Ingénierie des Systèmes d'Information

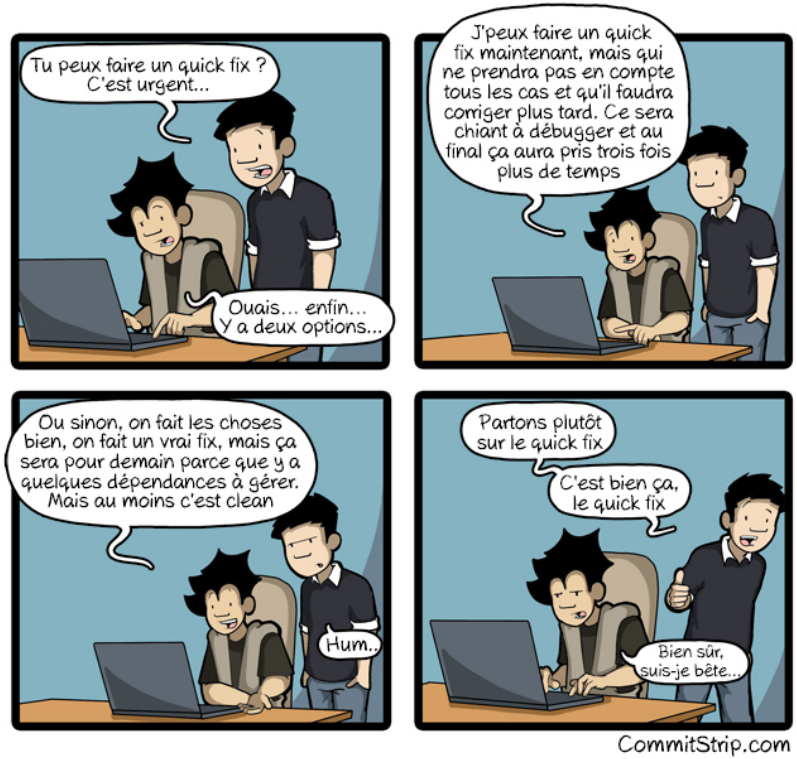
Laëtitia Matignon
laetitia.matignon@univ-lyon1.fr

Département Informatique - Polytech Lyon
Université Claude Bernard Lyon 1
2019 - 2020

Objectifs

Objectifs

- Maîtriser différents aspects du développement d'un logiciel
- Validation des acquis en POO, UML et programmation Java
- Compréhension des principes avancés de conception orientée objet
- Compréhension de l'intérêt des design pattern informatiques
- Mise en pratique des design pattern (UML et Java)
- Compréhension de l'intérêt des tests unitaire et des tests fonctionnels



Programme

- Rappels UML
- Mécanismes de base et principes avancées de Conception Orientée Objet
- Patrons de conception
- Tests logiciels, Programmation fonctionnelle

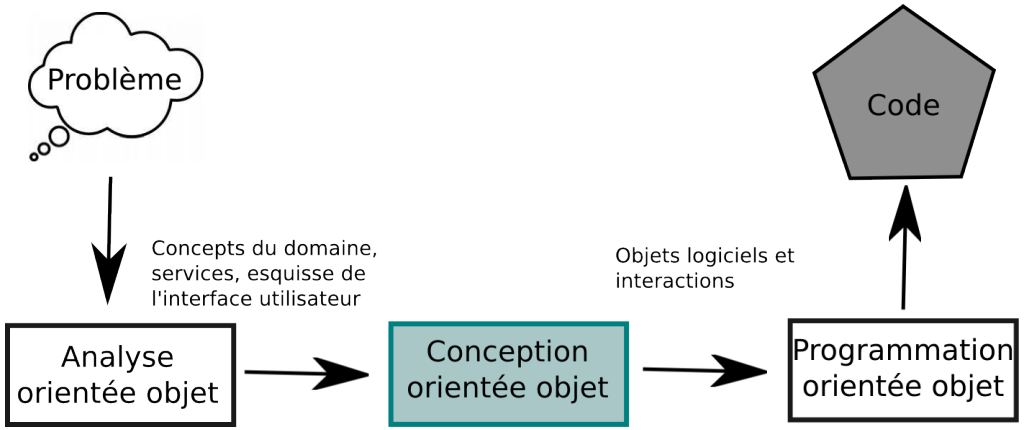
- liens vers cours, énoncés TDs : `https://projet.liris.cnrs.fr/sycosma/wiki/doku.php?id=isi3`
- Évaluation : une note TPs (coef. 0.5), une note CF (coef. 0.5)

1. Concepts fondamentaux et Principes avancés de Conception Orientée Objet

Laëtitia Matignon
laetitia.matignon@univ-lyon1.fr

Département Informatique - Polytech Lyon
Université Claude Bernard Lyon 1
2017 - 2018

Conception logicielle



Objectifs du cours

- Rappel des concepts UML relatifs à la vue structurelle
  - Préciser les différences sémantiques entre le langage UML (diagramme de classes) et les langages de POO (Java, C++)
  - Nécessaire pour *reverse engineering*
- Présenter les principes avancés de conception orientée objet
- Rappel sur les notions de couplage et dépendance

Plan

Qu'est-ce qu'UML ?

- UML = Unified Modeling Language
- UML = **Langage** universel pour la modélisation objet ... pas une méthode

Différence Langage – Méthode

- Langage de modélisation = notations, grammaire, sémantique
- Méthode = comment utiliser le langage de modélisation (recueil des besoins, analyse, conception, mise en oeuvre, validation, ...)

- UML est une unification de quasiment tous les langages de modélisation d'applications orientées objet.
- UML n'est pas un langage de programmation, n'est pas un processus de développement
- UML est indépendant d'un langage de programmation
- UML est une **norme** maintenue par l'OMG
  `http://www.omg.org/uml`

Classes et objets

- On définit des classes = regroupement d'objets similaires (appelés instances)
  - **Abstraction** : factorisation des caractéristiques communes à un ensemble d'objets similaires
  - Une classe décrit une infinité d'instances
- On instancie des objets à partir des classes :
  - Objet = etat + comportement + identité
  - Attributs, méthodes, référence

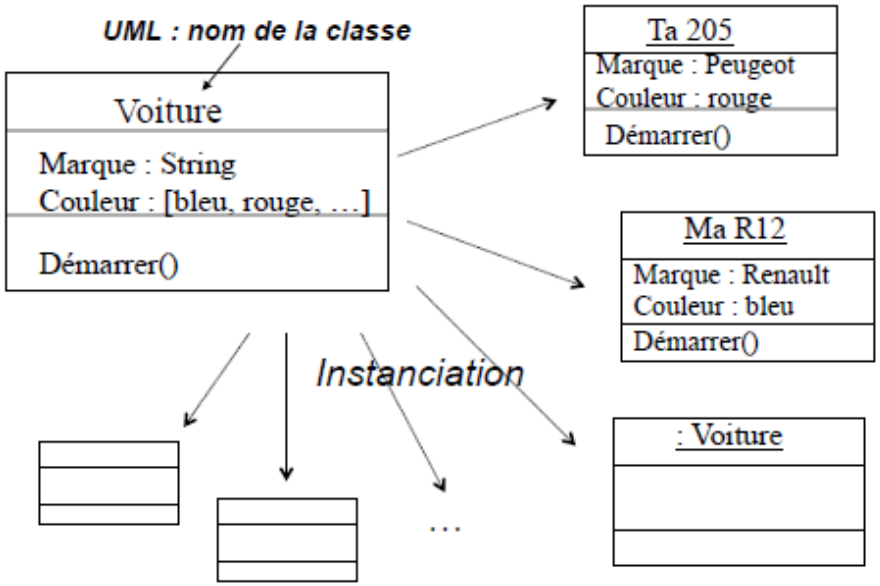
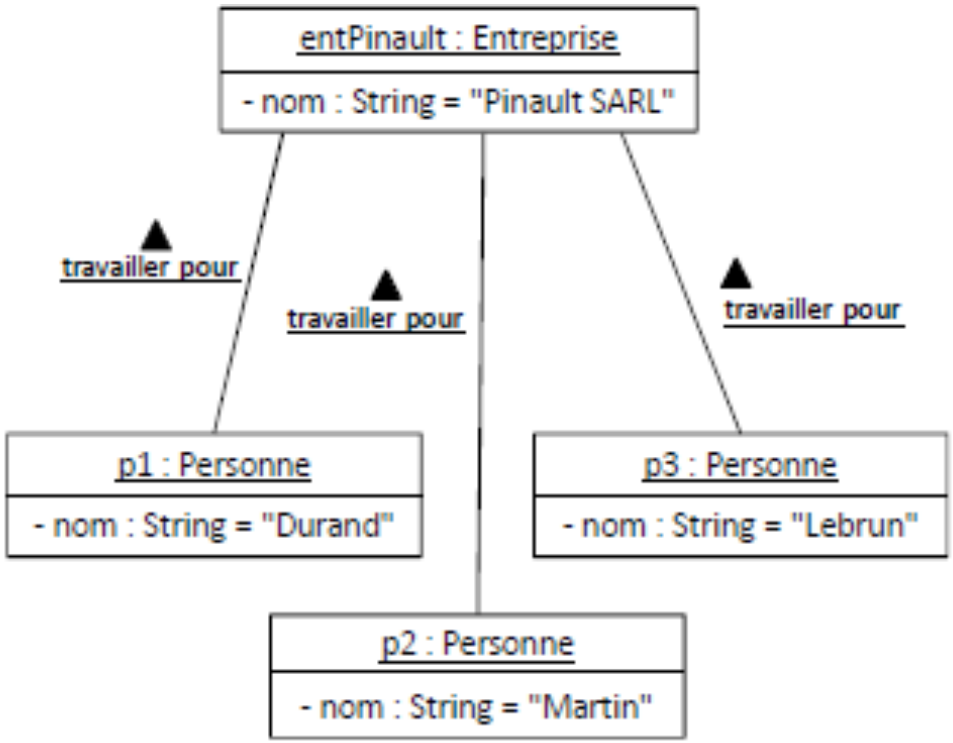


Diagramme d'objets

Objectifs

- Représente les objets/**instances** et leurs liens **à un instant donné**

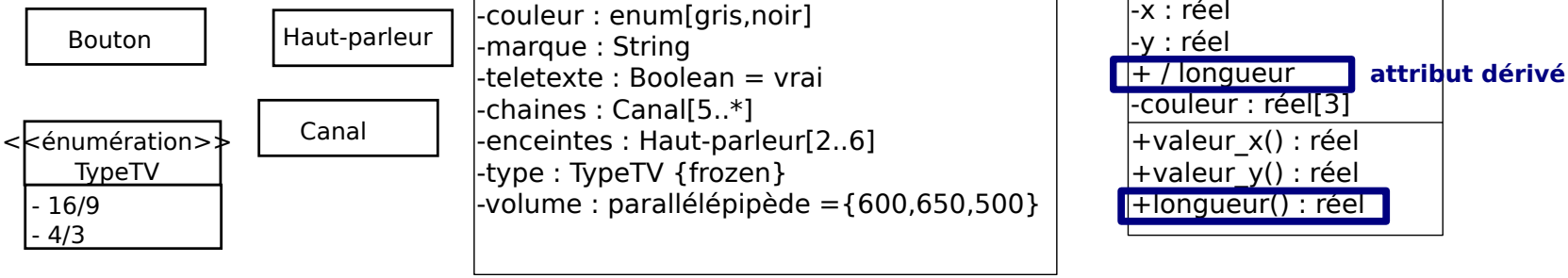


Représentation UML des attributs

- Format de description d'un attribut :

visibilité nom : type [multiplicité] = valeur_initiale {propriétés}

- public + privé - protégé # paquetage ~
- facultatif mais impératif pour l'implémentation
- facultatif ex. couleurs : Saturation[3] points: Points[2..*]
- facultatif
- facultatif ex. {frozen} mise à jour interdite



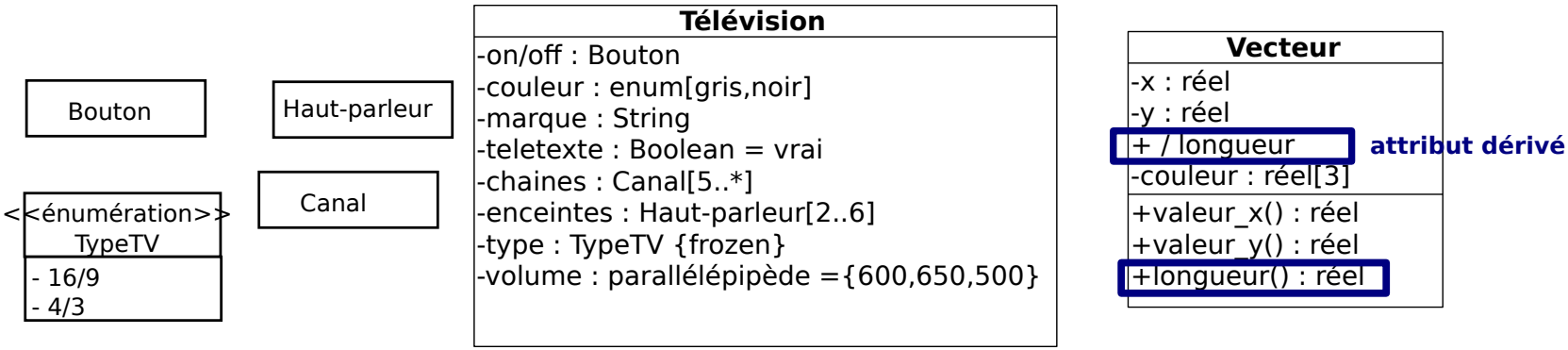
Représentation UML des attributs

- Format de description d'un attribut :

visibilité nom : type [multiplicité] = valeur_initiale {propriétés}

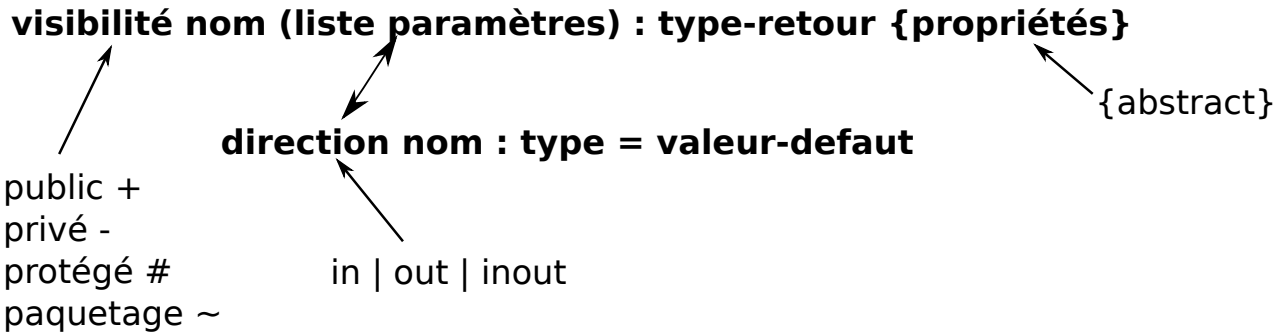
- public + / privé - / protégé # / paquetage ~
- facultatif mais impératif pour l'implémentation
- facultatif ex. couleurs : Saturation[3] points: Points[2..*]
- facultatif
- facultatif ex. {frozen} mise à jour interdite

- Attributs de classe (statiques) soulignés
- Attributs dérivés (calculés) précédés de " /"
- Enumération : stéréotype ≪enumeration≫

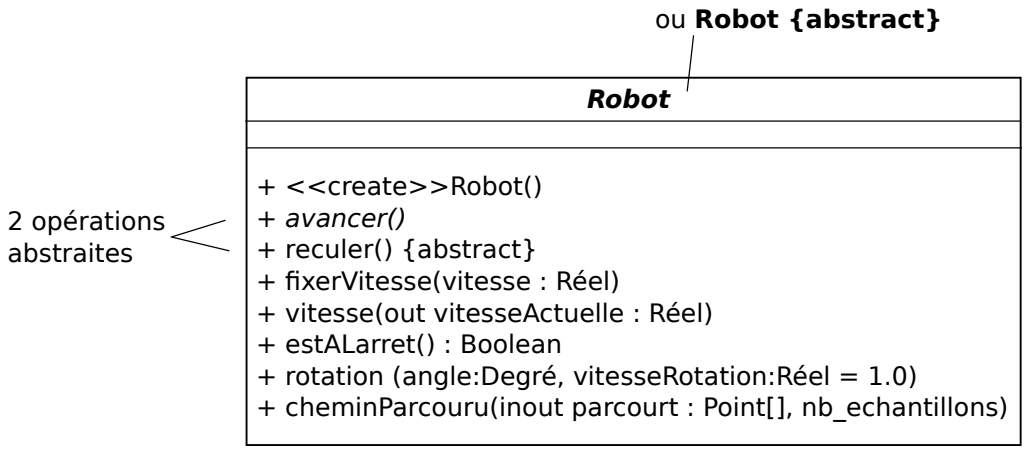


Représentation UML des opérations

- Format de description d'une opération :



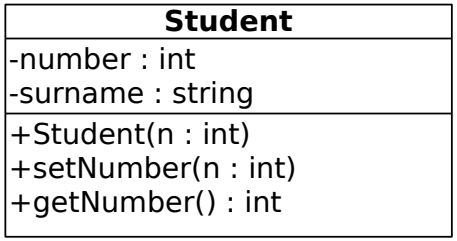
- *opérations abstraites* (non implémentées) / opérations de classe (statiques)
- Propriétés : {abstract}, {query} (l'opération n'altère pas l'état de l'instance concernée), pré- et post-conditions, description du contenu (commentaires, OCL)
- Stéréotypes d'opérations : constructeur ≪create≫, destructeur ≪destroy≫



Les classes : Implémentation Java

Student.java

```
...
public class Student {
  // Attributs
  private int number;
  private String surname;
  ...
  // Methodes
  public Student(int _n)
      {
    number = _n;
    surname = "STUDENT";
  }
  public void setNumber(
      int n) {
    number = n;
  }
  public int getNumber()
      {
    return number;
}
...
};
```

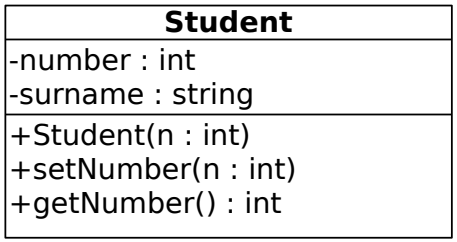


- Mode d'accès (`public`, `private`, `protected`), également appelé **visibilité**, spécifié avant chaque attributs ou méthodes
- Définition des méthodes (écriture du corps des fonctions) dans la définition de la classe, dans un unique fichier `.java`
- Obligatoirement un fichier .java par classe

Les classes : Implémentation C++

Student.h

```
class Student
{ // Attributs
  private:
    int number;
    std::string surname;
    ...
  // Methodes
  public:
   Student(int _n) {
     number = _n;
     surname = "STUDENT";
   }
   void SetNumber(int n);
   int GetNumber() const;
    ...
};
```



Student.cpp

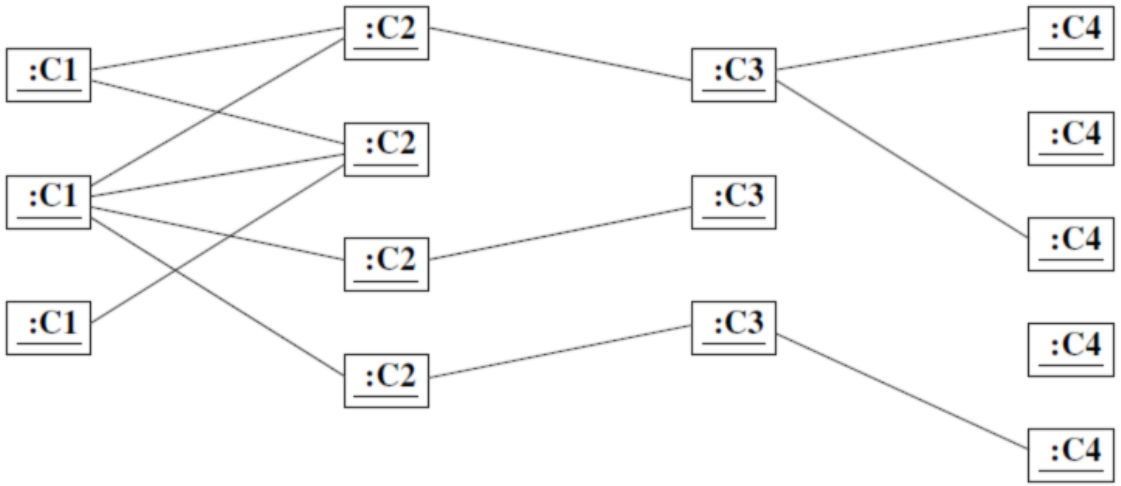
```
#include "Student.h"
void Student::SetNumber(
     int n)
{    number = n;  }
int Student::GetNumber()
     const
{    return number;  }
```

- Mode d'accès (`public`, `private`, `protected`) spécifié "par lot"
- Définition des méthodes (écriture du corps des fonctions)
  - Dans la définition de classe (.h) : méthodes inline (plutôt pour les petites fonctions)
  - En dehors de la définition de la classe (.cpp) : le plus courant
- Autant de classes que l'on veut par fichier .h

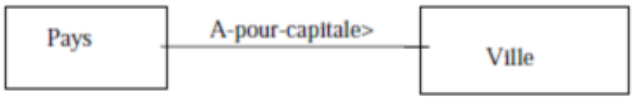
Associations entre classes

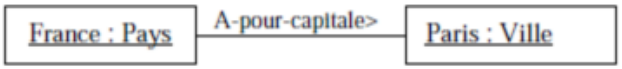
- Relation entre deux classes (binaire) ou plus (n-aire).
- Décrit les connexions structurelles entre les instances des classes associées
- Abstraction des relations définies par les liens entre objets

Liens entre objets :



Associations entre classes d'objets :

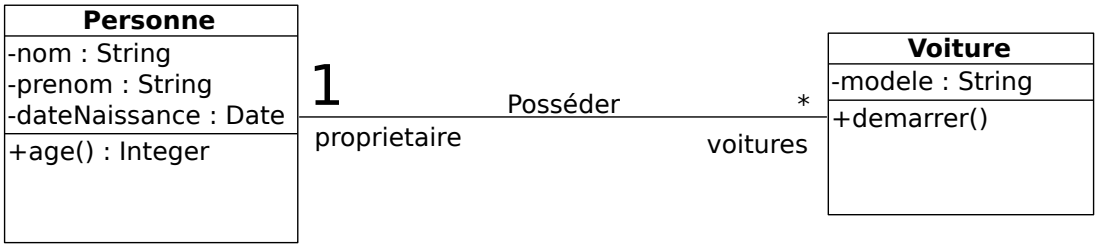


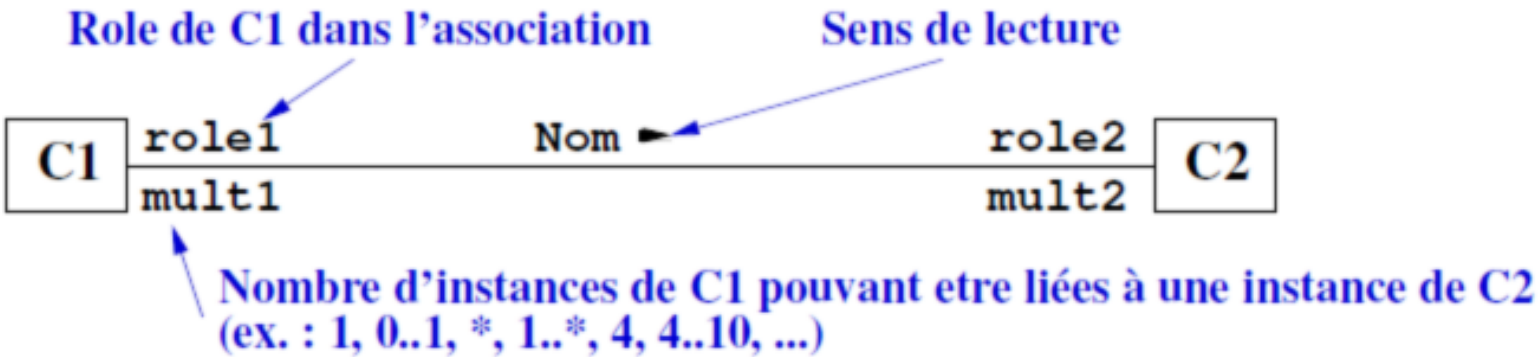


Associations entre classes

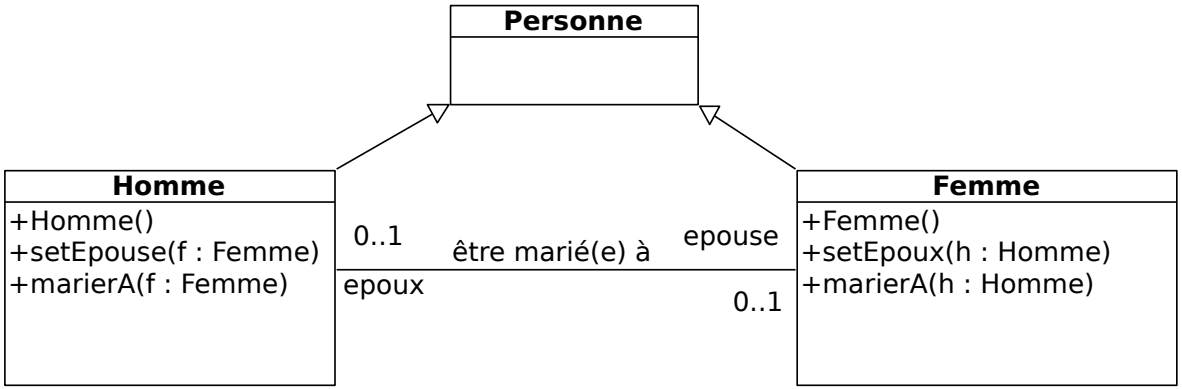
Notion d'associations

- Une association de la classe A vers B est équivalente à un attribut de type B dans la classe A





Association binaire



Homme.java

```
public class Home extends
      Personne {
  private Femme epouse;
  public Homme() {
// Celibataire par defaut
   epouse = null; }
//mutateur
  public void setEpouse(Femme f)
 {  epouse = f;}
  public void marierA(Femme f)
{   epouse = f;
   f.setEpoux(this); }
 };
```

Femme.java

```
public class Femme extends
     Personne {
private Homme epoux;
public Femme() {
 // Celibataire par defaut
 epoux = null;  }
//mutateur
public void setEpoux(Homme h)
 { epoux = h; }
public void marierA(Homme h)
{epoux = h;
h.setEpouse(this);}
};
```

Association avec sens de navigation

Diagramme UML correspondant ?

Polygone.java

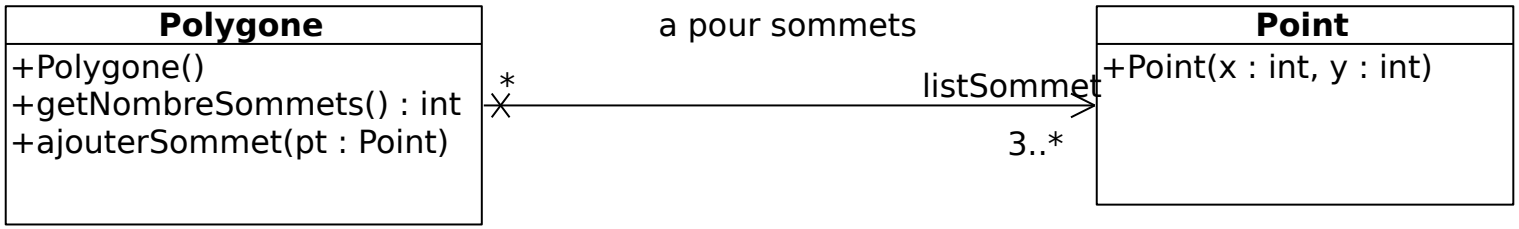
```
public class Polygone {
 protected ArrayList<Point> listSommet;
 public Polygone() {listSommet = new ArrayList<Point>();}
 public int getNombreSommets() {return listSommet.size();}
 public void ajouterSommet(Point pt) {listSommet.add(pt);}

};
```

Point.java

```
public class Point {
 public int x, y;
 ...
 public Point(int _x, int _y) {x=_x; y=_y;}
};
```

Association avec sens de navigation



Polygone.java

```
public class Polygone {
 protected ArrayList<Point> listSommet;
 public Polygone() {listSommet = new ArrayList<Point>();}
 public int getNombreSommets() {return listSommet.size();}
 public void ajouterSommet(Point pt) {listSommet.add(pt);}

};
```

Point.java

```
public class Point {
 public int x, y;
 ...
 public Point(int _x, int _y) {x=_x; y=_y;}
};
```

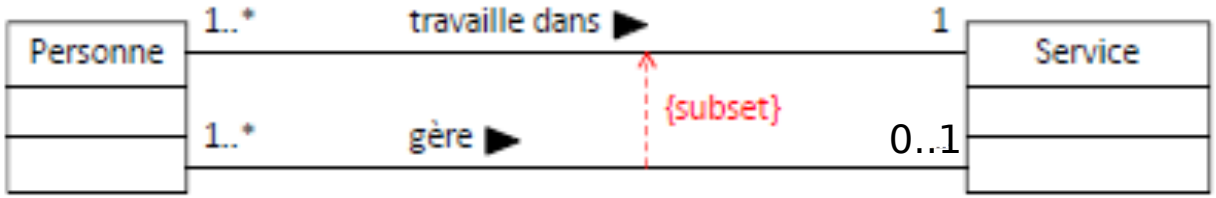
Contraintes sur les associations

- Exemple 1 : *les sommets dans un polygone sont ordonnés.*

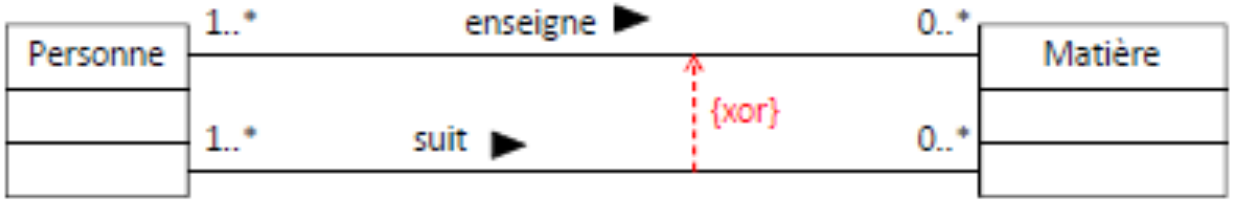
- La notion d'ordre est déjà présente dans la plupart des collections d'objet (`List` en Java ou std : :list en C++)

Contraintes sur les associations

- Contrainte d'inclusion : une personne travaille dans un service, et elle peut, en plus, gérer ce service. Plusieurs employés peuvent cogérer un service



- Contrainte d'exclusivité : une personne (sans distinction de classe entre étudiant et enseignant) est associée à une matière, soit parce qu'elle l'enseigne, soit parce qu'elle la suit (mais jamais les deux simultanément)

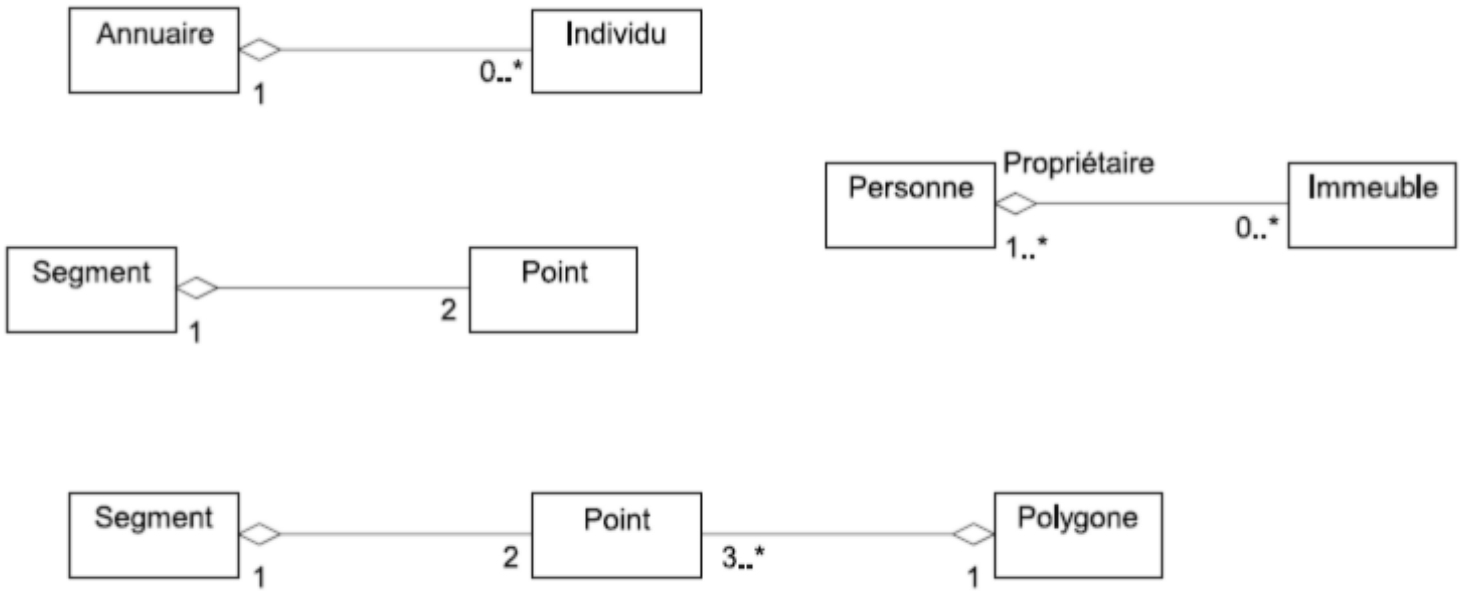


- Ces contraintes ne peuvent pas être représentée par le type de collection. Elles doivent êtres maintenues lors de la création des liens.

Associations particulières : agrégation

Agrégation

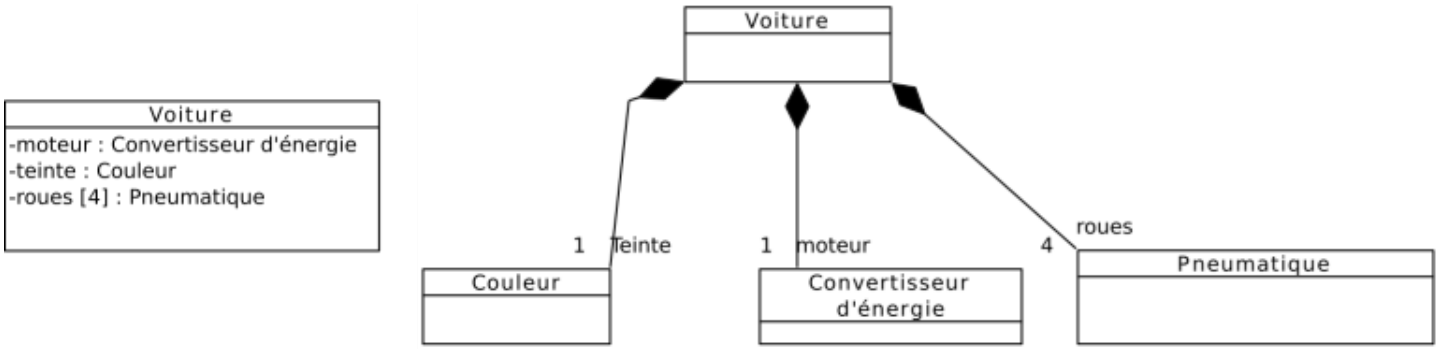
- Modéliser regroupement de parties dans un tout
- Association non-symétrique
  - Relation de dominance et de subordination
  - Une classe **fait partie** d'une autre classe
  - Une action sur une classe implique une action sur une autre classe
- Une classe peut appartenir à plusieurs agrégats



Associations particulières : composition

Composition

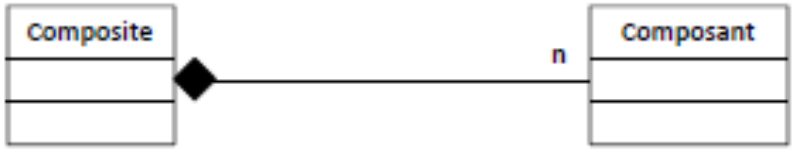
- Agrégation forte
- Contenance structurelle : Création/Copie/Destruction du composite → Création/Copie/Destruction de ses composants
- Un composant appartient à au plus un composite (mult. côté conteneur max 1)



Agrégation et composition

- Agrégation : Implémentation similaire à une association simple
- Composition :
  - Inclusion structurelle d'un composant dans un composite
  - La vie du Composant est liée à celle du Composite : la création (resp. destruction) du Composite entraîne la création (resp. destruction) de ses Composants
  - La composition peut être implémentée par l'ajout d'un attribut de classe Composant dans la classe Composite et instanciation du Composant dans la classe Composite.

Agrégation et composition



Composant.java

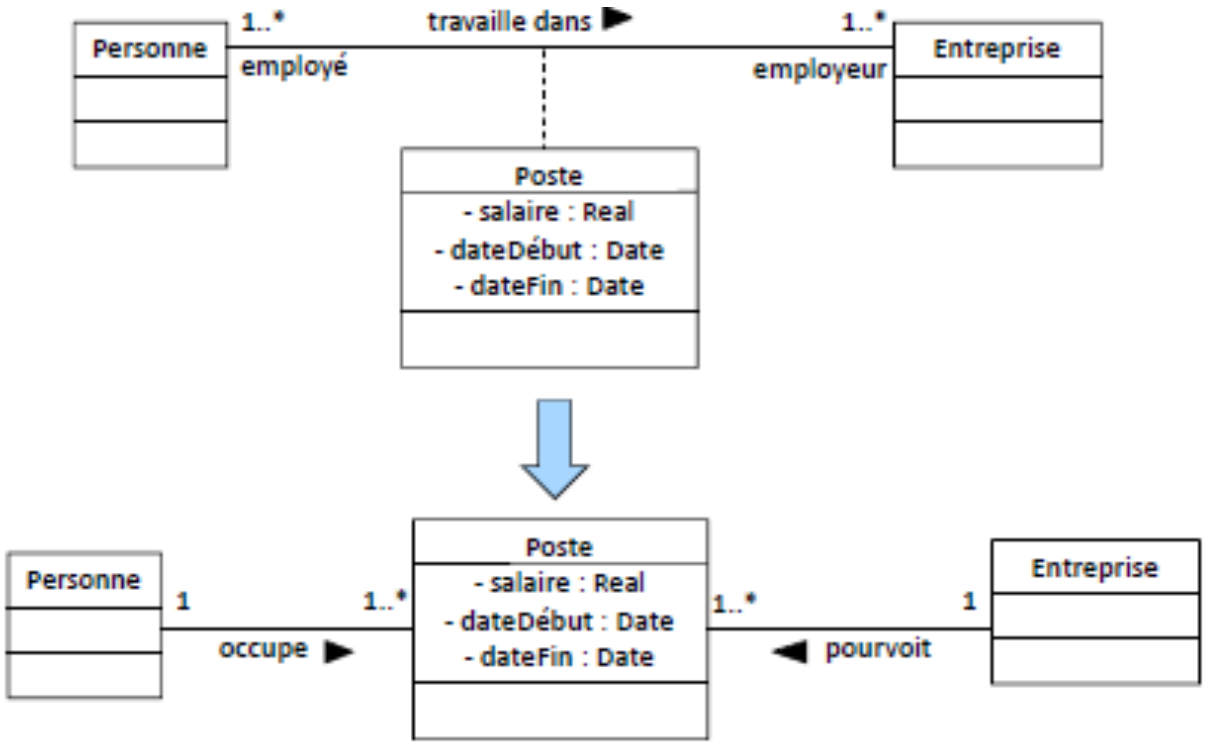
```
public class Composant {
  private Composite comp;
  public Composant() {comp = null;}
  void modifierComposite(Composite c) {comp = c;}
};
```

Composite.java

```
public class Composite {
  private Composant tabComposants[];
  public Composite() {
    tabComposants = new tabComposants[n];
    for (int i=0; i<n; i++) {
      tabComposants[i] = new Composant();
      tabComposants[i].modifierComposite(this);
    }
  }
...
}
```

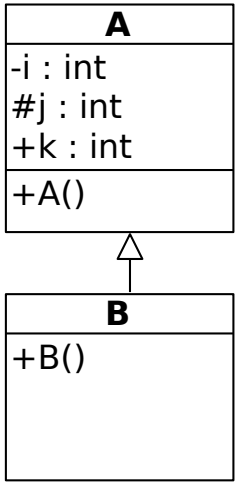
Classes-associations

- Une association peut être raffinée et avoir ses propres propriétés, qui ne sont disponibles dans aucune des classes qu'elle lie.
- Une association peut être représentée par une classe pour ajouter attributs et opérations à des associations
- Exemple : *une personne à travaillé dans des entreprises, à des périodes données et avec un certain salaire.*

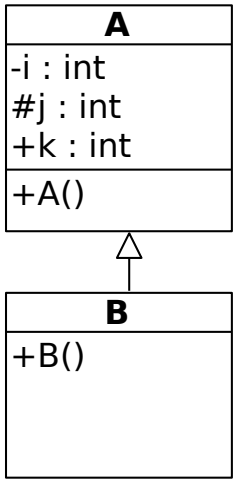


Héritage

- Concept fondamental : transmission des caractéristiques d'une classe vers une sous-classe
- Objectifs :
  - **spécialisation** d'une classe existante
  - factoriser des propriétés et comportements communs à plusieurs classes (**généralisation**)
  - hériter des attributs et des méthodes de sa super-classe
  - éviter la duplication et encourager la réutilisation
- Moyens :
  - relation de généralisation/spécialisation en UML
  - héritage en POO



Héritage simple : Implémentation



A.java

```
public class A {
  private int i;
  protected int j;
  public int k;
  public A() {
    i = 0;
    j = 0;
    k = 0;
  }
};
```

B.java

```
public class B extends
      A {
  public B() {
    i = 0; //???
    j = 0; //???
    k = 0; //???
  }
};
```

A.h

```
class A
{ private: int i;
  protected: int j;
  public: int k;
   A(){
    i = 0;
    j = 0;
    k = 0;
   } };
```

B.h

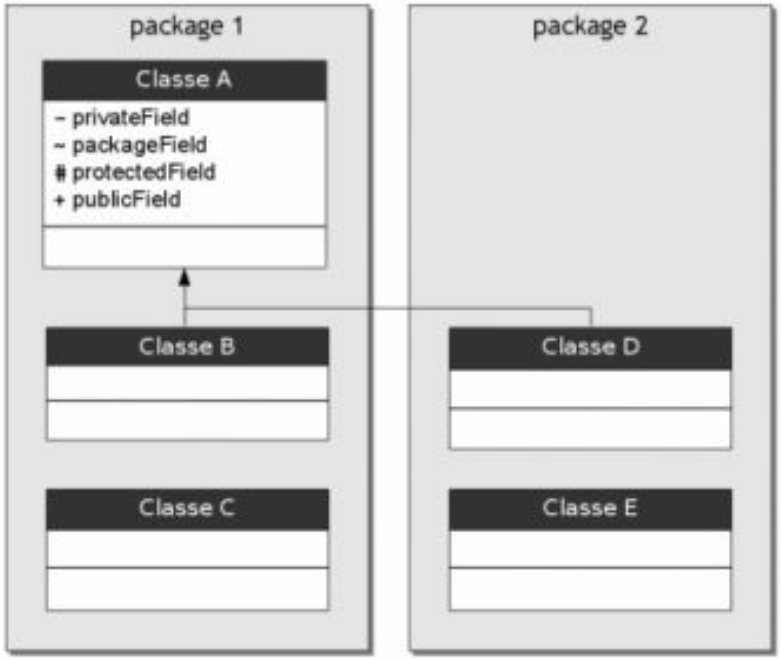
```
#include "A.hpp"
class B: public A
{
  public:
   B() {
    i = 0; // ???
    j = 0; //???
    k = 0; // ???
    }
};
```

Visibilité en Java

Quelles classes ont accès aux attributs protected de la classe A ?

Visibilité en Java

Les modificateurs de visibilité en Java



Visibilité des champs :

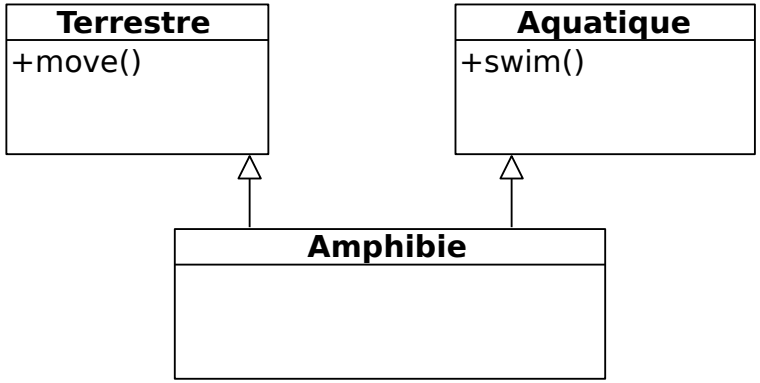
	Classe A	Classe B	Classe C	Classe D	Classe E
privateField	V				
packageField	V	V	V		
protectedField	V	V	V	V	
publicField	V	V	V	V	V

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Java Access to Members of a Class

Héritage multiple



- Héritage multiple autorisé en UML et en C++, pour bénéficier des opérations de plusieurs classes mères

Terrestre.h

```
class Terrestre {
  ...
 public:
   void move ();
}
```

Aquatique.h

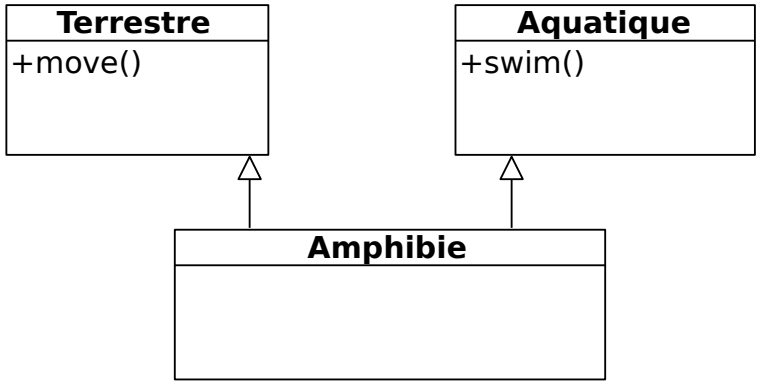
```
class Aquatique {
  ...
 public:
  void swim ();
 }
```

Amphibie.h

```
class Amphibie : public Terrestre, public Aquatique {...}
```

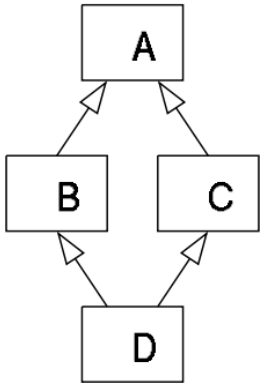
- `Amphibie` dispose à la fois des méthodes `move()` et `swim()`, qu'elle peut redéfinir

Héritage multiple : implémentation en C++



- La gestion des membres homonymes hérités des classes parentes
  - Si la classe `Amphibie` hérite à la fois des classes `Terrestre` et `Aquatique`. Il peut y avoir des problèmes de gestion des homonymes si :
    - `Terrestre` et `Aquatique` ont des attributs dont le nom est identique
    - `Terrestre` et `Aquatique` ont des méthodes dont le nom est identique
  - $\rightarrow$ on utilise le nom de la classe mère en préfixe afin de spécifier l'origine du membre requis (`Terrestre::attr` et `Aquatique::attr`)

Héritage multiple : implémentation en C++



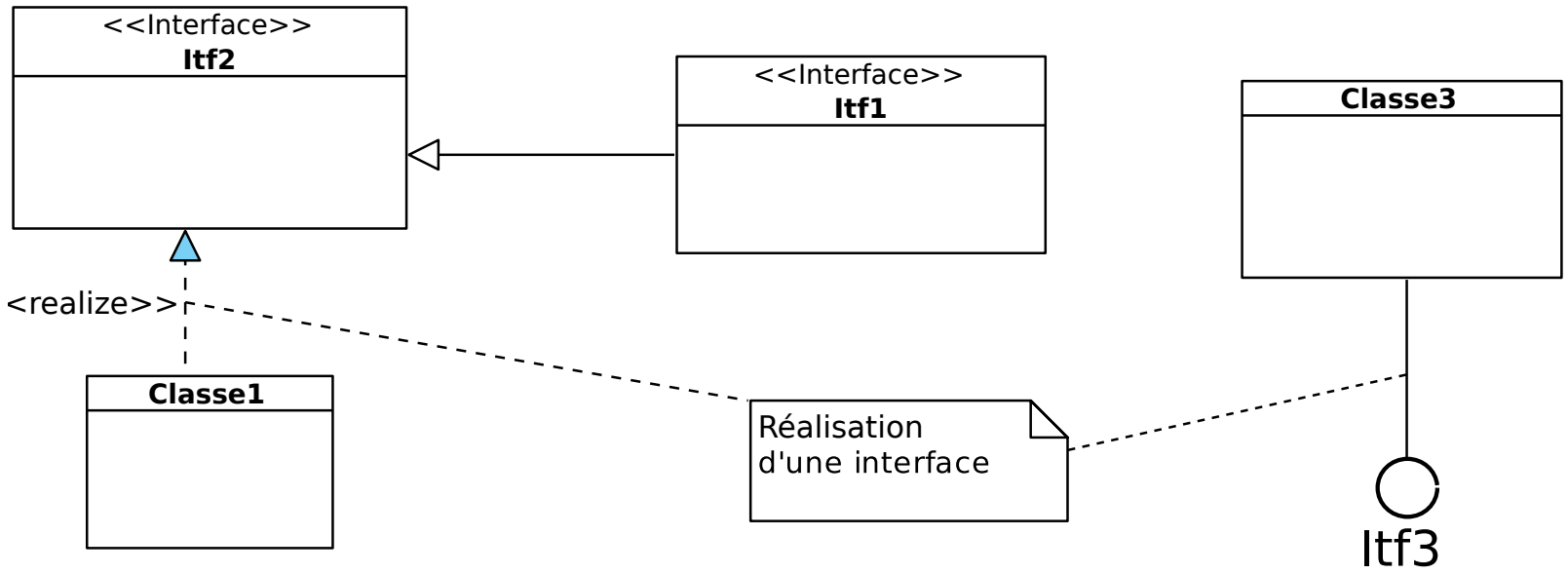
- La gestion de l'héritage à répétition
  - Les attributs de la classe A sont présents en double exemplaire dans la classe D, une fois à cause de l'héritage venant de B, une autre fois à cause de l'héritage en provenance de C
  - → Faire un héritage en mode `virtual` pour préciser de n'incorporer qu'une seule fois les membres de la classe A dans D
  - ```
    class B : public virtual A {...};
    class C : public virtual A {...};
    class D : public B, public C {...};
    ```
    ```

Interface

Qu'est-ce qu'une interface ?

Classe sans attributs dont toutes les opérations sont abstraites (classe abstraite pure)

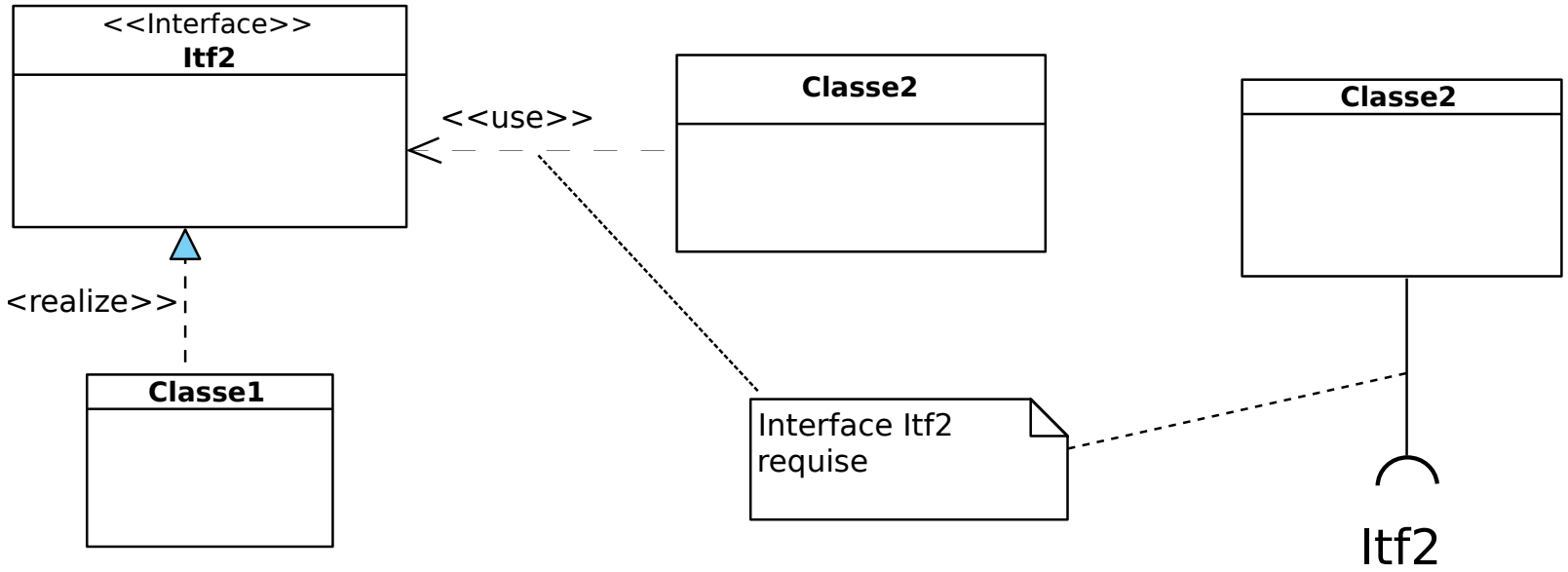
- Liste de services, savoir faire
- Ne peut pas être instanciée
- Doit être réalisée (implémentée) par des classes non abstraites
- Peut hériter d'une autre interface



Stéréotype `<<realize>>` facultatif.

Interface

Une classe peut aussi simplement dépendre d'une interface (interface requise).



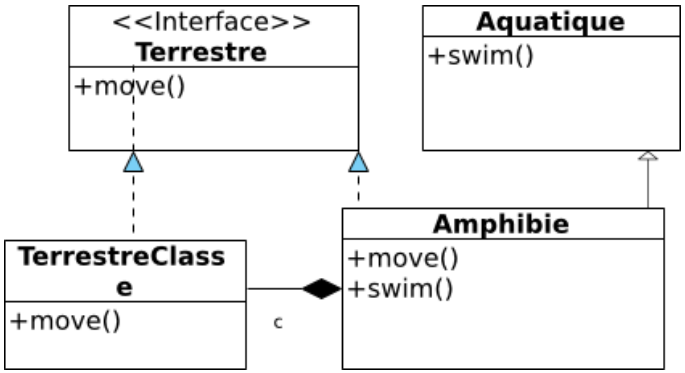
Réalisation

```
public class Classe1
    implements Itf2 {
  ...
}
```

Utilisation

```
public class Classe2 {
   public Classe2(Itf2 iface)
        { // public constructor
     ...
  }
```

Héritage multiple : implémentation en Java



- Héritage multiple impossible en Java : utilisation d'**interfaces** et **délégation**

Terrestre.java

```
public interface Terrestre {  public void move (); };
```

TerrestreClasse.java

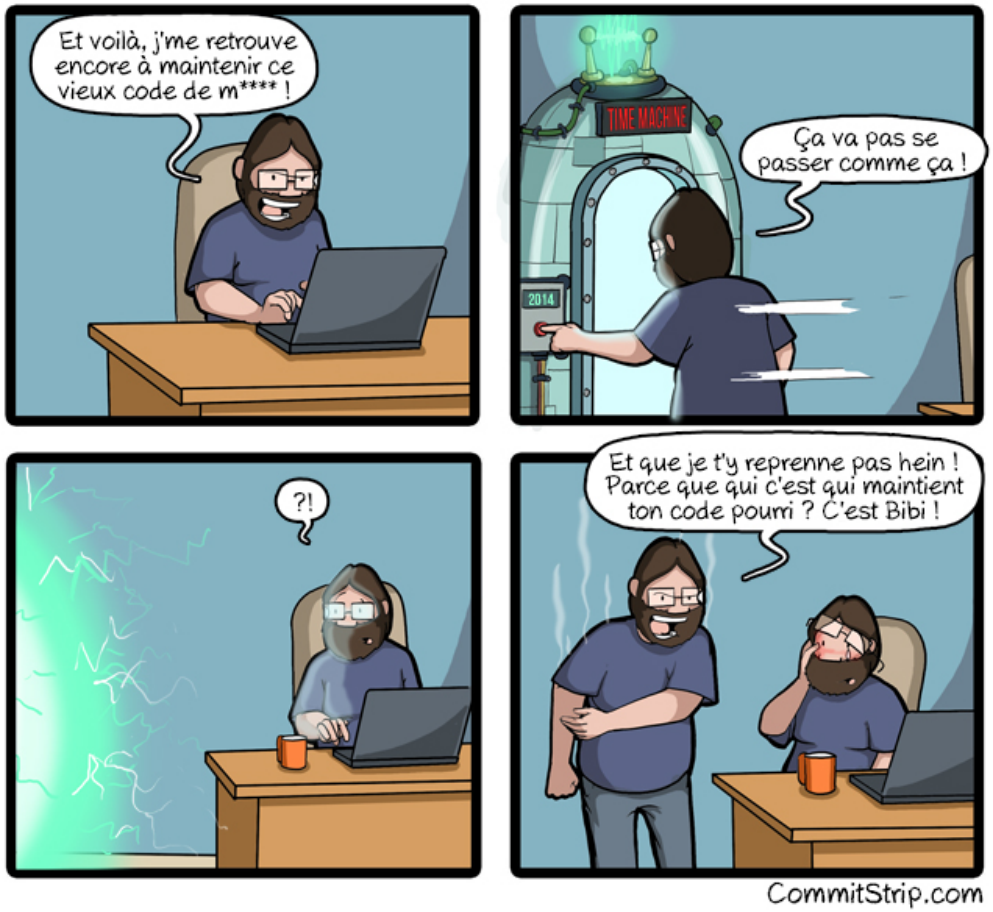
```
public class TerrestreClasse implements Terrestre {
//TerrestreClasse doit definir la methode move()
  public void move () {...};    }
```

Amphibie.java

```
    public class Amphibie extends Aquatique implements Terrestre {
     private TerrestreClasse c;     ...
     public void move() {c.move();}   }
```

Plan

Conception logicielle



Vie d'un source ... par exemple pendant votre dernier TP ...

- joli, "beau" ... une première "hérésie" ... de plus en plus d'horreurs ...
- Conséquences : de moins en moins **maintenable** et **évolutif**, effet "spaghetti"

Conception logicielle

- Les dégradations du design sont liées aux **modifications** des spécifications (ajout d'une nouvelle fonctionnalité, modification d'une partie du code)

Signes d'une mauvaise conception

- Rigidité : difficile d'ajouter une nouvelle fonctionnalité
- Fragilité : modification d'une partie du code → problème imprévisible dans un autre endroit

Signes d'une bonne conception

- Extensibilité : facile d'ajouter une nouvelle fonctionnalité
- Souplesse : permettre les changements, l'ajout de nouvelles fonctionnalités mais sans modification du code existant
- Flexibilité : modification d'une partie du code → peu ou pas d'effets sur les autres parties

Conception logicielle

- Les dégradations du design sont liées aux **modifications** des spécifications (ajout d'une nouvelle fonctionnalité, modification d'une partie du code)

Signes d'une mauvaise conception

- Rigidité : difficile d'ajouter une nouvelle fonctionnalité
- Fragilité : modification d'une partie du code $\rightarrow$ problème imprévisible dans un autre endroit

Signes d'une bonne conception

- Extensibilité : facile d'ajouter une nouvelle fonctionnalité
- Souplesse : permettre les changements, l'ajout de nouvelles fonctionnalités mais sans modification du code existant
- Flexibilité : modification d'une partie du code $\rightarrow$ peu ou pas d'effets sur les autres parties

Conception logicielle

- Les modifications sont à "faire vite" et par d'autres que les designers originaux
- Les dégradations dans le code sont dues aux dépendances entre les parties du code

Signes d'une mauvaise conception

- Immobilité : impossible ou difficile de réutiliser des parties

Signes d'une bonne conception

- Modularité : indépendance des sous-systèmes

→ **la conception doit permettre et anticiper les modifications**

Opérations pour améliorer un code existant

Reverse engineering ou retro-conception

Construire automatiquement une partie du modèle *i.e.* des diagrammes UML à partir du code d'une application (Java, C++, ...)
Objectifs :

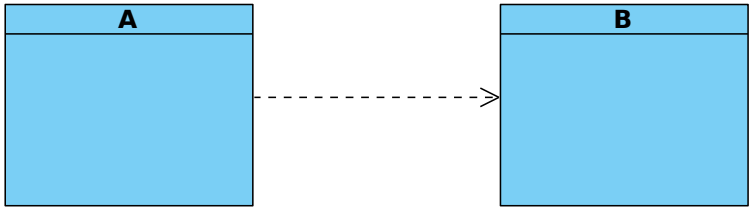
- établir un lien entre le code et le modèle d'une application,
- générer une documentation technique d'une application existante,
- comprendre le fonctionnement de l'application, analyser
- *refactoring* ou re-usinage

Refactoring ou re-usinage

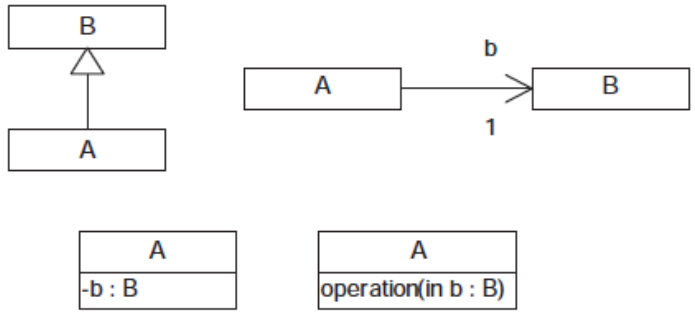
Retravailler le code source d'une application informatique sans ajouter de fonctionnalités mais en changeant sa structure interne pour améliorer sa lisibilité, souplesse, extensibilité, ...

Plan

Dépendance



Représentation graphique d'une dépendance (A dépend de B)



Causes d'une relation de dépendance entre deux classes

Dépendance

A dépend de B si le fonctionnement de l'élément A requiert la présence de l'élément B.

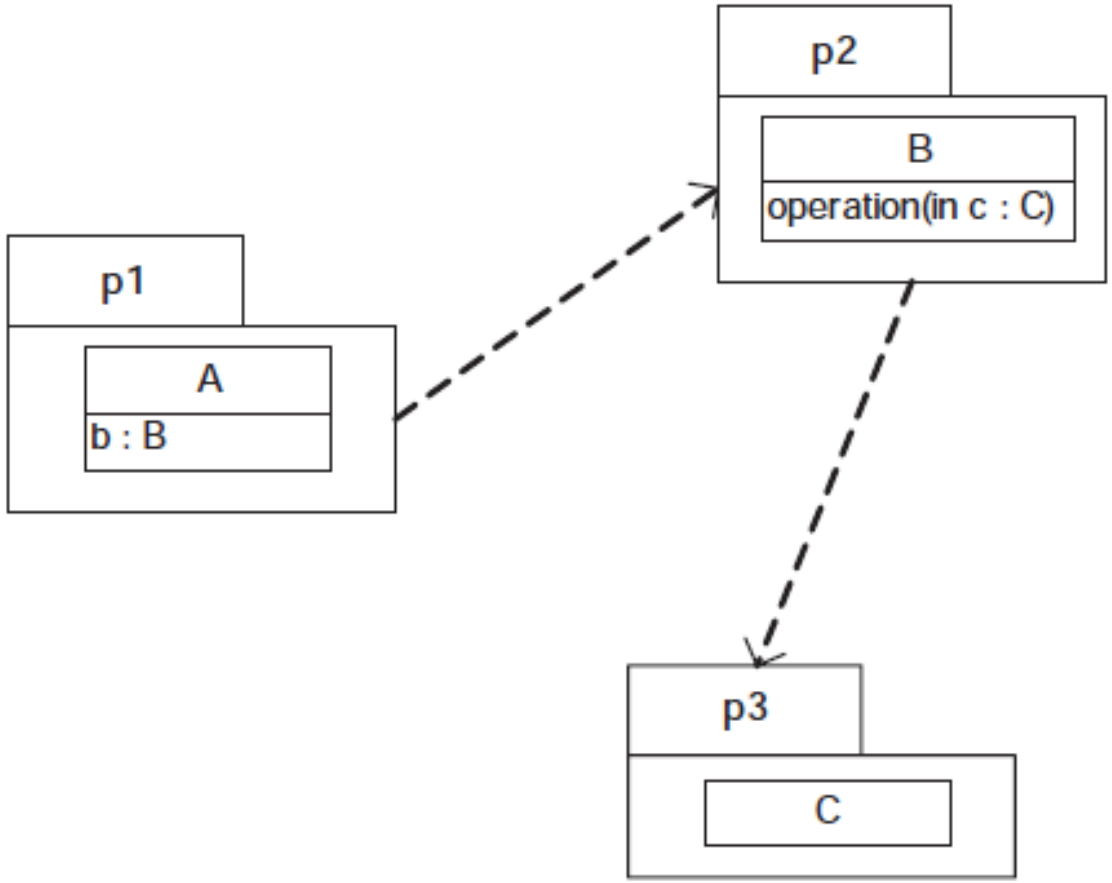
Conséquences :

- tout changement apporté à la partie publique de B peut aussi impacter A.
- A ne peut pas être utilisé dans un contexte autre que celui de B.

Dépendance entre package

Dépendances entre packages

Il y a une dépendance entre deux packages s'il existe des dépendances entre les classes qu'ils contiennent.



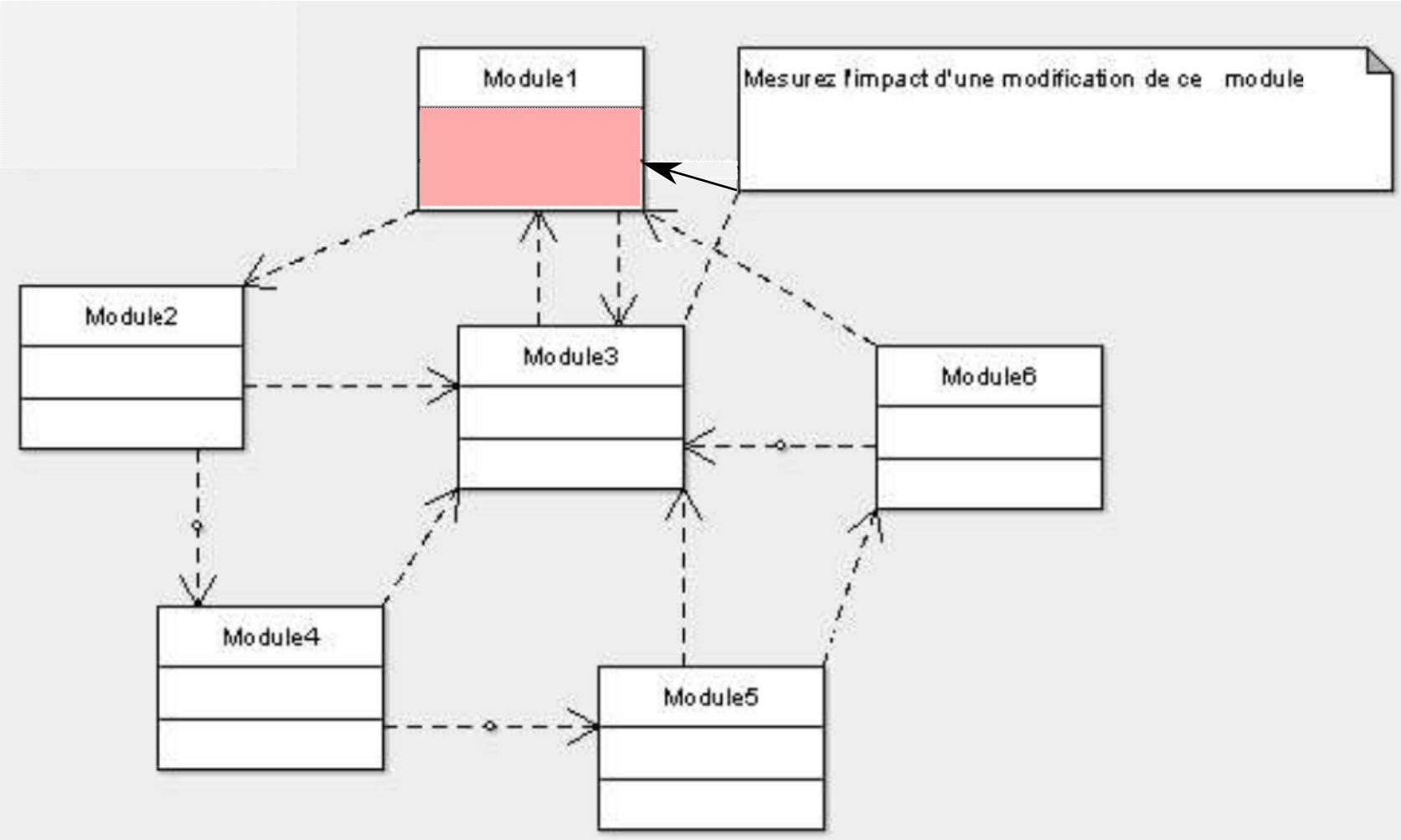
Couplage et dépendances

Couplage

- Mesure la quantité de dépendances qu'une entité (module, classe, package, composant) entretient avec un ou plusieurs éléments.
- Une entité est couplée à une autre si elle **dépend** d'elle.
- Mesure le degré d'interaction entre les entités dans le système.

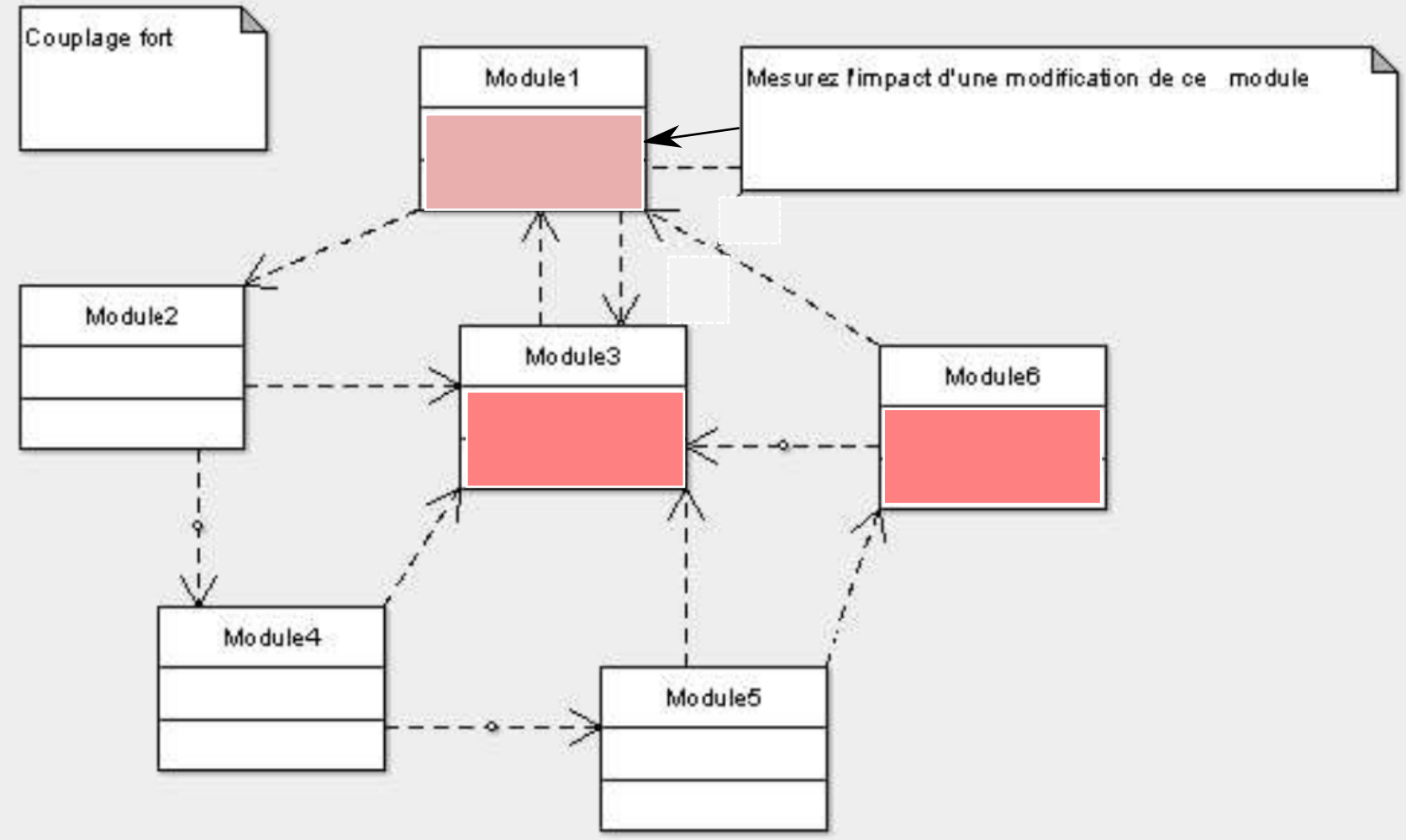
- Un couplage fort rend l'application plus **rigide** à toute modification de code.
- Un couplage faible permet une grande **souplesse** de programmation et de mise à jour

Couplage : Exemple

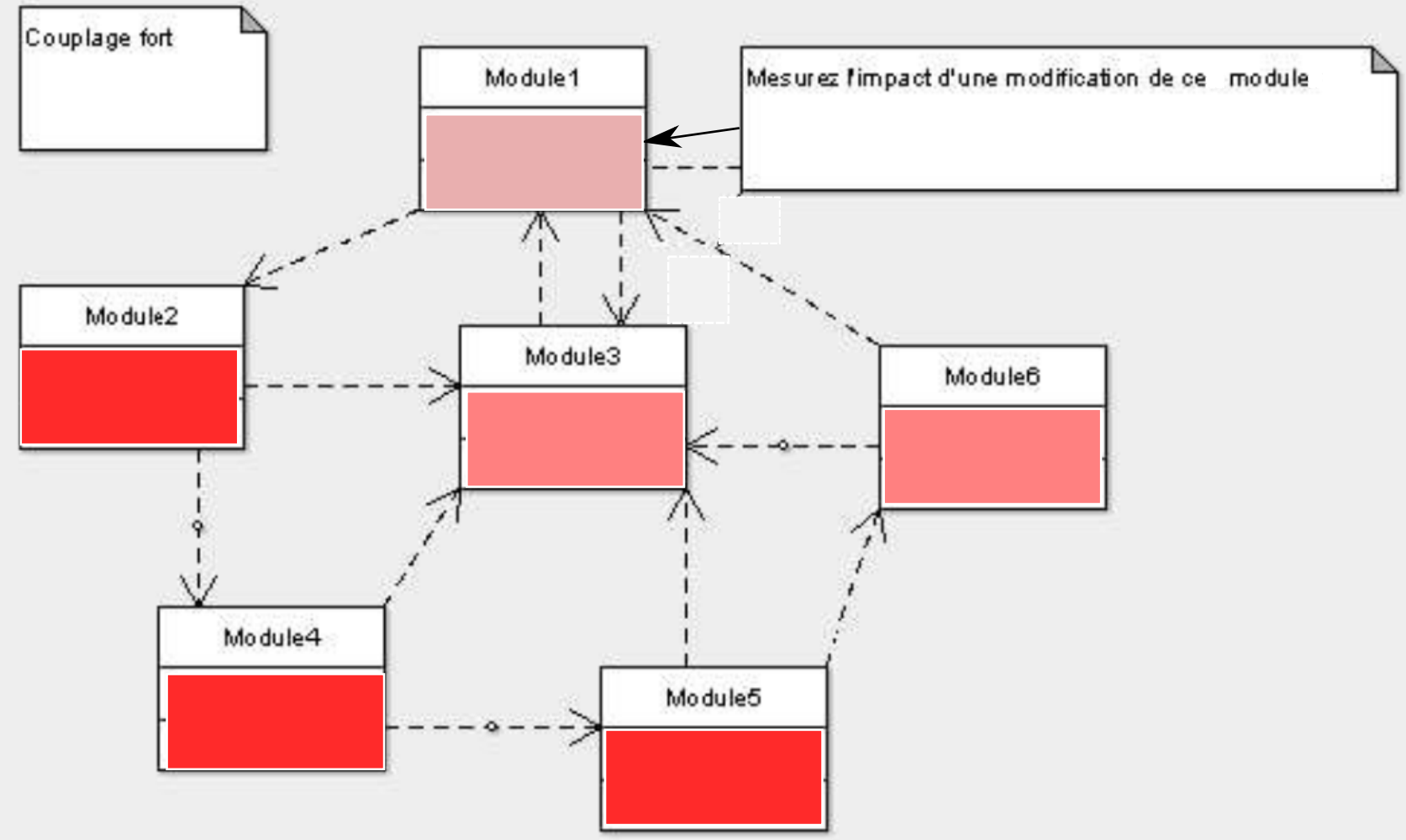


Couplage faible ou fort ?

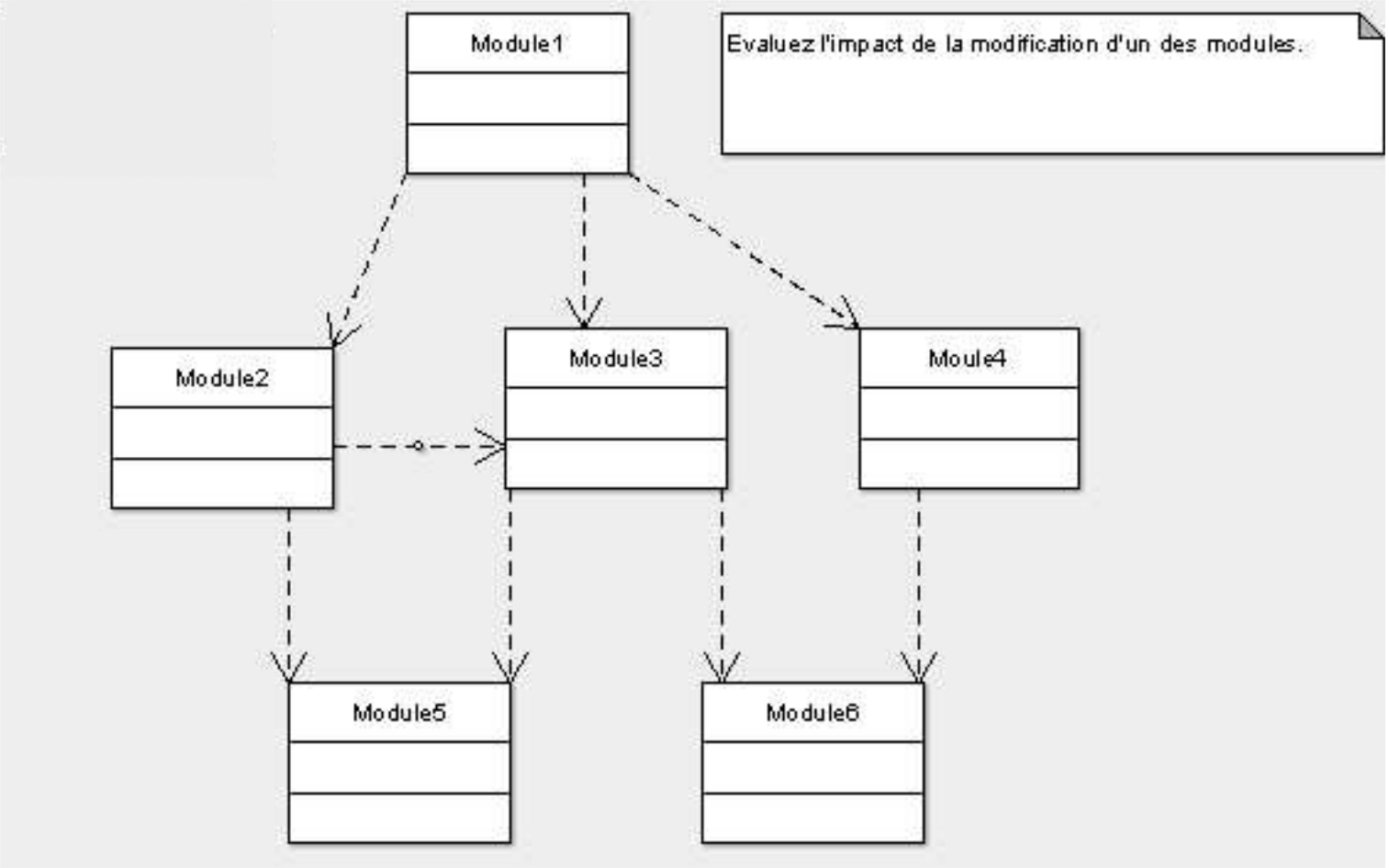
Couplage : Exemple



Couplage : Exemple



Couplage : Exemple



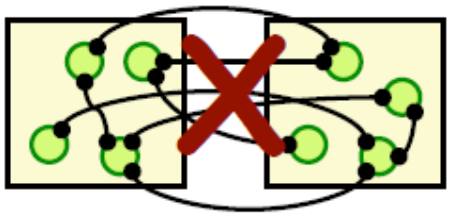
Couplage faible ou fort ?

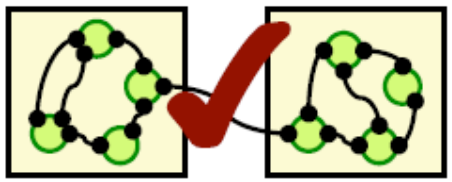
Couplage

Le principe du **faible couplage** est une maîtrise des dépendances qui facilite la réutilisation et augmente la souplesse. Surtout éviter un couplage fort à des éléments instables.

Modularité ou Indépendance

- technique de décomposition de systèmes
- un module ou sous-système présente un couplage avec les autres relativement faible (couplage faible entre module) par rapport au couplage de ses propres parties (cohésion interne forte)





Plan

Encapsulation

Encapsulation

- Séparer l'interface (partie publique) d'un module de son implémentation (partie privée)
  $\rightarrow$ L'implémentation (privée) peut évoluer sans conséquences sur les autres modules

Encapsulation

Encapsuler les parties qui peuvent varier indépendamment des parties stables.

- Technique pour favoriser la modularité des sous-systèmes et ajouter de la souplesse à l'application

Encapsulation : Exemple

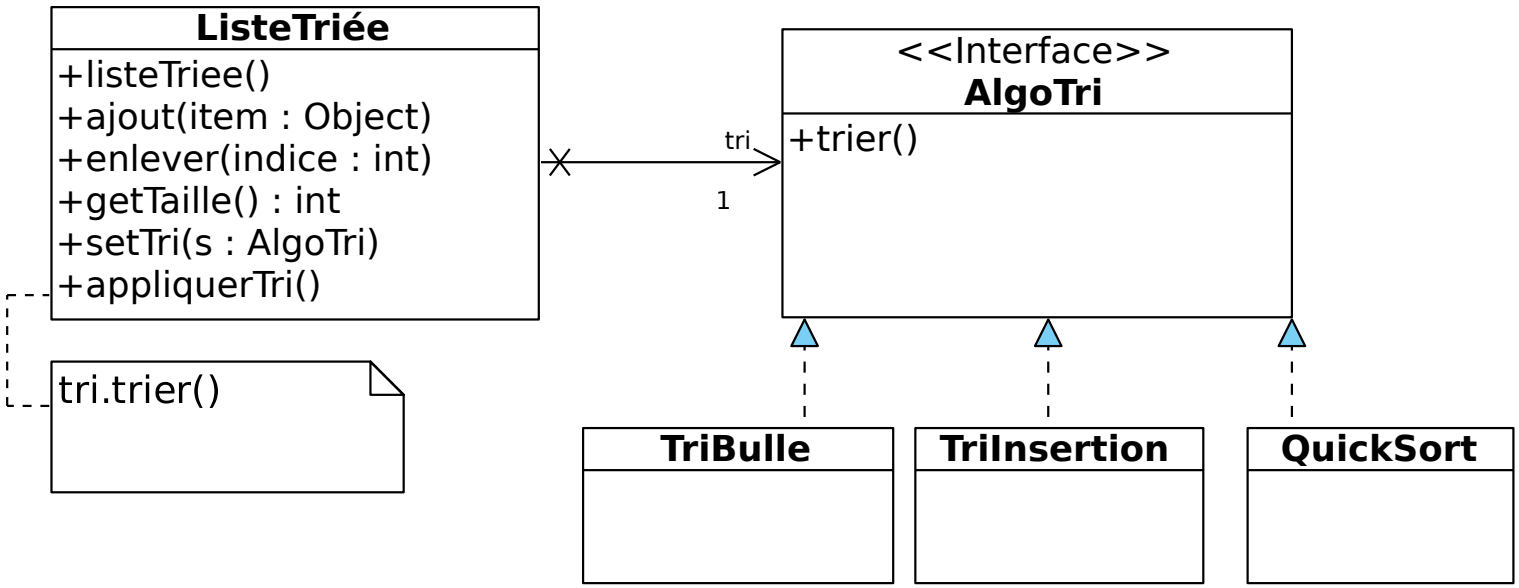
ListeTriée
+listeTriee() +ajout(item : Object) +enlever(indice : int) +getTaille() : int +trier()

```
public void trier(){
    triBulle();
}
```

La méthode de tri risque d'évoluer :

```
public void trier(){
    if (getTaille()<x) triBulle();
    else if (getTaille()<y) triInsertion();
    else quicksort();
}
```

Encapsulation : Exemple



Encapsuler la méthode de tri (instable) pour la séparer des autres méthodes sur la liste (stables)

Plan

Polymorphisme

- généricité : une même opération peut s'appliquer à des objets de classes différentes et avoir un comportement adapté à ces objets
- capacité à exécuter une méthode en fonction du type réel (ou type dynamique) de l'objet concerné par l'appel, et non en fonction du type déclaré (ou type statique).
- 2 types de polymorphisme :
  - polymorphisme de traitement : surcharge des méthodes
  - polymorphisme de données
    - polymorphisme d'héritage (redéfinition, sous-typage, spécialisation ...) : un même code peut être appliqué à des données de types différents liées entre elles par une relation d'héritage
    - polymorphisme paramétrique (généricité, *template)* : un même code peut être appliqué à n'importe quel type

Polymorphisme

- généricité : une même opération peut s'appliquer à des objets de classes différentes et avoir un comportement adapté à ces objets
- capacité à exécuter une méthode en fonction du type réel (ou type dynamique) de l'objet concerné par l'appel, et non en fonction du type déclaré (ou type statique).
- 2 types de polymorphisme :
  - polymorphisme de traitement : surcharge des méthodes
  - polymorphisme de données
    - polymorphisme d'héritage (redéfinition, sous-typage, spécialisation ...) : un même code peut être appliqué à des données de types différents liées entre elles par une relation d'héritage
    - polymorphisme paramétrique (généricité, *template)* : un même code peut être appliqué à n'importe quel type

Polymorphisme de traitement : surcharge de méthodes

- Définir deux méthodes ayant le même nom, mais pas la même signature (type et/ou nombre d'arguments différents)
- Le **compilateur** choisit la méthode à utiliser en fonction du type des paramètres

```
private static int addition(int x, int y)
{
  System.out.println("additionne des int");
  return x + y;
}

private static float addition(float x, float y)
{
   System.out.println("additionne des float");
   return x + y;
}
```

Polymorphisme d'héritage : redéfinition de méthodes

- Possibilité de définir le comportement d'une méthode selon le type d'objet l'invoquant
- Méthode redéfinie donne une nouvelle implémentation à une méthode héritée sans changer sa signature
- Une méthode redéfinie peut être complètement différente de la méthode de base, ou bien réutiliser celle-ci en effectuant des opérations supplémentaires (spécialisation) : `super` en Java, `::` en C++

Polymorphisme d'héritage : redéfinition de méthodes

Point.java

```
public class Point {
 private int x, y ;// Attributs de Point
 ...
 public Point(){...};
 public void affiche ( ) {
   System.out.println(x+" ,"+y);
 }
}
```

PointCouleur.java

```
public class PointCouleur extends Point {
 private int couleur ;// Attributs supplementaires
 ...
 @Override
 public void affiche () {
   super.affiche() ;
   System.out.println("Couleur:" + this.couleur);
 }
}
```

Polymorphisme d'héritage : redéfinition de méthodes

Point.h

```
class Point {
 private:
  int x,y ;
 public:
  Point() {...}
  void affiche () {
    std::cout<<x<<" "<<y<<std::endl;}
  ... };
```

PointCouleur.h

```
#include "Point.h"
class PointCouleur : public Point {
  private:
    ... // Attributs sup
  public:
   PointCouleur() {...}
   void affiche ()
   { Point::affiche(); // Appel a affiche() de Point
     std::cout<<"Couleur:"<<this->couleur;
   }
};
```

Polymorphisme d'héritage : méthode virtuelle

- Le type réel (type dynamique) de l'instance détermine la méthode à exécuter(**résolution dynamique de liens**)

```
Point p;
if (...)
//upcast
  p = new PointCouleur();
else
  p = new Point();
p.affiche(); //???
```

```
Point *pp;
if (...)
  pp = new PointCouleur();
else
  pp = new Point();
pp->affiche();   //???
```

Polymorphisme d'héritage : méthode virtuelle

- Le type réel (type dynamique) de l'instance détermine la méthode à exécuter(**résolution dynamique de liens**)

```
Point p;
if (...)
//upcast
  p = new PointCouleur();
else
  p = new Point();
p.affiche(); //???
```

```
Point *pp;
if (...)
  pp = new PointCouleur();
else
  pp = new Point();
pp->affiche();  //???
```

  - Méthode **virtuelle** est destinée à être redéfinie dans une classe dérivée
  - En Java, les méthodes sont virtuelles par défaut. `p.affiche()` fait appel à la méthode de la classe réellement instanciée avec `new` même si `p` a été déclaré de type `Point`.
  - En C++, les méthodes ne sont pas virtuelles par défaut. `pp->affiche()` appelle toujours la méthode de `Point` (résolution statique des liens). Pour y remédier, la méthode doit être déclarée `virtual` dans `Point`.

Polymorphisme d'héritage : méthode virtuelle pure, classe abstraite

- Une méthode **virtuelle pure** ou **abstraite** est déclarée sans être définie, et est donc destinée à être obligatoirement (re)définie dans une classe dérivée (`abstract` en Java, =0 en C++)
- Une classe est dite **abstraite** si elle contient au moins une méthode virtuelle pure (`abstract` en Java)
  - Sert à définir des concepts incomplets qui seront complétés dans les sous-classes.
  - Ne peut pas être instanciée, et est donc destinée à être dérivée.

```
public abstract class Point {
 private int x, y ;
 ...
 public Point(){...};
 public abstract void affiche
      ( );
}
```

```
class Point {
  private:
   int x,y ;
  public:
   Point() {...}
   virtual void affiche () = 0 ;
```

Plan

Principes avancés de conception orientée objet

- principes fondamentaux de COO : insuffisant pour guider la conception
- principes avancés de COO : préceptes globaux visant à faciliter la conception générale de systèmes plus souples, plus faciles à maintenir et à étendre.
- *design patterns* : application de ces principes à des problèmes ponctuels (cf. prochain cours)

Principes avancés de conception orientée objet

Principes de conception SOLID

- 5 principes de bases pour la POO identifiés par Robert C. Martin "Uncle Bob" (`http://www.cleancoders.com/`)
  - Principe de Responsabilité Unique - Single Responsibility Principle (SRP)
  - Principe d'ouverture/fermeture - Open-Closed Principle (OCP)
  - Principe de substitution de Liskov - Liskov Substitution Principle (LSP)
  - Principe de Séparation des Interfaces - Interface Segregation Principle (ISP)
  - Principe d'inversion des dépendances - Dependency Inversion Principle (DIP)

crédits image http ://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Principe de Responsabilité Unique

Principe de Responsabilité Unique

Une classe ne doit avoir qu'une seule raison de changer.

- Pour cela, une classe doit se focaliser sur un seul objectif fonctionnel.
- Fortement lié à l'idée de cohésion interne forte pour un module.

crédits image http ://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Exercice : Principe de Responsabilité Unique

```
public class Rectangle{
  //Coordonnees des points en haut a gauche et en bas a droite
  private double x1, y1, x2, y2;

  public Rectangle(double x1,double y1,double x2,double y2){
     this.x1 = x1; this.x2 = x2; this.y1 = y1; this.y2 = y2;
  }
  public double getX() { return x1; }
  public double getY() { return y1; }
  public double getLargeur() { return x2-x1; }
  public double getHauteur() { return y2-y1; }
  public double aire() { return (x2-x1) * (y2-y1); }
  public void dessiner(Graphics g){
     g.drawRect(x1, y1, x2-x1, y2-y1);
  }
}
```

Qu'implique un changement de représentation en `x, y, largeur, hauteur` ?

Exercice : Principe de Responsabilité Unique

```
public class Rectangle{
  //Coordonnees des points en haut a gauche et en bas a droite
  private double x1, y1, x2, y2;

  public Rectangle(double x1,double y1,double x2,double y2){
     this.x1 = x1; this.x2 = x2; this.y1 = y1; this.y2 = y2;
  }
  public double getX() { return x1; }
  public double getY() { return y1; }
  public double getLargeur() { return x2-x1; }
  public double getHauteur() { return y2-y1; }
  public double aire() { return (x2-x1) * (y2-y1); }
  public void dessiner(Graphics g){
     g.drawRect(x1, y1, x2-x1, y2-y1);
  }
}
```

Appliquer le principe de responsabilité unique pour limiter l'impact des changements.

Exercice : Principe de Responsabilité Unique

```
public class RectangleGeometrique{
    //Coordonnees des points en haut a gauche et en bas a droite
  private double x1, y1, x2, y2;

  public Rectangle(double x1,double y1,double x2,double y2){
     this.x1 = x1; this.x2 = x2; this.y1 = y1; this.y2 = y2;
  }
  public double getX() { return x1; }
  public double getY() { return y1; }
  public double getLargeur() { return x2-x1; }
  public double getHauteur() { return y2-y1; }

}
public class RectangleGraphique{
  private RectangleGeometrique rect;
  public RectangleGraphique(RectangleGeometrique r){ this.rect = r;}
  public double aire() { return rect.getLargeur() * rect.getHauteur
      (); }

  public void dessiner(Graphics g){
     g.drawRect(rect.getX(), rect.getY(), rect.getLargeur(), rect.
         getHauteur());
  }
}
```

Principe "open-closed"

Principe "open-closed" (Bertrand Meyer)

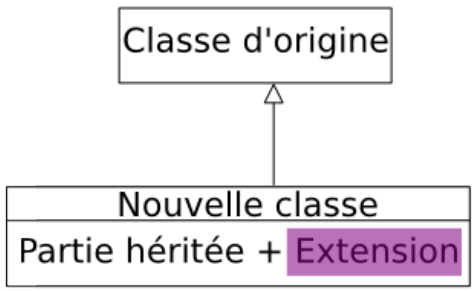
Les entités informatiques (paquetage, classe, méthode) doivent être ouvertes aux extensions mais fermées aux modifications.

crédits image http ://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

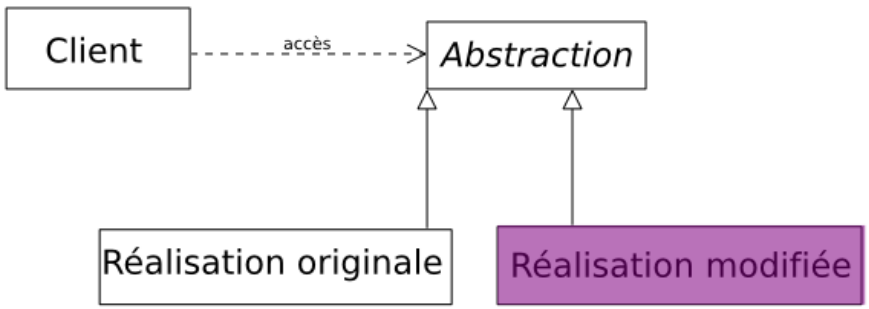
Principe "open-closed"

Principe "open-closed" (Robert C. Martin)

- Ouverture aux extensions :
  - le comportement d'une classe doit pouvoir être étendu
  - exemple : ajout de nouvelles méthodes
- Fermeture aux modifications :
  - le comportement d'une classe doit pouvoir être étendu mais sans modification de son code source.
  - exemple : mise en place d'une interface ou héritage
- En d'autres termes, l'ajout de fonctionnalités doit se faire en ajoutant du code et non en éditant du code existant.



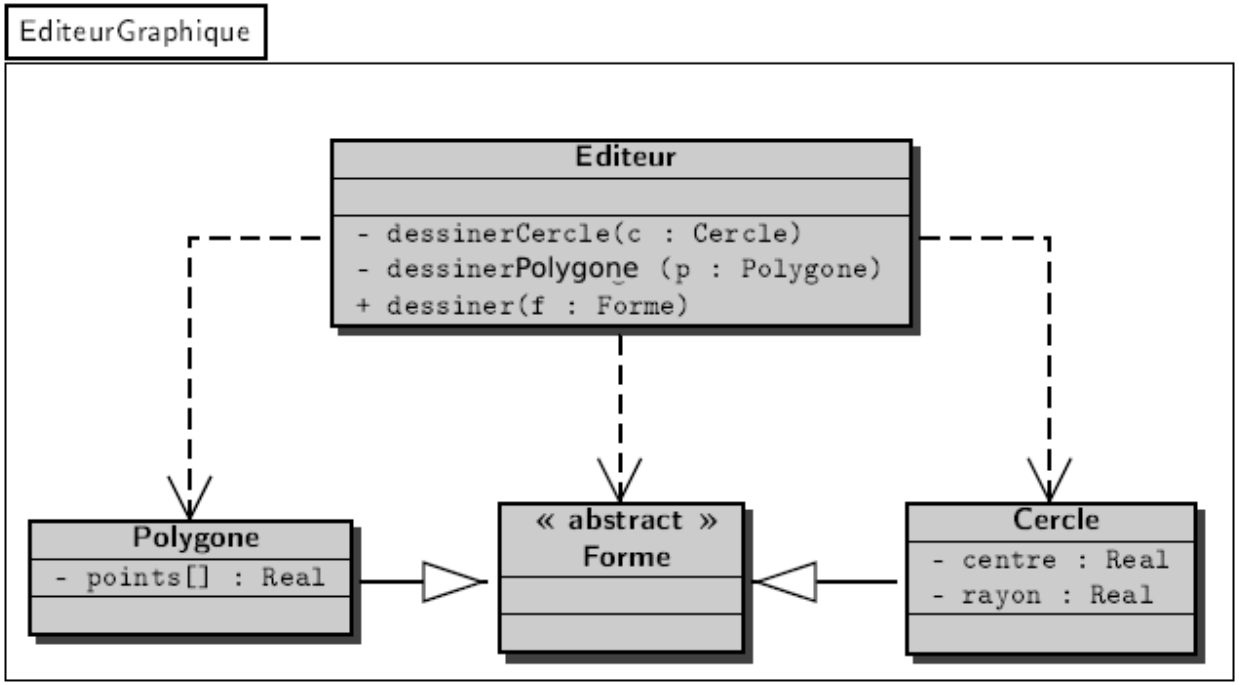
Héritage d'implantation



Héritage d'abstraction

Deux approches possibles pour la mise en oeuvre de ce principe concernant les classes.

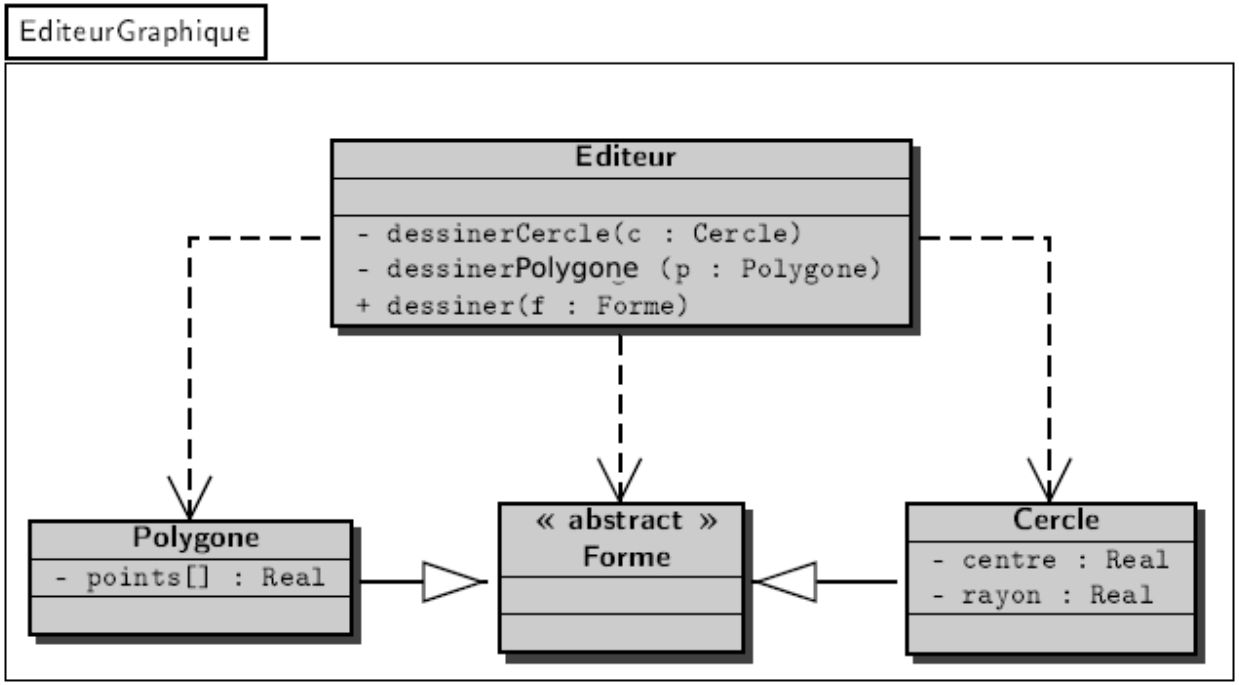
Exemple (avec une erreur de conception)




```
void dessiner(Forme f){
    if (f instanceof Cercle) dessinerCercle((Cercle)f);
    else if (f instanceof Polygone) dessinerPolygone((Polygone)f);
}
```

Le principe "open-closed" est-il respecté ici ? (on doit pouvoir facilement ajouter des formes à l'application sans modifier les classes existantes)

Exemple (avec une erreur de conception)

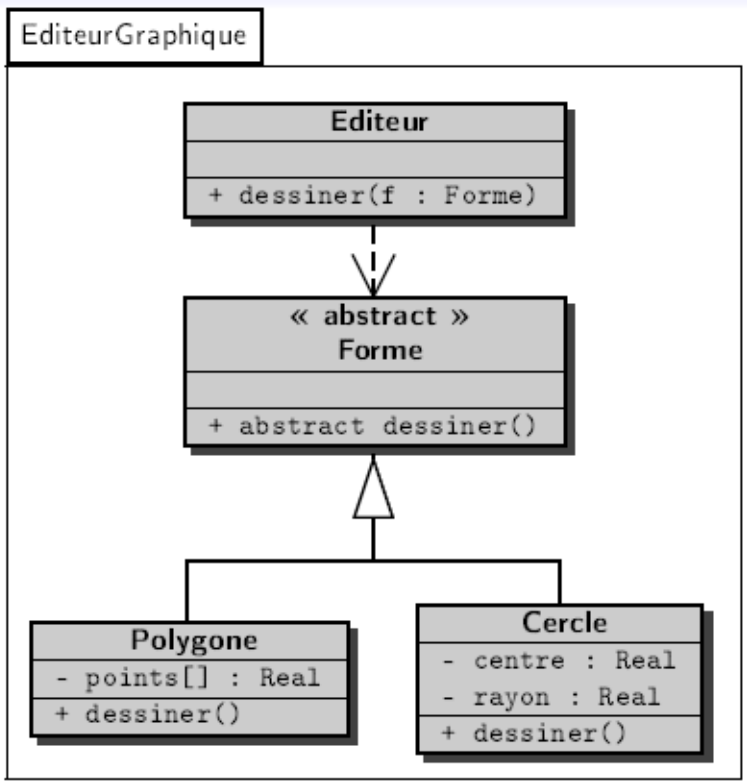



```
void dessiner(Forme f){
    if (f instanceof Cercle) dessinerCercle((Cercle)f);
    else if (f instanceof Polygone) dessinerPolygone((Polygone)f);
}
```

Le principe "open-closed" est-il respecté ici ? (on doit pouvoir facilement ajouter des formes à l'application sans modifier les classes existantes)

Principe "open-closed" non respecté : `Editeur` non fermée aux modifications. Proposer une solution respectant le principe OCP.

Exemple (corrigé)




```
void dessiner(Forme f){
   f.dessiner(); }
```

Principe "open-closed" respecté : `Editeur` est fermé aux modifications car la méthode `dessiner()` ne change pas si l'on ajoute une nouvelle `Forme`. `Editeur` est ouverte aux extensions : toutes les sous-classes de `Forme` peuvent changer le comportement de `dessiner()` .

Principe "open-closed"

A l'origine de plusieurs bonnes pratiques de développement objet :

- Encapsulation :
  - toute fonction qui dépend d'un membre ne sera pas fermée par rapport à ce membre : les membres d'une classe doivent être privés.
  - encapsuler les parties qui peuvent varier indépendamment des parties stables.
- Un certain nombre de *design patterns* sont une mise en pratique de ce principe :
  - `Décorateur` permet de rajouter à un objet de nouvelles fonctionnalités sans modifier son code, code cible fermé à l'ajout de nouvelles fonctionnalités ;
  - `Strategie` permet de modifier dynamiquement l'implantation d'un comportement sans modifier ses interactions avec les clients qui l'utilisent, client fermé aux changements/ajouts d'algorithmes ;
  - `Fabrique` permet de créer des objets sans savoir leur type précis, client fermé à la création de nouveaux objets
  - `Visiteur` permet d'étendre les capacités de la collection parcourue sans modifier l'implémentation des objets manipulés, structure visitée fermée à l'ajout de nouveaux algorithmes.

Principe de substitution de Liskov

Principe de substitution de Liskov (Barbara Liskov, 1988)

- Une instance d'une classe doit pouvoir être substituée par une instance d'une sous-classe sans que la compilation ne soit altérée ni que le programme ne soit altéré dans son comportement.
- Les méthodes qui utilisent les objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir.

crédits image http ://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Principe de substitution de Liskov

Principe de substitution de Liskov (Barbara Liskov, 1988)

- **Une instance d'une classe doit pouvoir être substituée par une instance d'une sous-classe** sans que la compilation ne soit altérée ni que le programme ne soit altéré dans son comportement.
- Les méthodes qui utilisent les objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir.

Autrement dit :

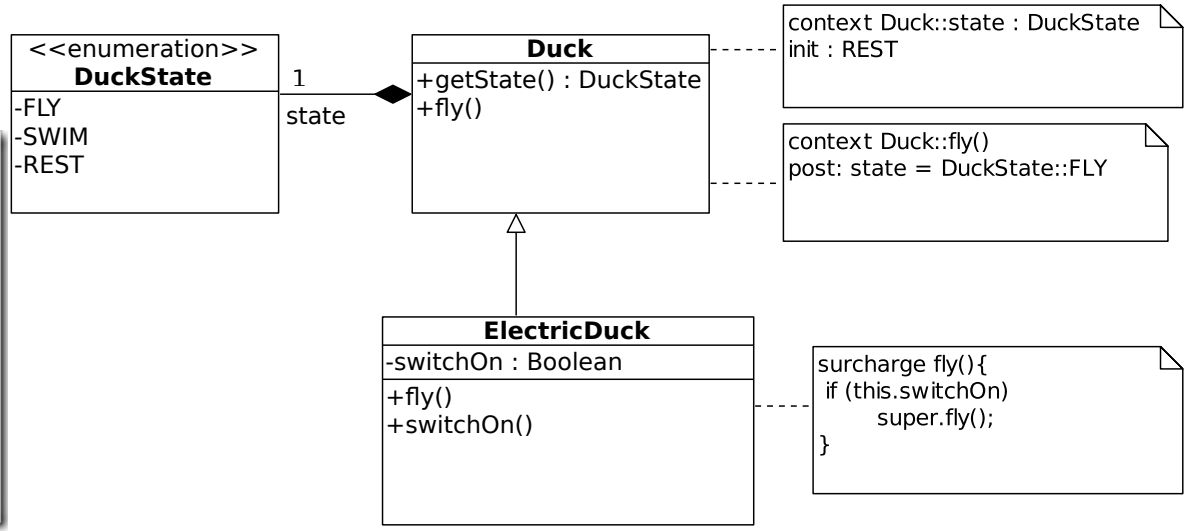
Si une classe B hérite d'une classe A, tout programme écrit pour traiter des instances de A doit pouvoir traiter des instances de B sans même savoir que ce ne sont pas des instances directes de A.

Conséquences

Ne pas introduire de modifications dans le fonctionnement de B qui rend inutilisable tout objet de type B utilisé comme un objet de type A.

Exemple

exemple inspiré de http ://blogs.developpeur.org/fathi/archive/2011/12/09/isp-liskov-substitution-principle.aspx

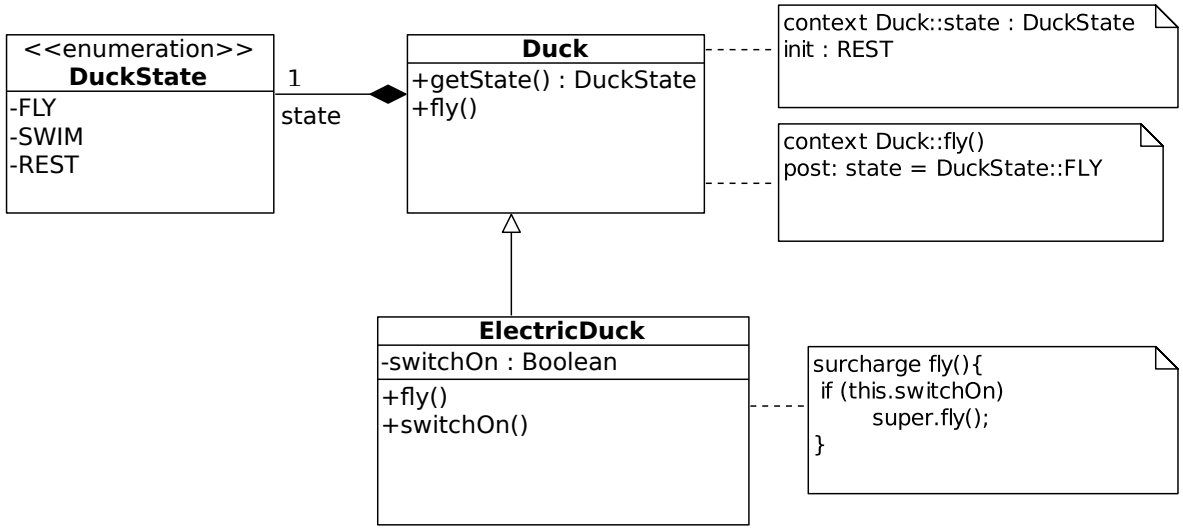



```
public class Test {
 public void test(List<Duck> myList ) {
   int nbFlyingDucks = 0;
   for (Duck duck : myList) {
     duck.fly();//on fait voler les canards
     if (duck.getState() == DuckState.FLY)       nbFlyingDucks++;   }
   if(nbFlyingDucks != myList.size()) //on verifie qu ils sont tous en train de voler
     throw new Exception("Some ducks not flying");}
 public static void main(String[] args){
   List<Duck> myList = new ArrayList<Duck>();
   myList.add(new Duck()); myList.add(new Duck());
   new Test().test(myList));
   myList.add(new ElectricDuck());
   new Test().test(myList));  }}
```

Le principe de Liskov est-il respecté ? (peut-on utiliser des instances de la classe dérivée `ElectricDuck` à la place d'instances de la classe `Duck` sans que le programme ne soit altéré dans son comportement ?)

Exemple

exemple inspiré de http ://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx

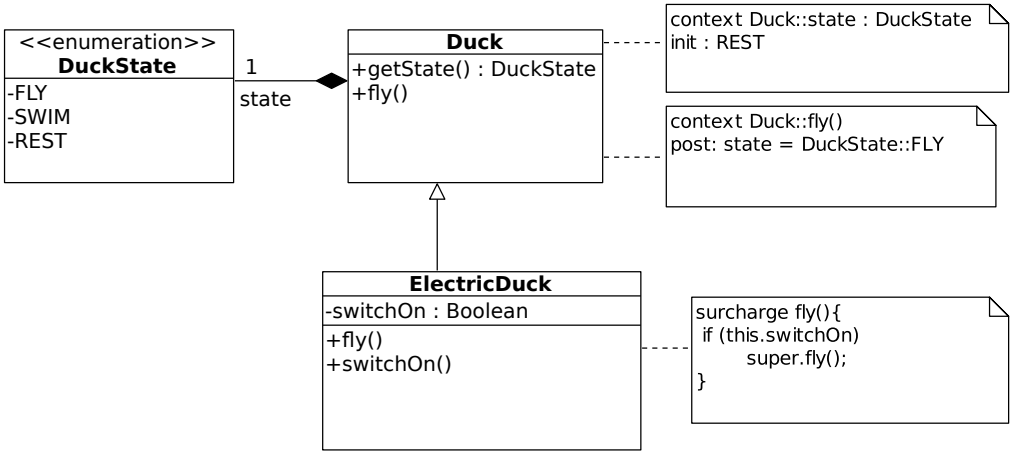



```
public class Test {
 public void test(List<Duck> myList ) {
   int nbFlyingDucks = 0;
   for (Duck duck : myList) {
     duck.fly();//on fait voler les canards
     if (duck.getState() == DuckState.FLY)       nbFlyingDucks++;   }
   if(nbFlyingDucks != myList.size()) //on verifie qu ils sont tous en train de voler
     throw new Exception("Some ducks not flying");}
 public static void main(String[] args){
   List<Duck> myList = new ArratList<Duck>();
   myList.add(new Duck()); myList.add(new Duck());
   new Test().test(myList));//OK
   myList.add(new ElectricDuck());
   new Test().test(myList));  }}
   //Exception levee: le canard electrique n a pas pu decoller car non allume
```

Principe de Liskov non respecté : la méthode `Test` qui utilise des instances de la classe `Duck` doit pouvoir utiliser des instances de la classe dérivée `ElectricDuck` sans que le programme ne soit altéré dans son comportement.

Exemple

exemple inspiré de http ://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx



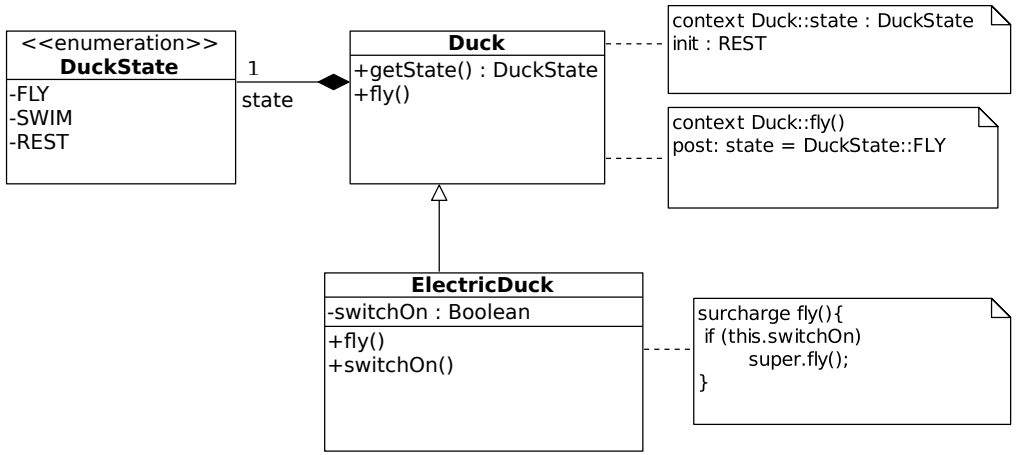
Solution :

```
public class Test{
 public void test(List<Duck> myList ) {
   int nbFlyingDucks = 0;
   for (Duck duck : myList) {
     if (duck instanceof ElectricDuck) { ((ElectricDuck) duck).switchOn();}
     duck.fly();
     if (duck.getState() == DuckState.FLY)  nbFlyingDucks++; }
   if(nbFlyingDucks != myList.size()) {
     throw new Exception("Some ducks not flying"); } }
```

Que pensez-vous de cette solution ?

Exemple

exemple inspiré de http ://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx



Solution :

```
public class Test{
 public void test(List<Duck> myList ) {
   int nbFlyingDucks = 0;
   for (Duck duck : myList) {
     if (duck instanceof ElectricDuck) { ((ElectricDuck) duck).switchOn();}
     duck.fly();
     if (duck.getState() == DuckState.FLY)  nbFlyingDucks++; }
   if(nbFlyingDucks != myList.size()) {
      throw new Exception("Some ducks not flying"); } }
```

La méthode `Test` a besoin de connaître le type réel des objets pour pouvoir les traiter correctement : méthode non-fermée aux modifications → principe *open closed* non respecté → `Duck` et `ElectricDuck` n'ont sans doute rien en commun ... à moins peut-être une interface `canFly` ?

Principe de substitution de Liskov

- Principe basé sur une bonne utilisation de l'héritage. Il révèle les problèmes cachés par une mauvaise utilisation de l'héritage.
- Utiliser l'héritage peut violer ce principe de conception.
- L'héritage doit être utilisé pour étendre une classe mère par ajout de fonctionnalités mais sans modification de fonctionnalités existantes.
- L'héritage ne doit pas être utilisée uniquement pour mettre du code en commun.

Principe de substitution de Liskov

- Principe basé sur une bonne utilisation de l'héritage. Il révèle les problèmes cachés par une mauvaise utilisation de l'héritage.
- Utiliser l'héritage peut violer ce principe de conception.
- L'héritage doit être utilisé pour étendre une classe mère par ajout de fonctionnalités mais sans modification de fonctionnalités existantes.
- L'héritage ne doit pas être utilisée uniquement pour mettre du code en commun.

- Principe fortement lié au principe ouvert-fermé.
- La redéfinition d'un comportement dans une sous-classe contredit le "contrat" qui caractérise la classe dont on dérive. La super-classe devrait alors être modifiée mais cela violerait principe ouvert-fermé.
- La conception par contrats précise cette notion de contrat et reformule LSP.

Conception par contrats (*Design by contract*)

Conception par contrats (Bertrand Meyer)

- Paradigme de programmation dans lequel des contrats définissent les conditions correctes d'utilisation des objets (spécifications formelles (OCL)).

Contraintes, assertions ou contrats :

1. préconditions : conditions d'utilisation de la méthode
2. postconditions : propriétés toujours vraies une fois la méthode exécutée
3. invariants : propriétés que toute instance d'une classe doit respecter

Conception par contrats (*Design by contract*)

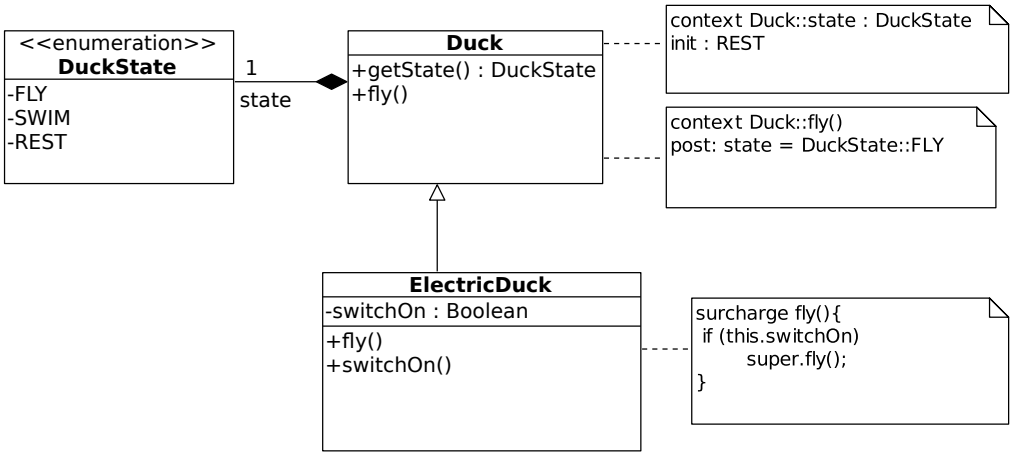
Principe de substitution de Liskov et conception par contrats

Toutes les clauses du contrat satisfaites par les sur-classes doivent être satisfaites par les sous-classes

Conséquences :

1. les préconditions ne peuvent pas être renforcées dans une sous-classe
2. les postconditions ne peuvent pas être affaiblie dans une sous-classe
3. les invariants de la sur-classe doivent être strictement vérifiés par la sous-classe

exemple inspiré de http ://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx



Conception par contrats (*Design by contract*)

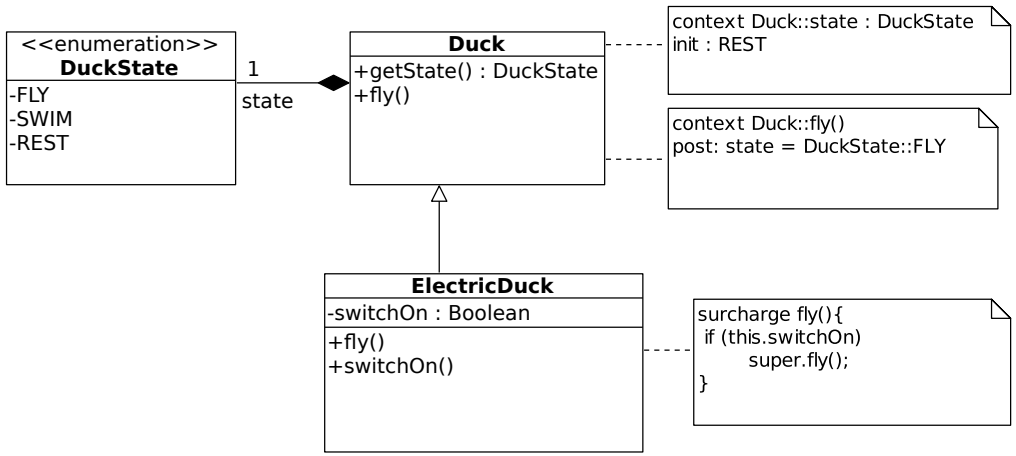
Principe de substitution de Liskov et conception par contrats

Toutes les clauses du contrat satisfaites par les sur-classes doivent être satisfaites par les sous-classes

Conséquences :

1. les préconditions ne peuvent pas être renforcées dans une sous-classe
2. les postconditions ne peuvent pas être affaiblie dans une sous-classe
3. les invariants de la sur-classe doivent être strictement vérifiés par la sous-classe

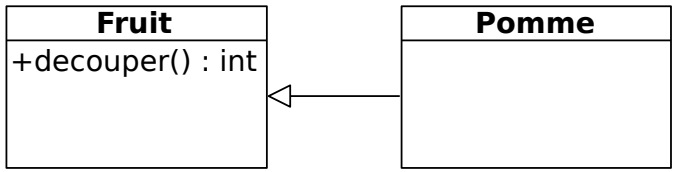
exemple inspiré de http ://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx



- Les postconditions de la classe `Duck` sont affaiblies dans la classe `ElectricDuck`
- Exemple du carré qui hérite du rectangle proposé par Robert C. Martin `https://web.archive.org/web/20151128004108/http:`

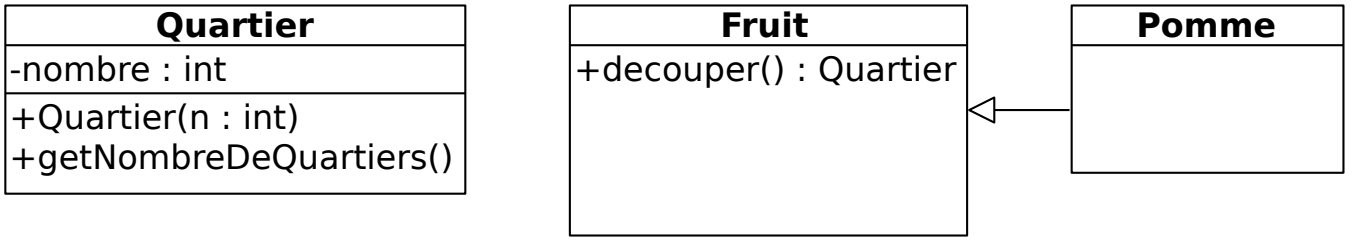
Alternatives à l'héritage : Délégation

Exemple : Héritage



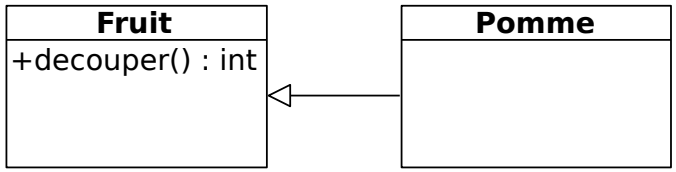

```
public class Example1Fruit {
 public static void main(String[] args) {
Pomme pomme = new Pomme();
 int nbquartiers = pomme.decouper(); }}
```

Que se passe-t-il si on change l'interface de `Fruit` ?



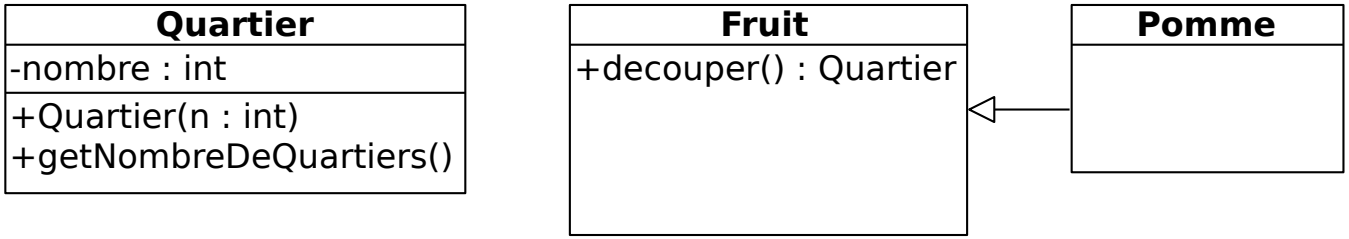
Alternatives à l'héritage : Délégation

Exemple : Héritage




```
public class Example1Fruit {
 public static void main(String[] args) {
 Pomme pomme = new Pomme();
 int nbquartiers = pomme.decouper(); }}
```

Changement de l'interface de `Fruit` : il faut changer l'interface de la sous-classe



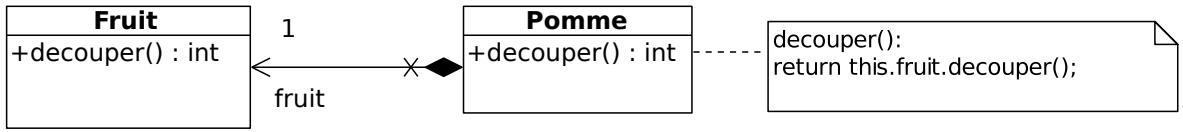

```
public class Example1CorrompuFruit {
 public static void main(String[] args) {
 Pomme pomme = new Pomme(); // ERREUR : CODE CORROMPU
 int nbquartiers = pomme.decouper();}}
```

Héritage ou Délégation ?

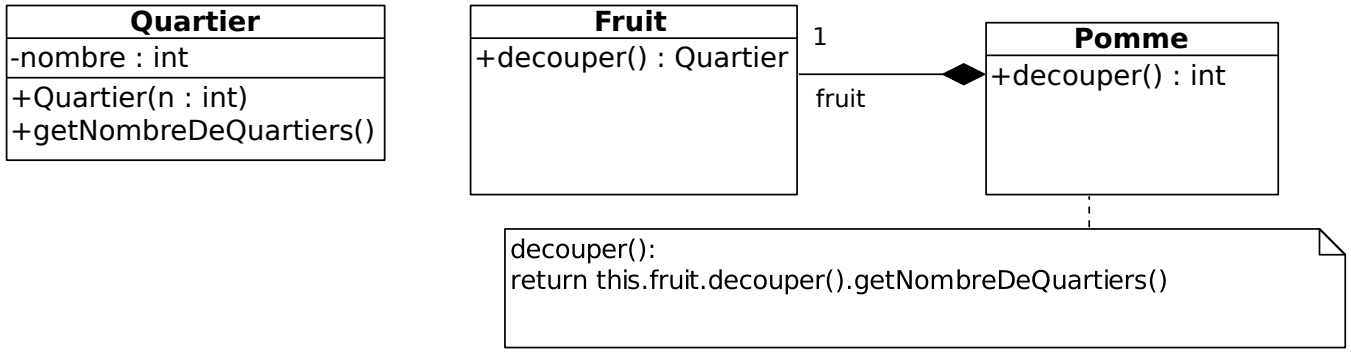
- Réutilisation de code à travers l'**héritage** : **white-box reuse**
  - liaison dynamique et polymorphisme
  - les sous-classes peuvent briser l'encapsulation
  - n'est pas toujours possible (p. ex. code propriétaire dont la documentation interne n'est pas disponible)
  - fragilité de l'interface de la super-classe - **encapsulation faible** : un changement de l'interface de la super-classe corrompt le code qui utilise la sous-classe

Héritage ou Délégation ?

Exemple : Délégation



Si on change l'interface de `Fruit`, l'interface de `Pomme` peut ne pas changer (mais il faut changer le code de la méthode de délégation)




```
public class Example1Fruit {
 public static void main(String[] args) {
 Pomme pomme = new Pomme();//ok
 int nbquartiers = pomme.decouper();}}
```

La classe `Pomme` est appelée la classe **front-end** et la classe `Fruit` la classe **back-end.**

Héritage ou composition ?

- Réutilisation du code à travers la **délégation** (association/composition/agrégation) : **black-box reuse**
  - pas de récupération automatique de toute l'interface publique de la super-classe, elle est explicite via la **délégation**
  - encapsulation plus forte : on peut ne pas changer l'interface de la classe *front-end* lorsque l'on change celle de la classe *back-end*
    - plus facile de changer l'interface d'une classe *back-end* que d'une super-classe
    - plus facile de changer l'interface d'une classe *front-end* que d'une sous-classe
    - plus difficile d'ajouter des classes *front-end* par composition que d'ajouter des sous-classes par héritage
  - la délégation d'invocation de méthode de la composition a un coût surtout si elle est systématique

Héritage ou composition ?

- Hériter pour substituer, sinon, déléguer.
- L'héritage doit être utilisé pour étendre une classe mère par ajout de fonctionnalités mais sans modification de fonctionnalités existantes.
- Ne pas utiliser l'héritage pour factoriser/"récupérer" du code, ou pour restreindre.
- Penser à combiner composition et interface.

Principe de Séparation des interfaces

Principe de Séparation des interfaces

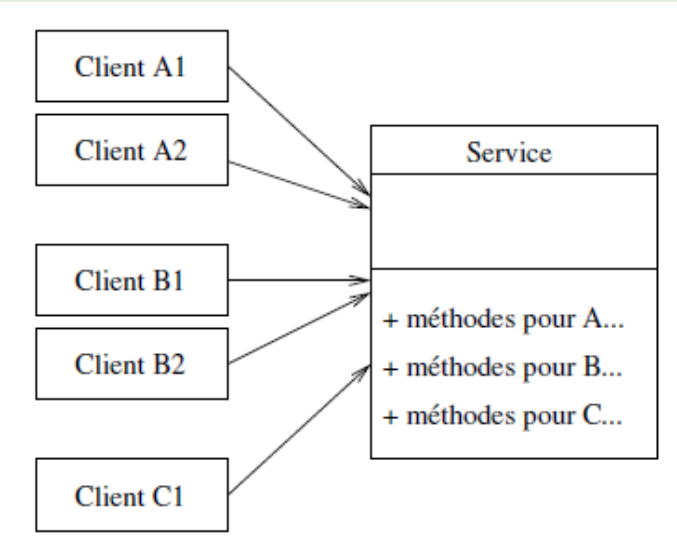
Un client ne doit jamais être forcé de dépendre d'une interface qu'il n'utilise pas.

crédits image http ://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Principe de Séparation des interfaces

- Interfaces avec multiples méthodes : symptôme de mauvaise conception.

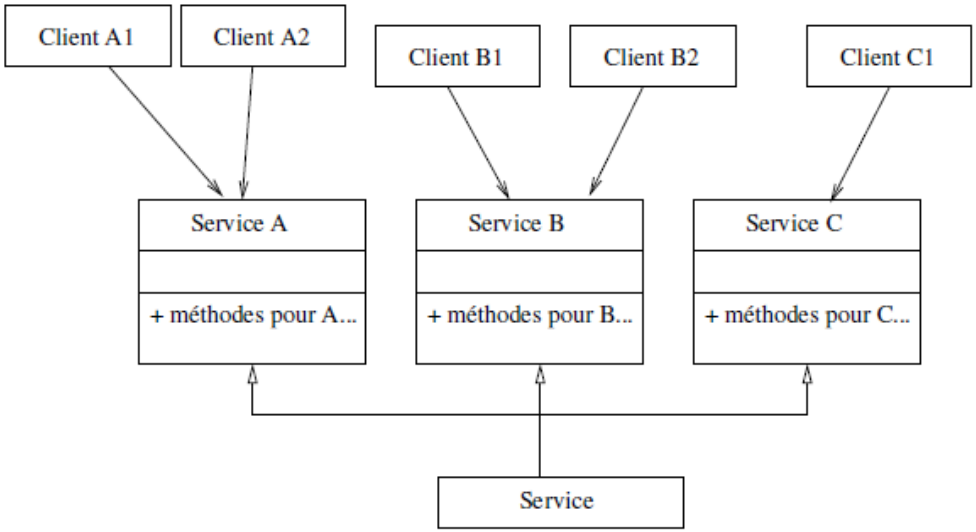
Pollution d'interface par agrégation de services :



Principe de Séparation des interfaces

Solution : séparation des services de l'interface

- Plusieurs interfaces client spécifiques valent mieux qu'une seule interface générale
- Les classes clientes ne doivent pas être forcées de dépendre d'interfaces qu'elles n'utilisent pas.



Intérêts :

- Limitation du couplage : éviter que les évolutions dues à une partie du service aient un impact sur un client alors qu'il n'est pas concerné
- Liens avec le principe de responsabilité unique : Interfaces spécialisées avec rôles bien définis

Principe d'inversion des dépendances

Principe d'inversion des dépendances

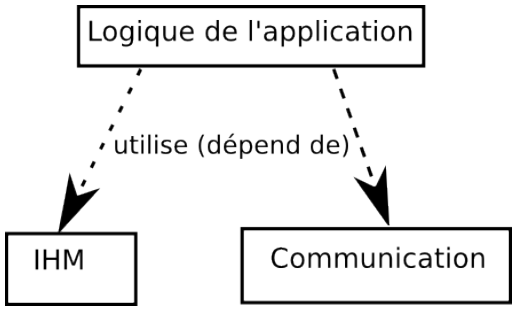
Dépendez des abstractions, ne dépendez pas des concrétisations.

crédits image http ://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Principe d'inversion des dépendances

- Modules de haut niveau (logique de l'application, aspects métier)
- Modules de bas niveau implémentations dépendantes de la machine, du stockage, de la communication ou des serveurs externes) (exemple : base de données, serveur de logs).

Les modules de haut niveau sont souvent construit à partir des modules fournissant des services de bas niveau :



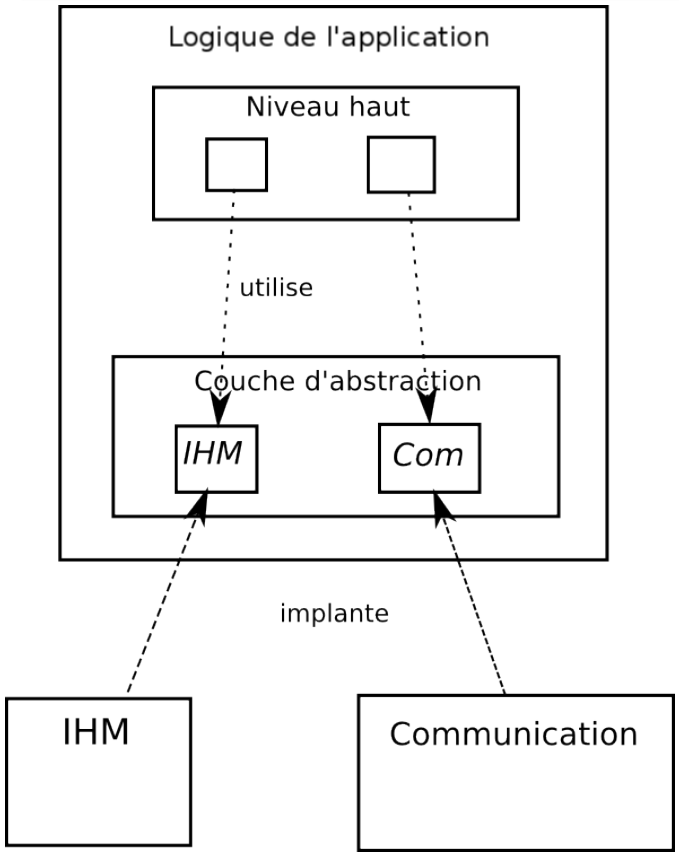
Conception néfaste :

- il faut modifier les modules de haut niveau quand les modules de bas niveau évoluent
- on ne peut pas réutiliser les modules de haut niveau (logique métier) dans contexte technique autre que celui d'origine.

Principe d'inversion des dépendances

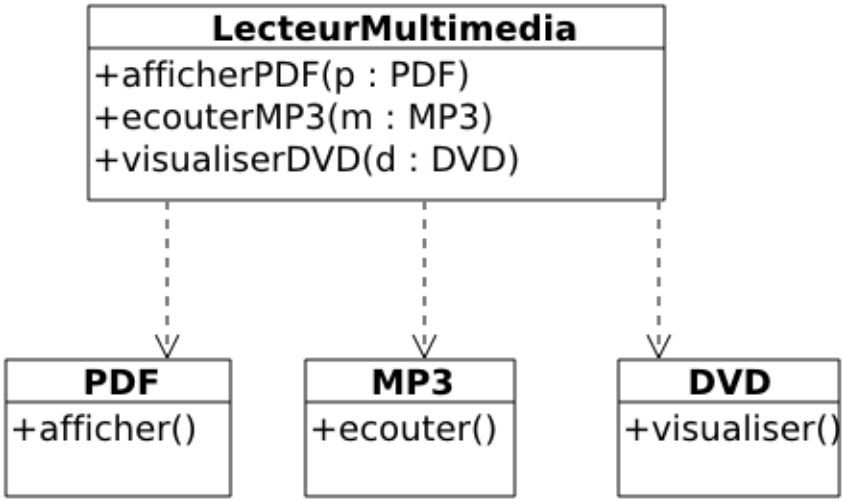
Dépendez des abstractions, ne dépendez pas des concrétisations :

- Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.



- une couche d'abstraction (interfaces, classes abstraites) doit séparer les niveaux.
- dépendances inversées : couche métier indépendante de l'implantation des couches basses

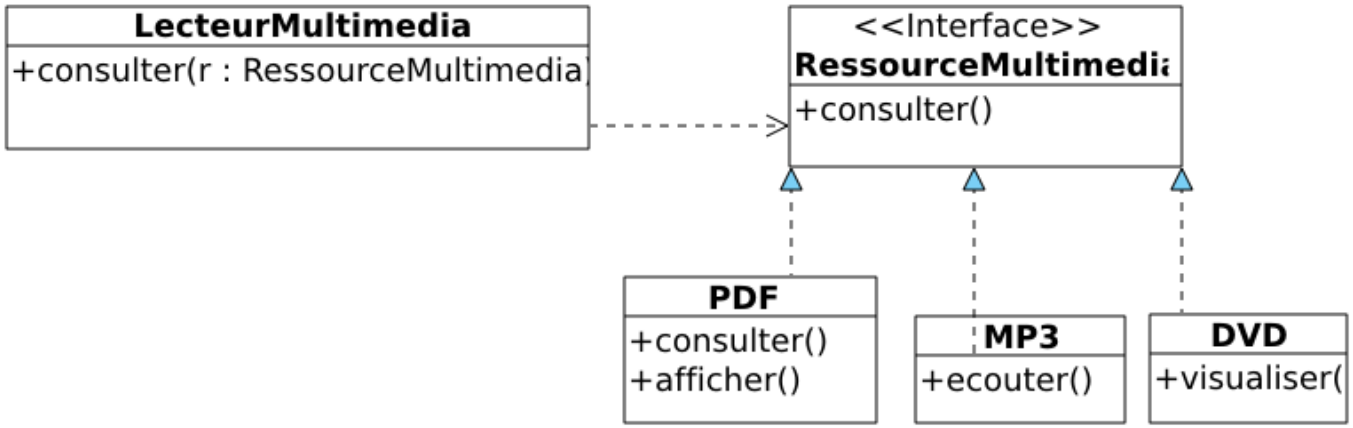
Exemple




```
public class LecteurMultimedia{
    ...
    public void afficherPDF(PDF p){ p.afficher();}
    public void ecouterMP3(MP3 m){ m.ecouter();}
    public void visuaiserDVD(DVD d){ d.visualiser();}
}
```

Exemple de mauvaise conception : le haut niveau dépend du bas niveau. Toute modification ou extension de la couche basse peut entrainer une modification de LecteurMultimedia.

Exemple

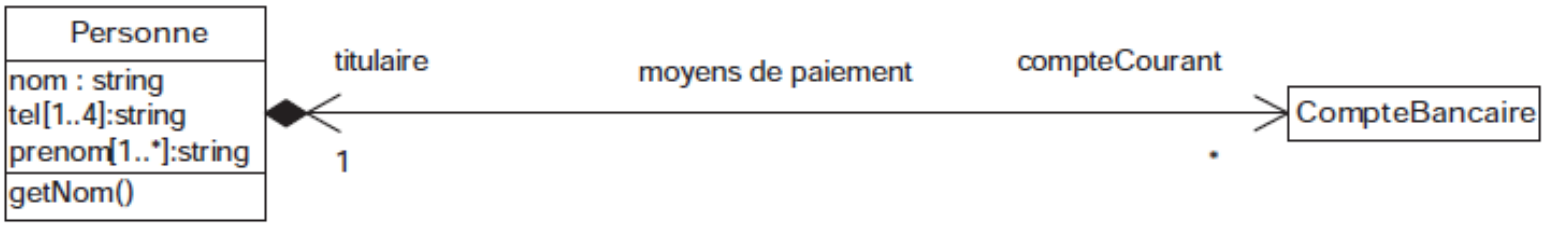



```
public class PDF implements RessourceMultimedia{
    public void consulter(){ this.afficher();}
}
```

Solution qui exploite l'inversion de dépendance en introduisant une couche d'abstraction `RessourceMultimedia`. La classe de haut niveau n'est plus impactée par des modifications d'une classe de bas niveau.

Principe de conception des package

Quel est l'impact des dépendances mutuelles, et plus généralement des cycles de dépendances ?



Dépendances entre classes :

- Si deux classes A et B dépendent mutuellement l'une de l'autre, cela signifie qu'il est impossible de les séparer.
- La dépendance mutuelle entre classes ne représentent pas une faute de conception.

Principe de conception des package

Quel est l'impact des dépendances mutuelles, et plus généralement des cycles de dépendances ?

Dépendances entre classes :

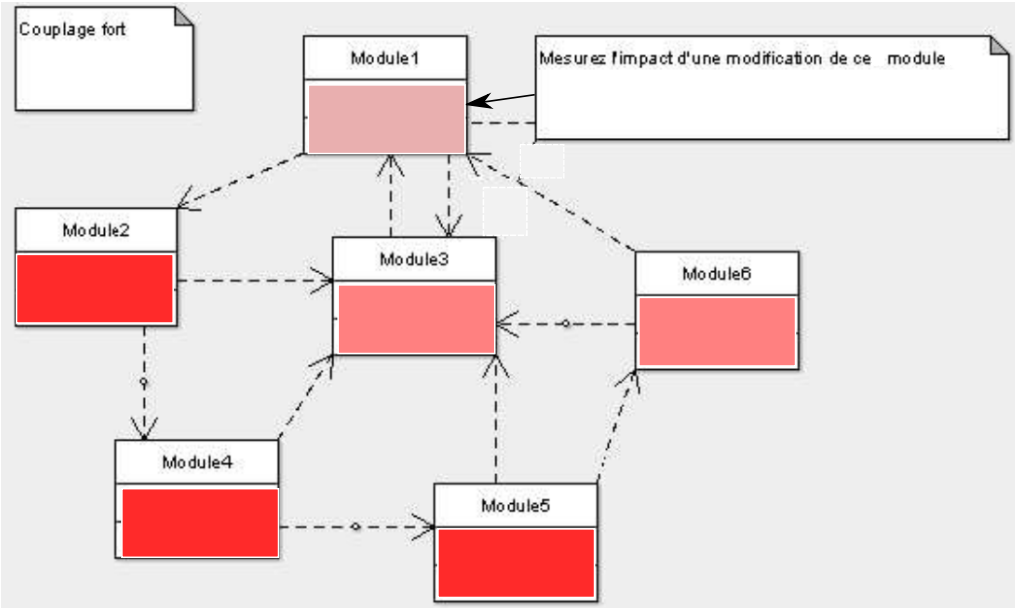
- Si deux classes A et B dépendent mutuellement l'une de l'autre, cela signifie qu'il est impossible de les séparer.
- La dépendance mutuelle entre classes ne représentent pas une faute de conception.

Principes de cohésion de package

Principe de fermeture commune - Common Closure Principle (CCP) : Les classes impactées par les mêmes changements doivent être placées de préférence dans un même package.

Principe de conception des package

Quel est l'impact des dépendances mutuelles, et plus généralement des cycles de dépendances ?



Dépendances entre package :

- Si dépendance circulaire, une modification d'un package est susceptible d'impacter tous les autres. L'application devient en quelque sorte monolithique.

Principe de conception des package

Principe de couplage entre package

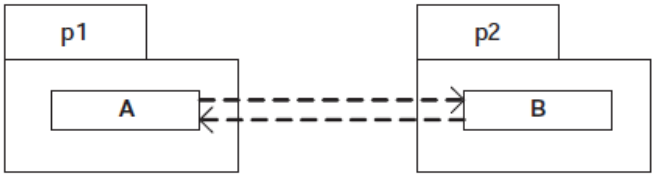
Principe des dépendances acycliques - Acyclic Dependencies Principle (ADP) : Les dépendances entre package doivent former un graphe acyclique.

Conception des packages

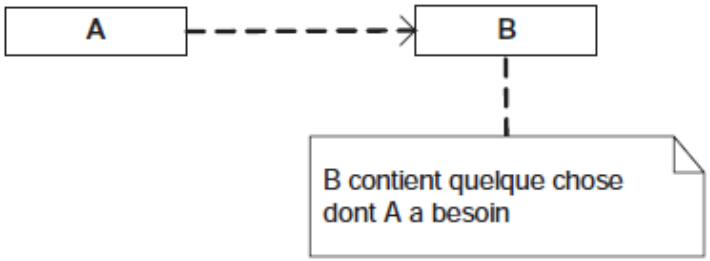
- Minimiser les dépendances qui existent entre les classes situées dans des packages différents.
- Eviter les cycles de dépendances entre packages
- Objectif : réaliser une découpe en package de meilleure qualité, réduire l'impact des changements et donc réduire les coûts d'évolution de maintenance, faciliter les tests.

Casser les cycles de dépendance

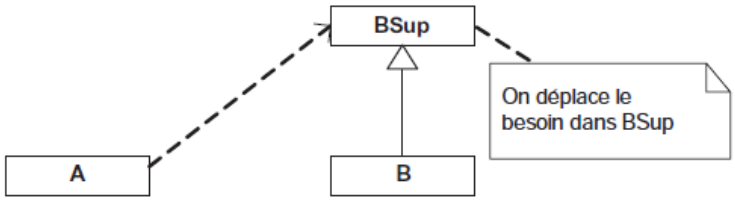
Déporter une des causes du cycle de dépendances hors de sa classe d'origine en utilisant l'inversion des dépendances :



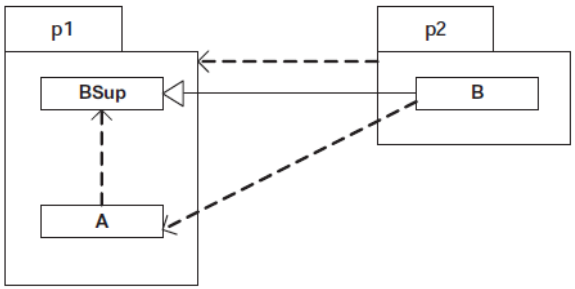
cycle de dépendance à casser



1- travailler sur une dépendance particulière



2- changer cette dépendance en une indirection



3- Placer BSup dans le package de A

Conception logicielle

Comment concevoir des logiciels maintenables et réutilisables ?

Maitriser la nature et le nombre des dépendances :

- Respecter les concepts fondamentaux de la conception orientée objet
- Suivre les principes avancés de conception orientée objet
- Appliquer des patrons de conception (*design patterns*) qui répondent concrètement à des problèmes récurrents
- Suivre des patrons d'architecture