Livre livre) {
        throw new LireEmprunteException ();
}}
```

```
public class Emprunte implements EtatLivre {
    public void retourner (Livre livre) {
        livre.setEtat (new Disponible() );
    }
    public void emprunter (Livre livre) {
        throw new LireEmprunteException ();
    }
}
```

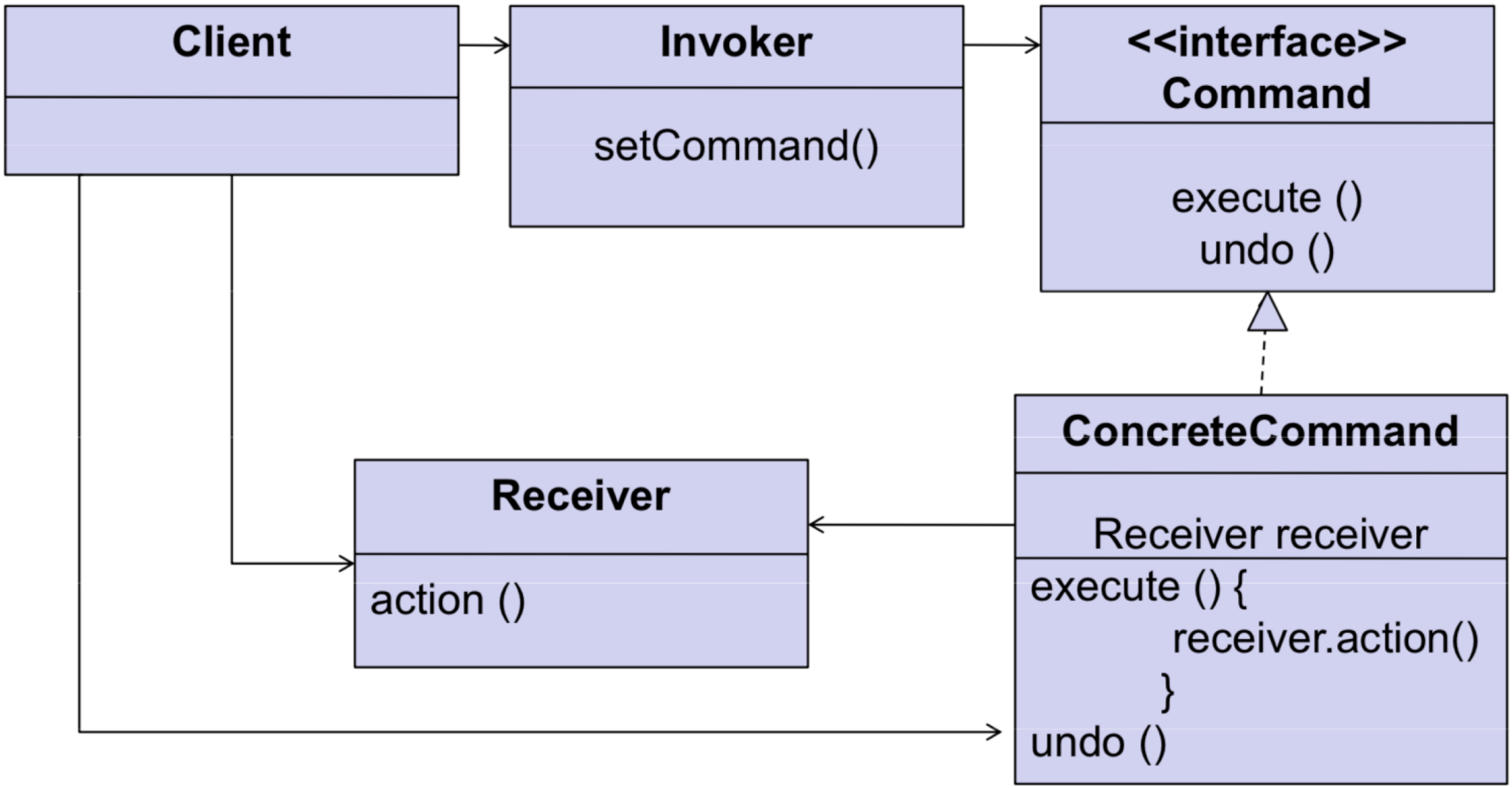
Pattern « Command »

Principe

- Objectif
  - Enregistrer et tracer (*logs*) les opérations effectuées
  - Rendre les opérations réversibles : pouvoir les annuler si besoin
  - Décharger le client de cette responsabilité (annulation, traçage, logs)
  - Découpler (au niveau des méthodes) l'appelant (client) de l'appelé (classe réalisant les traitements)
- Principe
  - Chaque appel de méthode est enregistré/encapsuler dans un objet *Command*
  - L'objet *Command* sauvegarde l'appel et le transmet à l'objet appelé
  - L'objet *Command* dispose d'une méthode d'annulation de l'appel réalisé

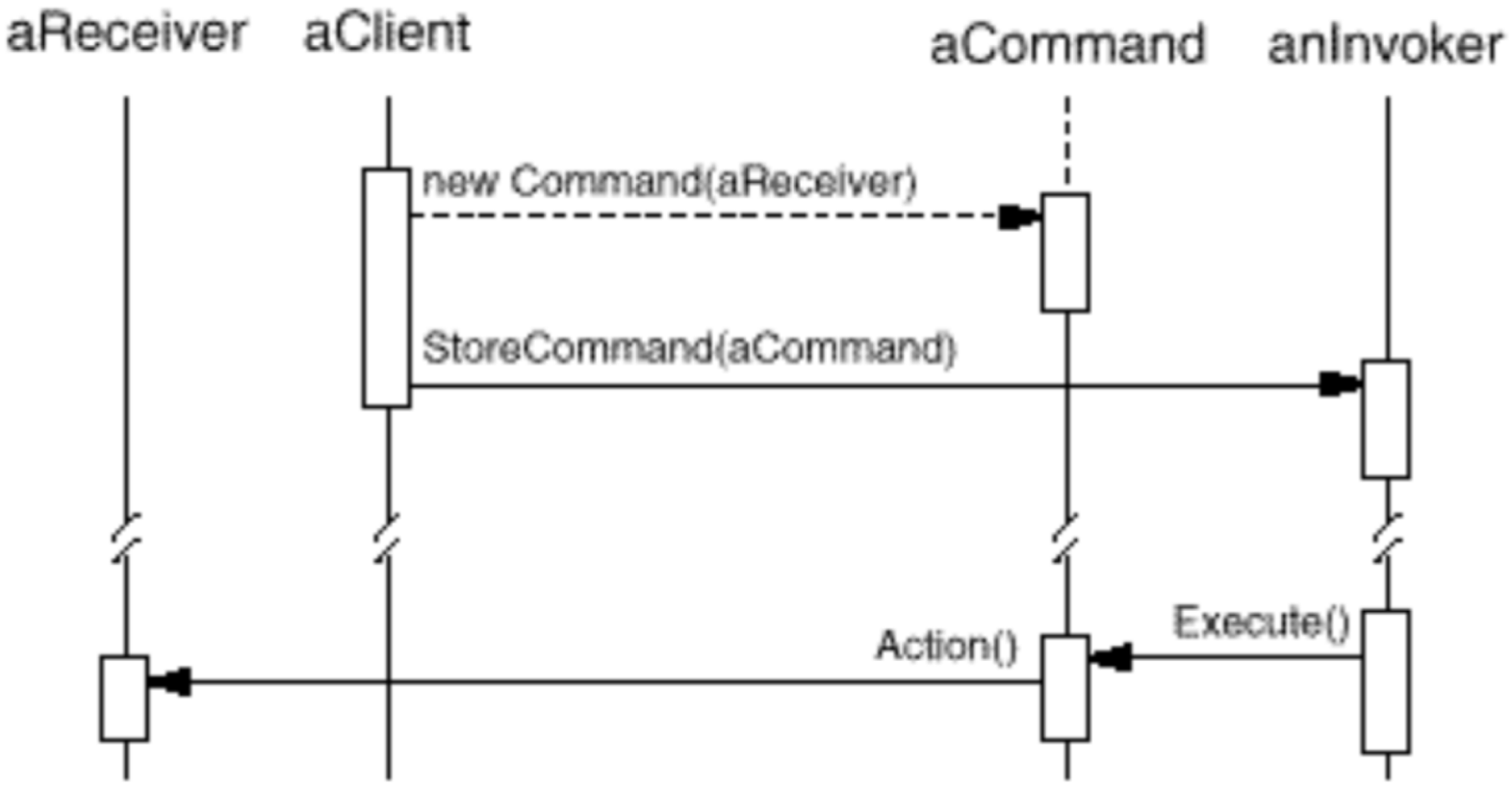
Pattern « Command »

Schéma du pattern



Pattern « Command »

Schéma d'exécution d'une opération/appel



Pattern « Command »

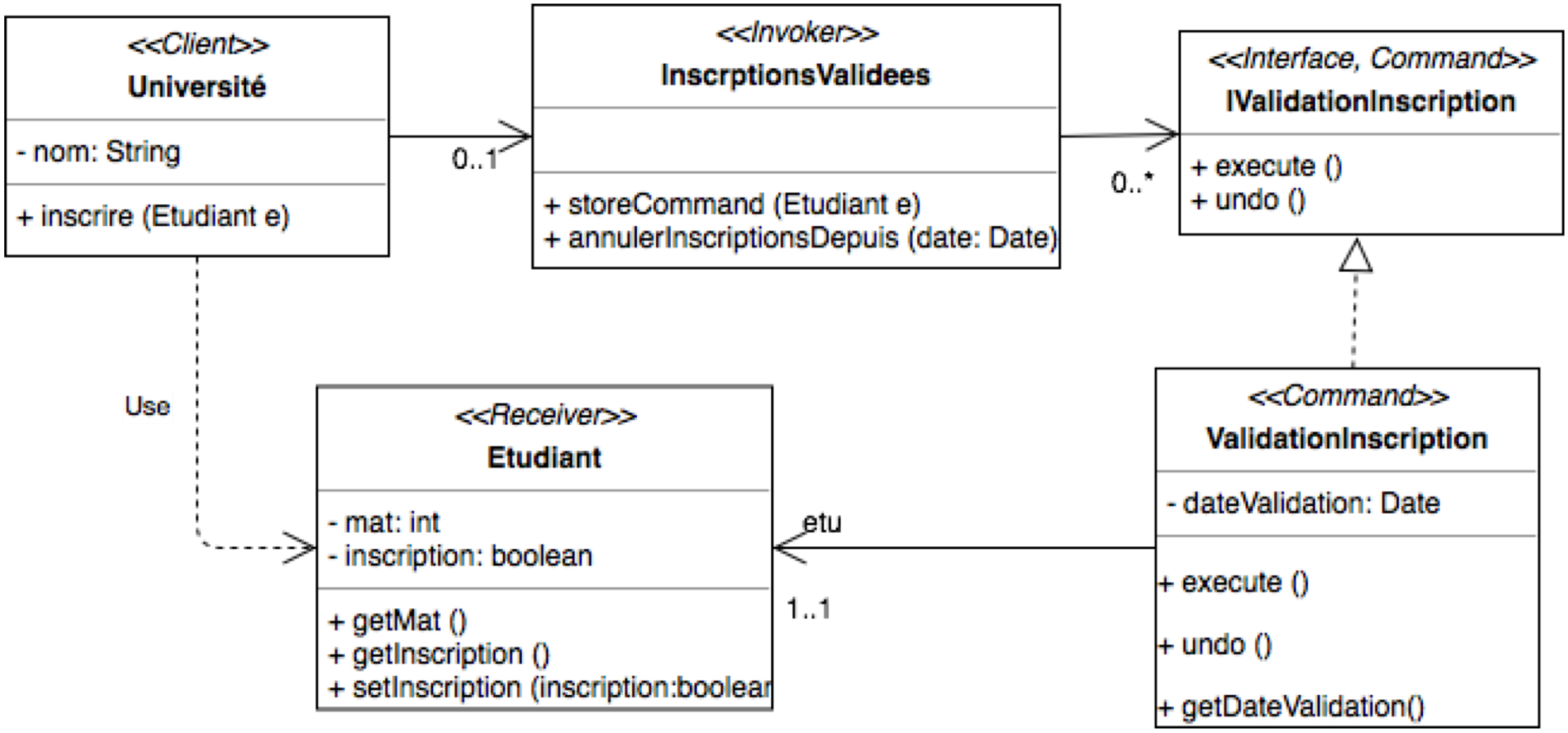
Rôles des composants du pattern

- *Receiver* : classe réalisant les traitements *reversibles*
- *Client* : classe cliente des traitements du *Receiver*
- *Command* : sauvegarde une action (appel au *Receiver*) venant du *Client*
- *Invoker* : sauvegarde les commandes passées pour un éventuel retour en arrière (annulation)

Pattern « Command »

Exemple

☞ Validation des inscriptions avec possibilité d'annulation



☞ Annuler les inscriptions validées dans la journée !

Pattern « Visitor »

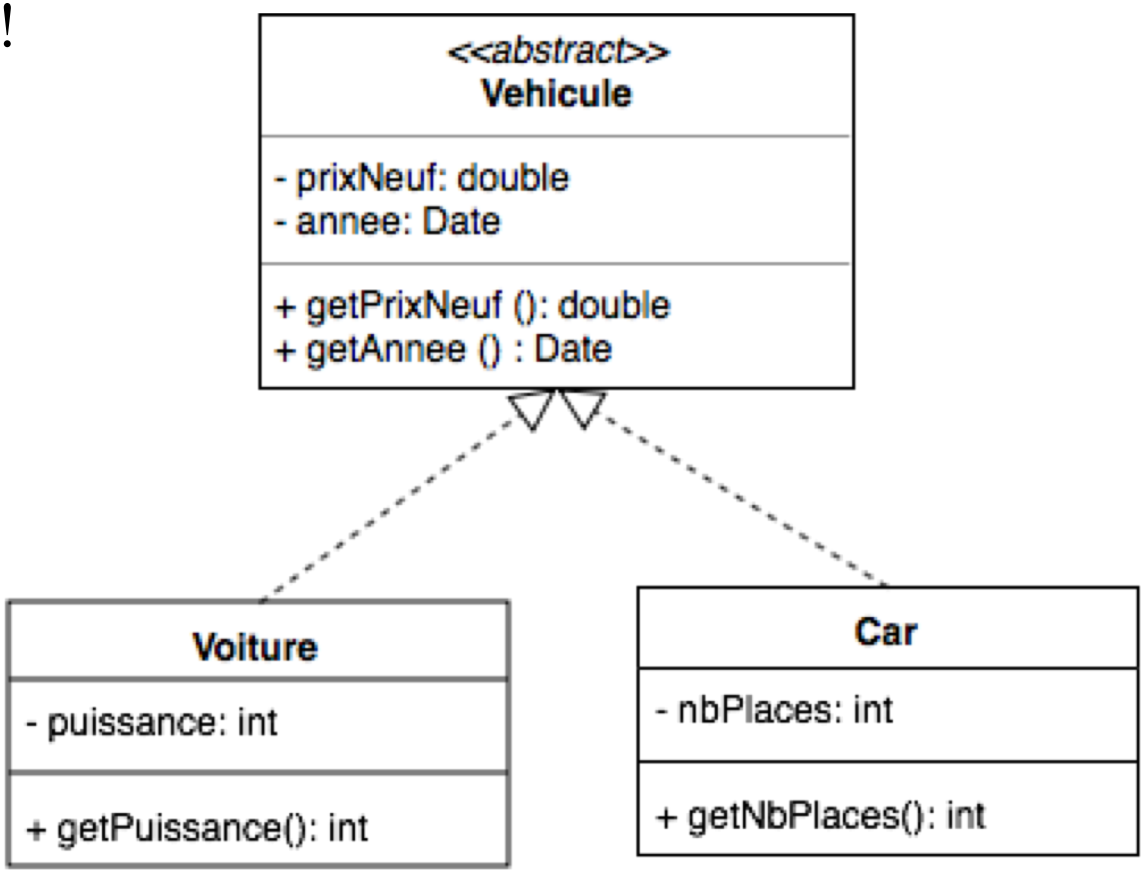
Principe

- Objectif
  - Ajouter de nouvelles opérations à une hiérarchie de classes
  - Les classe changent peu, mais de nouvelles opérations peuvent être définies
- Principe
  - Prévoir une méthode générique qui introduit (accepte) de nouvelles fonctions non-définies à la conception initiale
  - Coder les nouvelles fonctions dans de nouvelles classes (Visiteur)
  - Les fonctions ajoutées pour les sous-classes peuvent être différentes

Pattern « Visitor »

Exemple

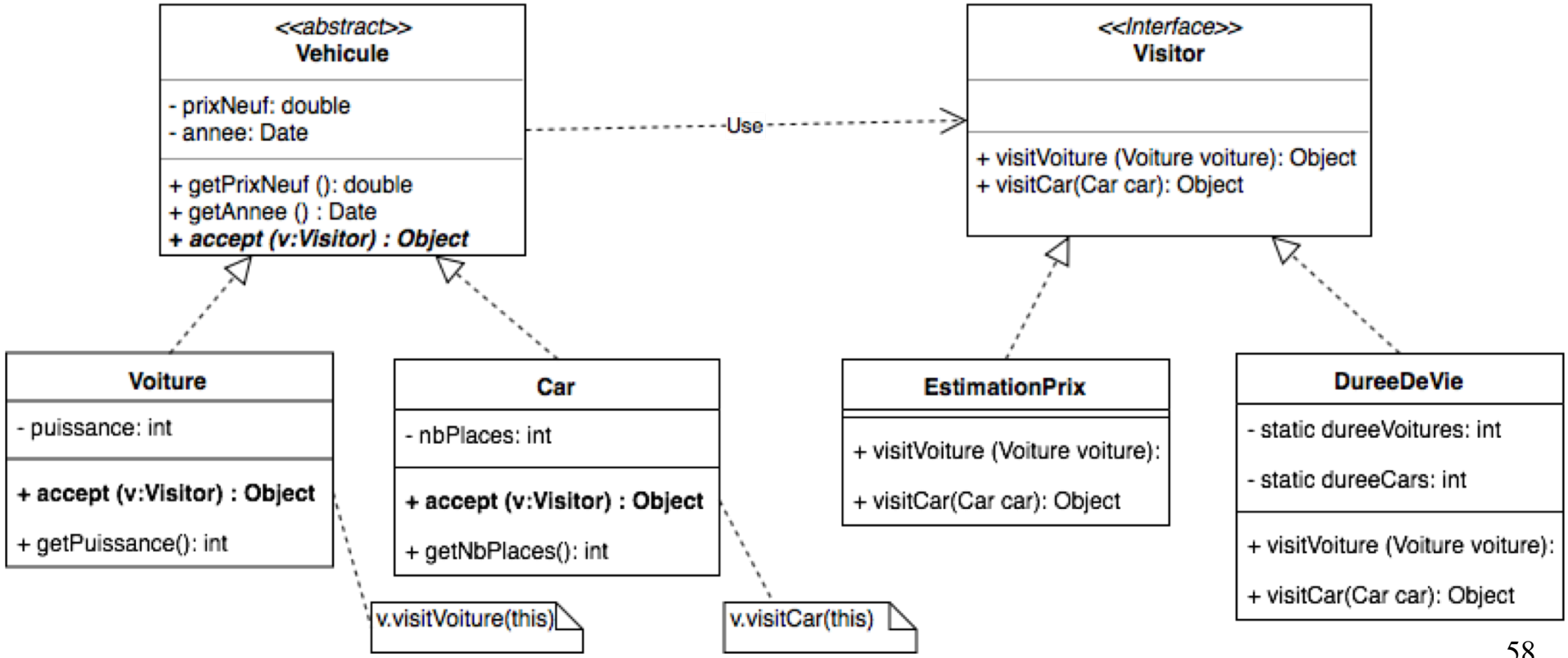
- Exemple : gestion de parc de véhicules
- Problématique : ajout de traitements dans le futurs sur les classes de véhicules !



Pattern « Visitor »

Schéma des classes

- Exemple : gestion de parc de véhicules
- Solution : injecter (via *accept()*) un traitement inconnu (non implémenté, méthode *visit()*) via une interface (*Visitor*) qui sera implémentée plus tard (à chaque nouveau traitement)



Pattern « Visitor »

Implémentation

```
public abstract class Vehciule {
   …
   public abstract Object accept (Visitor v);
}
```

```
public class Voiture extends Vehciule {
   …
   public Object accept (Visitor v) {
       return v.visitVoiture (this)
   }}
```

```
public class Car extends Vehciule {
   …
   public Object accept (Visitor v) {
       return v.visitCar (this)
   }}
```

```
public class UtilisationVistitor {
   public static void main (String [] args) {
      Voiture v = new Voiture (5000, 2018);
     int prixEstime = (int) v.accept (new EstimationValeur());
} }
```

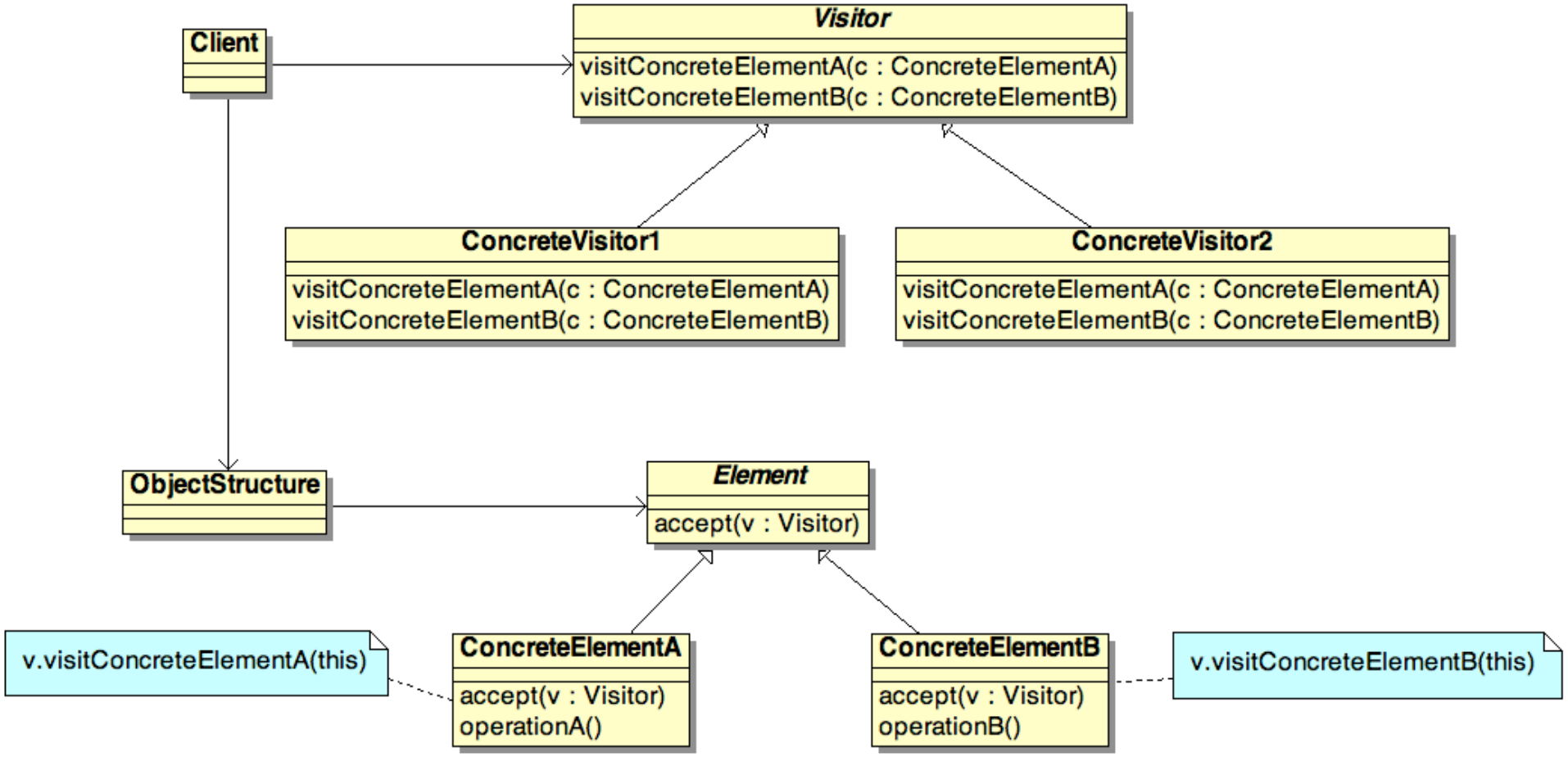
```
public interface Visitor {
    public Object VisitVoiture (Voiture voiture);
    public Object VisitCar (Car car);
}
```

```
// nouevau traitement !
public class EstimationValeur implements Visitor {
    public Object VisitVoiture (Voiture voiture) {
        int anciennete = new Date().getYear() - voiture.getAnnee();
        double estimation = voiture.getPrixNeuf() * (1 – anciennete * 5%);
        return estimation >100 0 ? estimation : 100;
    }
   public Object VisitCar (Car car) {
        int anciennete = new Date().getYear() - car.getAnnee();
        double estimation = car.getPrixNeuf() *  (1 – anciennete * 1000);
        return estimation >2000  ? estimation : 2000;
    }
}
```

Pattern « Visitor »

Schéma standard

☞ Architecture de classes



Pattern « Visitor »

Synthèse

- Cas d'application
  - Ajout de nouvelles opérations à des classes structurellement différentes
  - Hypothèse : classes stables !
- Avantages
  - Cohésion : groupement des opérations commune
  - Comportement localisé donc facile à maintenir
- Inconvénients
  - Complexité d'ajout et suppression de classes concrètes
  - Peut nécessiter de gérer l'état interne des classes concrètes dans les visiteurs (encapsulation)