Introduction à la programmation objet

Jean-Philippe Domenger

Université Bordeaux I &
LaBRI

Objectifs de ce cours

- Présenter les principaux concepts utilisés par les langages objets :
  - Modularité et encapsulation
  - Objets, classes, et messages
  - Interfaces, héritage, et polymorphisme
  - Réutilisabilité
  - Approfondissement de l'héritage
  - Réutilisation

Évolution du génie logiciel

- Passage de la programmation à petite échelle à la programmation à grande échelle
  - Évolution des langages de programmation
  - Évolution de la technologie
- Accroissement de la complexité
  - Nécessité de l ’abstraction

Historique des langages (1)

- Première génération : 1954-1958
  - Fortran I Expressions mathématiques
  - Algol 58 Expressions mathématiques
  - FlowMatic Expressions mathématiques
  - IPL V Expressions mathématiques

Historique des langages (2)

- Deuxième génération : 1959-1961
  - Fortran II: Sous programmes, compilation séparée
  - Algol 60: Structure en blocs, types de données
  - COBOL:Descriptions de données, gestion de fichiers
  - LISP: Traitement de listes, pointeurs, ramasse-miettes

Historique des langages (3)

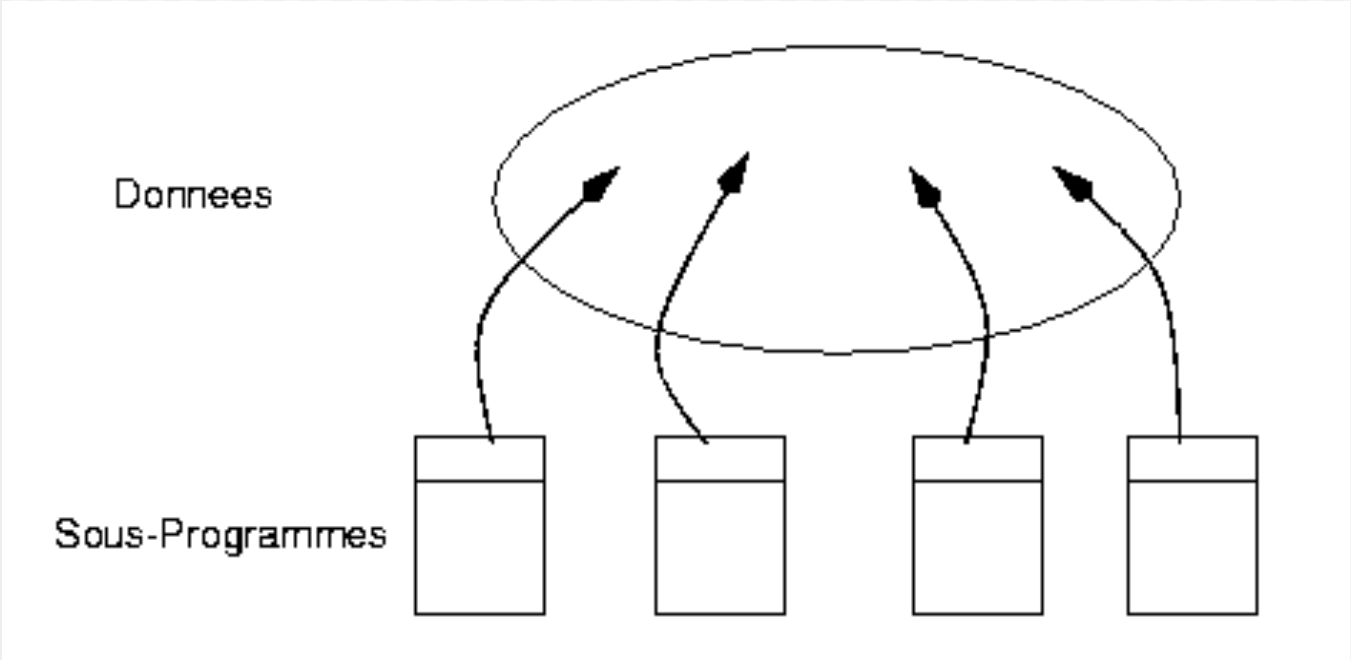
- Troisième génération : 1962-1970
  - PL/I Fortran + COBOL + ALGOL
  - Algol 68 Successeur rigoureux d'Algol 60
  - Pascal Successeur simple d 'Algol 60
  - *C* Successeur de B
  - Simula Classes, abstraction de données

Ancêtre des langages Objets

Historique des langages (4)

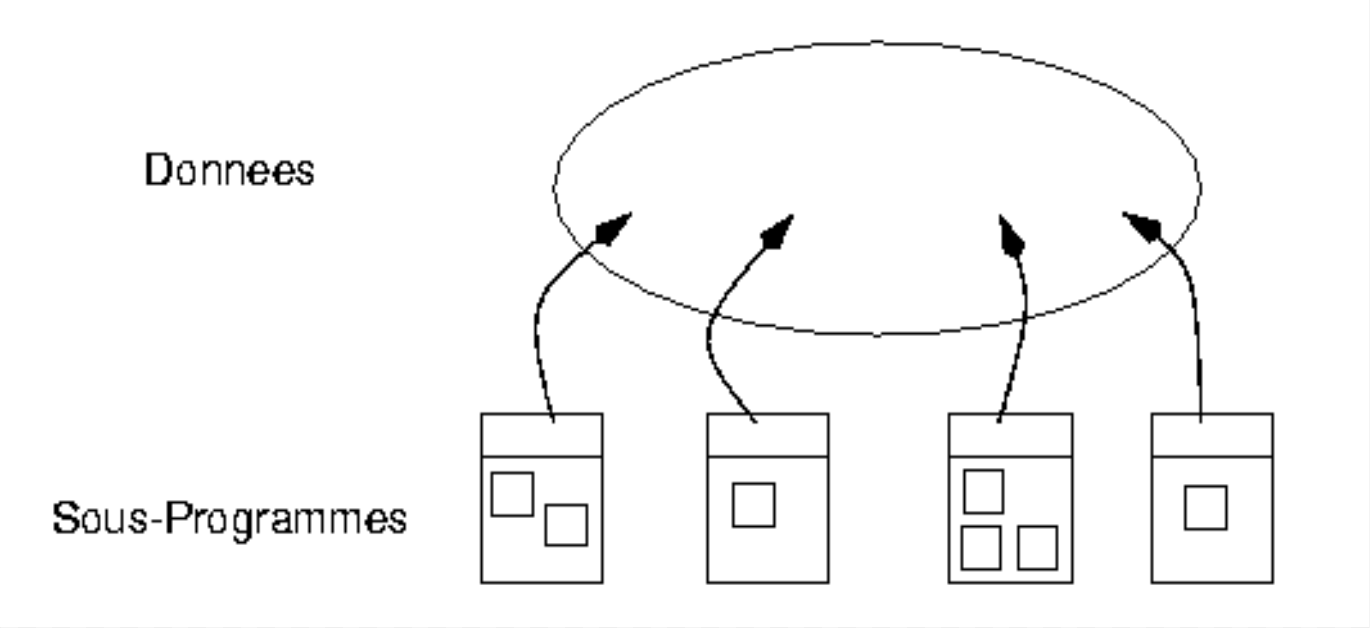
- Quatrième Génération : 1970-
  - ADA Successeur de Pascal et Algol 68
  - SmallTalk Successeur de Simula
  - C++ Mariage de Simula et C
  - Objective C Variante à C++
  - Eiffel Dérivé de Simula et ADA
  - CLOS LIPS + Flavors
  - Java Dérivé de simula et C

Topologie 1,2 Génération



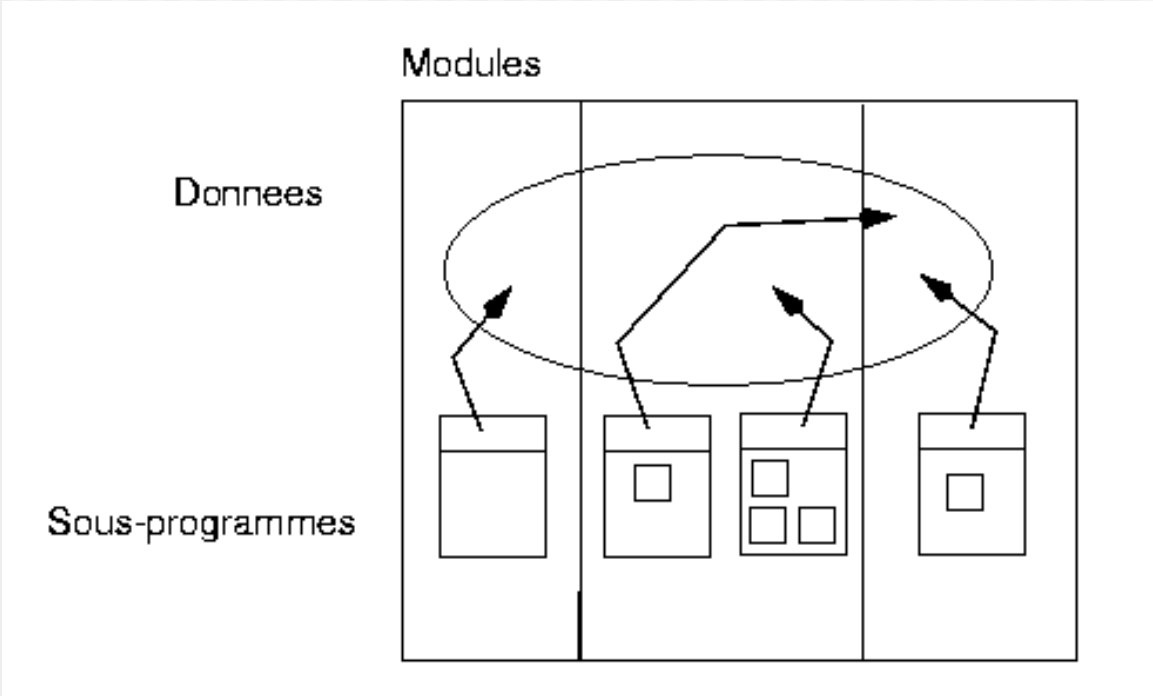
- Sous programmes = blocs élémentaires
- Pas de hiérarchie
- Propagations d ’erreurs
- Extensibilité difficile

Topologie 2,3 Génération



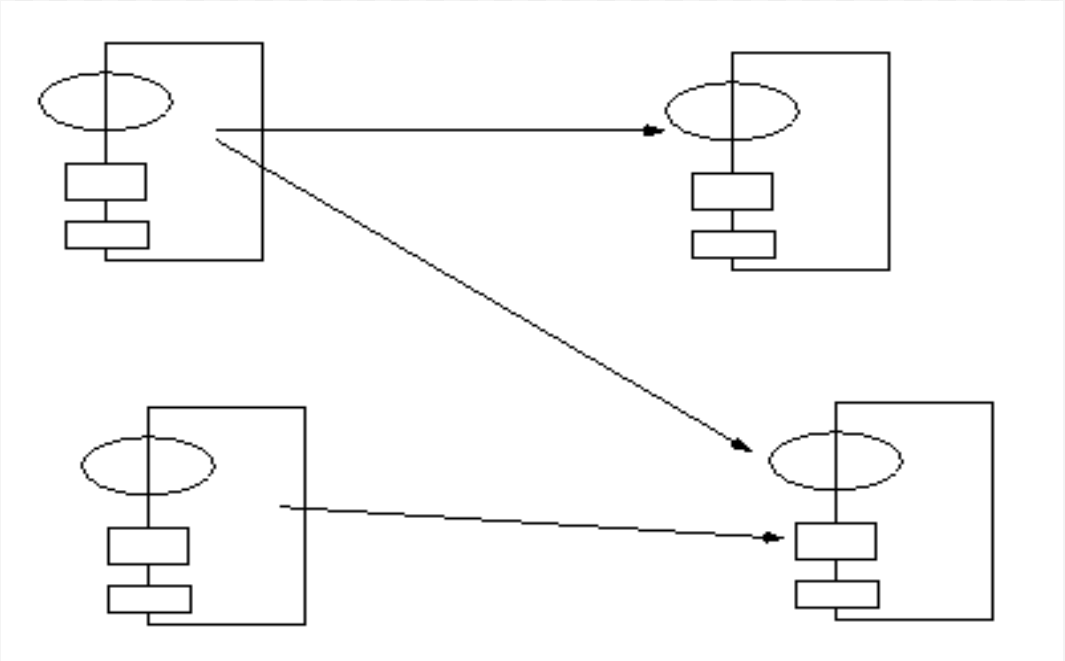
- Séparation des données globales
- Sous programmes = boîtes noires
  - Passages de paramêtres
  - Retour de fonctions, imbrications de fonctions
- Découpage difficile du développement

Topologie 3 Génération



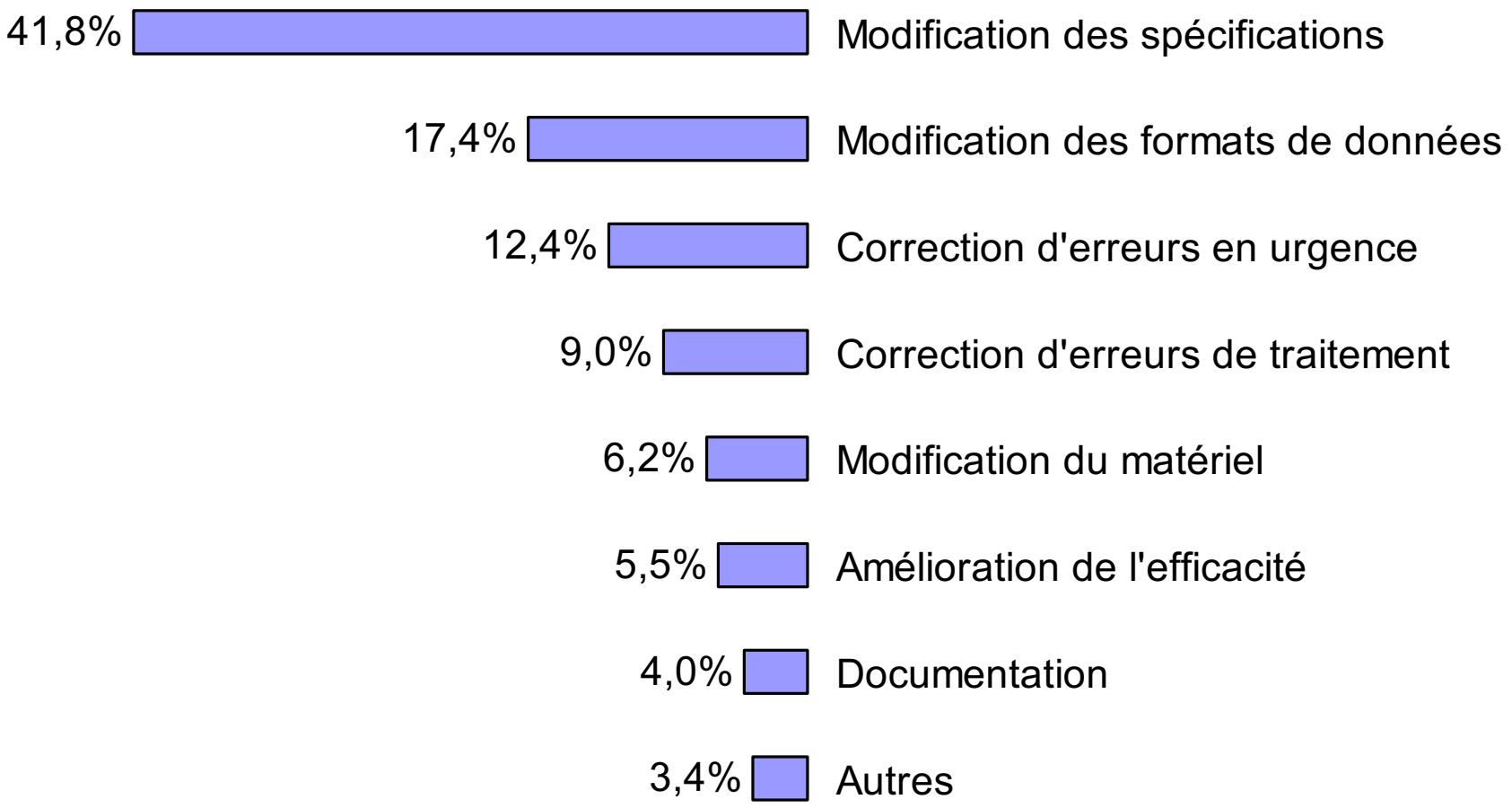
- Séparation des données locales, globales
- Abstraction modulaire
  - Interface Modulaires
  - Données internes
  - Fonctions internes

Topologie Objets



- Architecture basée sur les données **(Objets)**
- Types Abstraits **(Classes)**
- Protections de données **(Encapsulation)**
- Hiérarchie de classe **(Héritage)**

Répartition des coûts de maintenance



Facteurs externes de qualité

- **Validité**: aptitude à réaliser les tâches définies par les spécifications
- **Robustesse**: aptitude à fonctionner dans des conditions non spécifiées
- **Extensibilité**: facilité d'adaptation aux changements de spécifications
- **Réutilisabilité**: aptitude à être réutilisé en tout ou partie pour de nouvelles applications

Approche fonctionnelle versus approche objet

On veut réaliser un éditeur de dessin.
On crée des cercles, des rectangles,
…..
On les déplace, on les agrandit,….

Approche fonctionnelle

- Découpage en fonction des traitements.

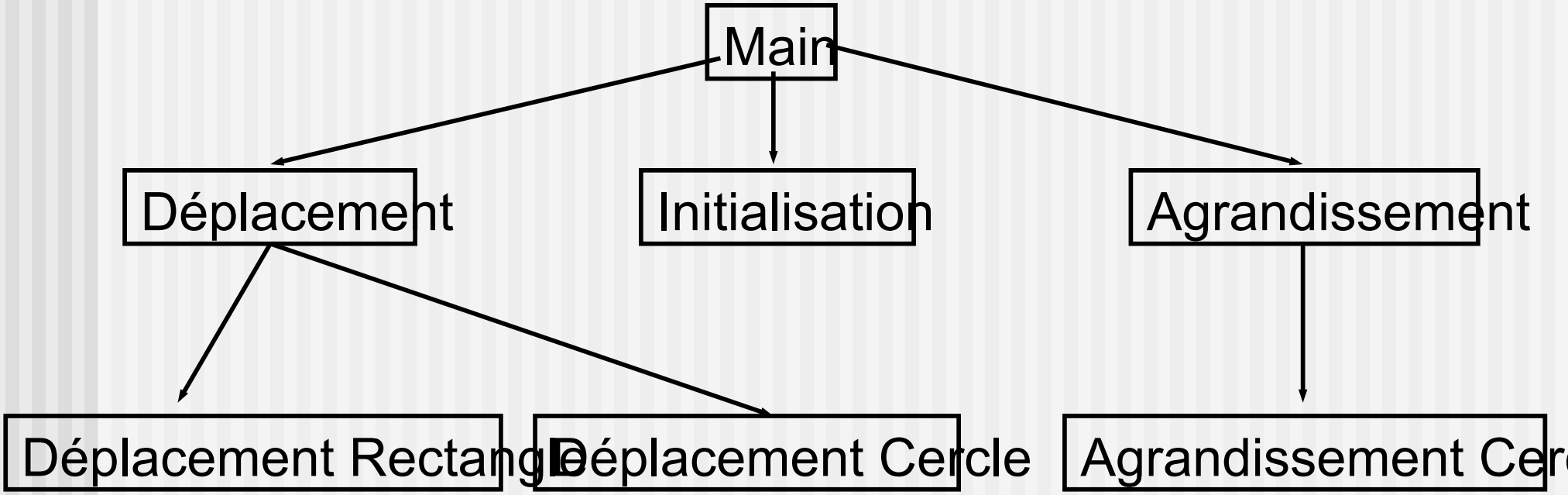
  Un module = Un traitement

  - Traitements
    - Initialisation
    - Déplacement
    - Agrandissement
    - …….

Approche fonctionnelle

- Les données circulent à travers les traitements
  - Données
    - Liste des cercles
    - Listes des rectangles

Exemple de structuration fonctionnelle



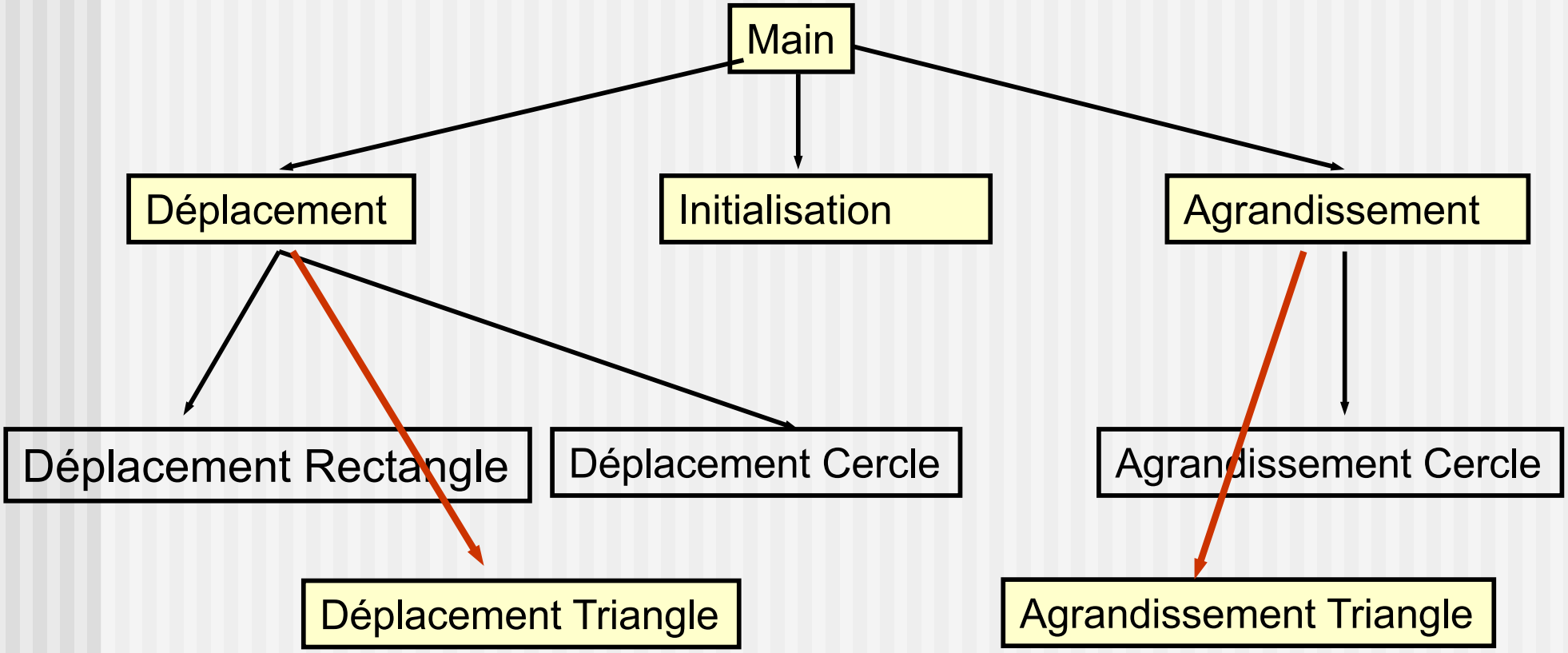
Exemple de pseudo code

```
Module Initialisation
Carre  [] listeCarre;
Cercle [] listeCercle;
void initialisation(){
  ajouterCarre(listeCarre);
  ajouterCercle(listeCercle);
}
Module Déplacement
void Déplacement(listeCarre,
listeCercle){
 DéplacementCarre(listeCarre);
 DéplacementCercle(listeCercle);
}
```

Approche Fonctionnelle

- Avantages
  - Adapté à un algorithme
- Inconvénients
  - Programme difficilement exprimable par un unique algorithme
  - basée uniquement sur les traitements
  - partage d'une même structure de données
  - extensibilité et réutilisabilité sont pénalisées

Extensibilité ?



Eléments modifiés pour prendre en compte un triangle

Approche objet

- Regroupe dans le même module le traitement et les données
- Les objets représentent des abstractions du domaine du problème
- Un programme objet consiste à faire collaborer des objets pour réaliser un traitement

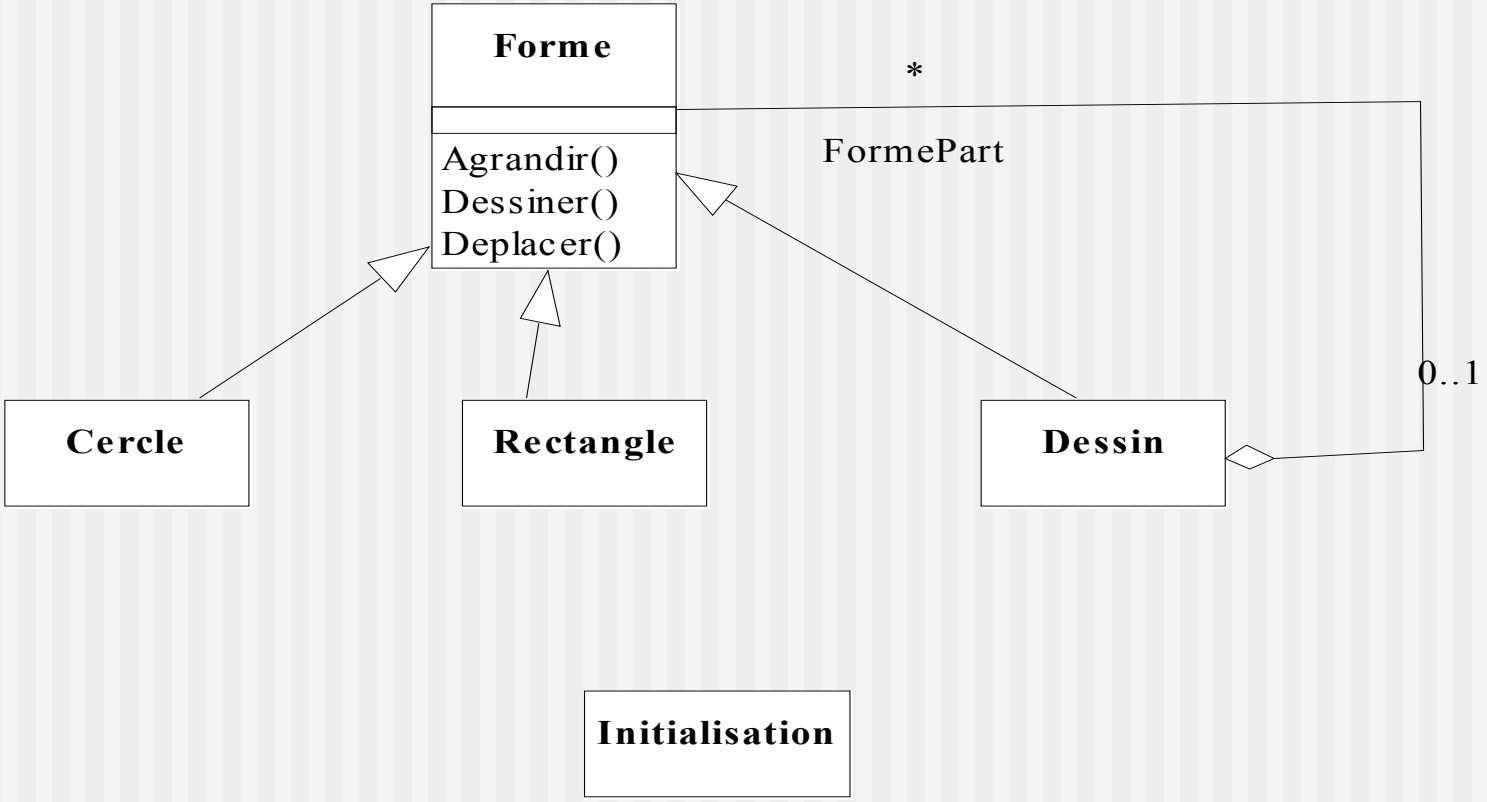
Les questions de l'approche objet

- Qui ?
- Que fait-il ?
- Avec qui ?
- Est ce que d'autres font la même chose ?
- Comment le fait-il ?

Approche Objet

- Qui ? Les cercles, les rectangles,
- Comportement ? Déplacement, agrandissement, affichage
- Généralisation ? Les cercles et les carrés sont des Formes
- Composition ? Un dessin contient des formes

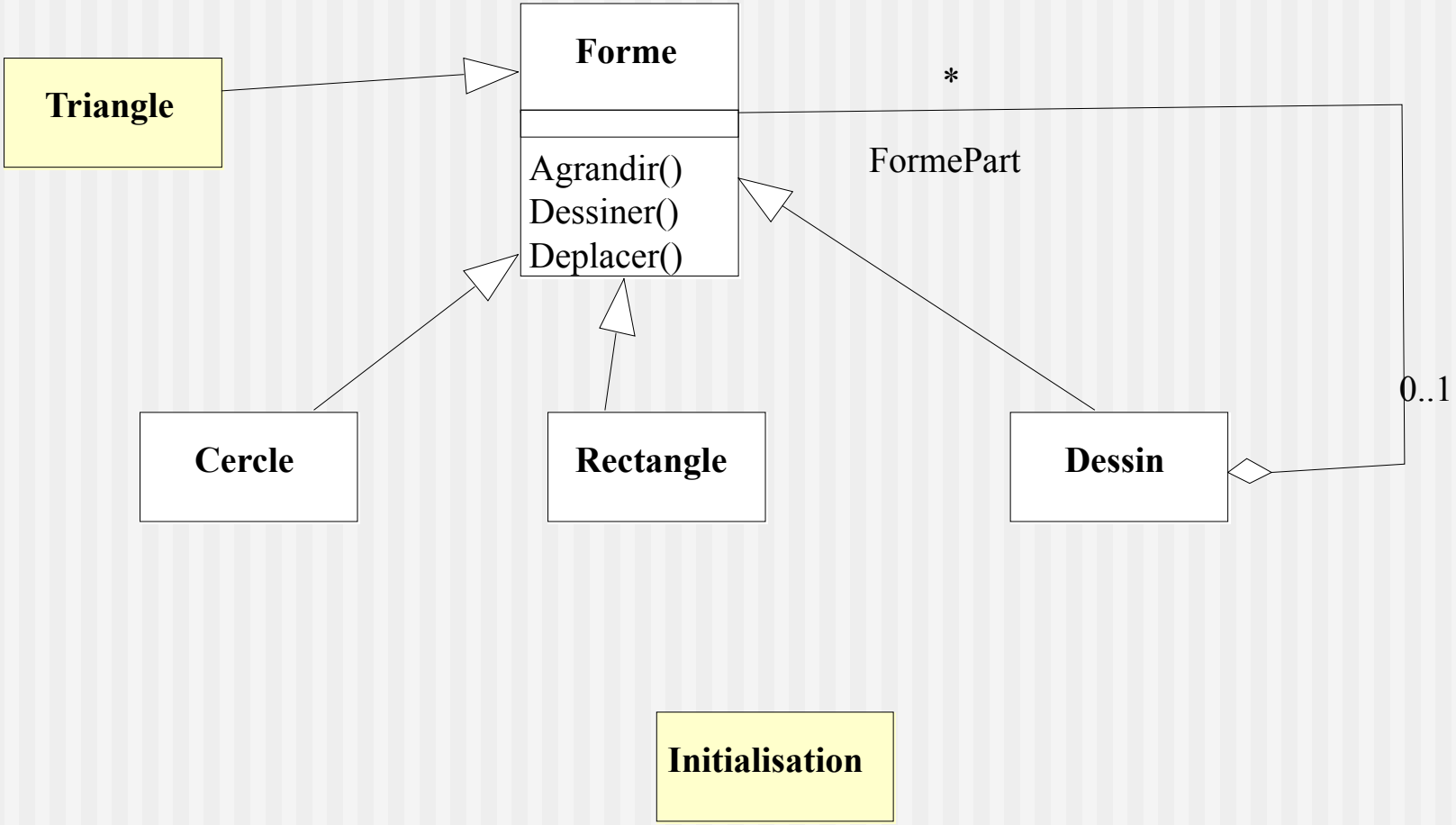
Structuration approche Objet



Exemple de pseudo code

```
Class Dessin{
Forme  [] listeForme;
void Dessine(){
  for(int i =0; i < listeForme.size;i++)
    listeForme[i].Dessine();
}
void Deplace(){
  for(int i =0; i < listeForme.size;i++)
    listeForme[i]. Deplace();
}
void Agrandir(float f){
  for(int i =0; i < listeForme.size;i++)
    listeForme[i]. Agrandir(f);
}
```

Extensibilité ?



Approche Objet

- Séparation interface/ implémentation
- stabilité des données par rapport aux traitements
- réutilisabilité
- extensibilité

Modularité

La modularité est permise par l ’utilisation de l ’abstraction et de l ’encapsulation

Modularité

- Technique de décomposition visant à réduire la complexité d'un système en le considérant comme un assemblage de sous-systèmes plus simples
- Un module est un sous-système dont le couplage avec les autres est faible par rapport au couplage de ses propres parties
- La capacité à modulariser dépend du degré de couplage entre les sous-systèmes

Abstraction

- Une abstraction fait ressortir les caractéristiques essentielles d'un objet
- Définition des frontières conceptuelles par rapport au point de vue de l 'observateur
- L'abstraction est représentée par les types abstraits

Type abstrait (1)

- Un type abstrait est composé :
  - D'un ensemble de valeurs possibles
  - D'un comportement, c'est-à-dire d'un ensemble d'opérations applicables sur ce type :
    - Constructeurs
    - Accesseurs
    - Modificateurs d'état

Type abstrait (2)

- Exemple : type abstrait **`Point`**
  - Construction

```
point: Float X, Float Y ➪ Point
```

  - Assesseurs

```
abscisse: Point ➪ Float
ordonnée: Point ➪ Float
```

  - Modificateurs d ’état

```
translater: Point P, Float DX, Float DY ➪ Point
pivoter: Point P, Point C, Float A ➪ Point
```

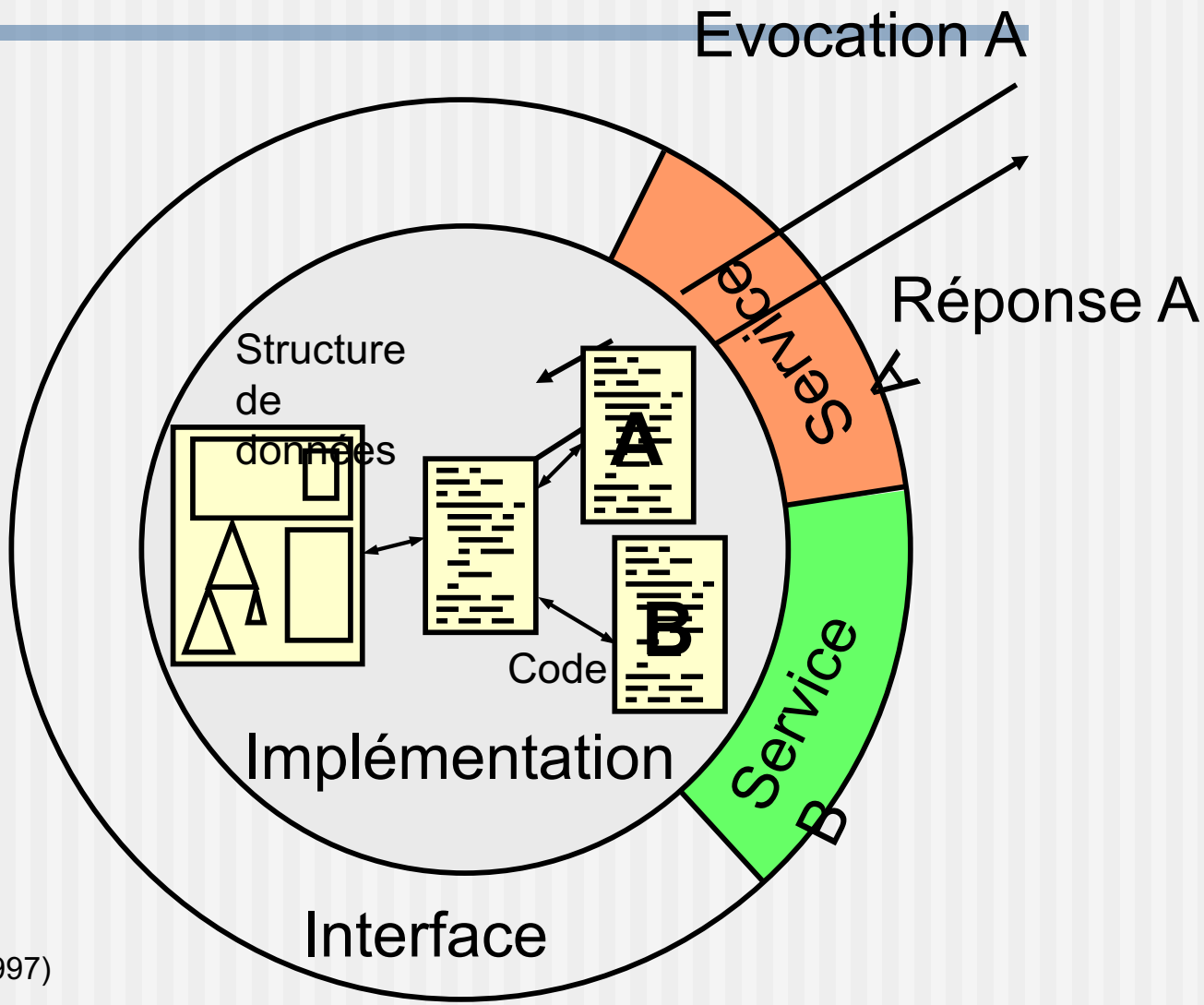
Encapsulation (1)

- Technique favorisant la modularité des sous-systèmes, par séparation de l'interface d'un module de son implémentation
- Interface (partie publique)
  - Liste des services offerts
  - Présentation du type Abstrait
- Implémentation (partie privée)
  - Réalisation des services offerts
    - Structures de données
    - Algorithmes et implémentation

Encapsulation (2)

- L'encapsulation doit être assurée :
  - Au niveau du langage : mots-clés qualificateurs **`public`**, **`private`**, …
  - Au niveau de l'exécution : interdiction de manipulations hasardeuses de pointeurs, …
- Les modules communiquent par évocation du comportement et non par accès mutuels à leurs données

Module



D'après G. Falquet (1997)

Intérêt de la Modularité

- Séparation entre les services et leur réalisation.
  - Evolution de l ’implémentation du fournisseur sans remise en cause des clients.
- Compatibilité Ascendante de l ’interface
  - Services fournis à To doivent perdurer

Intérêt de la Modularité

- Programmation contractuelle
  - Pré-Condition, vérification par le module que l ’appel est conforme au contrat. Evite la propagation des erreurs.
  - Post-Condition, phase de mise au point

    Vérifie que l ’objet est dans un état conforme.

Objets et classes

Comment les langages objets formalisent les notions de modules, d'encapsulation, et de typage

L’âge de raison

- L’approche orientée-objets à 30 ans
  - 1966 Une idée à Oslo (naissance de Simula)
  - 1969 La recherche à Xerox Park
  - 1980 Version industrielle de Smalltalk
  - 1986 1200 personnes pour OOPSLA’86
  - 1996 Omniprésence des solutions objets
  - 2006 Stabilisation de la transition vers l’objet

Approche orientée objets

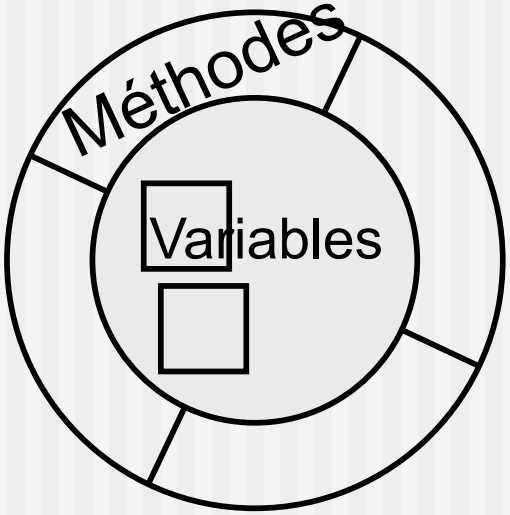
- Synthétise les notions de module et de type abstrait
- Permet la structuration du logiciel en termes de relations clients/fournisseurs
- Permet le découplage entre spécification et implémentation
- Autorise le polymorphisme simple
  - Héritage
  - Liaison dynamique

Concepts clés

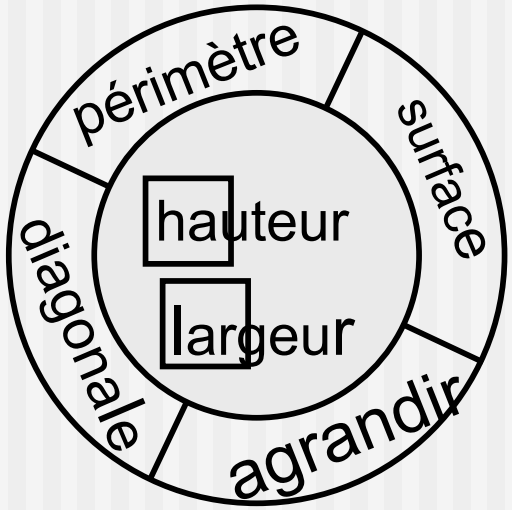
- Un programme est organisé comme un ensemble coopérant d'objets
- Chaque objet est instance d'une seule classe
- Une classe est la concrétisation d'une ou plusieurs interfaces
- Chaque interface représente un type
- Les types sont ordonnés partiellement par la relation d'héritage

Objet (1)

- Entité contenant des données (état) et des procédures associées (comportement)



Objet



rectangle

Objet (2)

- Objet : instanciation de modules
  - Partie privée :valeur des variables (d'instance)
  - Partie publique : noms et prototypes des méthodes publiques
  - Communication : invocation (appel) de méthodes, retour de résultats

Identité et référence

- Tout objet possède une identité qui lui est propre et qui le caractérise
- L'identité permet de distinguer tout objet de façon non ambiguë, indépendamment de son état
- Une référence est l'identifiant unique d'un objet
- L'adresse mémoire n'identifie pas systématiquement l'objet

État

- Chaque objet a un état qui lui est propre
- L'état regroupe les valeurs instantanées de tous les attributs d'un objet
- L'état d'un objet peut évoluer au cours du temps
- L'état d'un objet représente les effets cumulés de son comportement

Comportement

- Décrit les actions et les réactions d'un objet
- L'ensemble des opérations applicables à un objet définit son comportement
- Les opérations peuvent soit :
  - Accéder à l'état de l'objet sans le modifier
  - Modifier l'état de l'objet
- L'état d'un objet peut influencer le résultat du comportement de celui-ci

Encapsulation

- Un objet contient à la fois son état et son comportement
- Seules les opérations appartenant au comportement d'un objet peuvent modifier son état (*encapsulation pure et dure*)
- Séparation entre interface et implémentation

Message (1)

- Les objets collaborent en échangeant des messages

```
p.translation (3, 2); // Envoie à l'objet p le message
                      // « translation » avec les
                      // paramètres (3, 2)
```

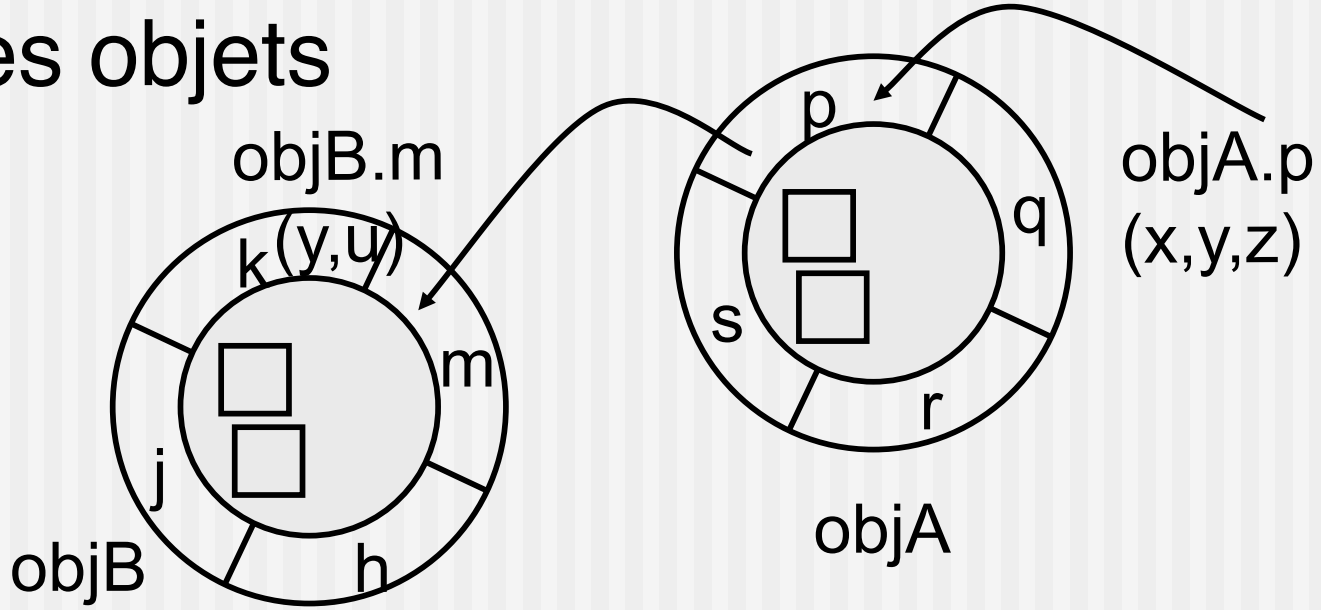
- Tout message reçu et accepté par un objet donné déclenche l'exécution d'une méthode

Message (2)

- La méthode à exécuter est choisie en fonction :
  - Du type du message (nom de la méthode)
  - Du type des paramètres contenus dans le message (signature de la méthode)
  - Du type de l'objet destinataire du message :
    - Type déclaré : liaison statique
    - Type réel : liaison dynamique

Message (3)

- Une méthode peut envoyer des messages à d'autres objets



- Un système est constitué d'objets communiquant entre eux

Classe (1)

- Définition conceptuelle
  - Une classe est l'implémentation d'un ou plusieurs types abstraits
  - C'est une définition statique de la structure et du comportement d'un ensemble d'objets
- Définition ensembliste
  - Une classe est un ensemble d'objets ayant exactement la même structure et le même comportement

Classe (2)

- Classe concrétisation d'un ou plusieurs types abstraits
  - Définit la structure et le comportement des objets instances de la classe
  - Réalise l'encapsulation en définissant la visibilité des constituants de l'objet : publique, privée, ...
  - La partie publique fournit une spécification (partielle) du ou des types abstraits

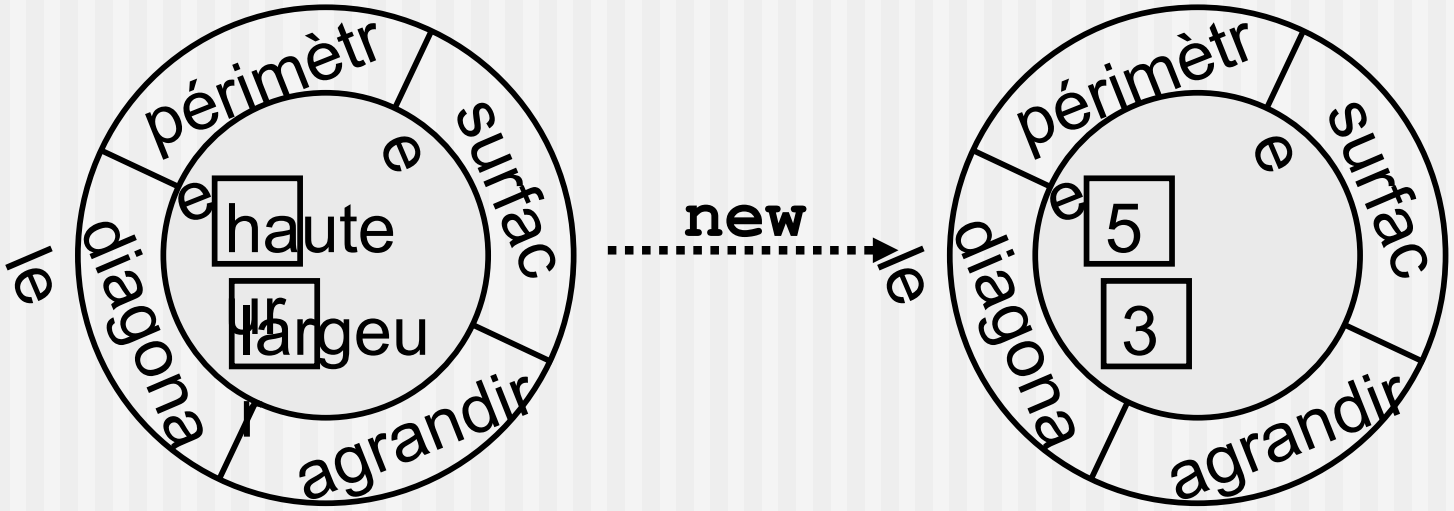
Classe (3)

- Une classe peut réaliser simultanément plusieurs rôles définis par des types abstraits différents
  - Implémentation des méthodes publiques correspondant aux opérations des différents types
  - Conflits à résoudre si opérations communes

```
class Pointvoiture: Point,Voiture
{ // Réalise Point + Voiture
 abscisse() // Abscisse du point
 ordonnée() // Ordonnée du point
 démarre()  // Démarre la voiture
 arrête()   // Arrête la voiture
 afficher() // Conflit entre représentations possibles
}
```

Classe (4)

- Une classe est un moule pour fabriquer des objets de même nature
- Un objet est une instance d'une unique classe



Classe (5)

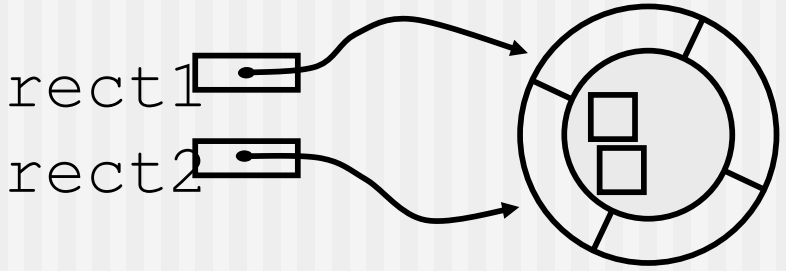
- Un objet peut tenir simultanément tous les rôles concrétisés par sa classe
- L'objet peut être considéré comme une instance de chacun des types abstraits concrétisés par la classe

```
Pointvoiture p = new Pointvoiture (); //
Instanciation
…
x = p.abscisse ();              // Obtension de
l'abscisse
y = p.ordonnée ();              // Obtension de
l'ordonnée
p.démarre ();                   // Démarre la
```

Références (1)

- Une classe C définit un type C
- Une variable de type C peut faire référence à un objet de la classe C
- Deux variables différentes peuvent faire référence au même objet

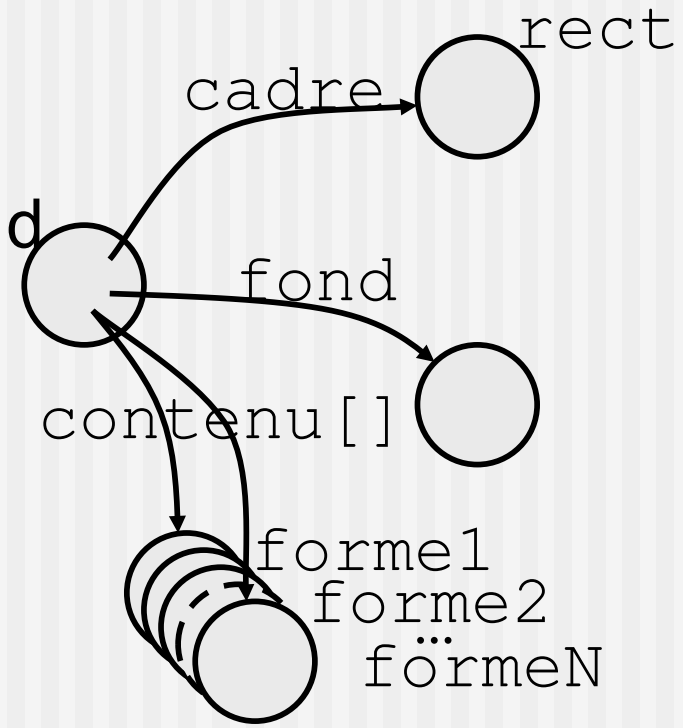
```
Rectangle rect1;                    // Déclaration
Rectangle rect2;
rect1 = new Rectangle (3, 5);       // Instanciation
rect2 = rect1;                      // Référence
```



Références (2)

- Les variables d'instances établissent des relations entre objets

```
class Dessin {
   Rectangle  cadre;
   Couleur    fond;
   Forme []   contenu;
}
…
d = new Dessin ();
d.cadre = rect;
d.fond  = new Couleur
("Rouge");
d.contenu[0] = forme1;
d.contenu[1] = forme2;
…
```



Identité et égalité

- Il y a identité d'objet lorsque deux références distinctes pointent sur le même objet
- Deux objets sont égaux s'ils sont dans le même état

```
rect1 = new Rectangle (3, 5);
rect2 = new Rectangle (6, 2);
rect3 = new Rectangle (3, 5);
rect4 = rect1
…
(rect1 == rect3) ➪ false   // Égalité des références
(rect1 == rect4) ➪ true
rect1.equals (rect2) ➪ false  // Égalité des champs
rect1.equals (rect3) ➪ true
rect1.equals (rect4) ➪ true
```

Propriétés de l'égalité

- L' égalité doit toujours vérifier les trois propriétés suivantes
  - Réflexivité : **`a.equals(a)`** doit toujours être vrai
  - Symétrie : **`a.equals(b)`** et **`b.equals(a)`** doivent toujours donner le même résultat
  - Transitivité : si **`a.equals(b)`** et **`b.equals(c)`** sont vrais, alors **`a.equals(c)`** doit toujours être vrai aussi

Test d'égalité

- Le test d'égalité peut être :
  - Simple : comparaison des variables d'instance
  - Complexe : calcul sur les variables d'instance

```
class Fraction {
  int       numérateur;
  int       dénominateur;
  …
  public boolean equals (Fraction f) {
    return ((this.numérateur   * f.dénominateur) ==
            (this.dénominateur * f.numérateur));
  }
}
```

Égalité de surface

- Il y a égalité de surface si on ne réalise qu'un test d'identité sur les champs des objets

```
((dess1.cadre == dess2.cadre)                    &&
 (dess1.fond  == dess2.fond)                     &&
 (dess1.contenu[0] == dess2.contenu[0])  &&
 …)
```

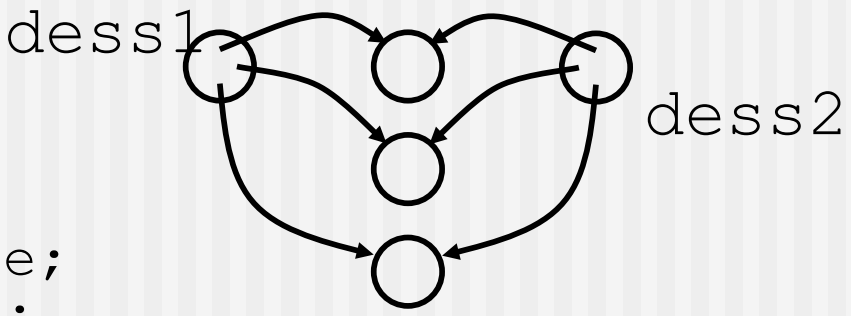
Égalité profonde

- Il y a égalité profonde si on effectue (récursivement) des tests d'égalité

```
(dess1.cadre.equals          (dess2.cadre)
&&
 dess1.fond.equals           (dess2.fond)
&&
 dess1.contenu[0].equals (dess2.contenu[0])
&&
 …)
```

Copie de surface

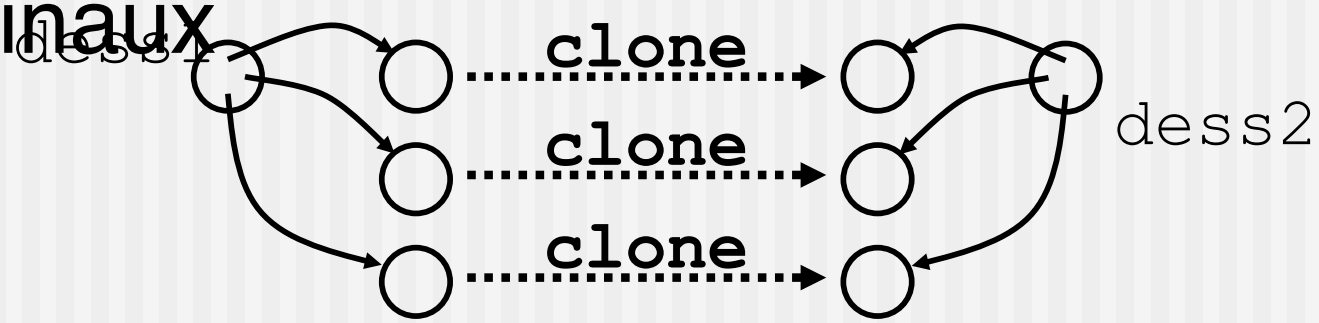
- Les considérations sur l'égalité s'appliquent aussi à la copie d'objets
- Il y a copie de surface si les champs de la copie font référence aux champs originaux, qui sont alors partagés par les deux objets




```
dess2.cadre       = dess1.cadre;
dess2.fond        = dess1.fond;
dess2.contenu[0]  = dess1.contenu[0];
…
```

Copie profonde

- Il y a copie profonde si les champs de la copie sont des clones des champs originaux




```
dess2 = new Dessin ();
dess2.cadre      = dess1.cadre.clone ();
dess2.fond       = dess1.fond.clone ();
dess2.contenu[0] = dess1.contenu[0].clone ();
…
➪ dess2 = dess1.clone ();
```

Interfaces, classes sous-typage et héritage de code.

Comment les relations entre types et sous-types augmentent la modularité et la réutilisabilité

Interface (1)

- La notion de classe mélange spécification (déclaration de type) et implémentation
- Nécessité de séparer les deux pour clarifier la notion d'héritage :
  - Héritage d'interface : relation type/sous-type
  - Héritage de code

Interface (2)

- Interface : spécification de type abstrait
  - Ensemble de signatures d'opérations publiques ayant une sémantique commune (service)
  - Aucune définition de code
  - Une interface I définit un type `I`
- Classe : implémentation du type abstrait
  - Une classe peut réaliser le comportement de plusieurs interfaces, en définissant le code de toutes les méthodes déclarées

Concrétisation

- Une classe peut réaliser une ou plusieurs interfaces

```
interface Vivant {
  void     naît ();
  void     meurt ();
  boolean  estVivant ();
}
interface Mobile {
  void     bouge (double x, double y);
}
class Lapin implements Vivant, Mobile {
  boolean  coeurBat = false;
  void     naît  () { coeurBat = true; }
  void     bouge (double x, double y) { …
```

- La gestion des conflits dépend des langages

Extension

- Une interface peut étendre une ou plusieurs autres interfaces

```
interface Mobile {
  void     bouge (double x, double y);
}
interface Vivant {
  void     naît ();
  void     meurt ();
  boolean  estVivant ();
}
interface Quadrupède extends Vivant, Mobile {
  void     allongé ();     // Extensions en plus de
  void     debout  ();     // ce qui vient des deux
}
class Lapin implements Quadrupède { …
```

Hiérarchie des types

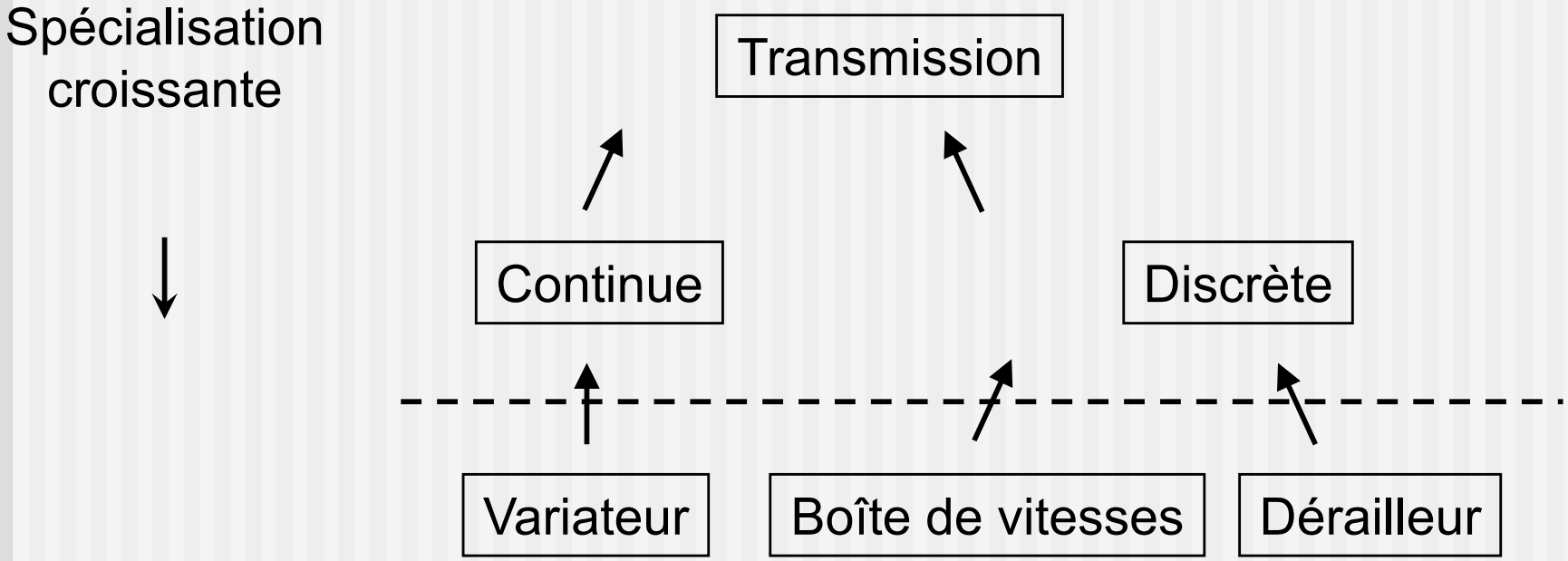
- Classification des interfaces en fonction de leurs caractéristiques communes
- Relations de type *est-un* ou *est-une-sorte-de*
- Arborescence des interfaces par abstraction croissante
- Ordre partiel

Sous-typage

- Conformité des spécifications des interfaces
  - L'interface du type est toujours incluse dans l'interface du sous-type
- Principe de substitution
  - Il est possible de référencer une instance du sous-type en utilisant le sur-type. Une variable ou un paramètre de type `T` peut donc indifféremment faire référence à un objet de type `T` ou d'un sous-type de `T`

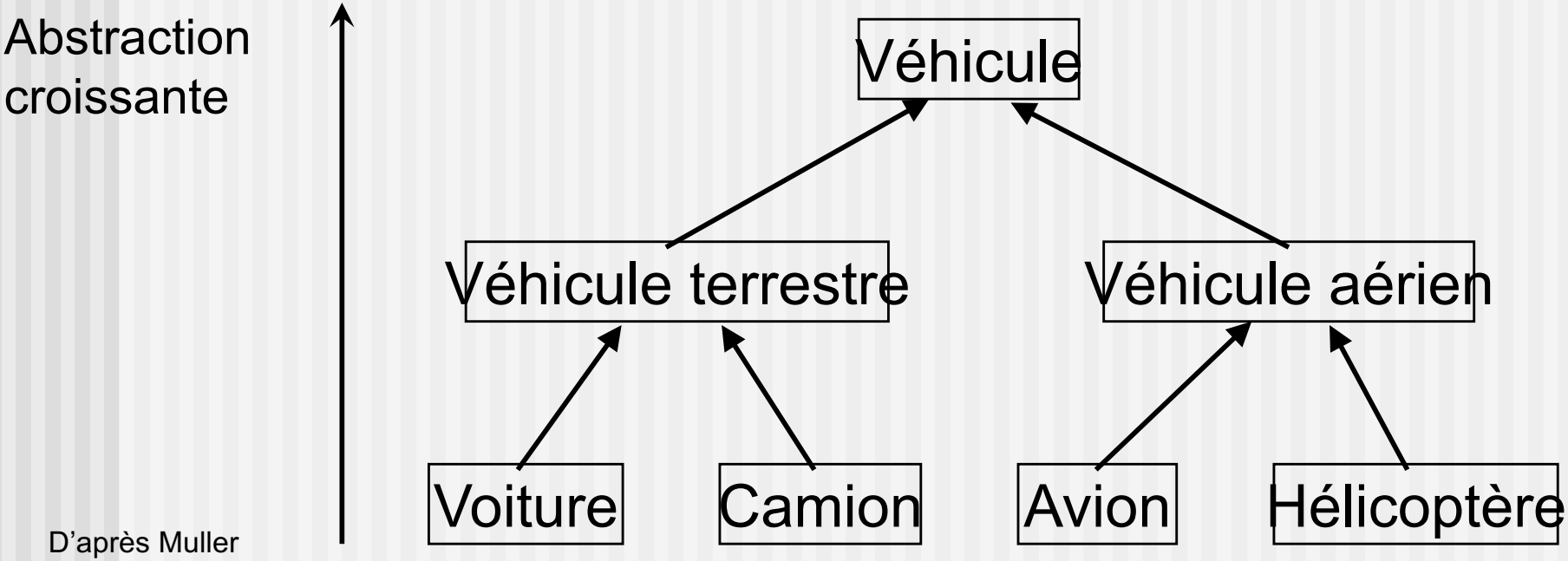
Spécialisation

- Extension cohérente (par niveaux) d'un ensemble de classes



Généralisation

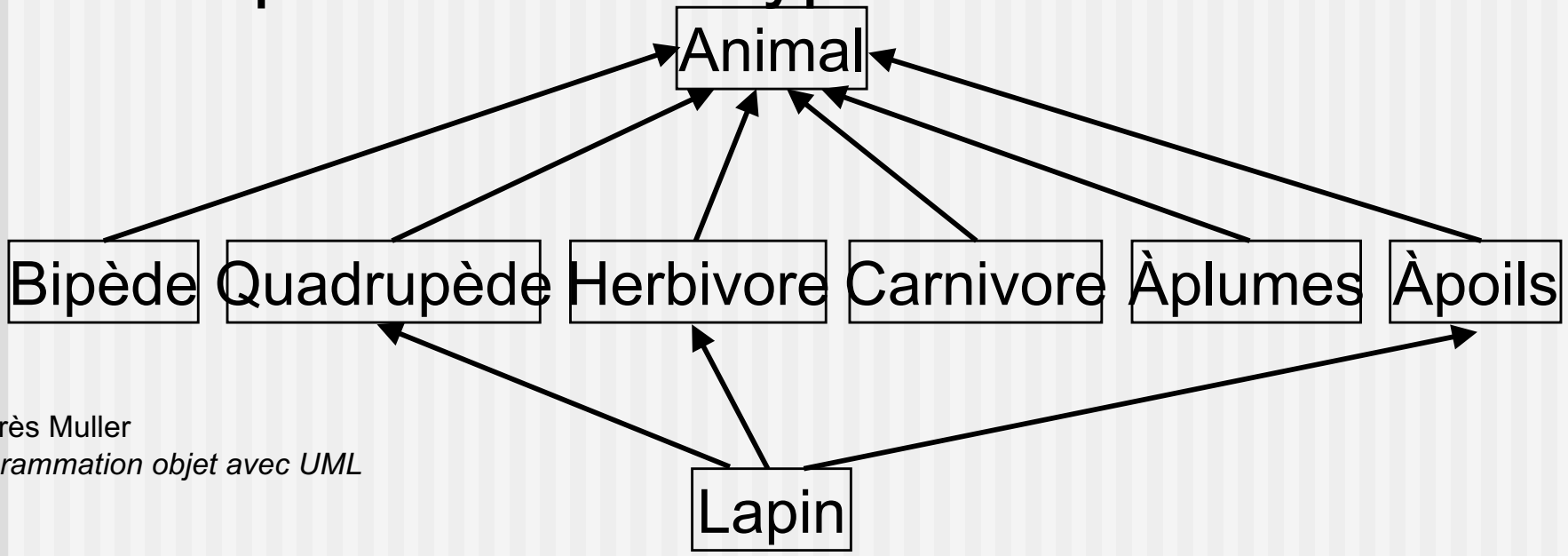
- Une super-classe est une abstraction de ses sous-classes



D'après Muller
Programmation objet avec UML

Généralisation multiple

- Un sous-type peut être la spécialisation de plusieurs sur-types



D'après Muller
Programmation objet avec UML

- Et le cochon ?

Héritage de code

- L'héritage (de code) permet :
  - La réutilisation de code existant (factorisation)
  - La spécialisation du code existant
  - La simplification du code par suppression des tests de typage, qui sont pris en charge par le système d'exécution en utilisant les liaisons dynamiques

Sous-classe (1)

- La définition d'une sous-classe fille par héritage lui permet d'hériter du code de sa classe mère

```
class Position {
  double   latitude;
  double   longitude;
  double   distance (Position p) {…}
  double   distancePôleNord () {…}
}
class Géodésique extends Position {
// Hérité: double   latitude, longitude;
// Hérité: double   distance (Position p) {…}
// Hérité: double   distancePôleNord () {…}
  double   altitude;
  double   modifieAltitude (double alt) {…}
}
```

Sous-classe (2)

- La sous-classe peut faire tout ce que peut faire sa sur-classe
- Elle peut le faire :
  - En utilisant le code de la sur-classe
  - En spécialisant le code de la sur-classe : redéfinition

Redéfinition

- La sous-classe peut redéfinir des méthodes de sa classe mère, sauf avis contraire de celle-ci

```
class Géodésique extends Position {
  double   altitude;
  double   modifieAltitude (double alt) {…}
  double   distance (Position p) {…}
// La distance entre un point géodésique et une
// position peut faire intervenir la hauteur du
// premier. On spécialise le code de la sous-classe.
}
```

Liaison dynamique

- Le choix de la méthode se fait par l'objet de manière dynamique à la réception d'un message
- Il y a conservation du comportement de l'objet indépendamment du type qui le référence

```
Animal a1, a2;
a1 = new Lapin (…);
a2 = new Tortue (…);
a1.bouge ();   // Méthode bouge() de Lapin
a2.bouge ();   // Méthode bouge() de Tortue
```

Liaison statique

- Le choix de la méthode se fait statiquement en utilisant le type déclaré de l'objet durant la phase de compilation
- On n'a pas conservation du comportement de l'objet indépendamment du type qui le référence

```
Lapin l  = new Lapin (…);
Animal a = l;
l.bouge ();     // Méthode bouge() de Lapin
a.bouge ();     // Méthode bouge() de
Animal
```

Surcharge (1)

- Définition de plusieurs méthodes de même nom mais avec des paramètres de types différents
  - Permet d'offrir le même service à partir de paramètres de types différents
  - Les méthodes de même nom dans le même contexte doivent réaliser des opérations de même nature

```
afficher: Int I ➪ Void
afficher: Char C ➪ Void
afficher: Float F ➪ Void
```

- Différent de la redéfinition !

Surcharge (2)

- La sélection d'une méthode surchargée se fait en fonction des types déclarés des paramètres et non en fonction des types réels

```
class Animal {…}
class Lapin extends Animal {…}
class Élevage {
  public void nourrir (Lapin l)  {l.mange (herbe);}
  public void nourrir (Animal a) {a.mange (viande);}
…
  Élevage élev = new Élevage ();
  Lapin  l = new Lapin ();
  Animal a = l;     // a et l référencent le même objet
  élev.nourrir (a);          // On lui donne de la viande
  élev.nourrir (l);          // On lui donne de l'herbe
```

Surcharge et héritage (1)

- La sous-classe peut surcharger des méthodes héritées de sa classe mère

```
class Géodésique extends Position {
  double   altitude;
  double   modifieAltitude (double alt) {…}

// Hérité: double   distance (Position p) {…}
  double   distance (Géodésique g) {…}
// La distance entre deux points géodésiques
// peut faire intervenir leurs deux hauteurs
}
```

Surcharge et héritage (2)

- La sélection de la méthode appelée s'effectue en deux temps :
  - En utilisant le type déclaré, on détermine une signature de méthode (réalisé à la compilation)
  - En partant du type réel, on remonte l'arbre d'héritage jusqu'à trouver une méthode satisfaisante (réalisé à l'exécution)

```
Position p = new Géodésique ();
Position q = new Géodésique ();
…
p.distance (q);
```

Surcharge et héritage (3)

- Détermination de la signature de la méthode :
  - **`Position::distance(Position)`**
- En partant du type réel, on remonte jusqu'à trouver une méthode satisfaisante
  - À partir de Géodésique :
    - **`Géodésique::distance(Position)`** n 'existe pas car toutes les **`Position`** ne sont pas des **`Géodésique`**
  - On remonte à Position :
    - on exécute **`Position::distance(Position)`**

Surcharge et héritage (4)

- Solution : redéfinir **`distance(Position)`** avec une méthode différenciant dynamiquement les **`Géodésiques`** des **`Position`**

```
class Géodésique extends Position {
  …
  double   distance (Position p) {
    if (p instanceof Géodésique)
      return (distance ((Géodésique) p)) // Transtypé
    else
      super (p);       // Appel de distance() de Point
  }
}
```

Héritage de code inutile

- On ne doit pas hériter de code et de données que l'on n'utilise pas

```
class PièceMécanique {          // Exemple à ne pas suivre
  String    nom;
  int       poids;              // Poids de la pièce
  boolean   estComposée;        // Vrai si pièce composée
  int       nombrePièces;       // Nombre de sous-pièces
  Pièce []  composantes;        // Tableau des composantes
  …
  int poidsTotal (Pièce p) {
    if (estComposée) {          // Test de type de pièce
      int       pt = 0;         // poids ne sert pas ici
      for (int i = 0; i < nombrePièces; i ++)
        pt += composantes[i].poidsTotal ();
      return pt;
    else return poids;          // poids ne sert qu'ici
```

Méthode abstraite

- On ne doit factoriser que le code nécessaire à l'ensemble des sous-classes
- Les méthodes du type abstrait dont l'implémentation ne sera définie que dans les sous-classes sont déclarées comme abstraites dans la classe mère
- Une classe possédant au moins une méthode abstraite doit être déclarée abstraite elle aussi

Classe abstraite

- Une classe déclarée comme abstraite ne peut jamais être instanciée
- Elle sert de conteneur à du code et des données qui seront utilisés par les sous-classes qui en hériteront
  - Uniformisation des noms
  - Factorisation du code

```
abstract class PièceMécanique {
  String    nom;                    // Factorisation
  abstract int poidsTotal ();   // Uniformisation
}
```

Implémentation

- Les sous-classes implémentent les méthodes abstraites de leur classe mère

```
class PièceMécaniqueSimple extends
PièceMécanique {
  int       poids;              // Poids de la
pièce
  int       poidsTotal () { // Implémentation
    return poids;
} }
class PièceMécaniqueComposée extends
PièceMécanique {
  int       nombrePièces;    // Nombre de
sous-pièces
  Pièce []  composantes;     // Tableau des
composantes
```

Héritage multiple

- Possibilité de faire hériter une classe du code de plusieurs sur-classes
- Complexité de la résolution des conflits en cas d'identité de noms de méthode
- On n'hérite souvent que de du code d'une seule classe, les autres étant purement abstraites ($\Rightarrow$ interfaces)

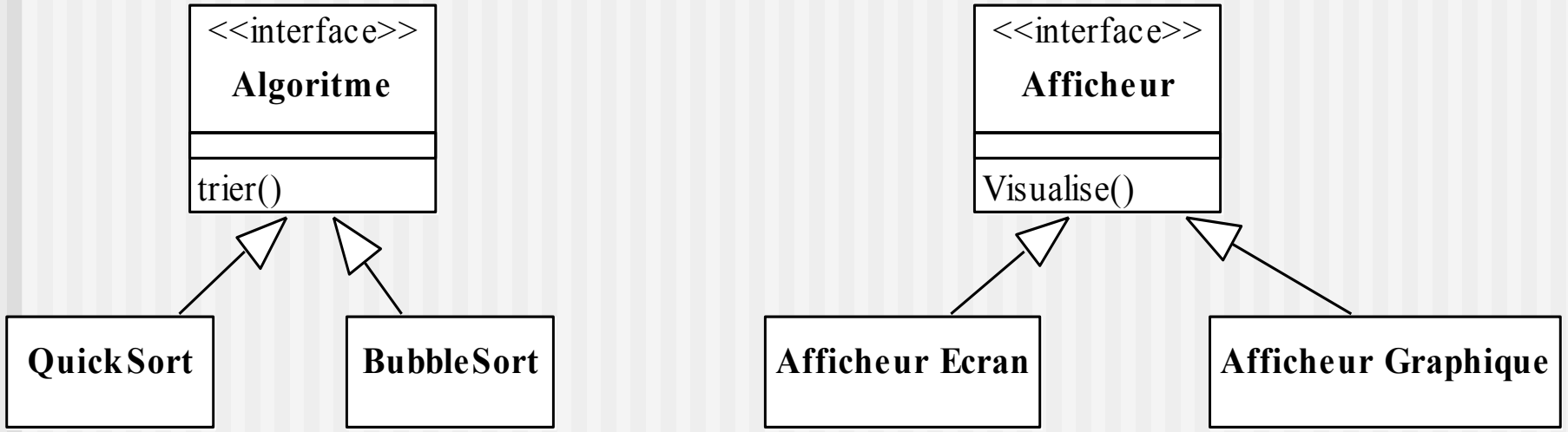
Héritage ou délégation

- L'héritage peut engendrer une prolifération de classes, surtout s'il y a composition de comportement (héritage multiple)
- La délégation est une alternative souple à l'héritage

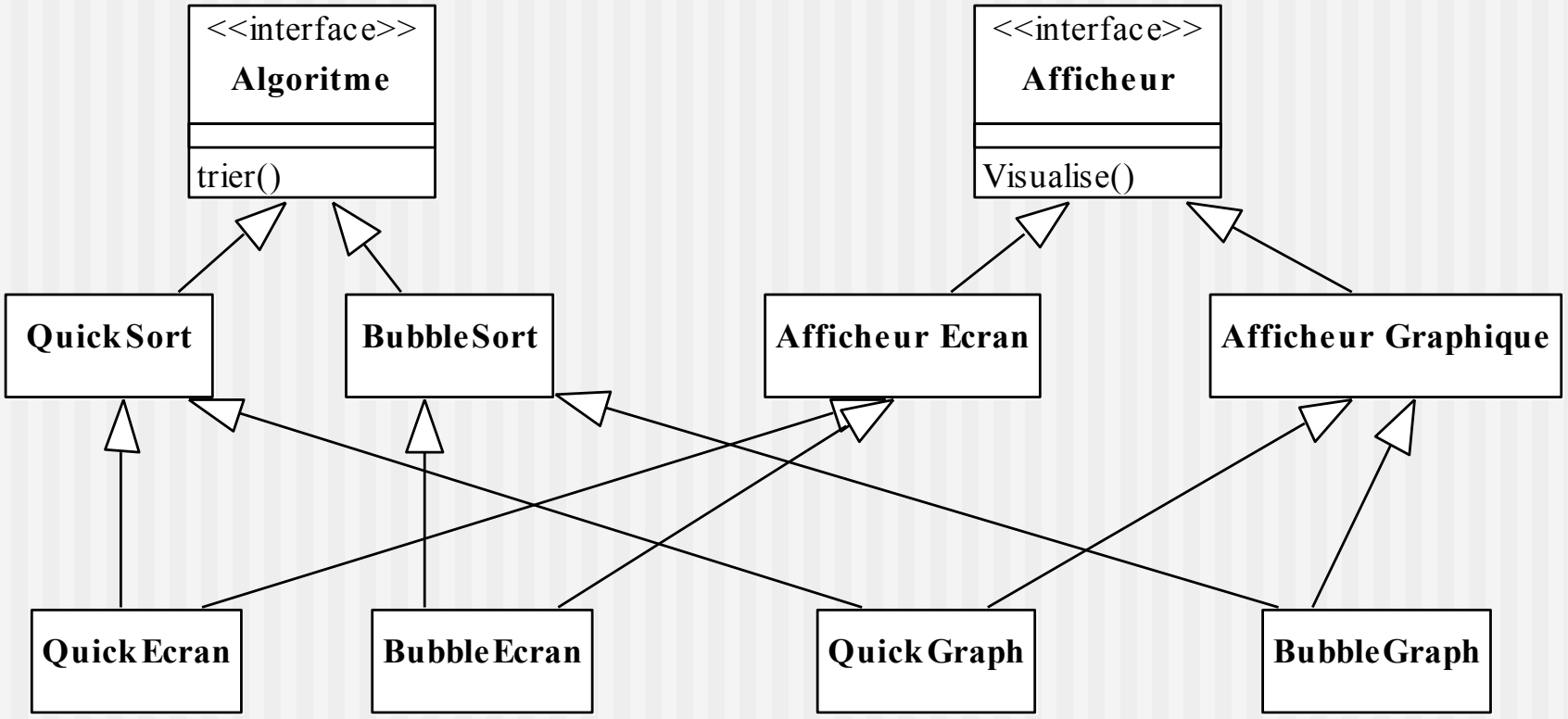
Exemple (1)

- On dispose d'une hiérarchie d'algorithmes de tri
- On dispose aussi d'une hiérarchie d'afficheurs
- On veut pouvoir composer un algorithme de tri avec un afficheur

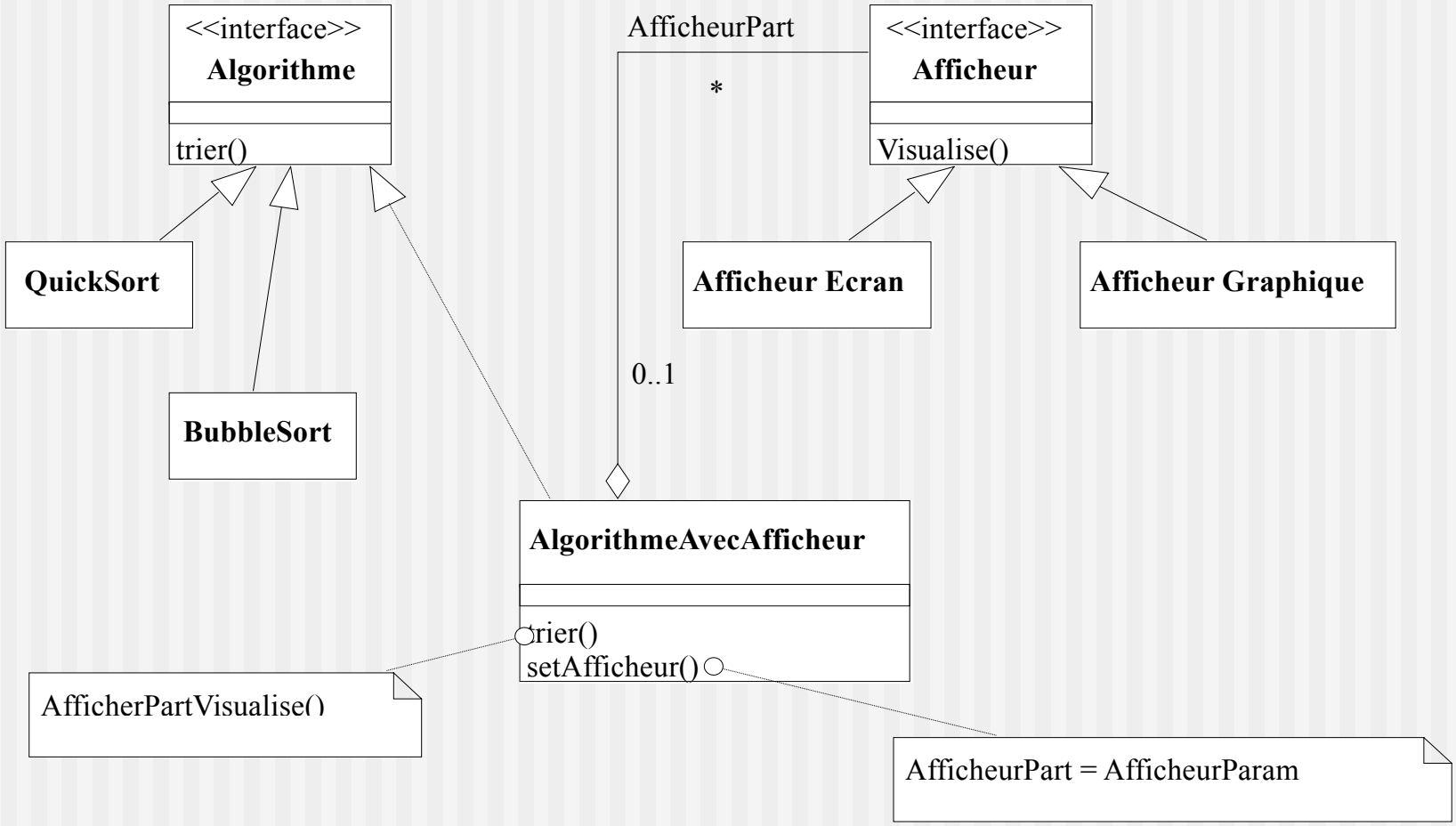
Exemple(2)


<<interface>>
Algoritme
trier()
QuickSort
BubbleSort
<<interface>>
Afficheur
Visualise()
Afficheur Ecran
Afficheur Graphique

Exemple(3) : Version héritage



Exemple(3) : Version avec délégation



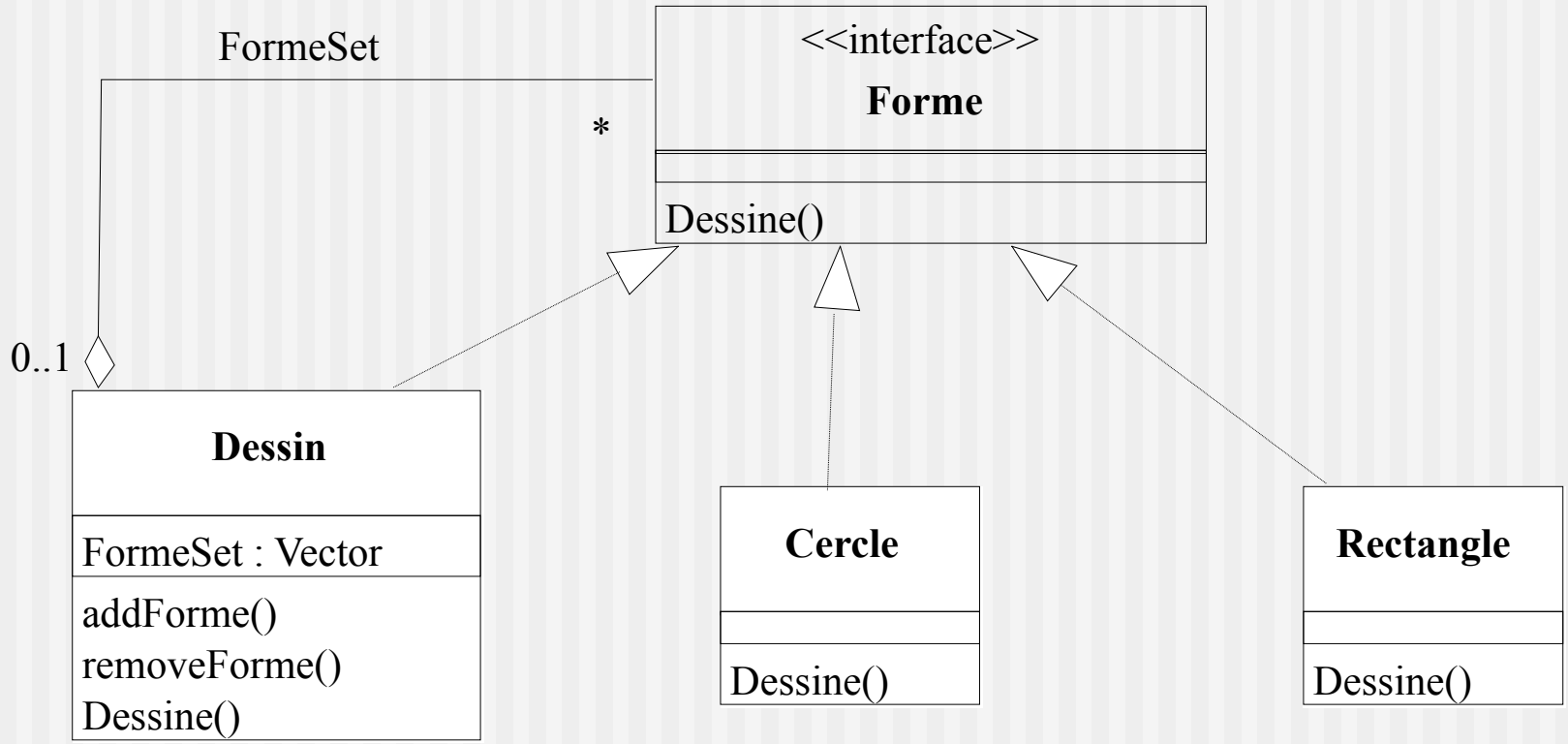
Polymorphisme

Le véritable intérêt de l'orienté objet

Polymorphisme

- Utilisation des relations types/sous-type pour obtenir l'unification des différentes classes
- Utilisation des liaisons dynamiques pour conserver la spécificité du comportement

Polymorphisme (exemple)



Polymorphisme (exemple)

- `Forme` est une interface qui admet plusieurs réalisation. On dit que `Forme` est **polymorphe**
- `Cercle` **et** `Rectangle` **sont des classes qui réalisent l 'interface** `Forme`
- `Dessin` est une classe composite qui contient des `Forme` et qui est une **`Forme`**

Le code de Dessin

```
Class Dessin implements Forme{
 private Vector FormeSet;

public void addForme (Forme f){
      FormeSet.AddElement(f);
   }
public void removeForme(Forme f){
    FormeSet.removeElement(f);
   }
public void Dessine(){
      for(int i = 0; i < FormeSet.size(); i++)
          ((Forme) FormeSet.elementAt(i)).dessine();
   }
}
```

Le code de l ’application

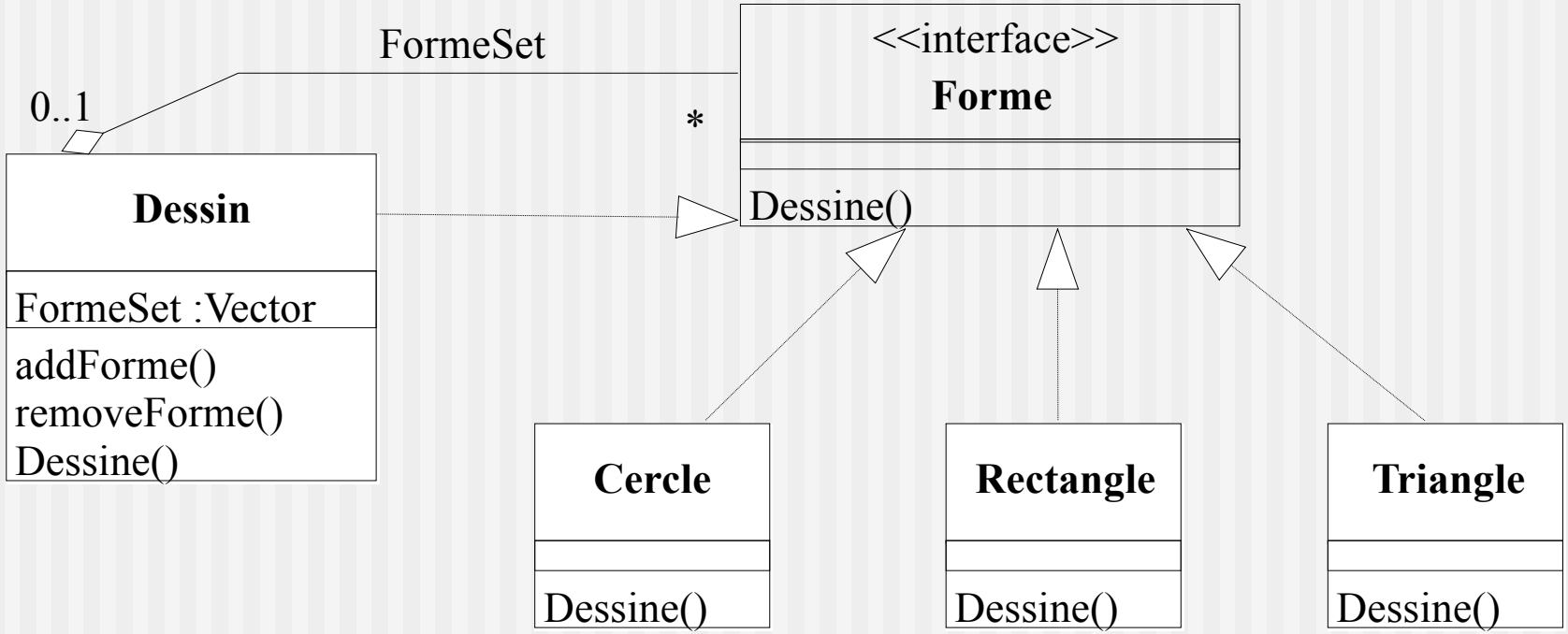
```
class Main {

static public void Main(String [] argv){
  Dessin d = new Dessin();
  d.addForme(new Cercle());
  d.addForme(new Rectangle());

                  ..............

}
```

Extension avec un triangle



Le nouveau code de l ’application

```
class Main {

static public void Main(String [] argv){
  Dessin d = new Dessin();
  d.addForme(new Cercle());
  d.addForme(new Rectangle());
  d.addForme(new Triangle());

                  ..............

}
```

Les techniques de réutilisation

Intérêt de l'héritage et du polymorphisme

Technique de Réutilisation

- Critère
  - Séparation de l'interface et de l'implémentation
- Différentes techniques
  - Boîte Noire : aucune adaptation à faire, pas de connaissance de l 'implémentation
  - Boîte Grise : adaptation au contexte par paramétrisation, pas de connaissance du source

Technique de réutilisation

- Autres techniques
- Boîte blanche : modification du code source, afin de l'adapter au contexte
  - Boîte en verre : consulter le code (l 'implémentation) sans le modifier

Les techniques d'extension et de réutilisation

- Boîtes Noires
  - Héritage public d'interface
- Boîtes Grise
  - Les poignées
  - La délégation

Héritage public

```
class AlgorithmeTri
{
    virtual void Tri()=0
};
class AlgorithmeTriAffiche : public
AlgorithmeTri
{
     private void afficheEtape(){……}
     virtual void Tri() {
          ……
          afficheEtape();
     }
}
```

Peut engendrer de la duplication de code

Les poignées

```
Class AlgorithmeTri {
    protected:
      virtual void poignéeAffiche(){}
    public
       void affiche() {……. poignéeAffiche();……}
}
class AlgorithmeTriAffiche:public
AlgorithmeTri{
     protected:
       virtual void poignéeAffiche(){……}}
```

On diminue la duplication de code, mais le nombre de classe augmente.
Il est nécessaire de prévoir toutes les extensions de la classe lors de la conception

La délégation

```
class AlgorithmeTri {
     Afficheur *affiche;
  public:
      void Tri() {  …… affiche-
>visualise();}
 };
       void setAfficheur(Afficheur *aff) {
affiche = aff;}
}
class Afficheur{
    public:
      virtual void visualise(){……}
}
```

On minimise la duplication de code et le nombre de classes

Exercices

- Un analyseur doit lire un message et déclencher l'action associée au message
  - Connect ➪ lancer une nouvelle connexion
  - Disconnect ➪ terminer la connexion
- Un nouveau message doit être pris en compte
  - Print ➪ affiche l'état des connexions

Héritage simple (1)

```
class Parser
{
   virtual boolean parse (String s){
      if( s==« Connect »){
          Connexion::NewConnexion();
           return true;
        }
      if(s==« Disconnect »){
          Connexion::DetruireConnexion();
          return true;
        }
      return false;
   }
};
```

Héritage simple (2)

```
class ParserPrint : public Parser
{
     virtual boolean parse(String s) {
          if(s==« Print ») {
               Connexion::Print();
               return true;
          }
      return Parser::parse(s);
    }
};
```

Poignée (1)

```
class Parser
{
    virtual protected boolean  handle(String s){
        return false;
    }
    virtual boolean parse (String s){
        if( s==« Connect  »){
            Connexion::NewConnexion();
            return true;
        }
        if(s==« Disconnect »){
            Connexion::DetruireConnexion();
            return true;
        }
        return handle(s);
    }
```

Poignée (2)

```
class ParserPrint : public Parser
{
     virtual boolean handle (String s) {
         if(s==« Print ») {
            Connexion::Print();
            return true;
        }
       return false;
    }
};
```

Délégation (1)

```
Class ParserInterface // une interface en
C++ {
     virtual boolean parse(String S) = 0;
}
class Parser Default: public ParserInterface
{
   virtual boolean parse (String s){
     if( s==« Connect »){
          Connexion::NewConnexion();
           return true;
      }
      if(s==« Disconnect »){
          Connexion::DetruireConnexion();
          return true;
```

Délégation (2)

```
class ParserDecorateur: public ParserInterface {
     private:
          ParserInterface component;
     public:
           ParserDecorateur(ParserInterface pi){
             component = pi;
           }
            virtual boolean parser (String s) {
              return component.parser(s);
            }
};
```

Délégation (3)

```
class ParserPrint: public ParserDecorateur
{
  public:
  ParserPrint(ParserInterface
pi):ParserDecorateur(pi)
  {};
  virtual boolean parser (String s) {
      if(s==« Print ») {
         Connexion::Print();
         return true;
      }
      return ParseDecorateur::parser(s);
  }
}
```

Approfondissement de l ’héritage

Les relations entre:

1- interfaces

2- classes

3- classes et interfaces

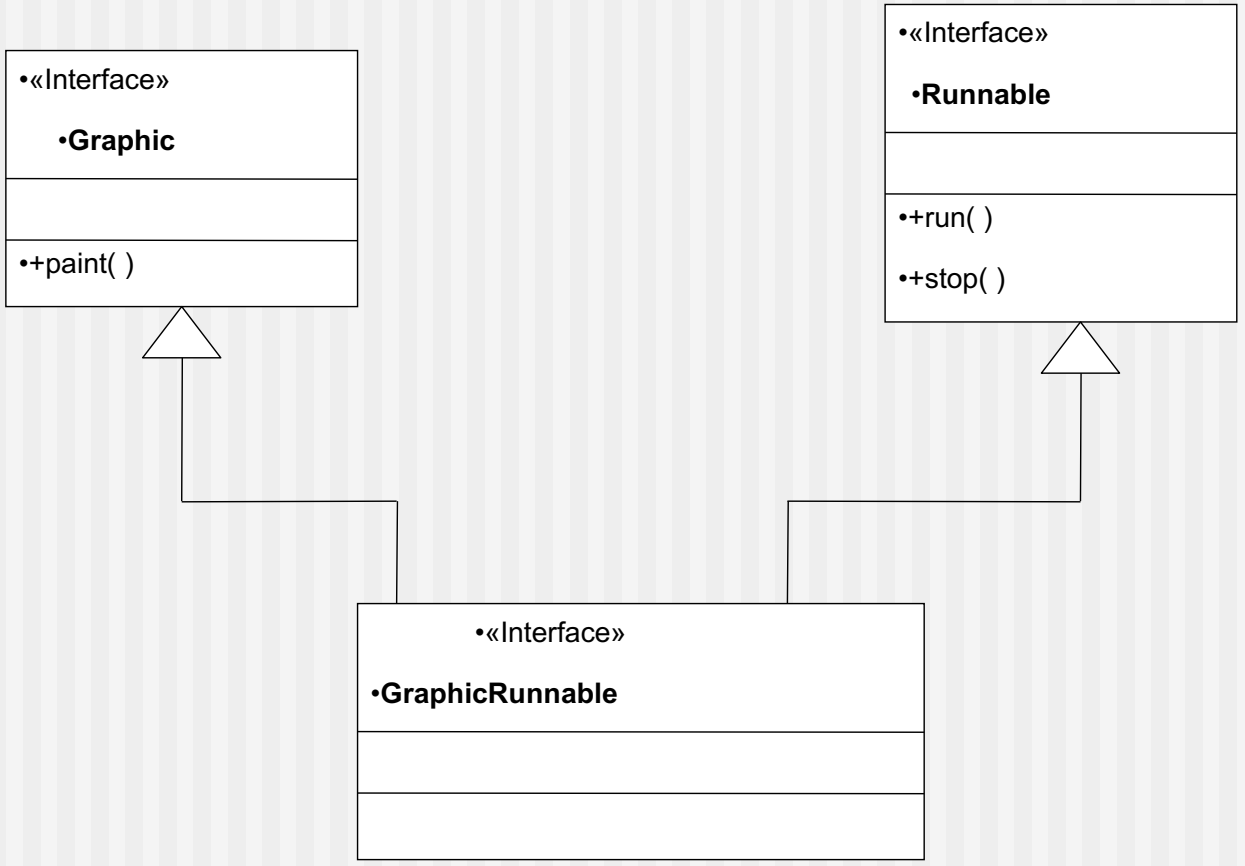
Spécification/Implémentation

- Interface type uniquement
- Classe ambiguë à la fois le type et le code
- Héritage public ambiguë à la fois type et code
- Héritage privé non-ambiguë, uniquement le code

Relations statiques

- Relation entre interfaces.
  - Héritage public (généralisation/spécialisation)
- Relation interface/classe
  - Concrétisation
- Relation classes/classes
  - Héritage privé

Relations entre Interface



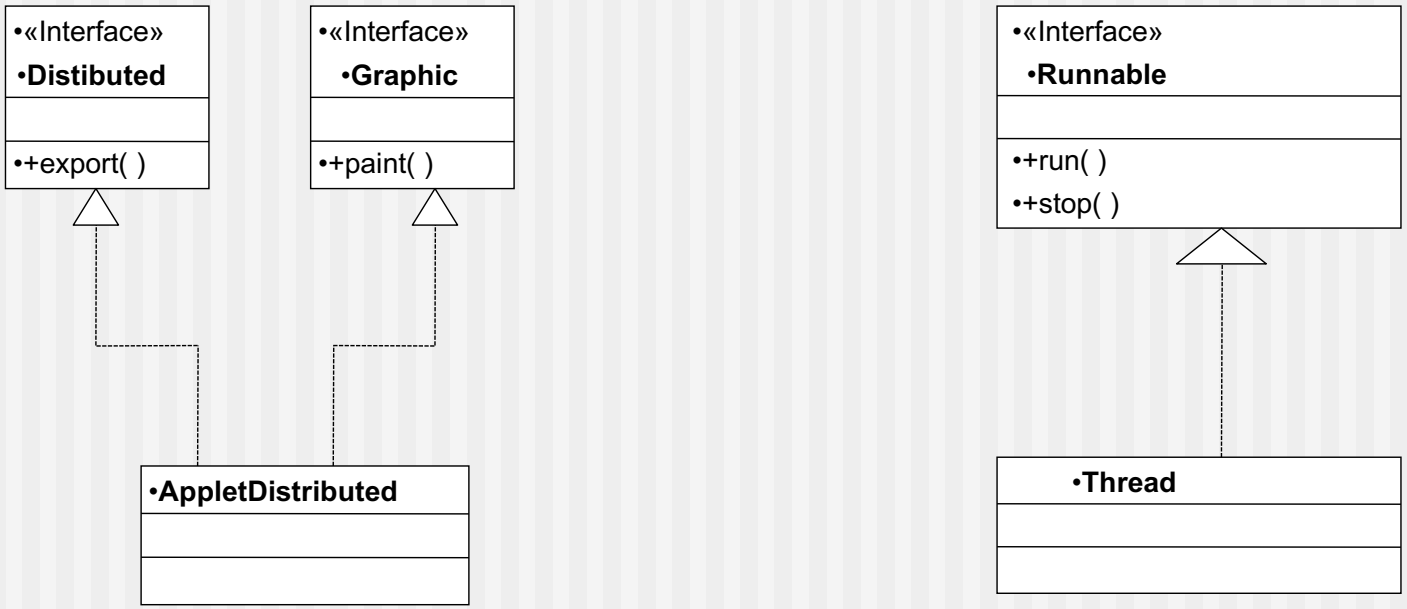
Relations entre interfaces

- L ’interface GraphicRunnable peut tenir les rôles de Graphic et de Runnable

Relations entre interfaces en C++

```
class Graphic
{
  public:
  virtual void paint() = 0;
};
class Runnable
{
 public:
   virtual void run() = 0;
   virtual void stop() = 0;
};
class GraphicRunnable: public Graphic, public Runnable
{}
```

Relation interface/classe



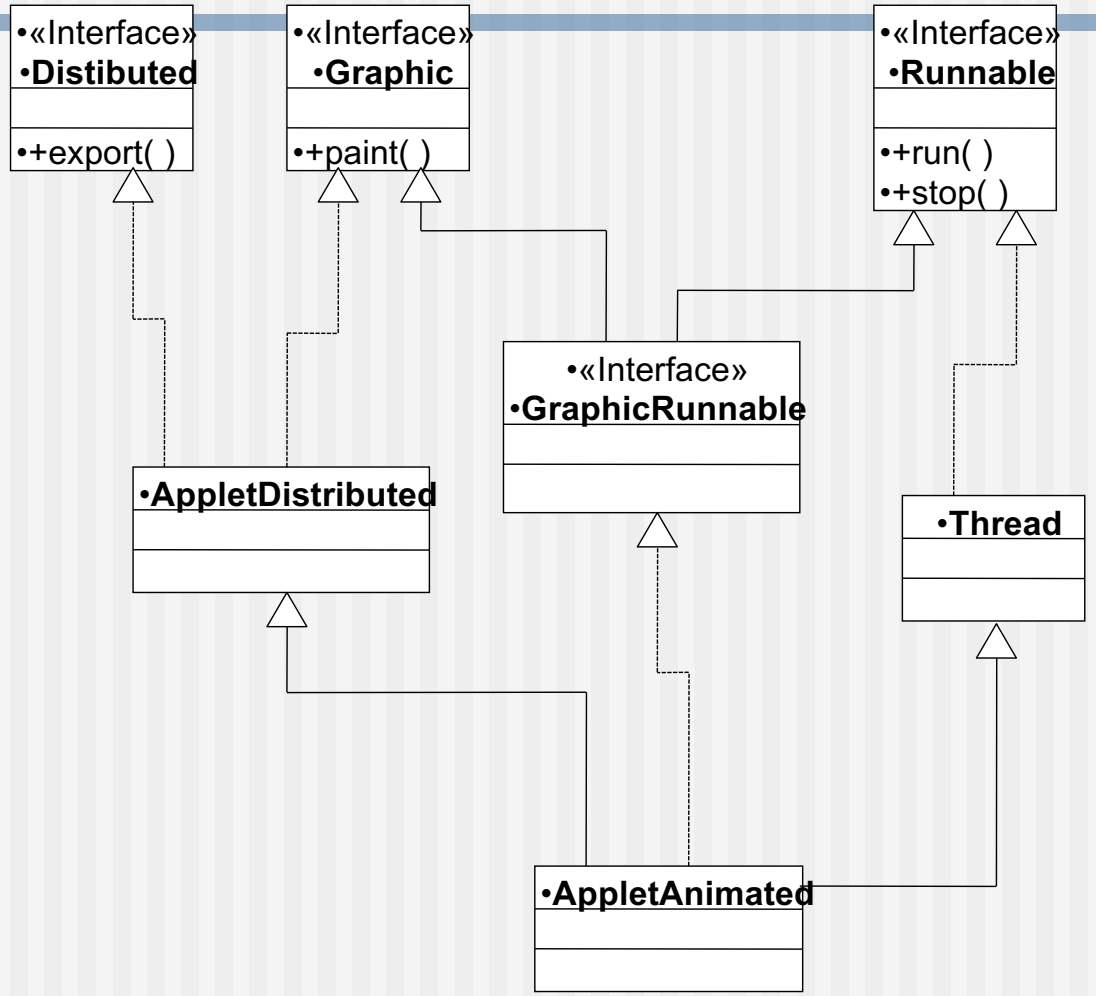
Relation interface/classe

- La classe AppletDistributed concrétise les classe Graphic et Distributed
- La classe Thread concrétise la classe Runnable

Relation interface/classe en C++

```
class Distributed
{public:
 virtual void export() = 0;
};
class AppletDistribué:public Applet, public Distributed
{ public:
  virtual void paint(){ implémentation de paint };
  virtual void export(){ implémentation de export};
};
class Thread:  public Runnable
{public:
  virtual void run()  { implémentation  de run };
  virtual void stop() { implémentation  de stop };
```

Relations classe/classe



Relations classe/classe

- La classe AppletAnimated hérite des classes AppletDistributed et de Thread

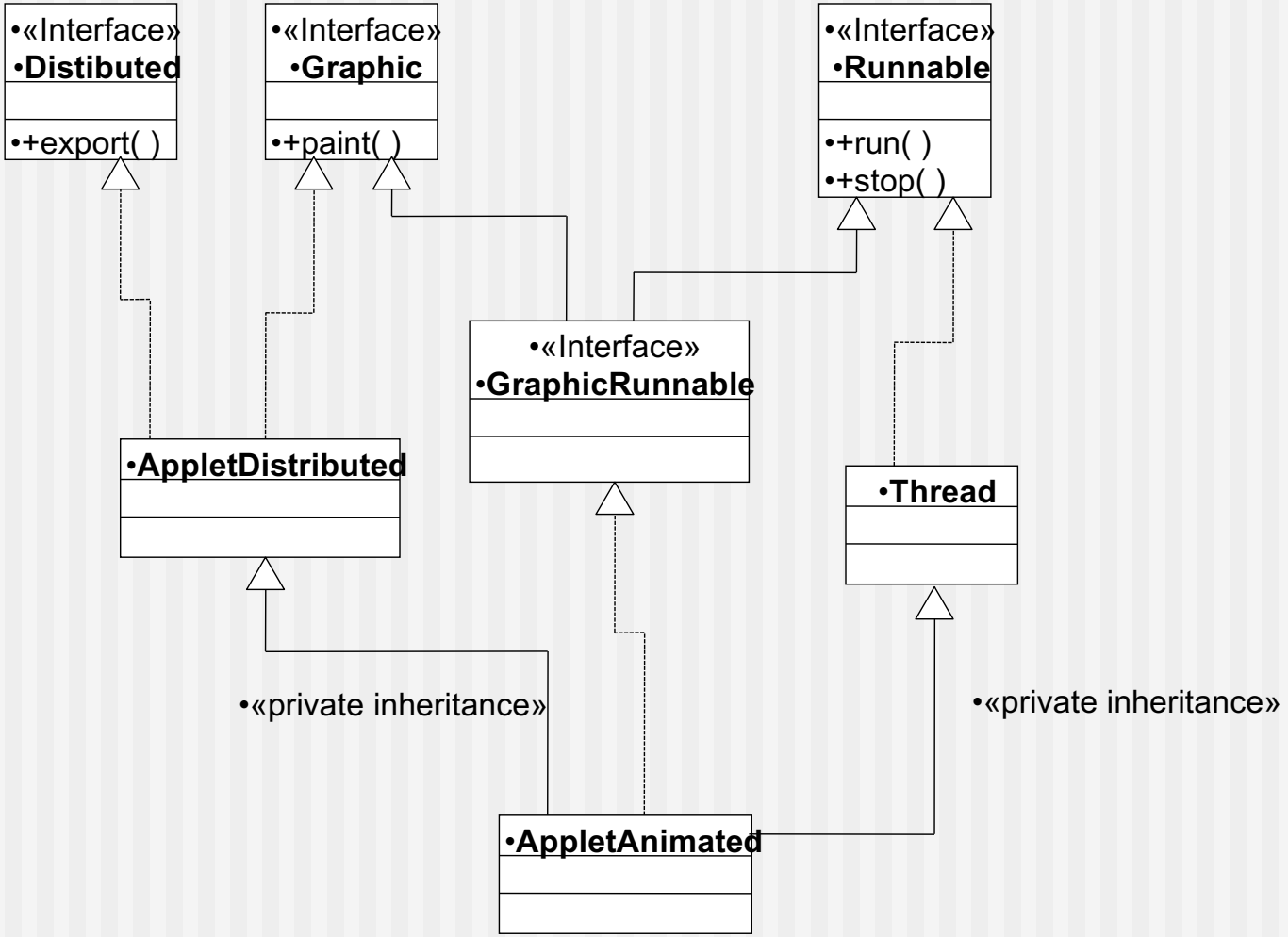
*Le problème est que les instances de **AppletAnimated** peuvent être distribuées. Comme l'héritage est une relation publique par transitivité on obtient des comportements supplémentaires non pertinents.*

Traduction en C++

```
Class AppletAnimated: public GraphicRunnable,
                                public AppletDistributed, public Thread
 {};


void function()
{
   Distributed *dis = new AppletAnimated();
}
```

Héritage privé



Héritage privé

- Si on ne veut que le code, et pas le type il faut utiliser l ’héritage privée.
- Relation entre classes donc statique

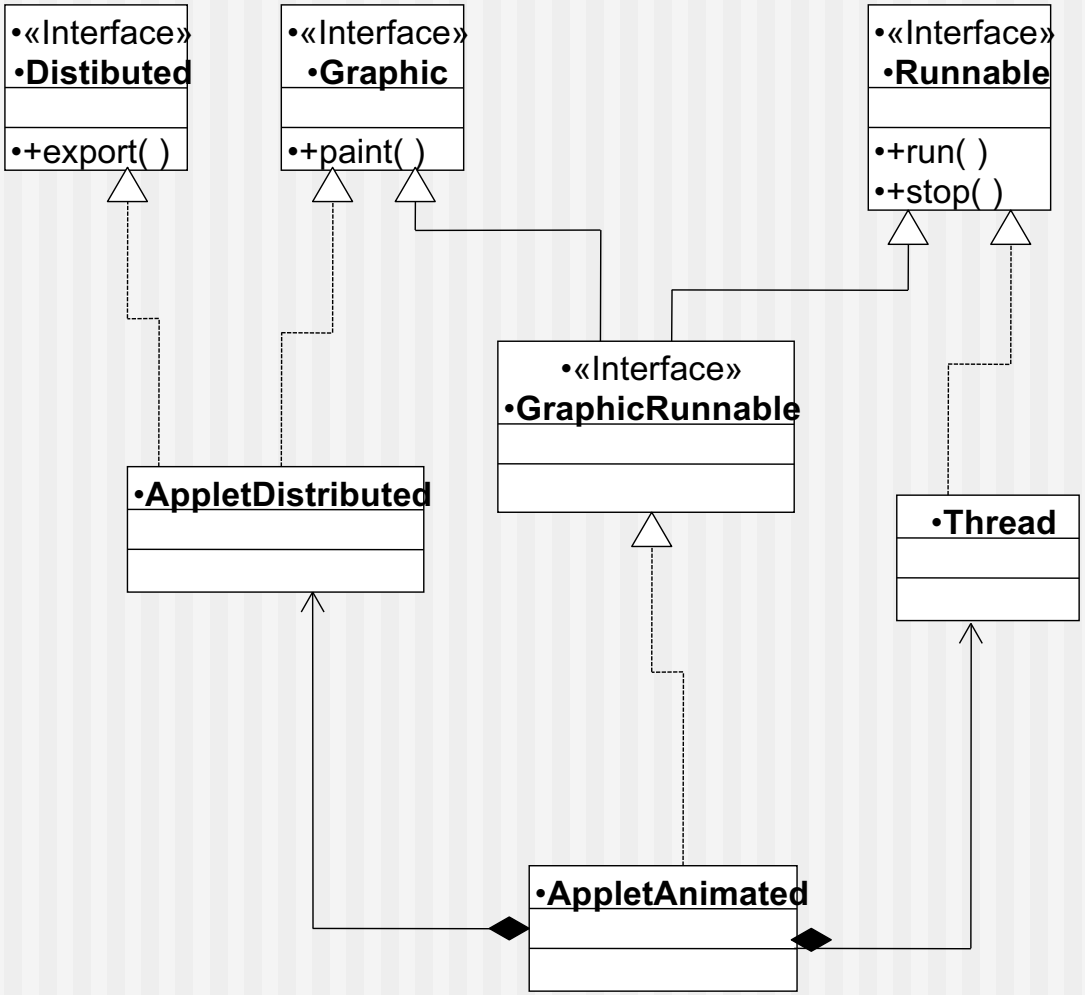
Héritage privée en C++

```
Class AppletAnimated: public GraphicRunnable,
                      private AppletDistributed,
                      private Thread
 {public:
  virtual void paint(){AppletDistributed::paint()};
  virtual void run  (){Thread::run()};
  virtual void stop (){Thread::stop()};
};
void function()
{
   Distributed *dis = new AppletAnimated();
   // erreur de compilation.
}
```

Délégation = Héritage privé

- Si on ne veut que le code, et pas le type il faut utiliser la délégation.
- Délégation statique
- Délégation dynamique

Délégation statique = Héritage privé



Délégation statique en C++

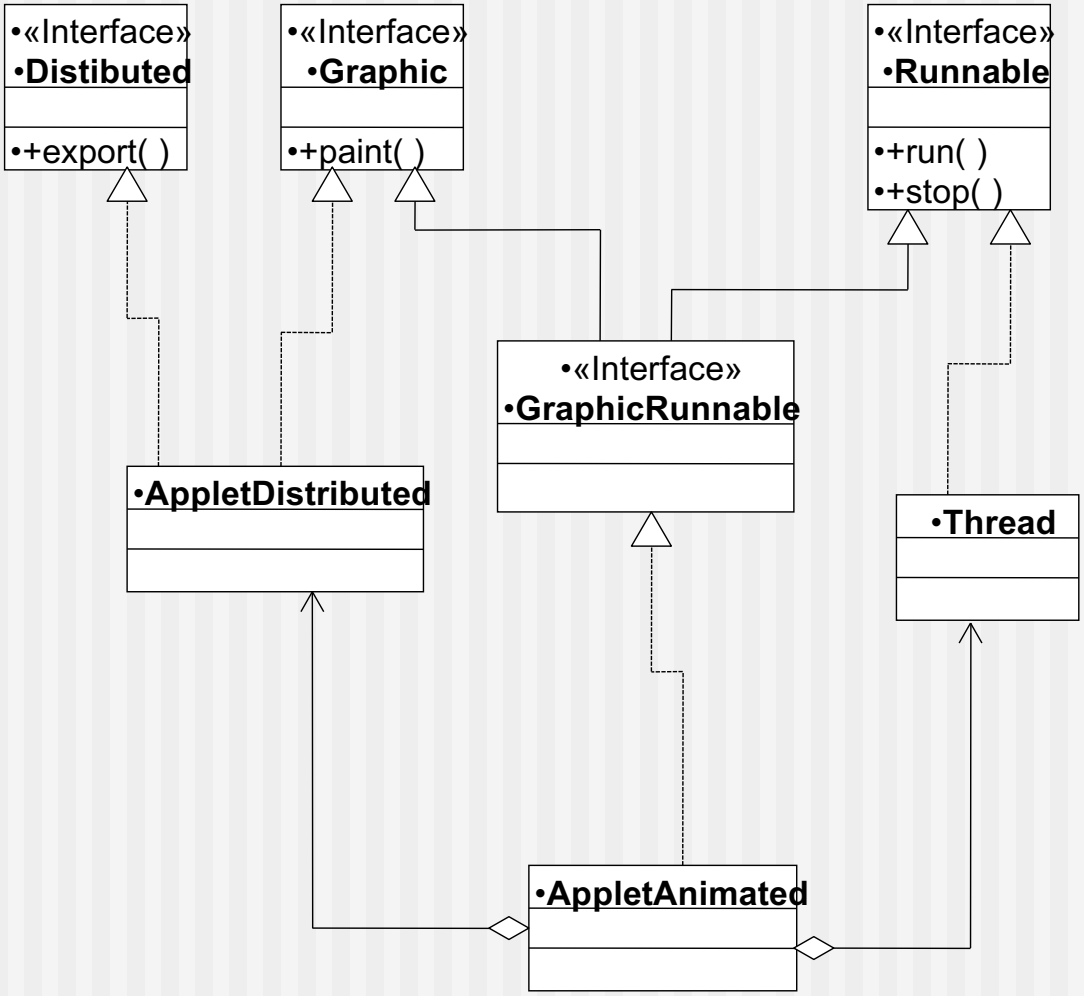
```
Class AppletAnimated: public GraphicRunnable,
{
   private:
   AppletDistributed appletDeleguated;
   Thread             threadDeleguated;
   public:
   virtual void paint(){appletDeleguated.paint()};
   virtual void run  (){threadDeleguated.run()};
   virtual void stop (){threadDeleguated.stop()};

};
```

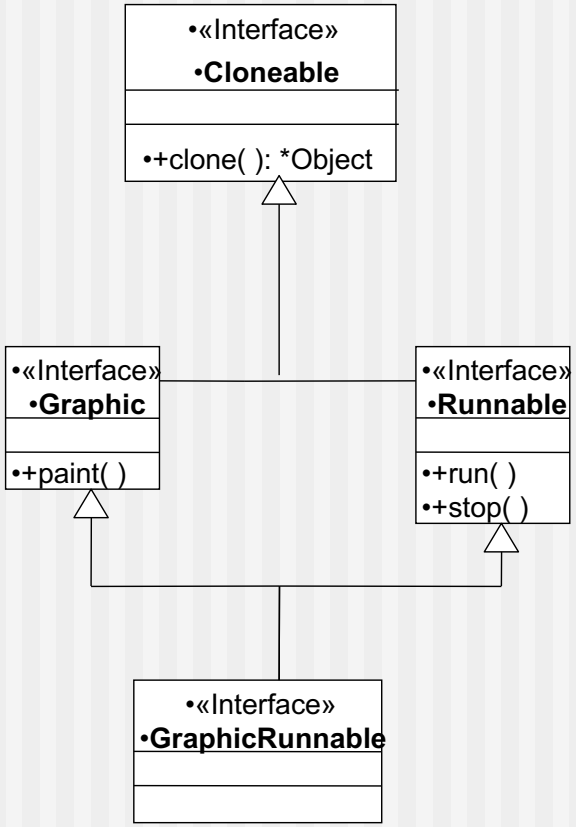
Délégation dynamique en C++

```
Class AppletAnimated: public GraphicRunnable,
 {
  private:
   Graphic    *graphicDeleguated;
   Runnable  * threadDeleguated;
  public:
  AppletAnimated(Graphic *g,Runnable *r)
  {
      graphicDeleguated   = g;
      runnableDeleguated = r;
}
  virtual void paint(){appletDeleguated->paint()};
  virtual void run  (){threadDeleguated->run()};
  virtual void stop (){threadDeleguated->stop()};
};
```

Le diagramme du code précédent



Une modification du schéma



Délégation dynamique en C++

```
Class AppletAnimated: public GraphicRunnable,
 {private:
   Graphic    *graphicDeleguated;
   Runnable  * threadDeleguated;
public:
  AppletAnimated(Graphic g,Runnable r)
  {
      graphicDeleguated   = g.clone();
      runnableDeleguated = r.clone();
}
  virtual void paint(){appletDeleguated->paint()};
  virtual void run  (){threadDeleguated->run()};
  virtual void stop (){threadDeleguated->stop()};
};
```

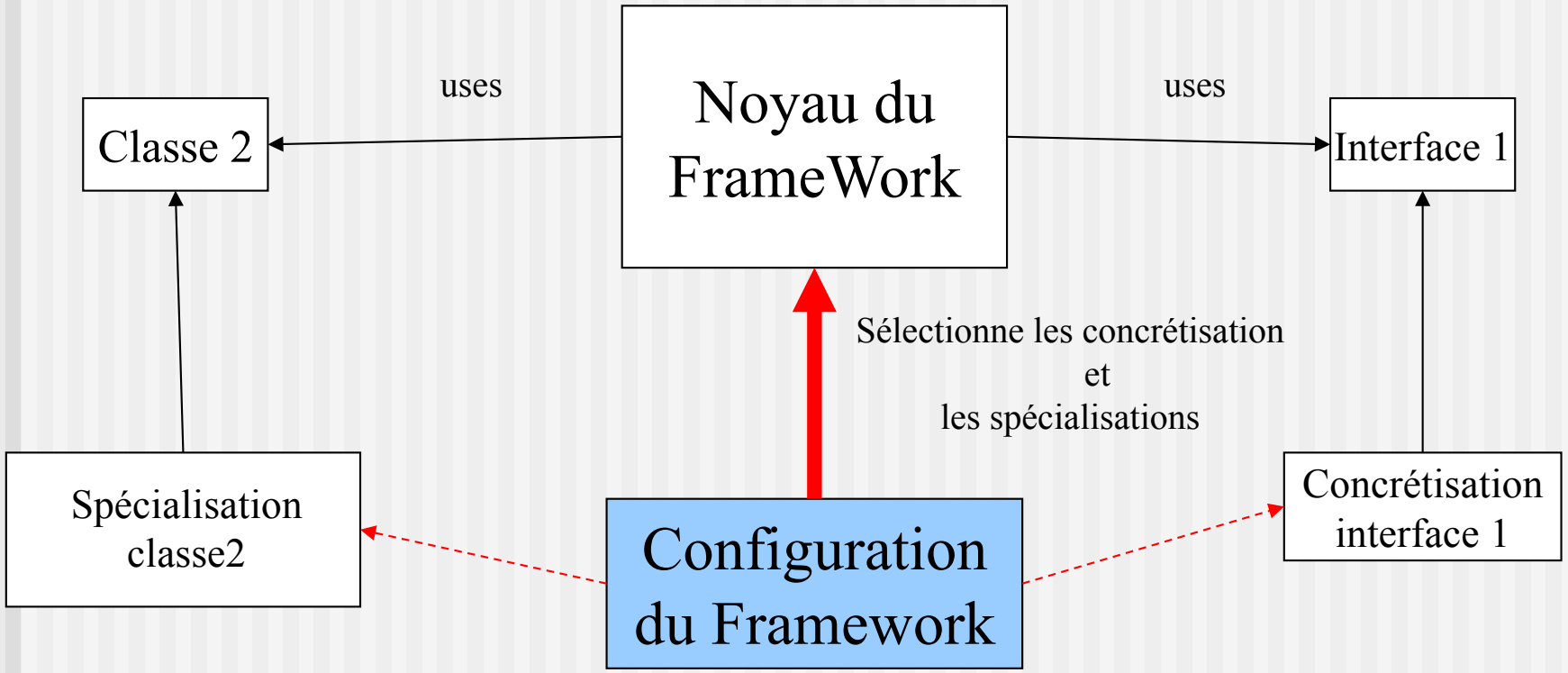
La réutilisation

- Réutilisation d ’architecture (framework)
- Réutilisation de composant d’architecture (pattern)
- Réutilisation de rôles (interface)
- Réutilisation de classes (bibliothèque)

Framework

- Framework
  - Un ensemble coopérant de classes, certaines peuvent êtres abstraites, cet ensemble est réutilisable pour certains type de problème.

Un framework



Patterns

- Proposer une micro architecture pour résoudre un problème donné.
  - Les patterns sont les principaux composants des frameworks.
  - Créationnels, Comportementaux, Structuraux

Pattern

- L ’initialisation est une abstract factory.

```
Class Configuration
{
      Classe2* creerClasse2()
       {
            return new
SpécialisationClasse2();
       }
      Interface1* creerInterface1()
       {
            return new
ConcretisationInterface1();
```

Une autre implémentation !

```
class Cloneable{
  public:
   virtual Cobject* clone() = 0;
};
class Classe2: public Cloneable {
   public:
     virtual Cobject* clone(){
          // allocation memoire spécifique;
     this-> StateCopy();
     }
   protected:
    void StateCopy(){ // Copie de l 'état}
};
```

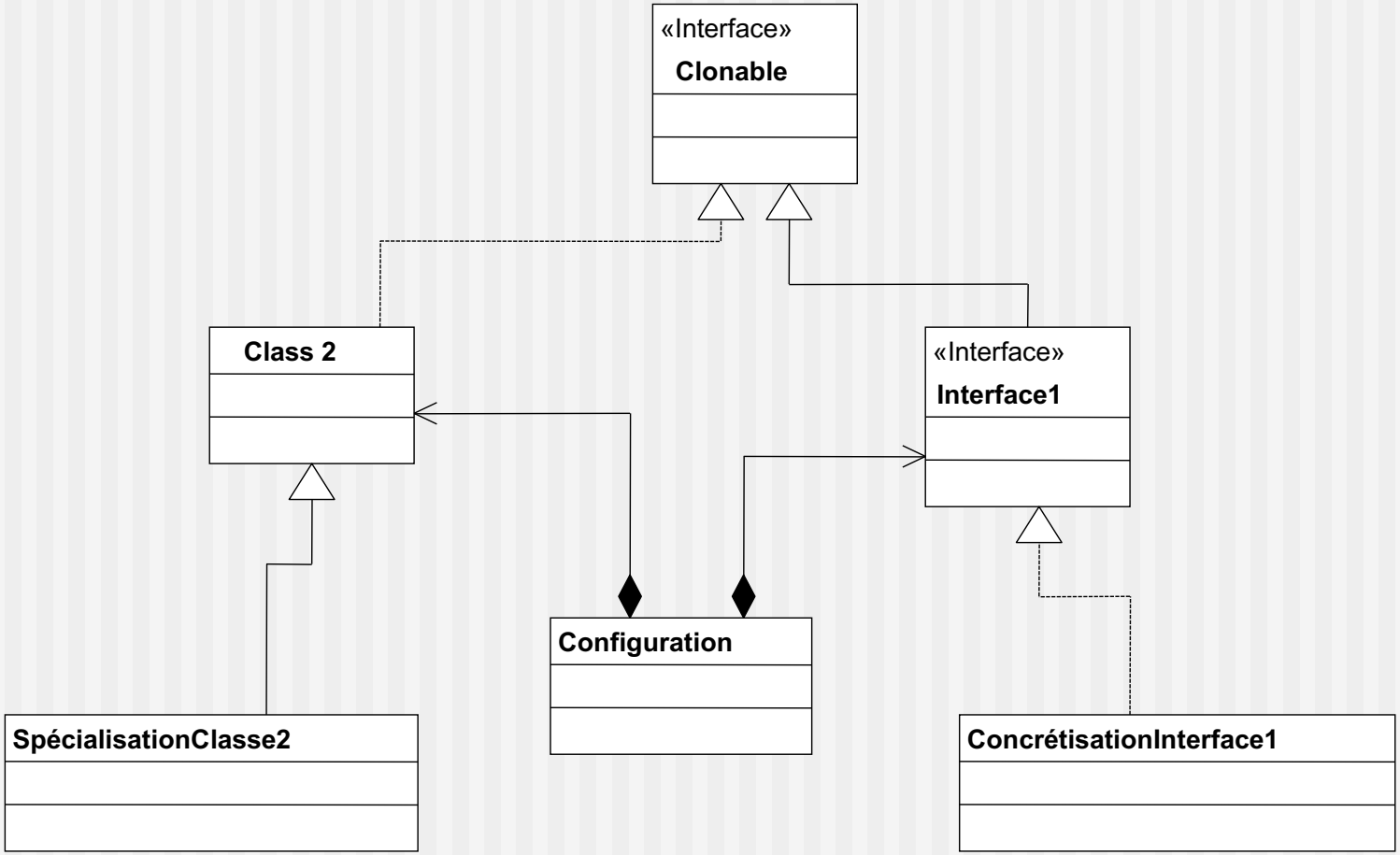
Une autre implémentation !

```
class SpecialisationClasse2: public Classe2
{
   public:
   virtual Cobject* clone(){
   SpecialisationClasse2 *tmp=new
SpecialisationClasse2();
   this.StateCopy();
  }
  protected:
  void StateCopy(){ Classe2::StateCopy();…….}
};
```

Une autre implémentation !

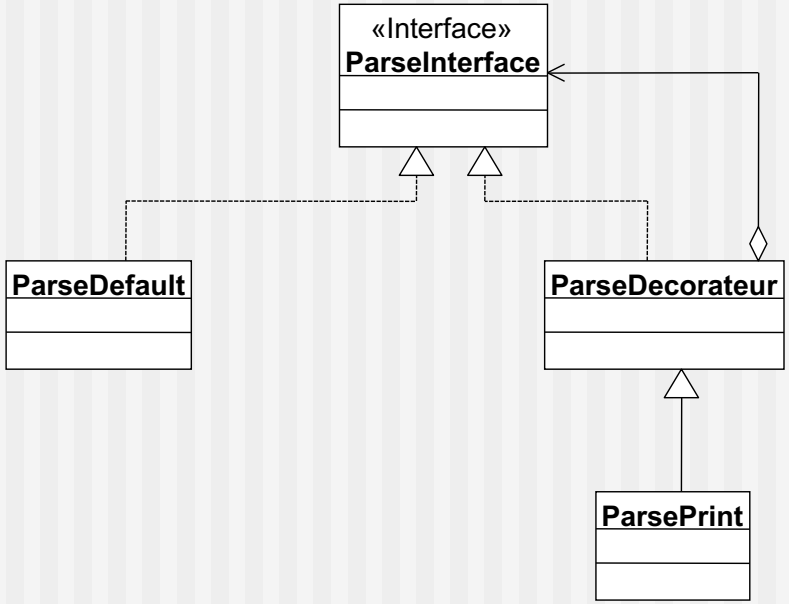
```
class Configuration {
private:
      Classe2     *protoClasse2;
      Interface1 *protoInterface1;
public:
      Configuration (Classe2 *c,
Interface1 *i){
      protoClasse2    = c;
      protoInterface1 = i;
   }
Classe2* creerClasse2() {
    return (dynamic_cast<Classe2 *>)
c.clone();
   }
```

Une autre implémentation !



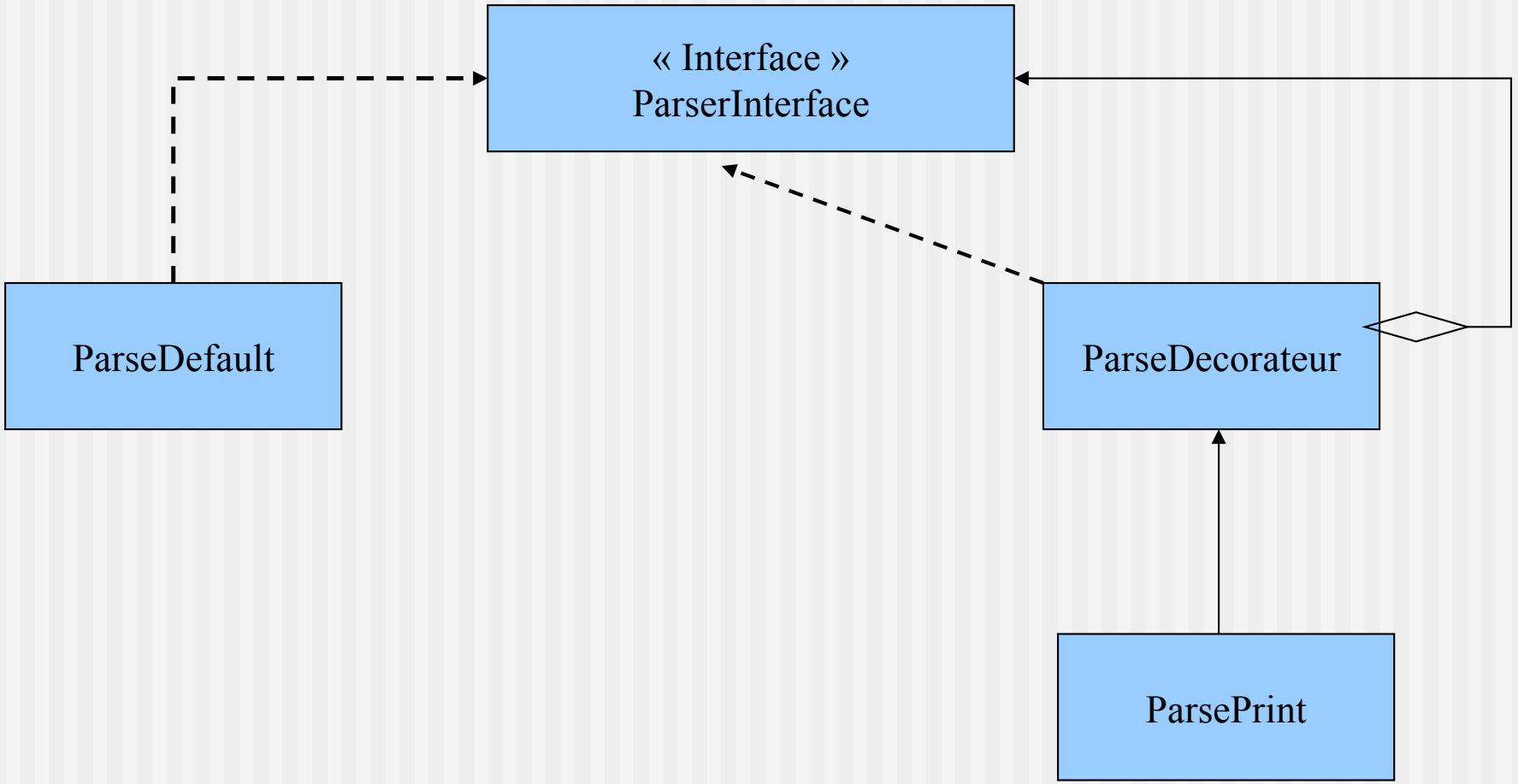
Pattern

- Un autre pattern le décorateur



Pattern

- Un autre pattern le décorateur



Relations

- Relation classes/classes
  - Héritage privé/ Délégation
  - Réutilisation de code
- Relation interface/classe
  - Concrétisation
  - Adaptation d'application
- Relation entre interfaces.
  - Héritage public (généralisation/spécialisation)
  - Réutilisation de partie d'application par une nouvelle application

Concevoir pour réutiliser

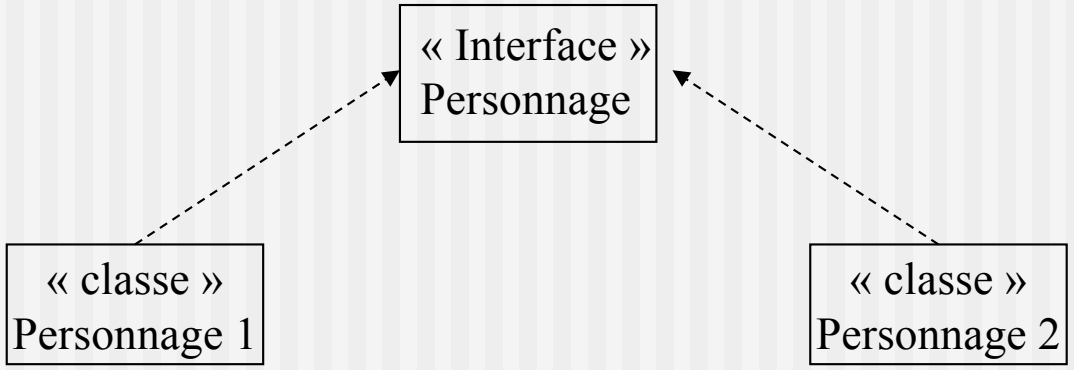
- Définir des rôles qui soient réutilisables
  - indépendant de l'application
  - représentant correct de l'abstraction
    - interface complète et minimale

Favoriser la réutilisation en phase d'analyse.

Permettre de conserver les réalisations de rôles

Exemple

- L ’interface *Personnage*



Pour un autre jeux si le rôle est correct, il peut être réutiliser

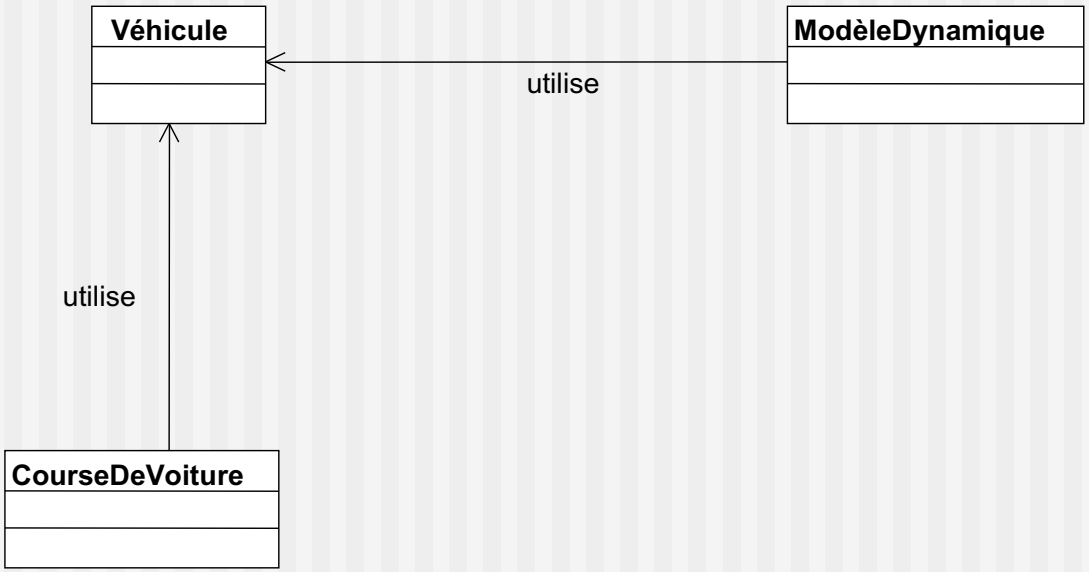
On peut jouer avec les deux personnages existants

Concevoir pour réutiliser

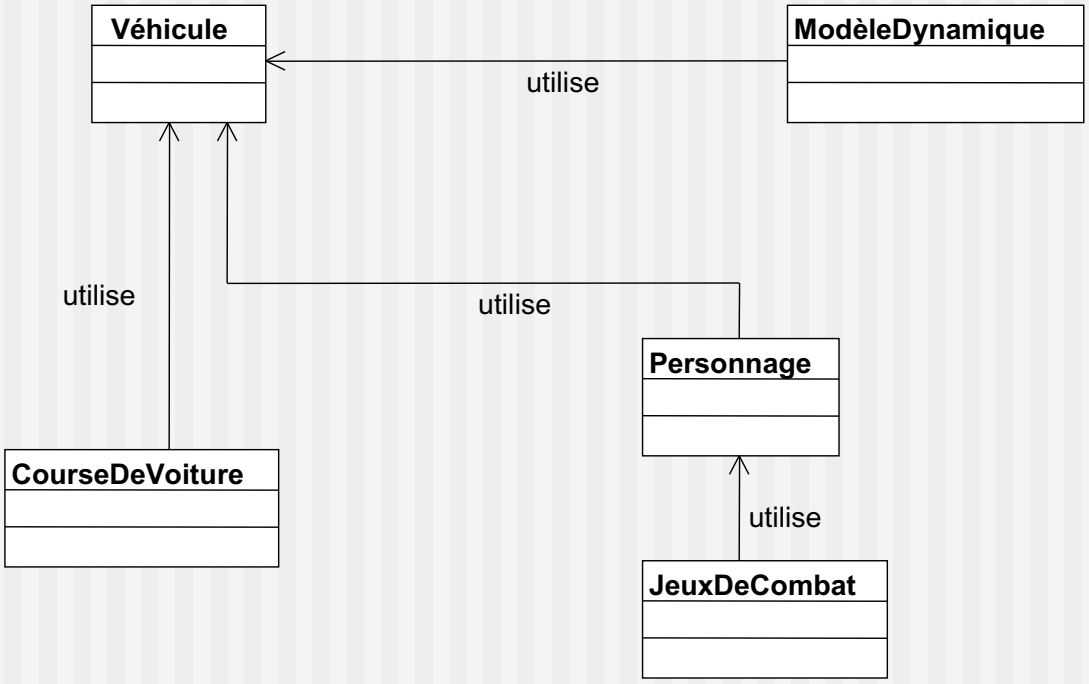
- Identifier clairement les relations de dépendances entre les composants du framework et les types
  - les sous-types bénéficient alors du composant dépendant du sur-type et de lui seul.

Diminue le couplage entre composant
Favorise la réutilisabilité

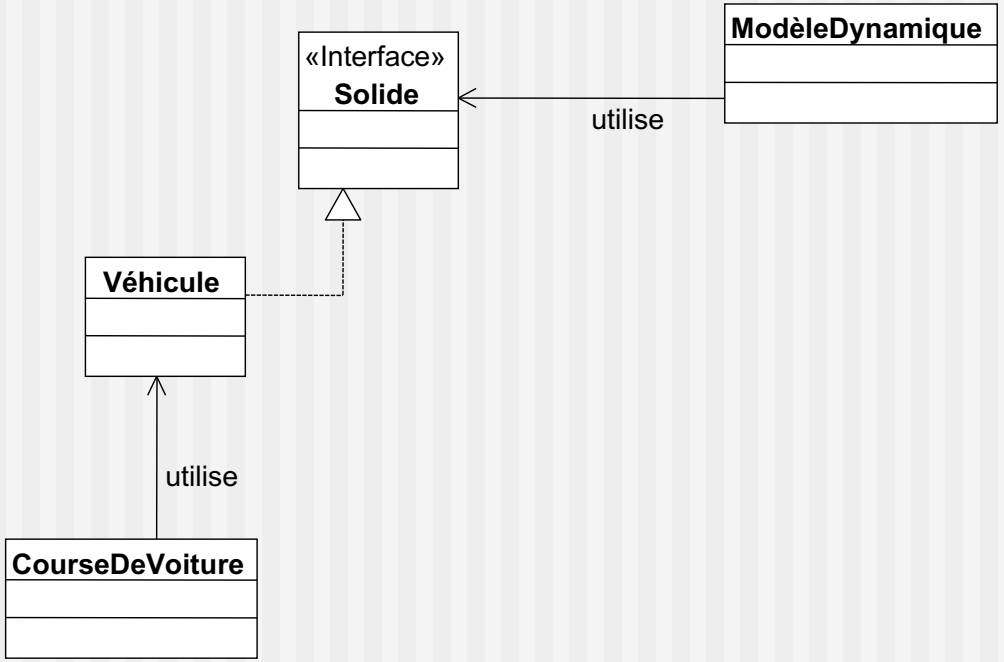
Exemples


Véhicule
ModèleDynamique
utilise
utilise
CourseDeVoiture

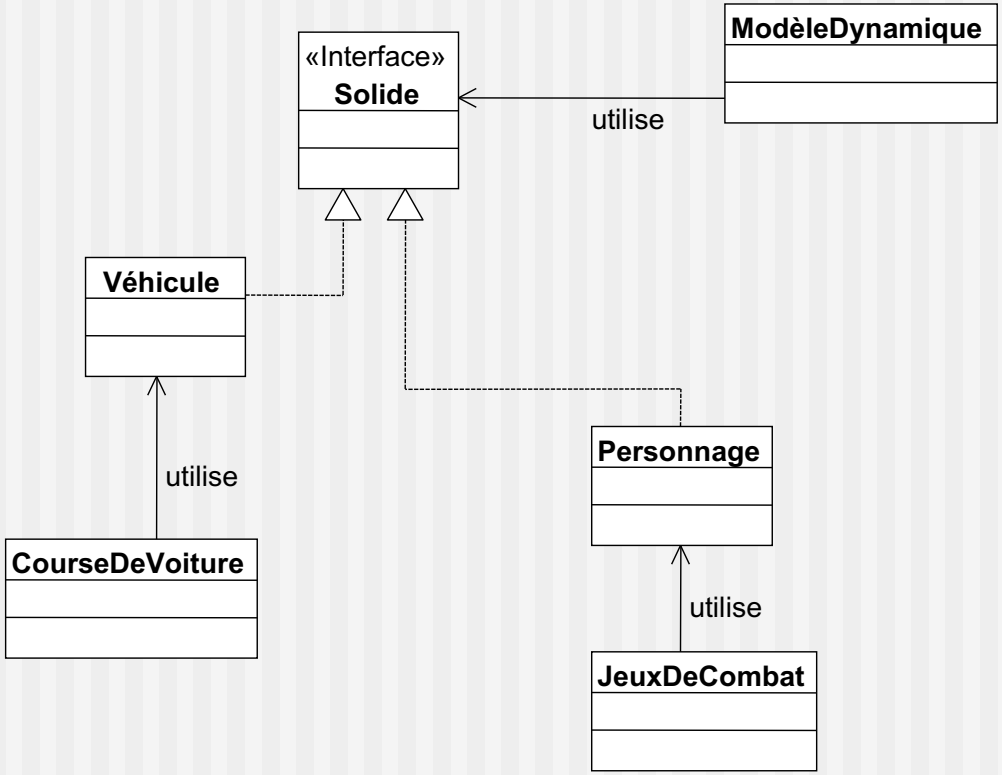
Exemples


Véhicule
ModèleDynamique
utilise
utilise
utilise
Personnage
CourseDeVoiture
utilise
JeuxDeCombat

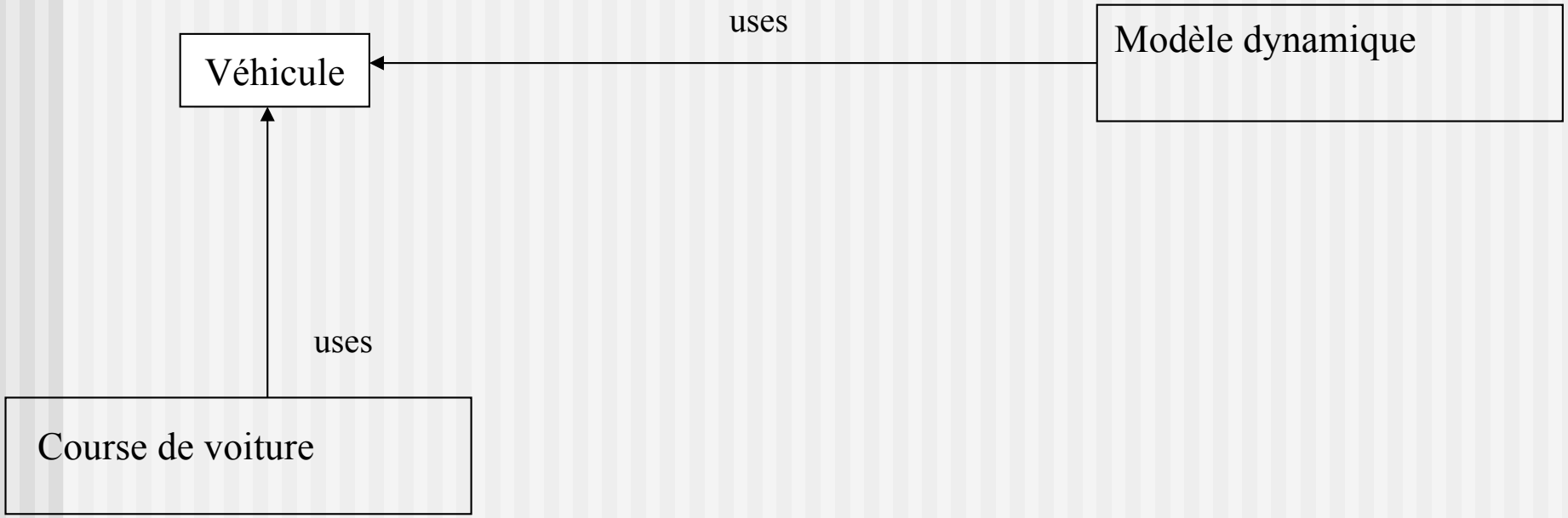
Exemples


«Interface»
Solide
ModèleDynamique
utilise
Véhicule
utilise
CourseDeVoiture

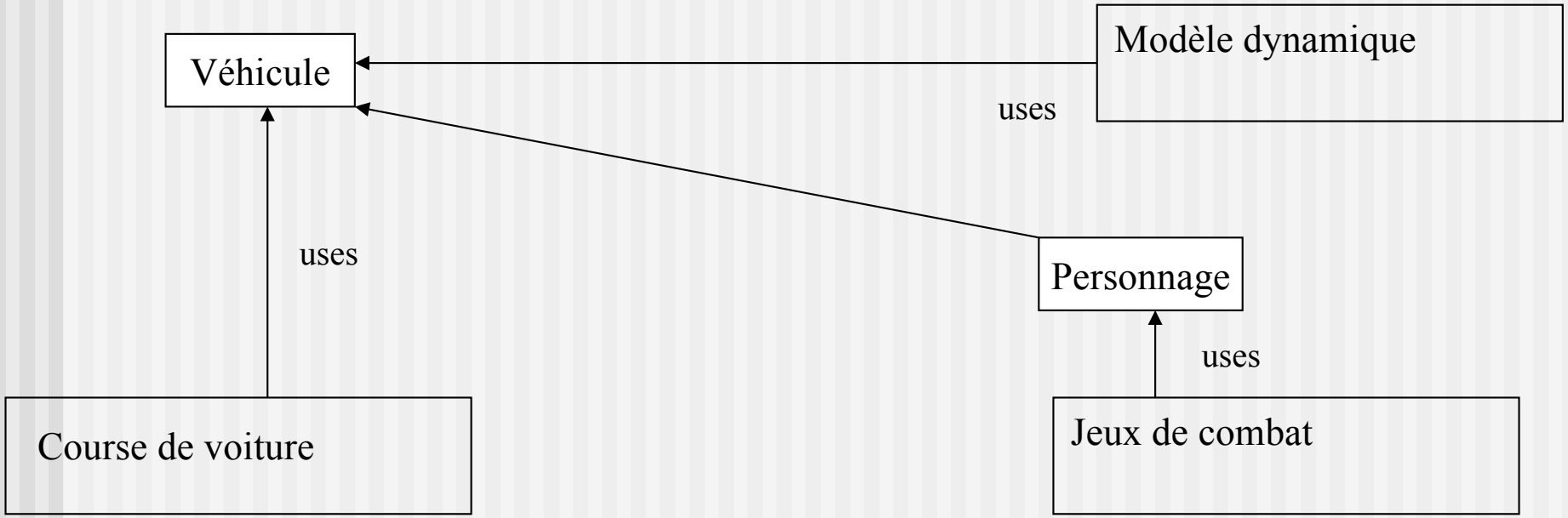
Exemples


«Interface»
Solide
ModèleDynamique
utilise
Véhicule
utilise
CourseDeVoiture
Personnage
utilise
JeuxDeCombat

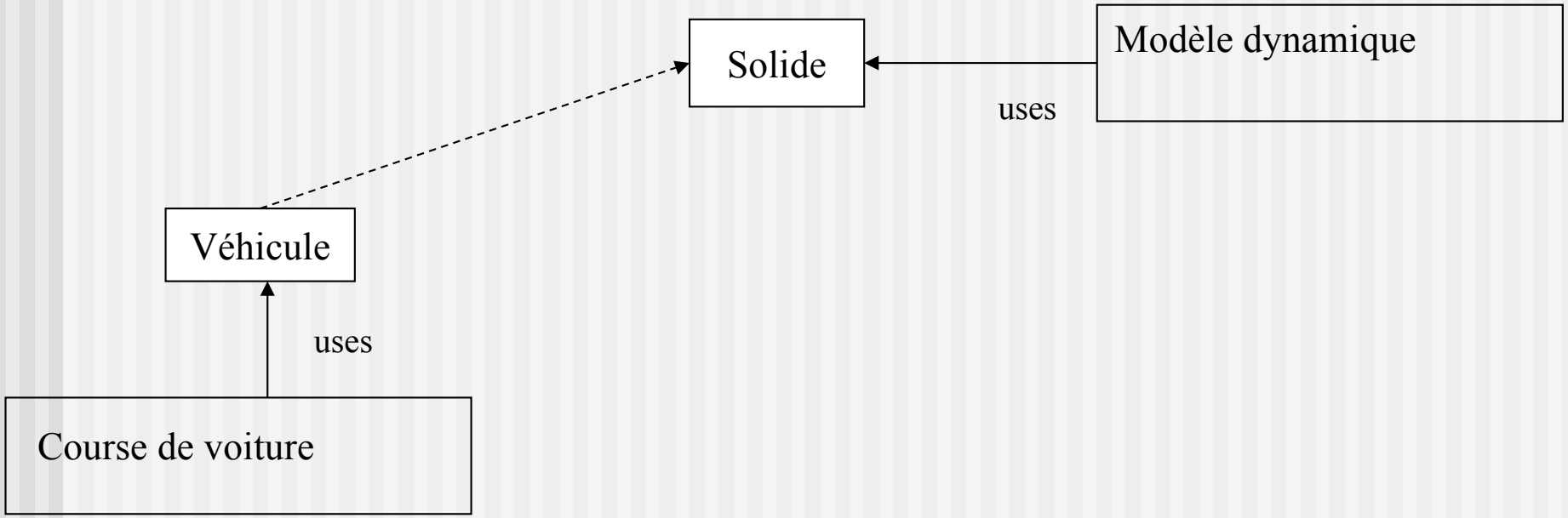
Exemples



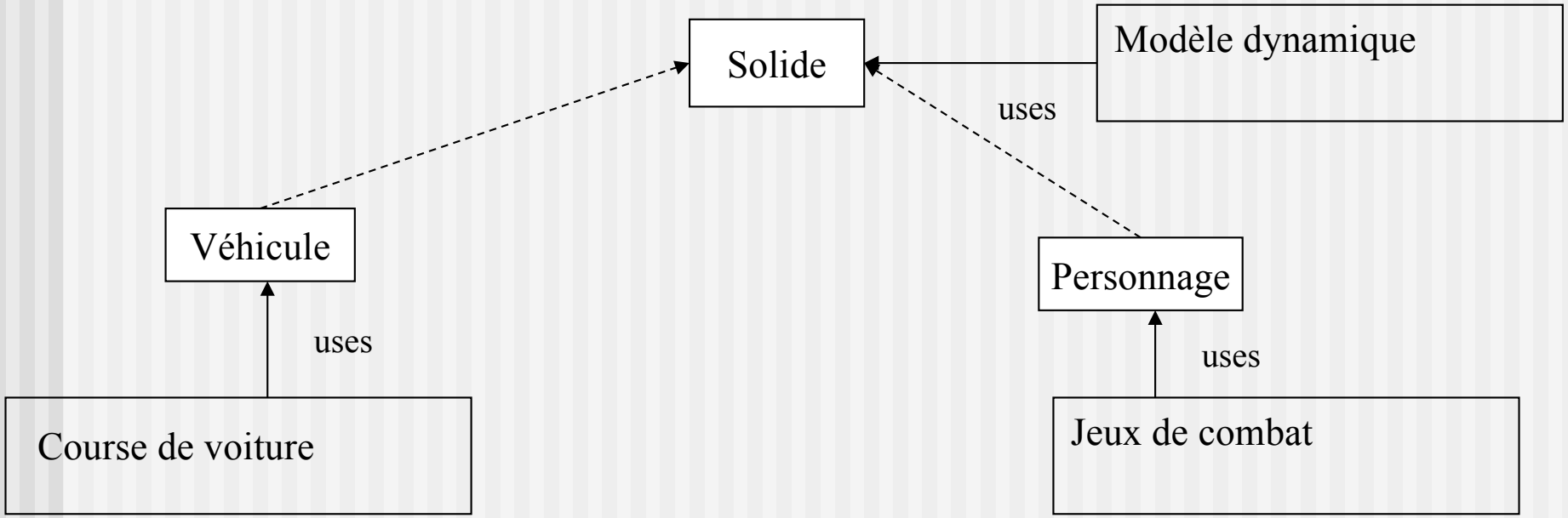
Exemples


Véhicule
Modèle dynamique
uses
uses
Personnage
uses
Course de voiture
Jeux de combat

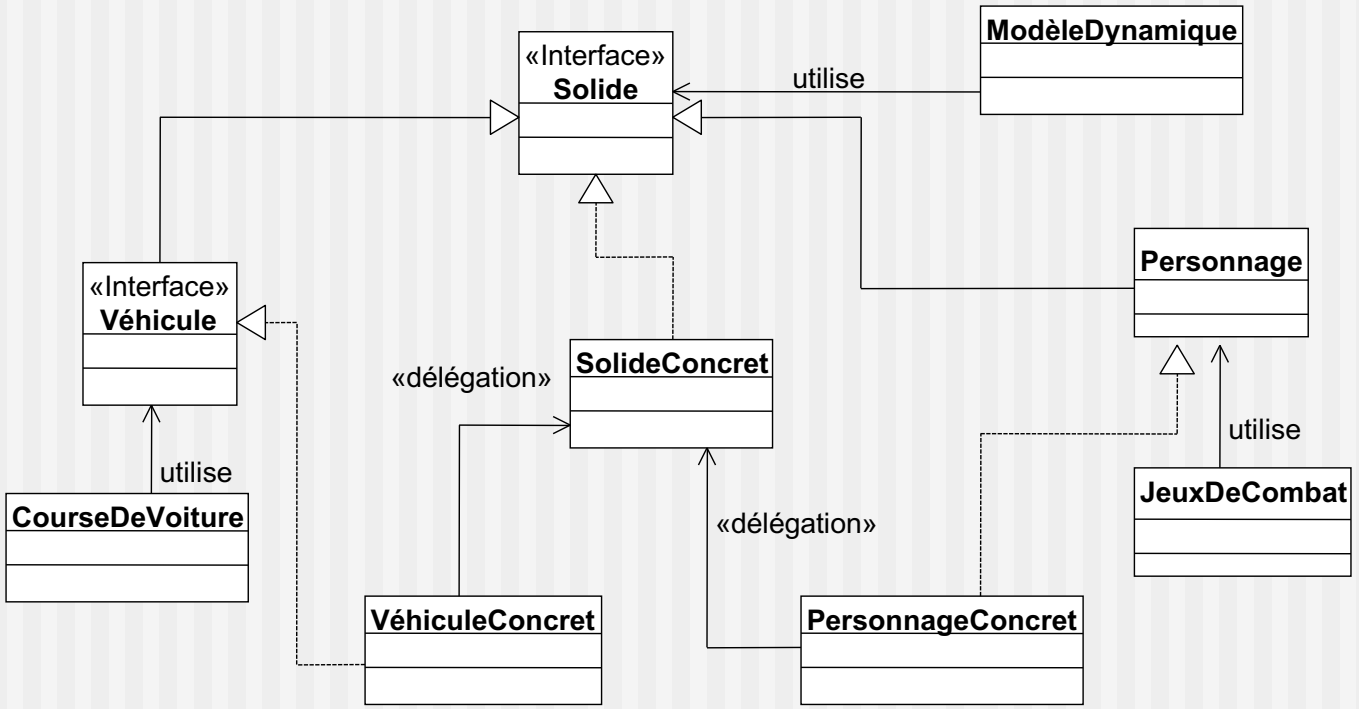
Exemples


Solide
Modèle dynamique
uses
Véhicule
uses
Course de voiture

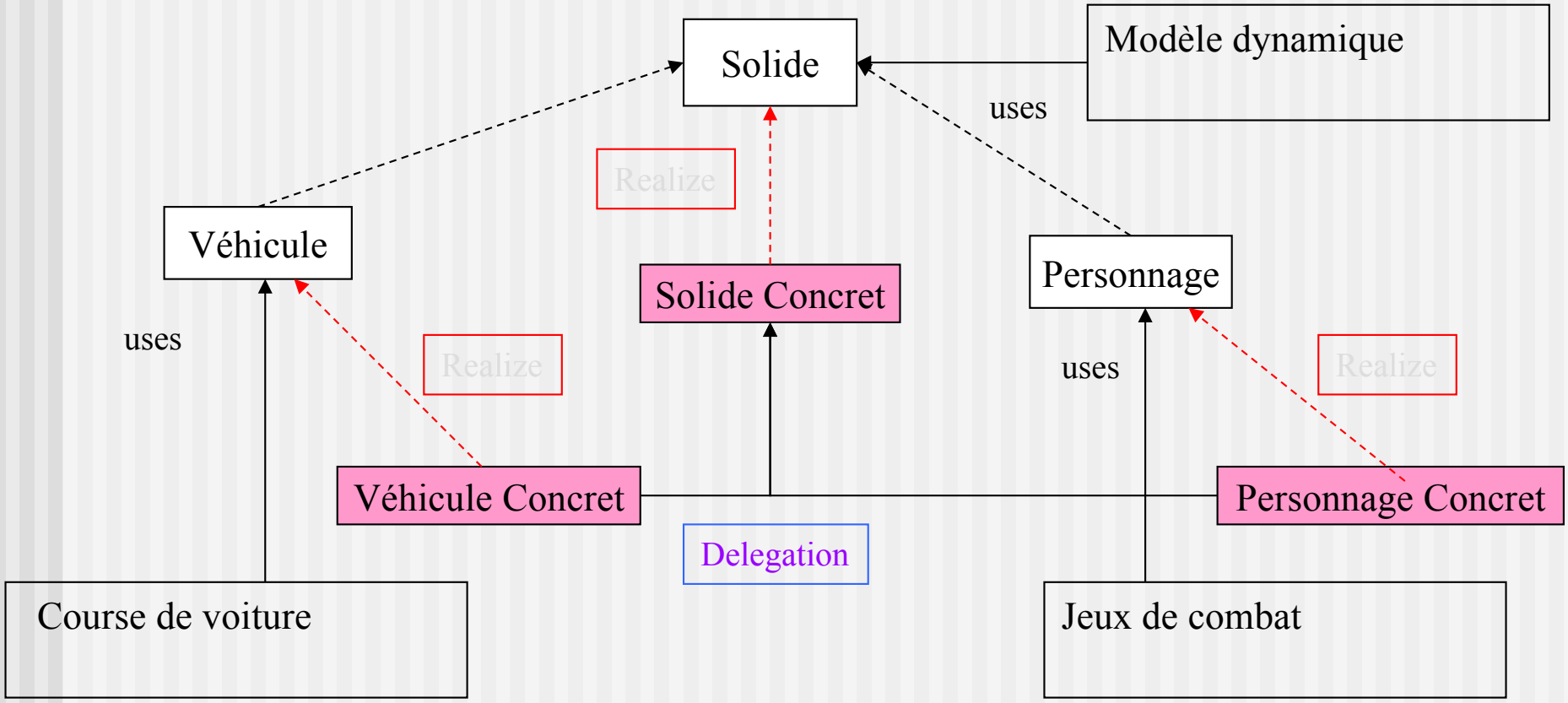
Exemples


Solide
Modèle dynamique
uses
Véhicule
Personnage
uses
uses
Course de voiture
Jeux de combat

Exemples


«Interface»
Solide
ModèleDynamique
utilise
«Interface»
Véhicule
Personnage
«délégation»
SolideConcret
utilise
utilise
CourseDeVoiture
JeuxDeCombat
«délégation»
VéhiculeConcret
PersonnageConcret

Exemples


Solide
Modèle dynamique
uses
Realize
Véhicule
Solide Concret
Personnage
uses
uses
Realize
Realize
Véhicule Concret
Personnage Concret
Delegation
Course de voiture
Jeux de combat

Concevoir pour réutiliser

- Prévoir des mécanismes pour insérer facilement de nouvelle réalisation d'interfaces
  - Séparation entre l'instanciation de classes et son utilisation
    - patterns créationels (Abstract factory, …..)

Favorise l'extension d 'application
Adaptation de framework

Concevoir pour réutiliser

- Prévoir des mécanismes pour pouvoir paramétrer le comportement d ’une classe
  - Poignée
  - Délégation du comportement à un objet concrétisant un rôle (Pattern comportementaux)

Favorise l’adaptation de classe