



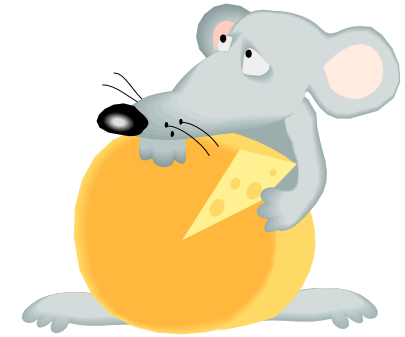
Introduction au modèle objet

loic.yon@isima.fr
<https://perso.isima.fr/loic>

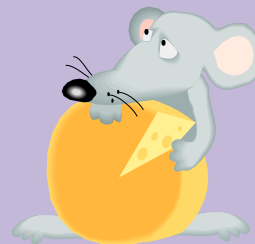


Plan

1. Génie logiciel et historique
2. Concepts objets



GÉNIE LOGICIEL & HISTORIQUE



Plan



- Pourquoi avons-nous besoin du génie logiciel ?
- Problèmes liés à la conception de logiciels ?
- Qualités d'un bon logiciel ?
- Évolution de la programmation au cours des âges

But : améliorer la qualité des logiciels

- Utilisateurs
 - Répondant bien aux besoins
 - Fiable et sûr
 - Bien documenté
- Développeurs
 - Facile à faire évoluer et à maintenir
 - Bien documenté

Génie logiciel

- Résoudre les problèmes liés à la complexité
 - ↓ Coûts de développement et de maintenance
 - Obtenir des logiciels satisfaisants pour tous
- Comment ?
 - Maîtriser la conception d'un logiciel
 - Prévoir son évolution
- Outil fondamental : l'abstraction
- Problème : subjectivité de l'abstraction

Facteurs de qualité

- Fiable
 - dès la conception !
- Efficace
- Modifiable
- Intelligible
- Interopérable

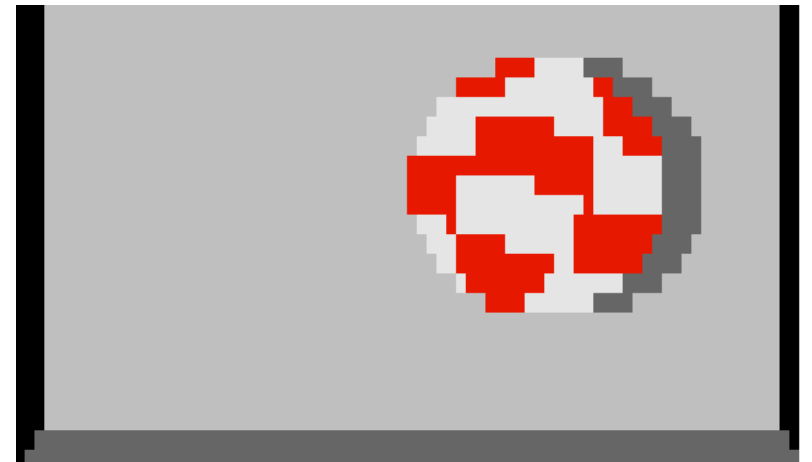


```
#include <stdio.h>
#include <math.h>
#include <unistd.h>
#include <sys/ioctl.h>
```

Ce qu'il ne faut pas faire

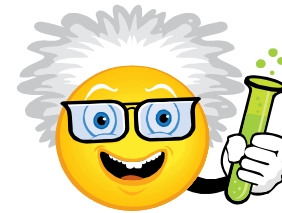
```
    main() {
        short a[4];ioctl
        (0,TIOCGWINSZ,&a);int
        b,c,d=*a,e=a[1];float f,g,
        h,i=d/2+d%2+1,j=d/5-1,k=0,l=e/
        2,m=d/4,n=.01*e,o=0,p=.1;while (
        printf("\x1b[H\x1B[?25l"),!usleep(
        79383)){for (b=c=0;h=2*(m-c)/i,f=-
        .3*(g=(1-b)/i)+.954*h,c<d;c+=(b==
        b%e)==0)printf("\x1B[%dm ",g*g>1-h
        *h?c>d-j?b<d-c||d-c>e-b?40:100:b<j
        ||b>e-j?40:g*(g+.6)+.09+h*h<1?100:
        47:((int)(9-k+(.954*g+.3*h)/sqrt
        (1-f*f))+(int)(2+f*2))%2==0?107
        :101);k+=p,m+=o,o=m>d-2*j?
        -.04*d:o+.002*d;n=(1+=
        n)<i||l>e-i?p=-p
        ,-n:n;}}
```

Peter Eastman
IOCCC 2011



Réutilisabilité (1)

- Temps de développement
 - Ne pas réinventer la roue à chaque étape
 - Même algorithme sous diverses formes
 - Différents langages de programmation
 - Différentes structures de données
 - Différents environnements
- Obstacles
 - Documentation inadéquate
 - Problèmes de diffusion
 - Interfaçage avec le code local



Réutilisabilité (2)

- Avantages
 - Plus un code est utilisé, plus il devient fiable
 - Détection précoce des bogues et manquements
 - Coûts de maintenance inférieurs
 - Un module validé n'a plus à être vérifié !
- Réutiliser le code c'est bien ...
- Réutiliser la conception c'est mieux !
 - Éliminer certaines étapes de conception sur des problèmes semblables
 - Englober en une unité fonctionnelle unique les travaux de conception et d'implémentation

Évolution du logiciel

- L'époque héroïque
- Les premiers langages évolués
- La notion de sous-programme
- La programmation modulaire
- Les types abstraits de données
- Les objets



L'époque héroïque

- Ordinateurs peu nombreux et réservés aux institutions gouvernementales
 - Peu de choses demandées aux ordinateurs
 - Interface sommaire voire inexistante
 - Petits programmes en langage machine puis en assembleur
 - Un seul concepteur ou une très petite équipe





```
0100010001010010
0001000111001110
1111010001010100
0101010010000101
1111000010101001
1010100101011010
1011101011111000
0101010101010101
1011011111110100
1010101010101010
0101010101111101
1001010101010101
```

```
mov ah, 00h
mov al, 13h
int 10h

mov ax, 0a000h
mov es, ax
mov ax, 320
mul 100
add ax, 160
mov di, ax
mov al, 255
mov es:[di], al
```

Les ordinateurs se démocratisent

- Formalismes plus simples
 - Premiers langages structurés (FORTRAN)
 - Règne du « spaghetti »
- Programmes plus ambitieux
 - Premières équipes de travail
 - Découpage en plusieurs programmes reliés
 - Premiers problèmes liés à la communication
 - Découpage / Interfaçage
- Abstraction de l'implémentation
= description des interfaces



$$PL(I, J) = (2 * I - 1) * 4 * PL(I - 1, J) / I - (I - 1) * PL(I - 2, J) / I$$

[illegible]

Sous-programme (1)

- Motivation
 - traitement des tâches apparaissant de manière répétitive dans les programmes
- Boîte noire
 - L'utilisateur ne connaît que le prototype :
 - Paramètres
 - Fonctionnalités



Sous-programme (2)

- **Abstraction procédurale**
 - Tout appel de sous programme apparaît atomique
- Problèmes
 - Nommage des identificateurs
 - Collisions possibles entre les identificateurs de l'utilisateur et ceux des sous-programmes.

Module (1)

- Apparition de "gros" logiciels de gestion
- Un module = une fonctionnalité
 - gestion client, gestion compte, ...
- Communication par messages
- Extension naturelle des sous-programmes

Module (2)

- Entité indépendante regroupant les sous-programmes et les données sur lesquelles ils travaillent
 - Séparation entre partie publique et partie privée
 - La partie publique définit l'interface ou contrat d'utilisation
- Problèmes majeurs
 - Impossible de dupliquer un module
 - Une seule copie des données sur lesquelles un module est capable de travailler
 - Attention au découpage ! il est tentant de vouloir faire des modules trop petits et de perdre ainsi en efficacité

Type abstrait de données (1)

- Abstraction des données

*"On connaît une donnée au travers
des opérations disponibles dessus"*

- Structure de données avec traitements associés
- Apparition du schéma modèle/instance
 - Autant d'instances que nécessaire :
 - Données dupliquées
- Notion d'interface / contrat
 - Liste des opérations disponibles
 - Implémentation cachée des données

Type abstrait de données (2)

La pile



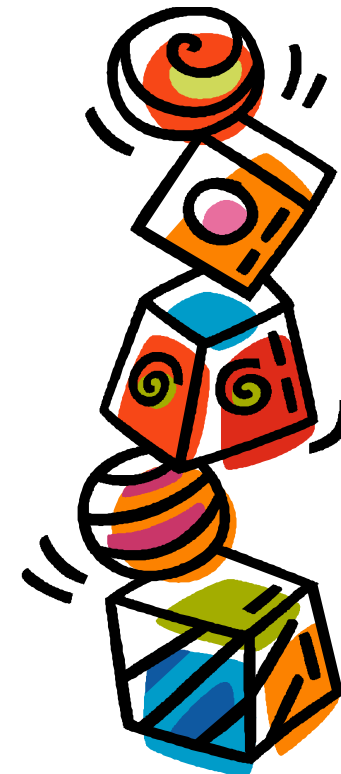
Sommet ?

créer()

empiler()

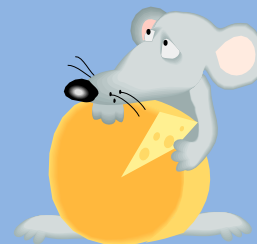
dépiler()

estVide()



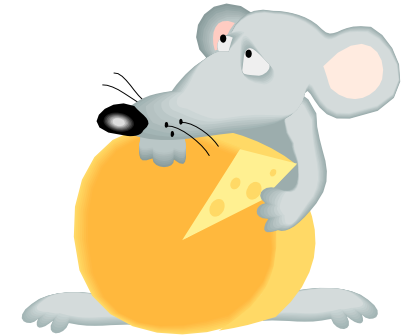
Autres exemples : une file, une liste chaînée ...

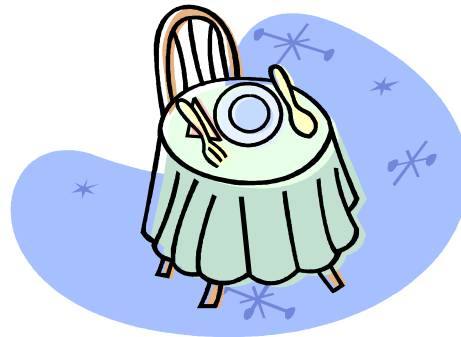
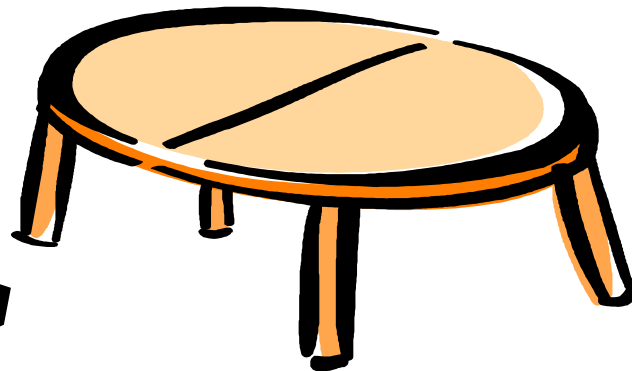
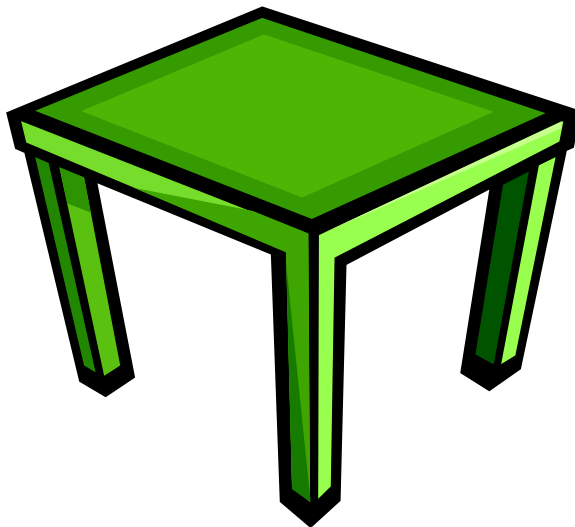
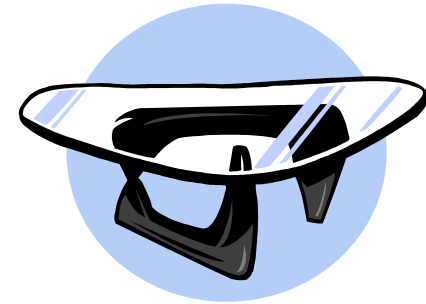
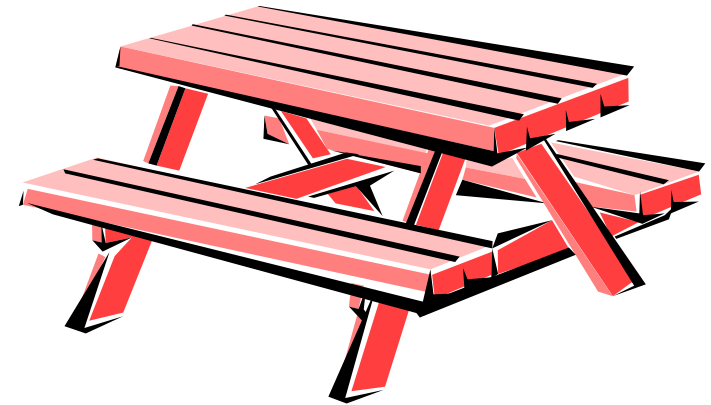
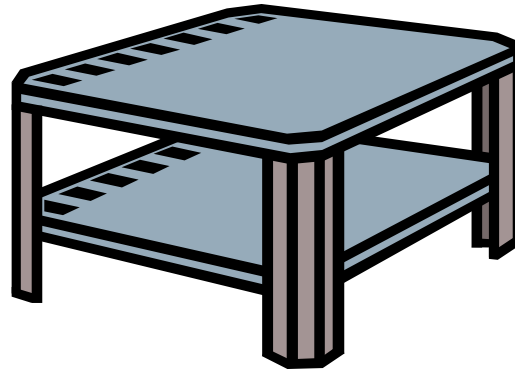
CONCEPTS OBJETS



Plan

- Définitions
 - Objet
 - Classe
- Notation (UML)
- Relations
 - Héritage
 - Agrégation / Composition
 - Association





Réification ?

Définition

“ Un objet est une capsule logicielle oblativ avec un tropisme conatif dont l'hétéronomie est la marque de la durée de l'éphémère et de la hoirie ! ”

-- Serge Miranda, Université de Nice

Objet

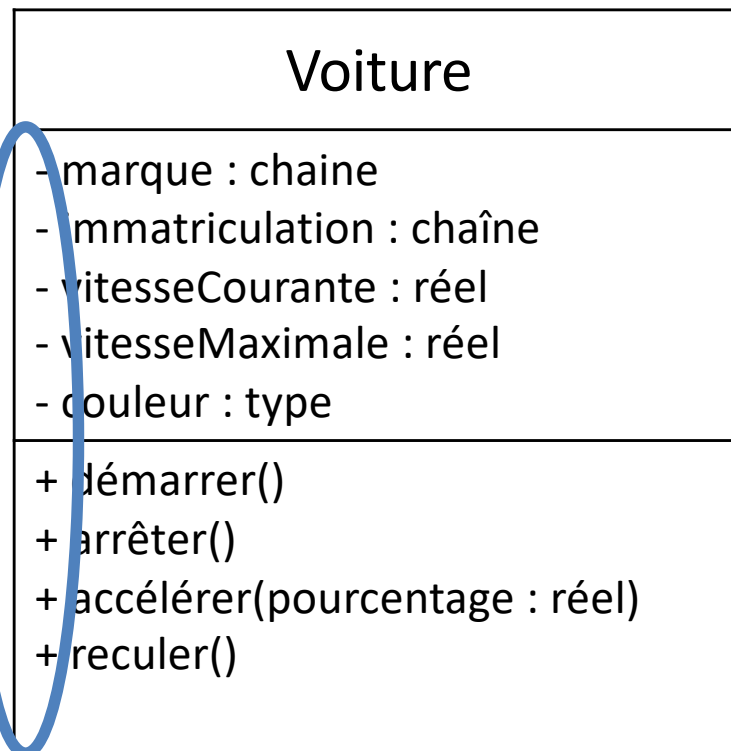
- Entité cohérente rassemblant des **données** et le **code** travaillant sur ces données
- Données
 - Attributs
 - Données membres
- Code
 - Méthodes
 - Fonctions membres

Classe

- Moule / Modèle / Fabrique à objets
 - donnée qui décrit des objets \approx Classe d'équivalence
- Comment représenter ?
 - *Unified Modeling Language* - UML



Une classe - UML



Nom de la classe

Description des attributs
ou données membres

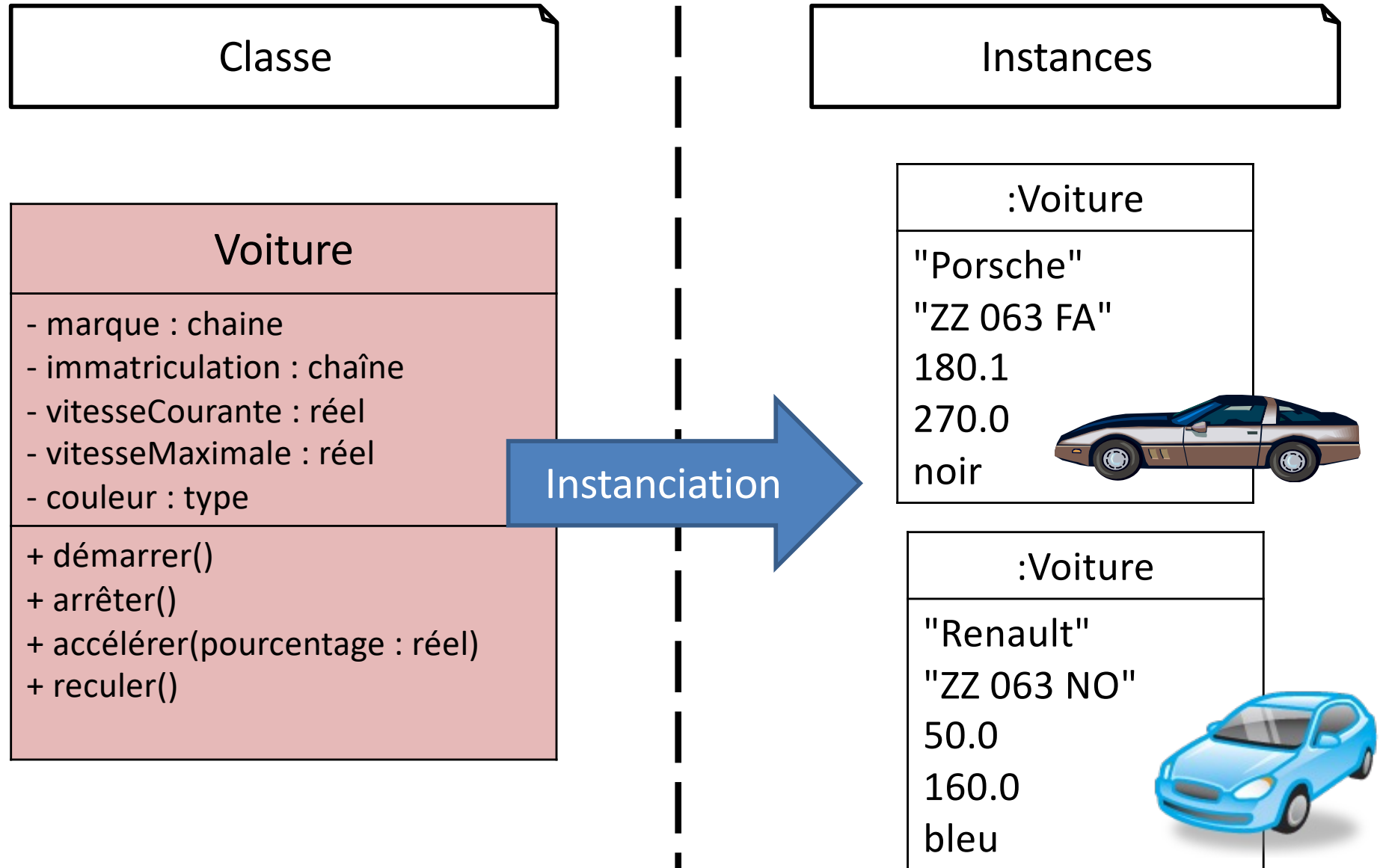
Description des méthodes =
code associé aux données

Visibilité vis-à-vis de l'extérieur
public / privé

Relation d'instanciation

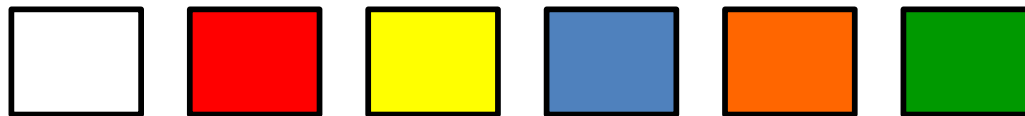
- Obtenir un objet à partir de la classe
 - La classe décrit un objet
 - Données membres / Attributs
 - Fonctions membres / Méthodes
 - L'objet est un **état** de la classe
 - Tous les objets d'une même classe ont la même structure
 - Les données prennent des valeurs significatives pour un objet particulier
 - C'est une **instance** de la classe

Exemples d'instanciation



Membre d'instance ou de classe (1)

- Attributs **d'instance** = une valeur par objet
 - vitesse d'un véhicule
 - couleur d'un véhicule
- Attributs de **classe** = partagés par tous les objets d'une même classe
 - nombre de véhicules présents à un instant donné
 - gamme de couleurs disponibles



Membre d'instance ou de classe (2)

- Méthode **d'instance** = agit sur un objet particulier
 - Accélérer, freiner
 - Donner la vitesse courante
 - Donner la couleur
- Méthode de **classe** = agit sur toute la classe
 - Donner la gamme de couleurs disponibles
 - Donner le nombre de voitures créées
 - [Créer ou détruire une voiture]

Membre d'instance ou de classe (3)



Voiture	
- <u>NombreDeVoitures</u> : entier	
+ <u>Gamme de couleurs</u>	
- Marque : chaîne	
- Immatriculation : chaîne	
- VitesseCourante : réel	
- VitesseMaximale : réel	
- couleur : Gamme	
+ démarrer()	
+ arrêter()	
+ accélérer(pourcentage : réel)	
+ reculer()	
+ <u>Créer une voiture()</u>	
+ <u>Détruire une voiture()</u>	

Attributs
ou méthodes
soulignés

Encapsulation (1)

- Séparation **forte** interface/implémentation
- **Interface** = partie visible d'un objet
 - Ensemble de messages paramétrés
= prototype des méthodes
 - Communiquer avec un objet
= envoi de **messages**
= appel direct de méthode (en pratique)
- **Implémentation** cachée
 - Attributs / Code des méthodes
 - Modifiable sans que l'utilisateur ne s'en aperçoive
 - ▷ ne pas modifier l'interface



Encapsulation (2)



- Abstraction procédurale

Appel de message **atomique**

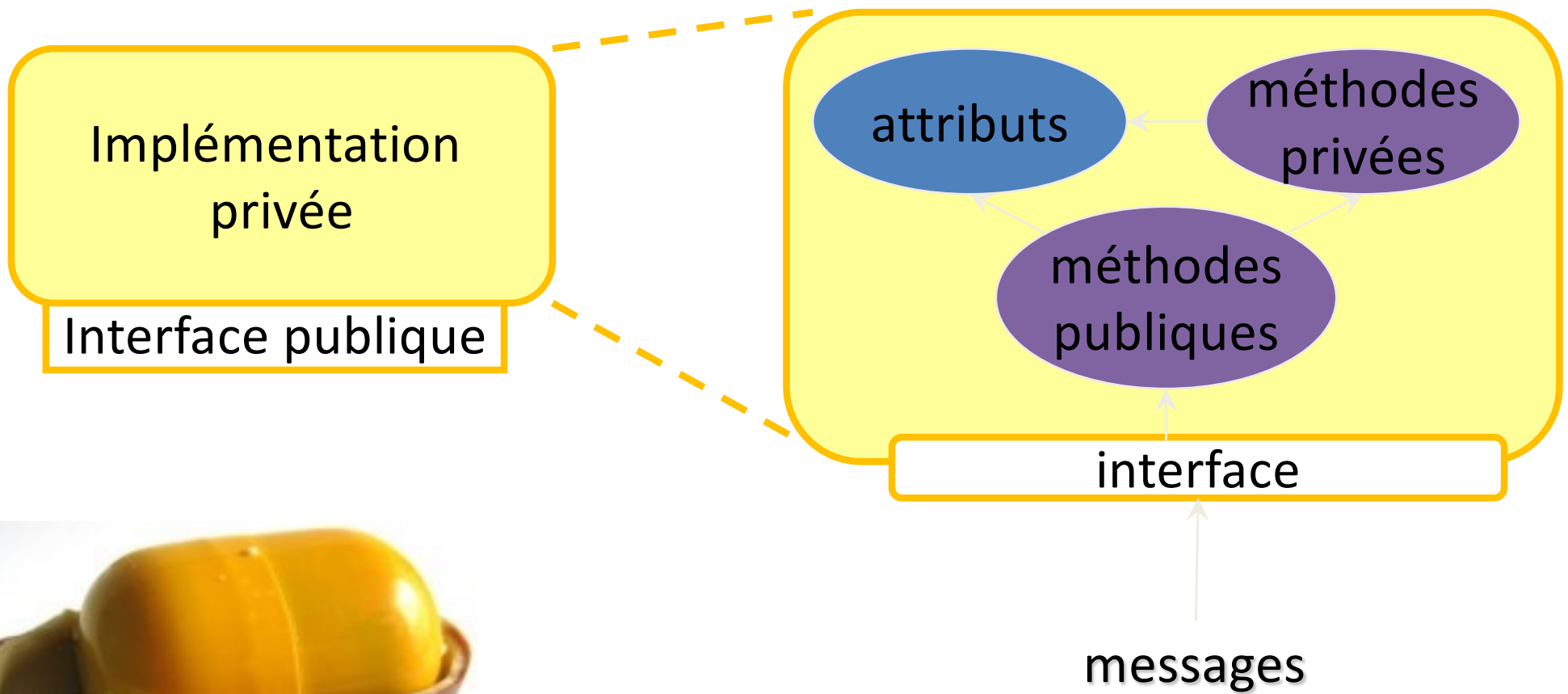
- Les objets agissent comme des boîtes noires
- L'utilisateur n'a aucun contrôle sur les traitements associés à son message

- Abstraction de données

Objet accessible uniquement par ses messages

- L'utilisateur n'a aucun renseignement sur la structure interne d'un objet

Encapsulation (3)



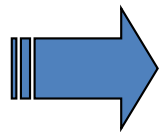
Principe de Demeter

- On ne parle qu'à ses amis !
- Une méthode accède exclusivement aux éléments suivants :
 - Ses arguments et les variables temporaires qu'elle crée
 - Les attributs de l'objet courant
 - Les attributs de la classe de l'objet courant
 - Les variables globales



Des interfaces multiples ?

- Certains langages permettent à un objet de disposer de plusieurs interfaces



Choix d'interface avant d'accéder aux messages

Développeurs



Chef de projet



Big boss



Notion de logiciel orienté objet

- Collection d'objets en interaction !

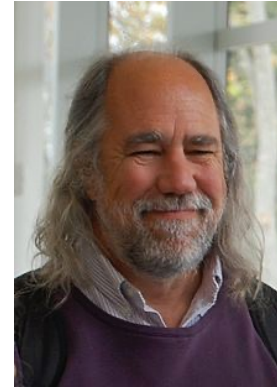
Une nouvelle abstraction : l'exécution

*Un système d'objets doit toujours donner l'impression d'être
monolithique*

- Un objet cible peut être :
 - Situé sur la machine courante ou distante
 - Actif / en sauvegarde (persistance) / pas encore créé

Créer un logiciel orienté objet (1)

- 5 grands principes de BOOCH



1. Décomposable

- Partitionner le logiciel en modules d'objets indépendants ne nécessitant que de connaître les interfaces

2. (Re)Composable

- Les différents modules doivent pouvoir s'assembler pour former de nouveaux modules

Créer un logiciel orienté objet (2)

3. Compréhensible

- Fondé sur la qualité de la documentation
- Chaque module doit pouvoir être compris par une personne n'appartenant pas à son équipe de développement.
- Les dépendances entre modules doivent être claires

Créer un logiciel orienté objet (3)

4. Continu

- Un changement faible de spécification doit entraîner le minimum de modifications dans les modules

5. Protégé

- Lorsqu'une erreur apparaît dans un module, celui-ci doit s'en rendre compte et avertir les autres

Typage des objets

- une classe = un type
- Deux catégories de langages à objets
 - Typage statique
[C++ , JAVA]
 - Typage dynamique
[Javascript, PHP, Python]

Typage statique

- Un objet est affecté à une classe à la **compilation**
- L'information de type est liée au nom (identificateur) de l'objet et donc à son adresse mémoire
- Vérification de l'adéquation objet/message à la **compilation**
 - Impossible d'envoyer un message non traitable
 - "Facile" à déboguer et à mettre au point

Typage dynamique

- L'information est liée non à l'adresse mais au contenu de l'objet
- Le type de l'objet peut être modifié au cours du temps
- Grande souplesse notamment lors du prototypage d'une application
- Vérification de l'adéquation objet/message à l'**exécution**
 - Erreur
 - Transmission ou délégation vers un autre objet

Les relations fondamentales

- Héritage

Concept de Généralisation/spécialisation

Relation : Est une version spécialisée de (Is A)

- Composition / agrégation

Concept de composition

Relation : Contient, regroupe, possède (Has A)

- Association

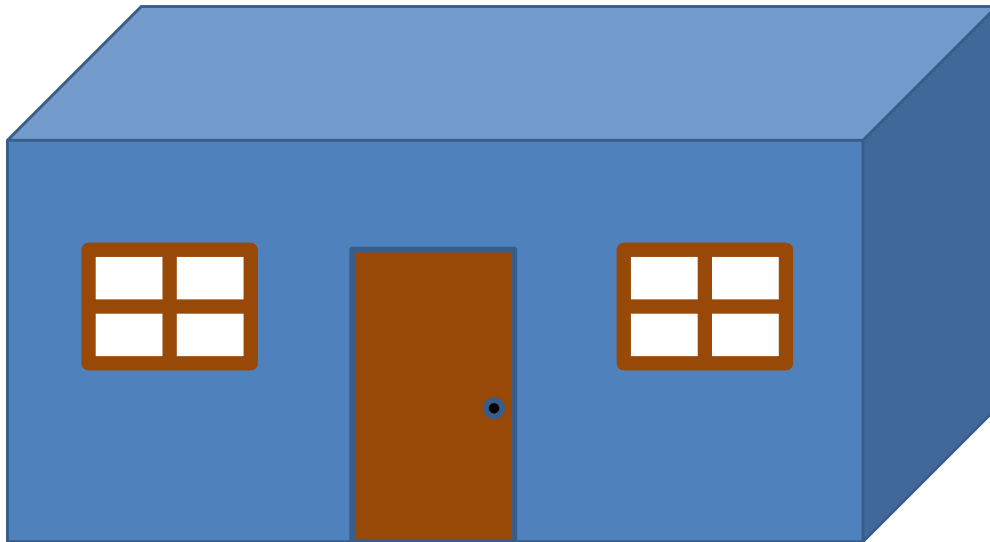
Concept d'objets communicants

Relation : communique avec (Uses A)

Héritage ?

Généralisation / Spécialisation

- Classe de base
- Classe générale
- Classe mère
- Super classe



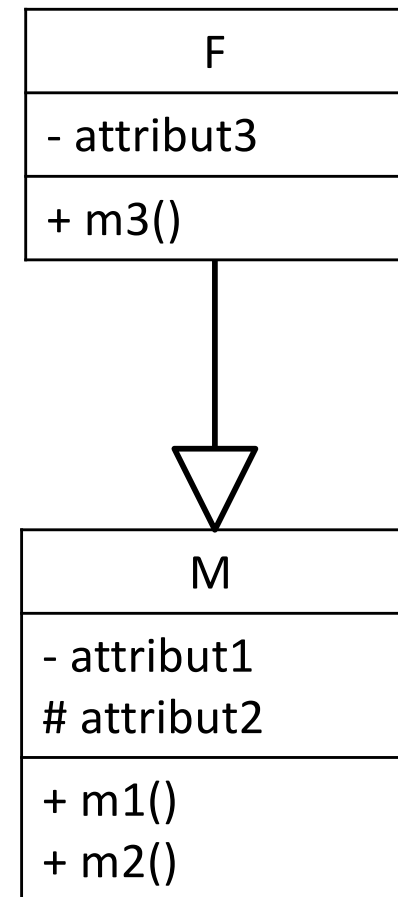
M
- attribut1 # attribut2
+ m1() + m2()

Une classe fille est une classe mère particulière

- Classe spécialisée
- Extension
- Classe fille
- Classe dérivée

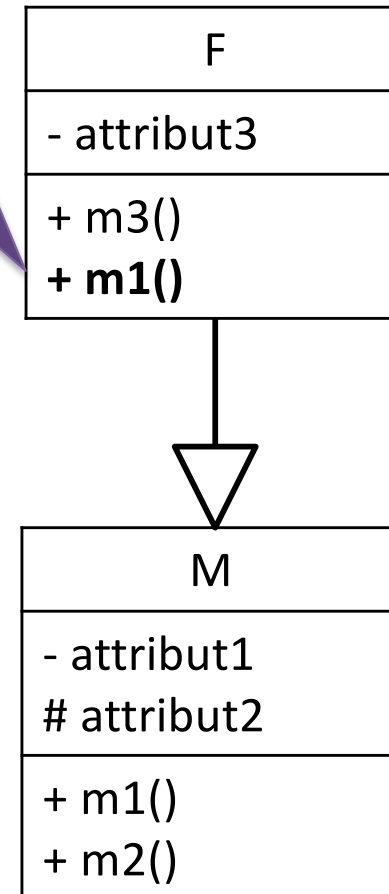
Nouveaux messages et attributs

Héritage des messages et attributs de la classe mère



Redéfinition de message

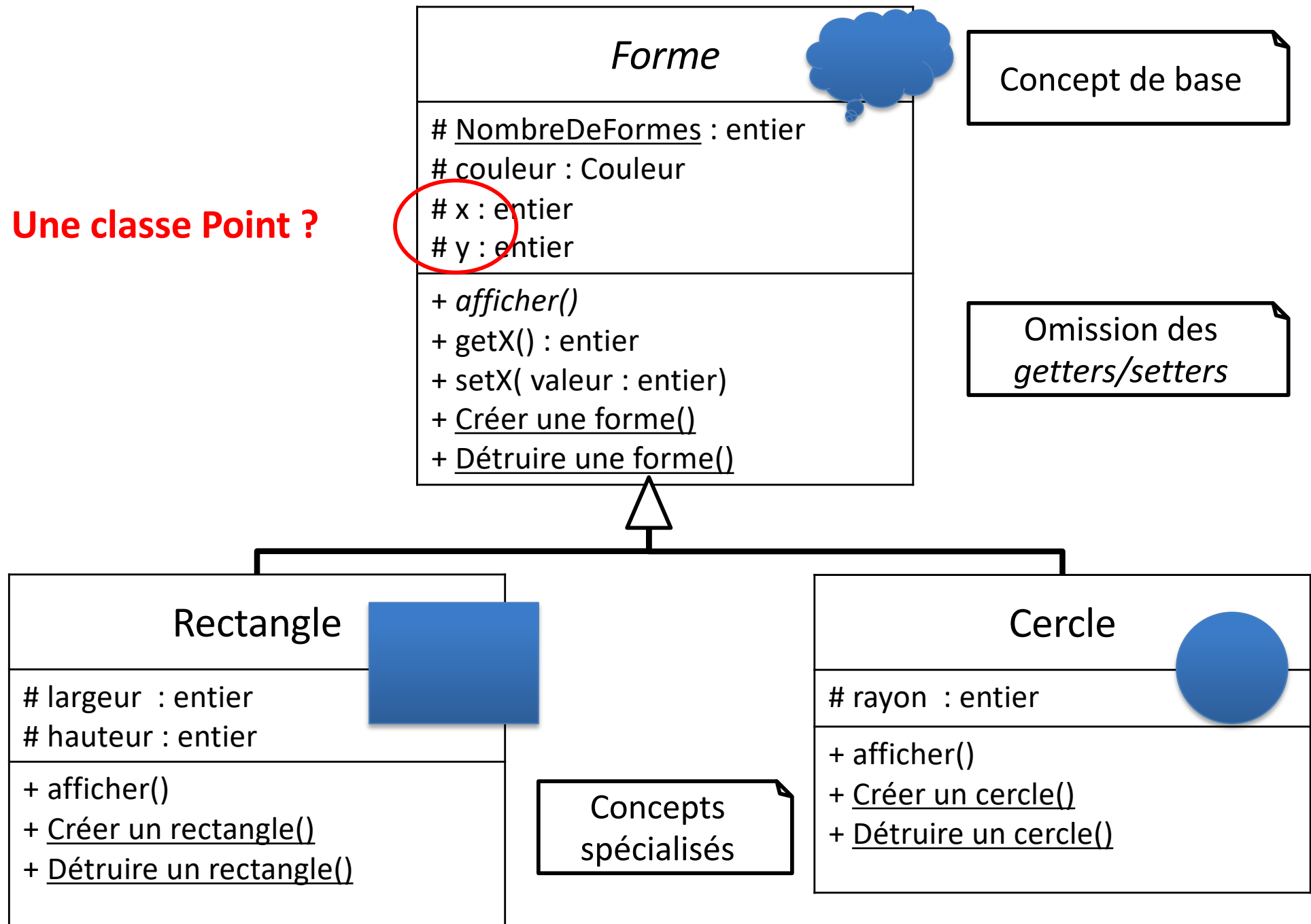
Réponse différente de la classe
fille à un message de la classe
mère



Visibilité des membres en UML ?

- + public : tout le monde
 - Méthodes de l'**interface publique**
- privé : classe seulement
 - **Attributs** -> getter / setter
 - Méthodes privées
- # protégé
 - Privé pour l'extérieur - **Attributs**
 - Transmis par héritage
 - Notion dépendante du langage
- ~ package

Une classe Point ?



Représentation de hiérarchie de classes – arbres généalogiques

Comment travailler avec l'héritage ?

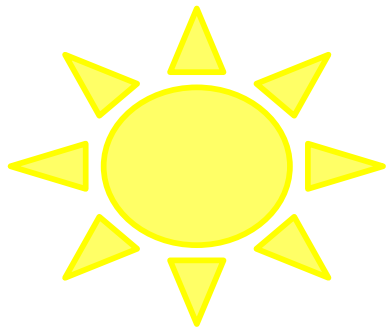
- Construire un système *ex nihilo*
 - Identifier tous les composants
 - Procéder par généralisation en factorisant les caractéristiques communes
- Étendre un système
 - Ajouter les nouvelles classes dans le graphe d'héritage en identifiant les différences avec les classes existantes
 - « programmation différentielle »



Un parc de véhicules

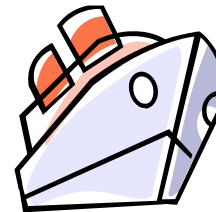
- On souhaite fournir un modèle de parc de véhicules pouvant fournir
 - des voitures
 - des camions
 - des bateaux
 - des hélicoptères
- Tous sont des véhicules
 - Factorisation au plus haut par une classe `Véhicule`
 - Les voitures et les camions sont des véhicules terrestres pouvant découler d'une même classe

Modélisation possible d'un parc
hétérogène de véhicules



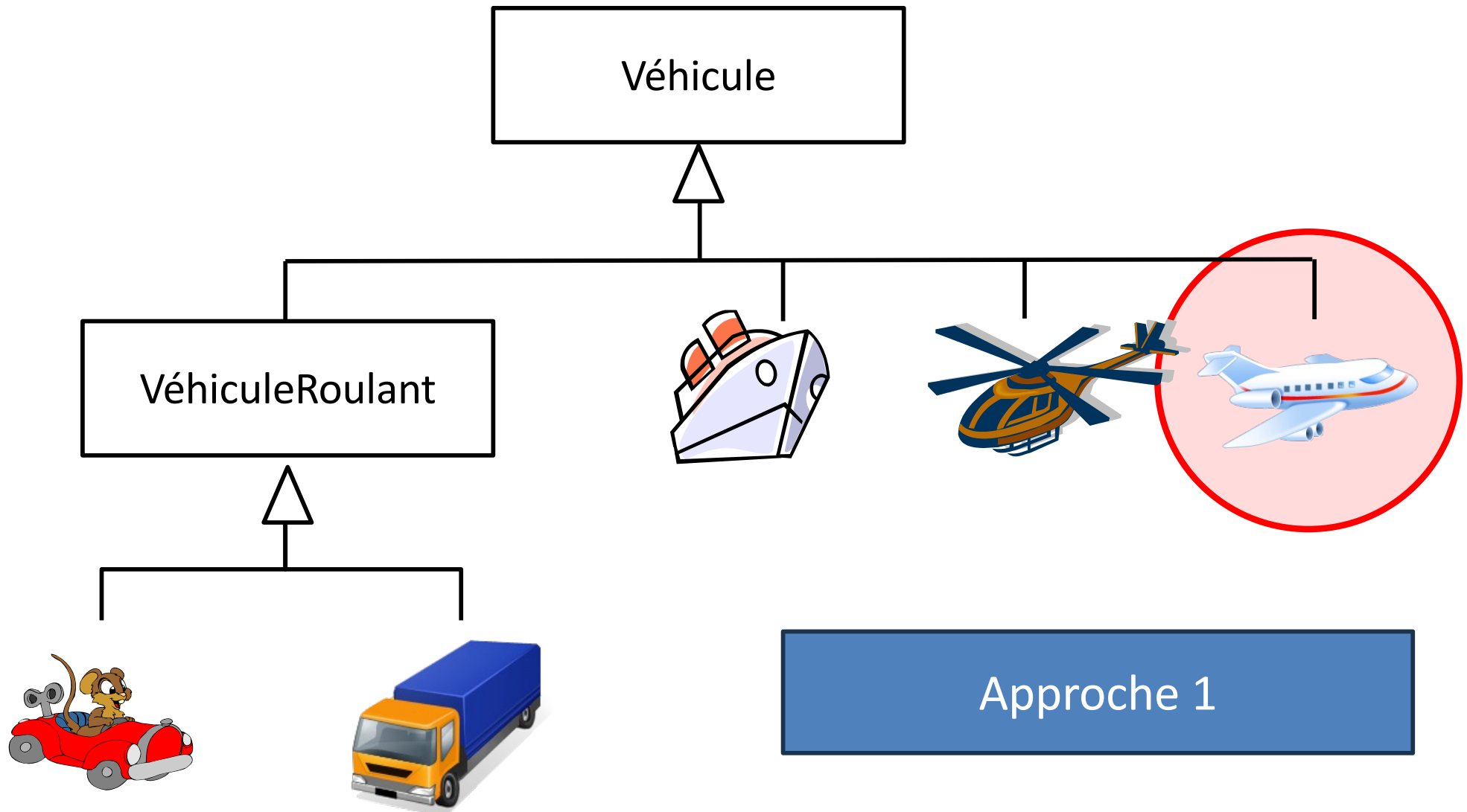
Véhicule

VéhiculeRoulant

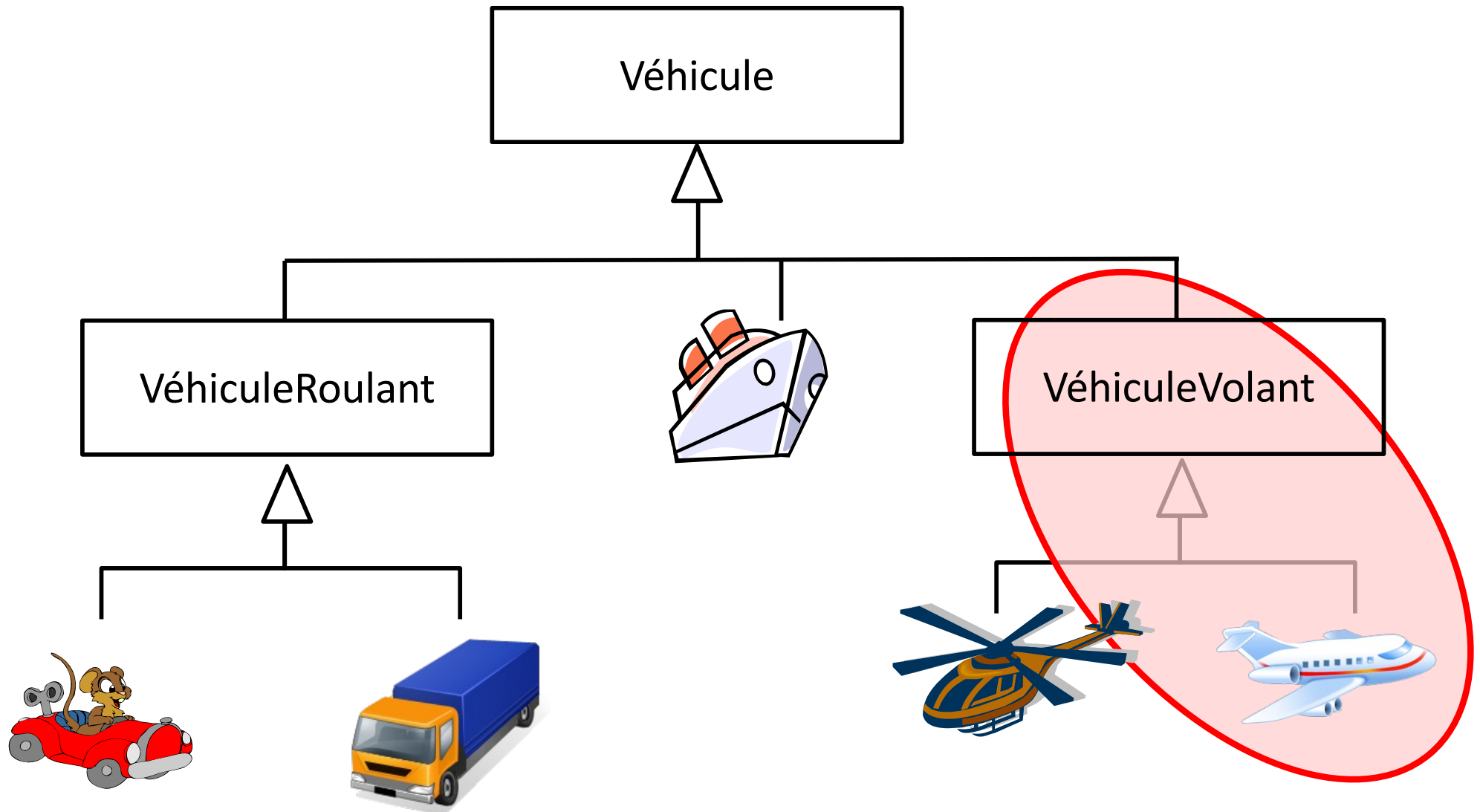


Ajouter un avion ?





Approche 2





Classe abstraite

- Modélisation d'un **concept** plutôt que d'objets réels
 - Pas d'instance
 - Définit des méthodes abstraites
 - Pas d'implémentation : uniquement une spécification
 - Peut définir des attributs
- Utilisée comme super classe d'une hiérarchie
 - Exemples : Véhicule, Forme
 - Aucun intérêt d'avoir des instances \Rightarrow classes dérivées
- Définit un cadre de travail pour le *polymorphisme*
 - ☞ Une même méthode prend plusieurs formes !

Polymorphisme

- Une même méthode prend plusieurs formes

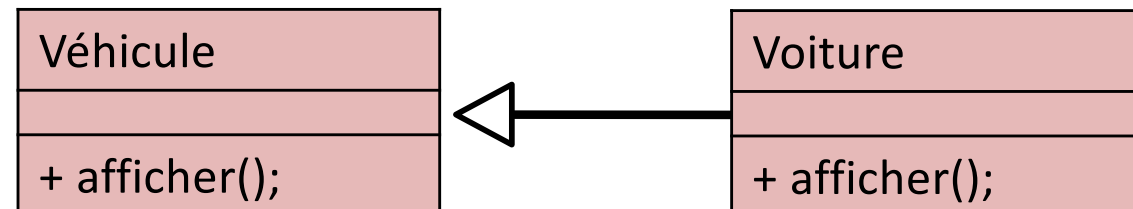
- Forme faible

- Surcharge de méthode – *overloading*
- Méthodes de signatures différentes

Voiture
+ avancer(temps : entier) + avancer(distance : réel)

- Forme forte

- Redéfinition – *overriding*
- Actions différentes pour des classes d'une même hiérarchie



Exemple (1)

- Méthode à différencier selon le type de l'objet `Forme` : l'affichage
- Code sans la notion d'objet :

```
Pour chaque objet :  
  Selon (TypeObjet)  
    cas CERCLE  
      AffichageCercle (objet)  
    cas RECTANGLE  
      AffichageRectangle (objet)  
    cas TRIANGLE  
      AffichageTriangle (objet)  
  Fin Selon  
Fin Pour
```

?

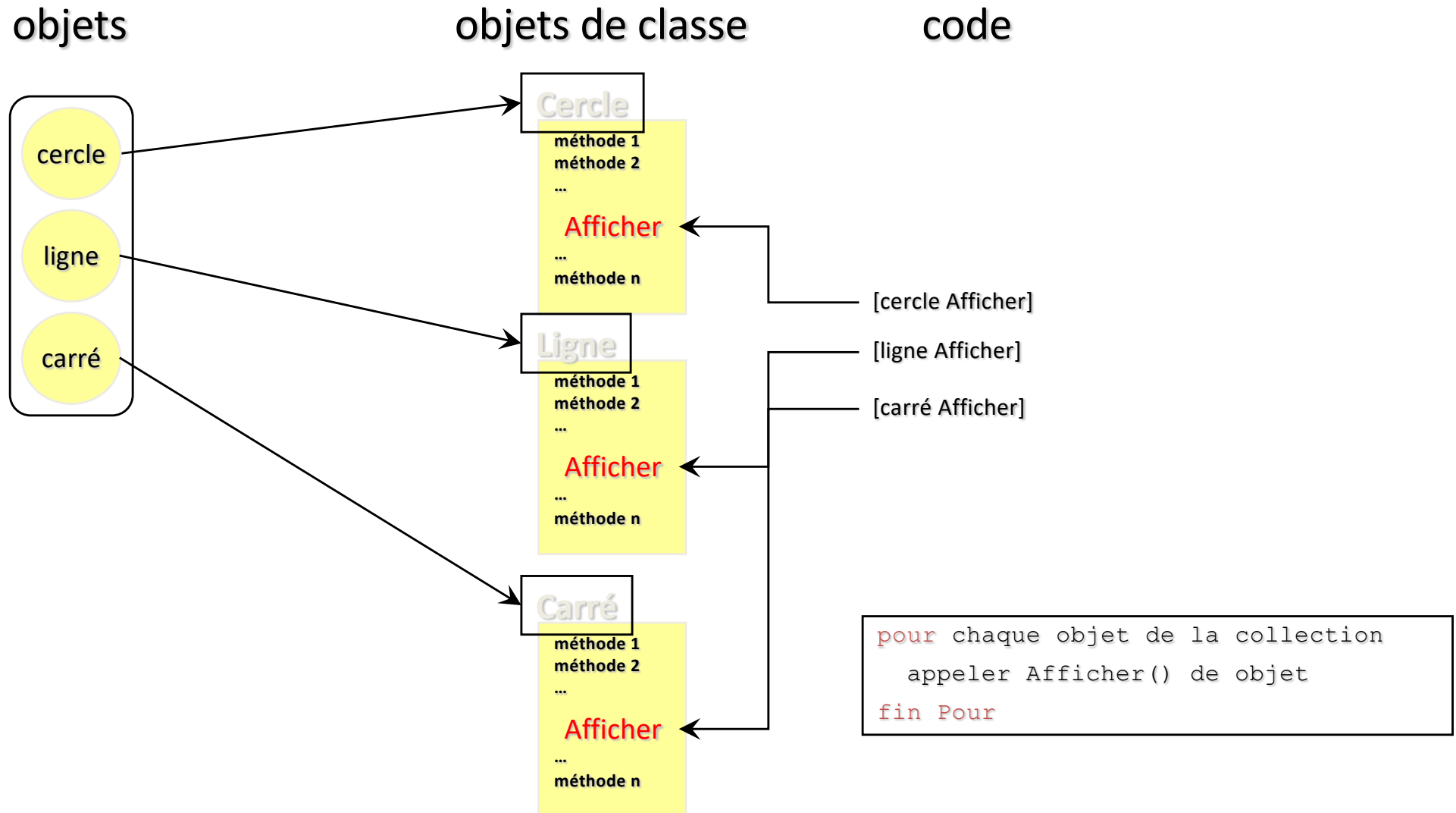
Exemple (2)

- Des méthodes spécifiques
 - Comportement propre pour chaque classe
 - Héritage possible du code de la classe mère si nécessaire
- Code très simplifié :

```
Pour chaque objet :  
    Appeler Afficher() de objet  
Fin Pour
```



Mise en œuvre

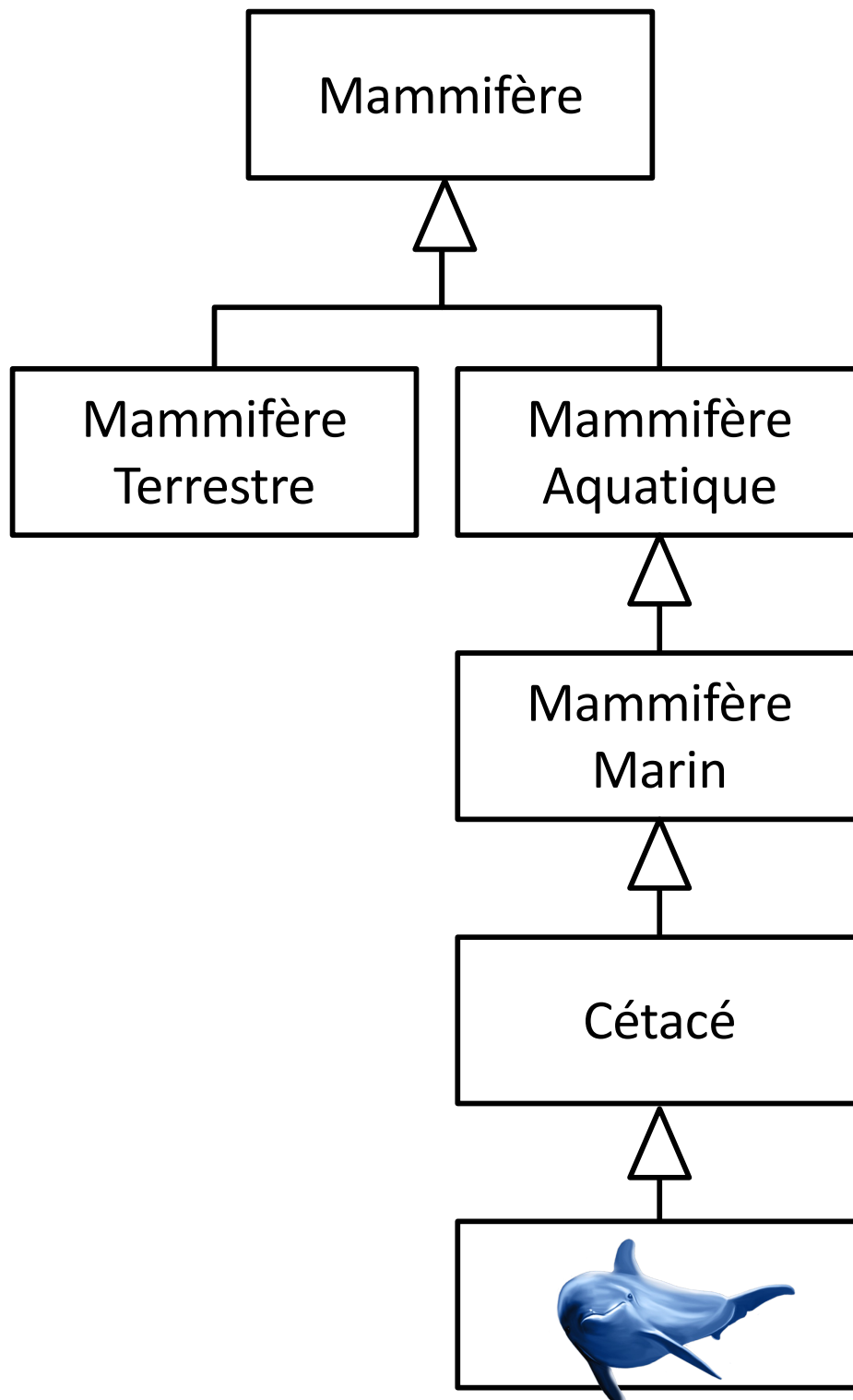


Avantages de l'héritage (1)

- Partage de code
 - Réutilisabilité
 - Fiabilité
 - Le code des classes les plus hautes dans la hiérarchie est utilisé le plus souvent : il gagne vite en fiabilité
- Modélisation d'un concept très naturel

Avantages de l'héritage (2)

- Taille de code source plus faible (factorisation)
- Maintenance facilitée
 - Modifier l'interface d'une classe fille n'impose pas de recompiler toutes celles qui sont au dessus
 - Modifier l'implémentation d'une classe n'impose pas de recompiler la hiérarchie



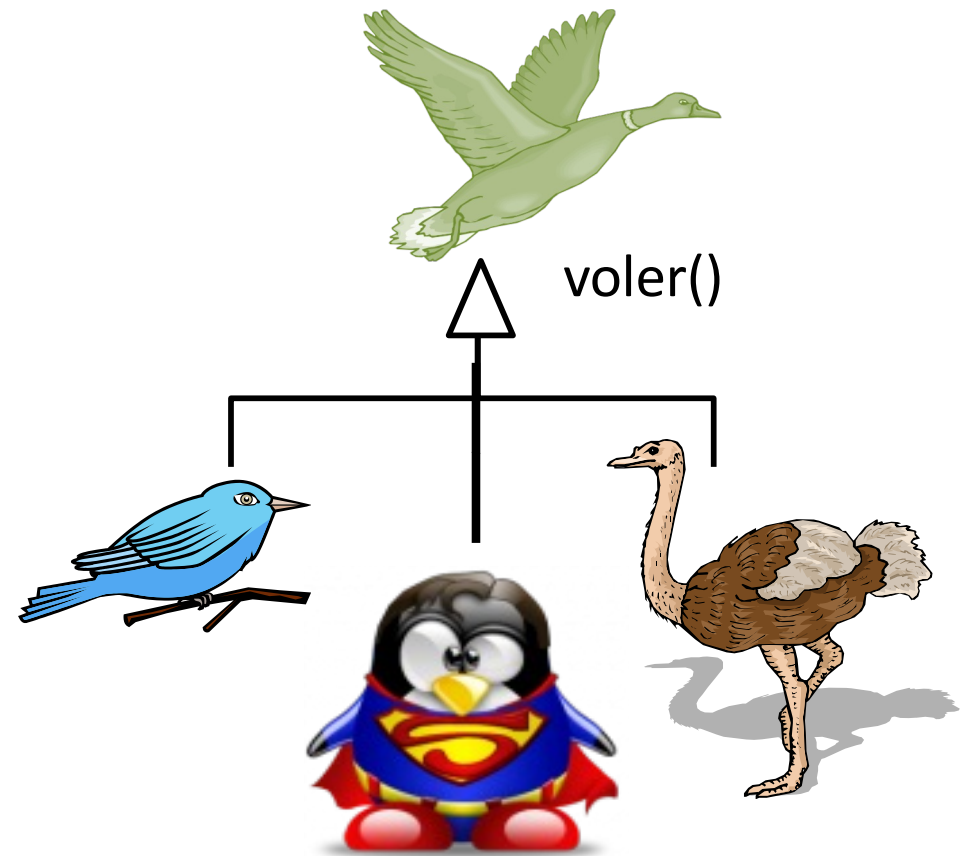
Dangers de l'héritage (1)

- Attention à la hiérarchie !
- Trop lourde, elle nuit à l'efficacité du code !

Dangers de l'héritage (2)

- Violation du principe d'encapsulation
 - Règle : transmission intégrale de la mère à la fille
 - Erreurs sémantiques (Manchot / *penguin*)
- Héritage **sélectif** avec certains langages !

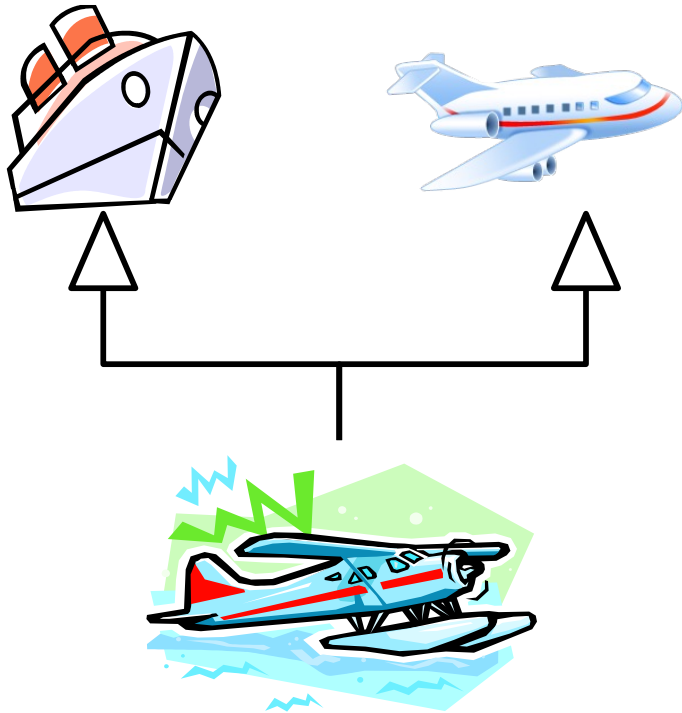
Choisir les méthodes ou les attributs à faire hériter



Dangers de l'héritage (3)

- Héritage de construction
 - Rajouter des méthodes sans conserver la notion de généralisation/spécialisation
 - Dériver rectangle de ligne en rajoutant une largeur
 - Dériver là où une différence d'attribut suffirait
 - Dériver des animaux en fonction de la couleur du pelage \Rightarrow manque de discrimination fonctionnelle

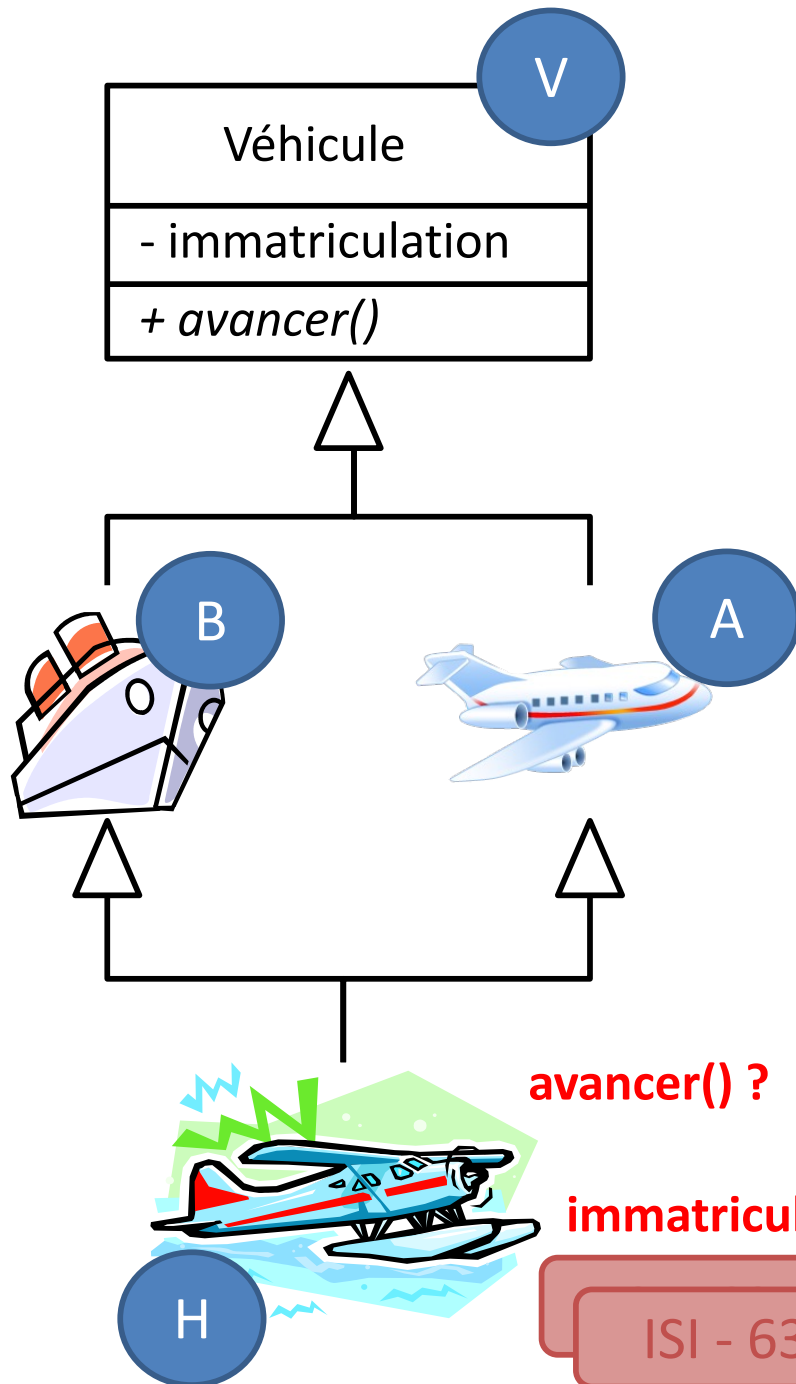




Héritage multiple

- Plusieurs classes mères pour une même classe fille !
- Modélisation de principes naturels
- Risques de collision de noms sur les classes parent
 - Certains langages permettent de préfixer les noms des membres
- Se transforme très rapidement en héritage à répétition !

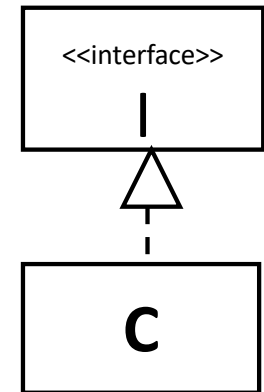
Héritage à répétition

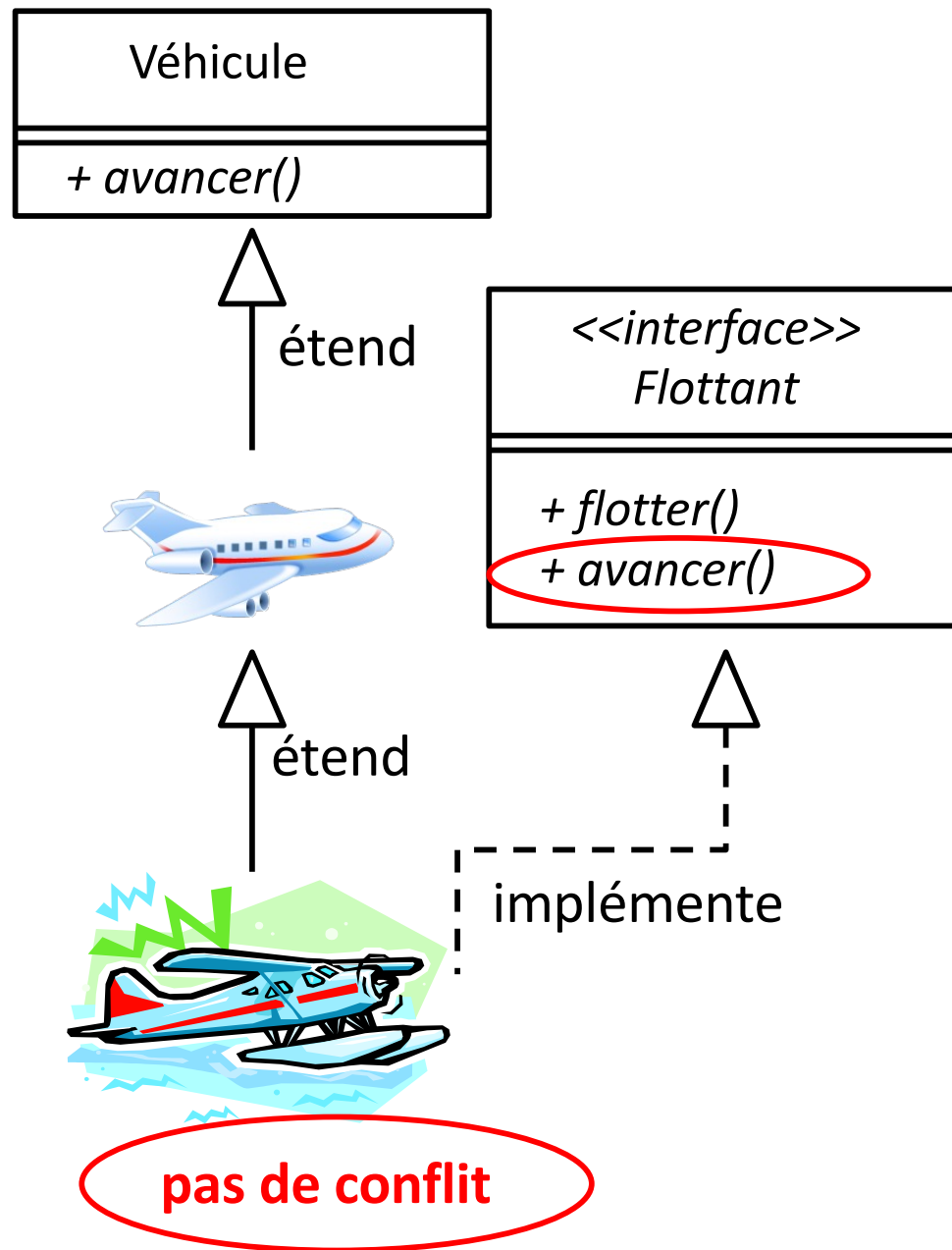


- Duplication de V dans H
 - (une fois via B et une fois via A)
 - `avancer()` ?
 - Immatriculation ?
- Très fréquent avec les bibliothèques
- Mécanismes spécifiques (C++)

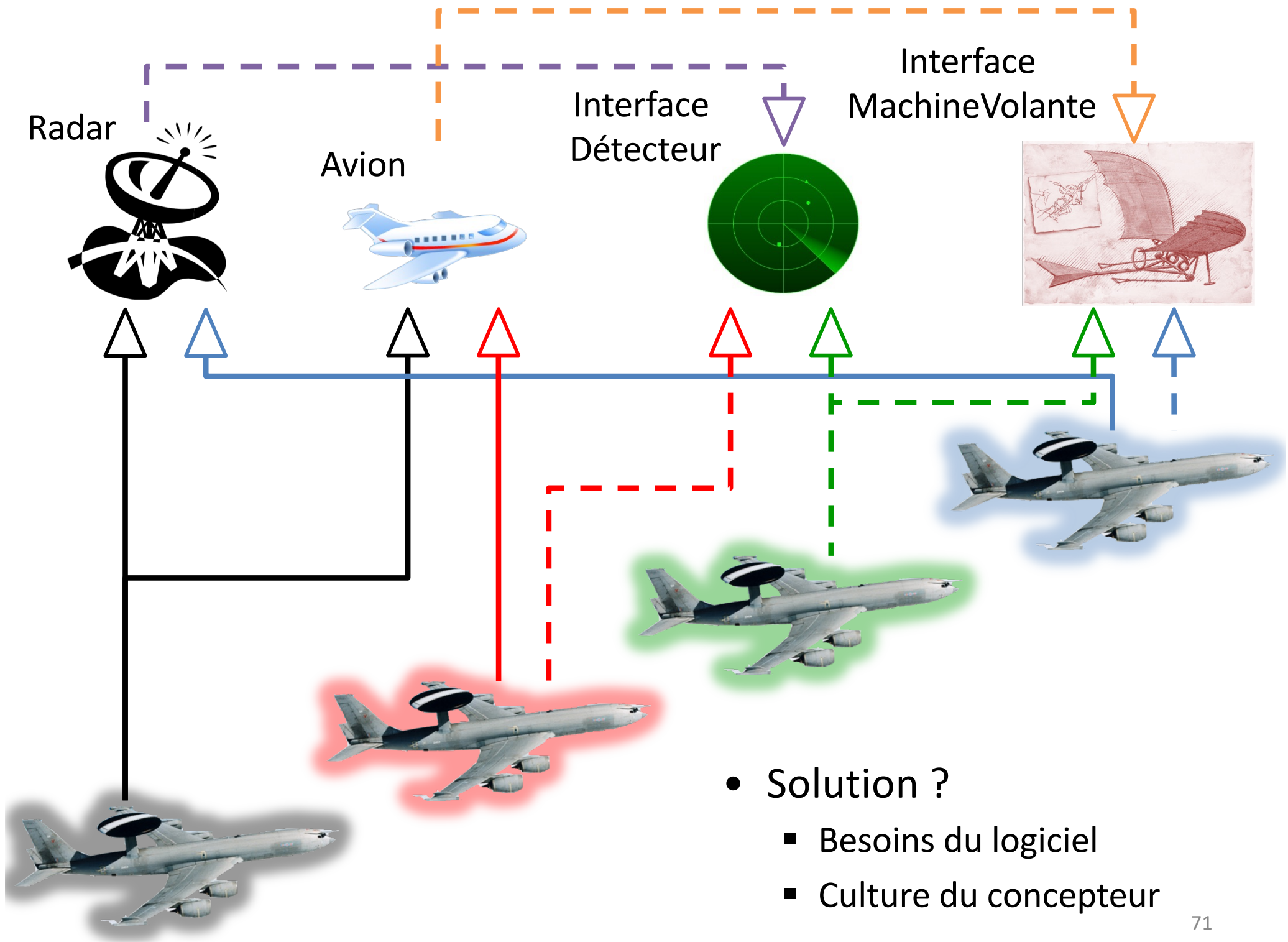
Interface

- Formalisation de la notion d'interface
 - Comportement de la classe
 - Ensemble de méthodes virtuelles
 - Similaire à une classe abstraite sans attribut
 - Suivant langage ? Attribut ? Code ?
- Vocabulaire :
 - Une classe **implémente** une interface
- Interfaces multiples
 - Alternative à l'héritage multiple !


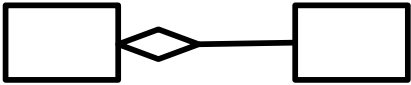




- Complémentarité de l'héritage simple et interfaces multiples
 - Hériter du concept primordial
 - Implémenter des interfaces



Composition / Agrégation

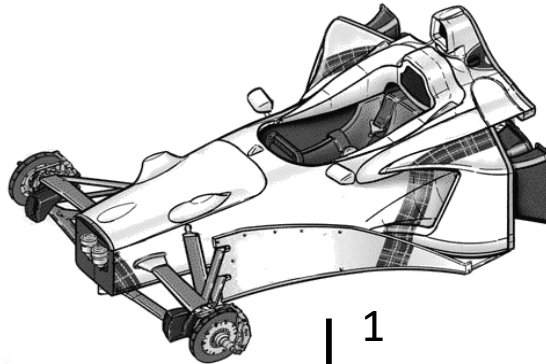
- Modélisation de la relation « est composé de »
 - Composition, appartenance, groupage, « Has A »
- Composition 
 - Les constituants disparaissent avec le composé
- Agrégation 
 - Les composants survivent à l'agrégat
- Modélisation des **conteneurs**
- Notion de cardinalité
 - Simple / multiple

Objet agrégé

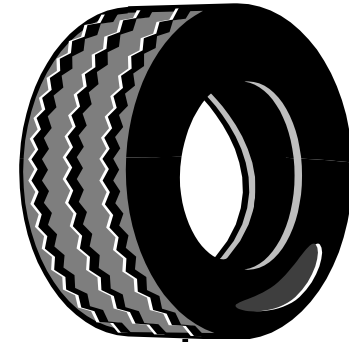
Moteur



Chassis



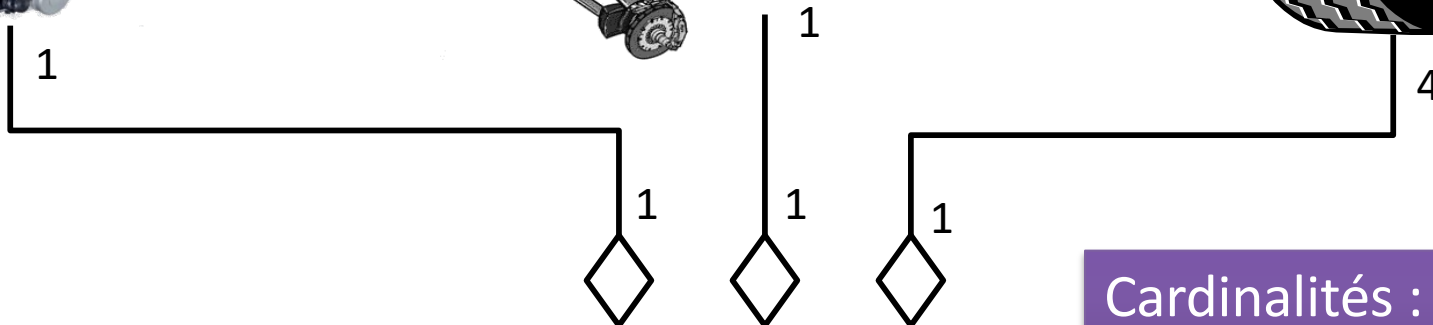
Roue



Agrégateur

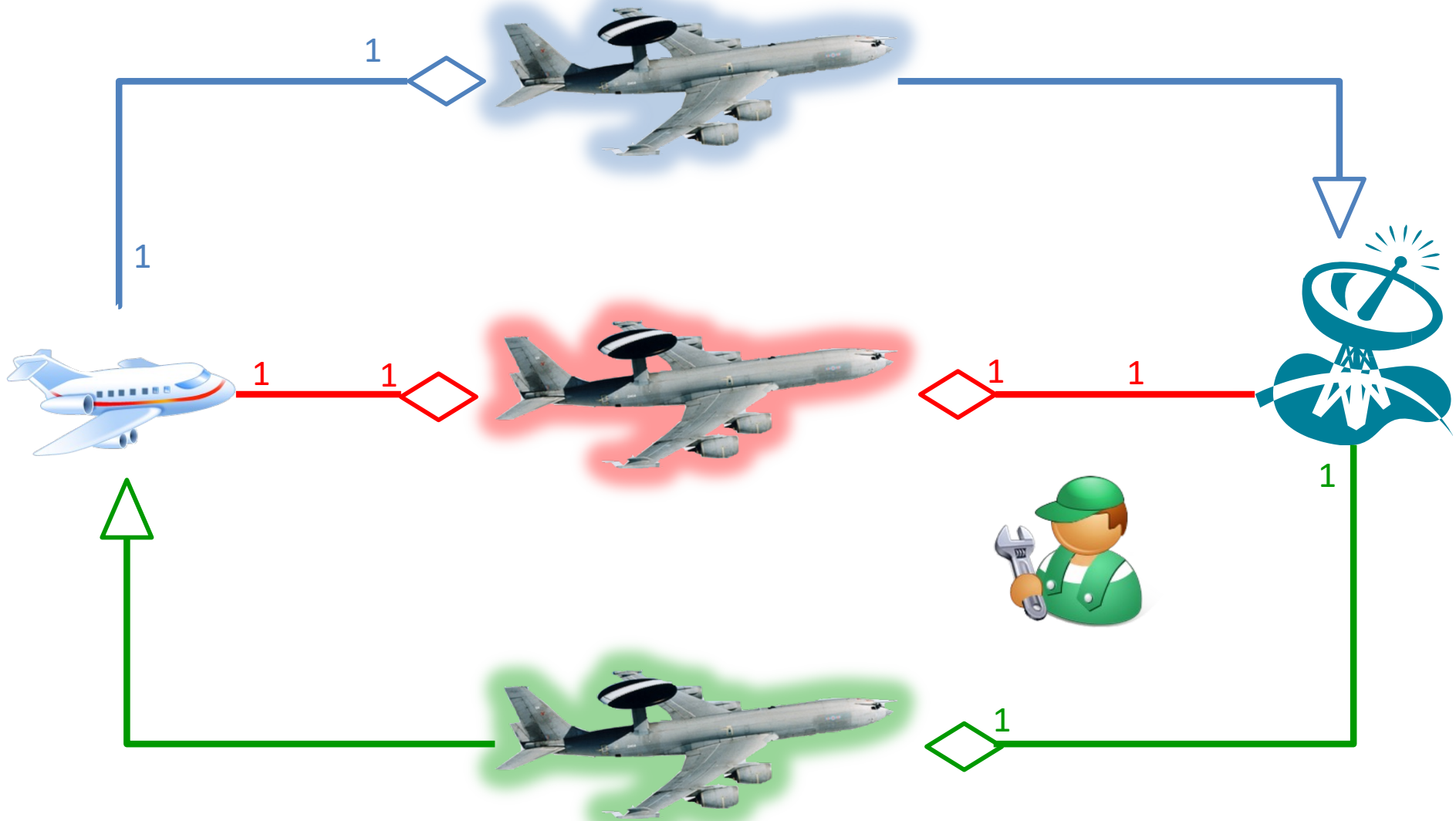


Voiture



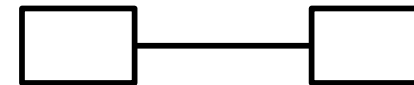
Cardinalités :
Une voiture agrège
quatre roues

Alternative à l'héritage multiple ?

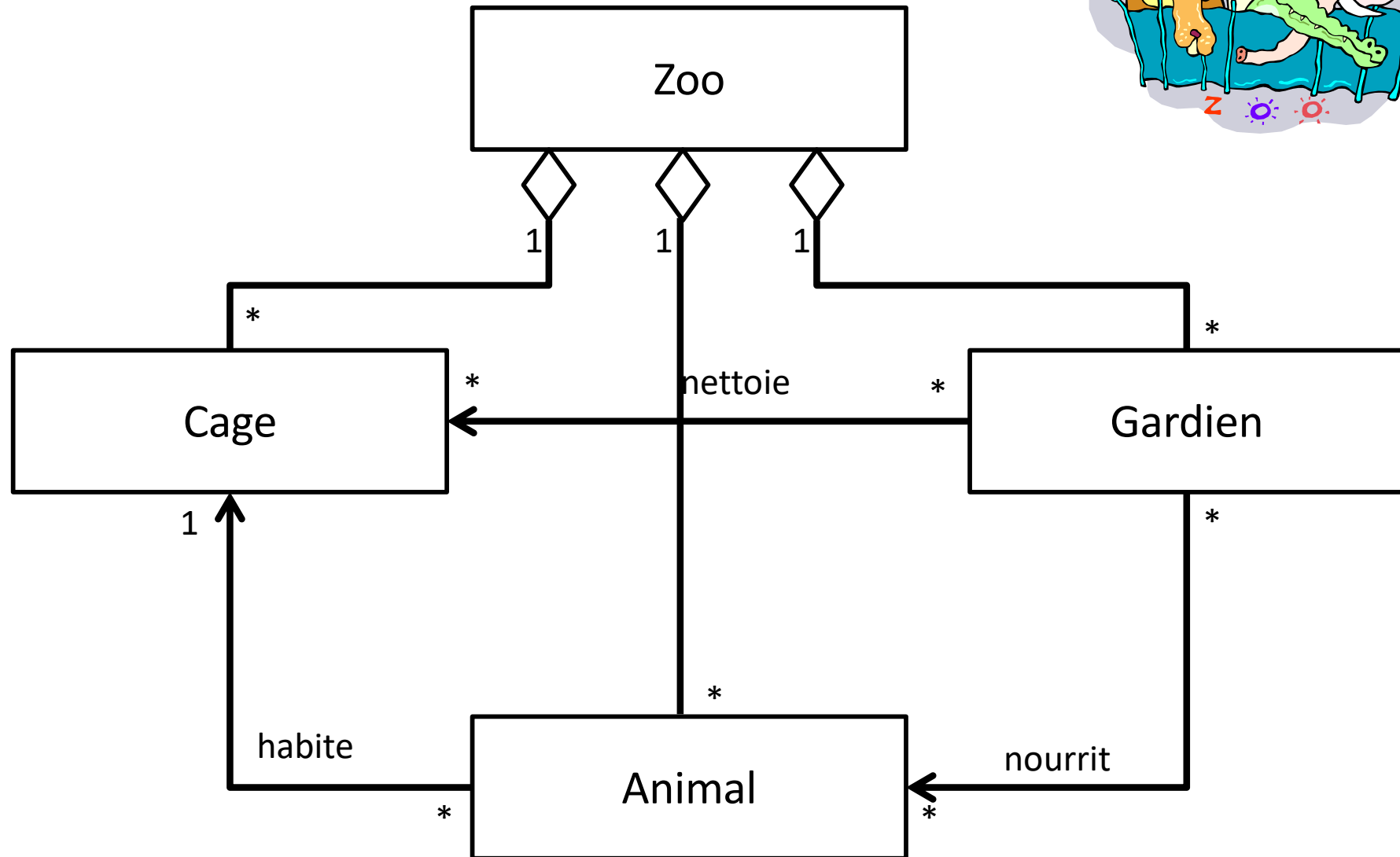
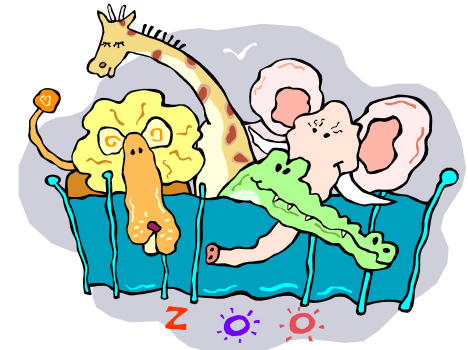


Association

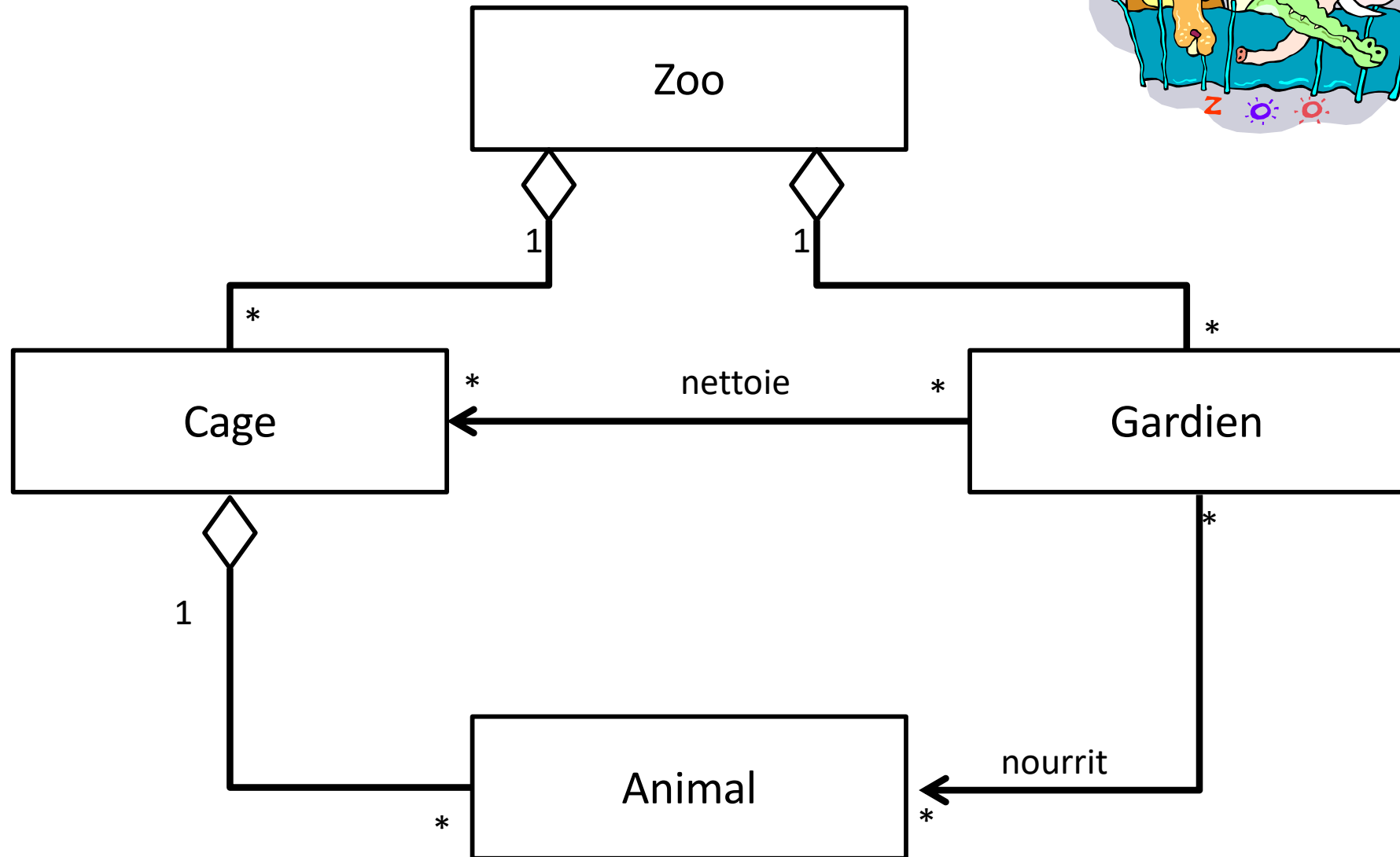
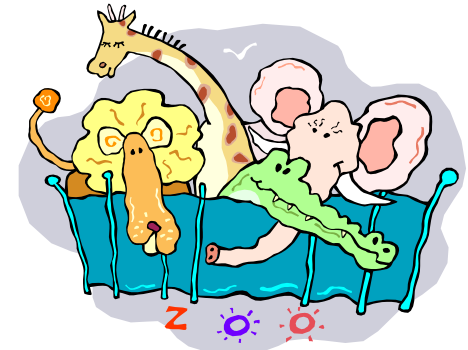
- Modélise des concepts assez flous
 - Uses A
 - Est en association avec
 - Communique avec
- Caractéristiques
 - Habituellement nommée (auto documentation)
 - Sens
 - Cardinalités : 1 .. 4
- Souvent difficile à différencier de l'agrégation
 - Concepts voisins
 - Même implémentation



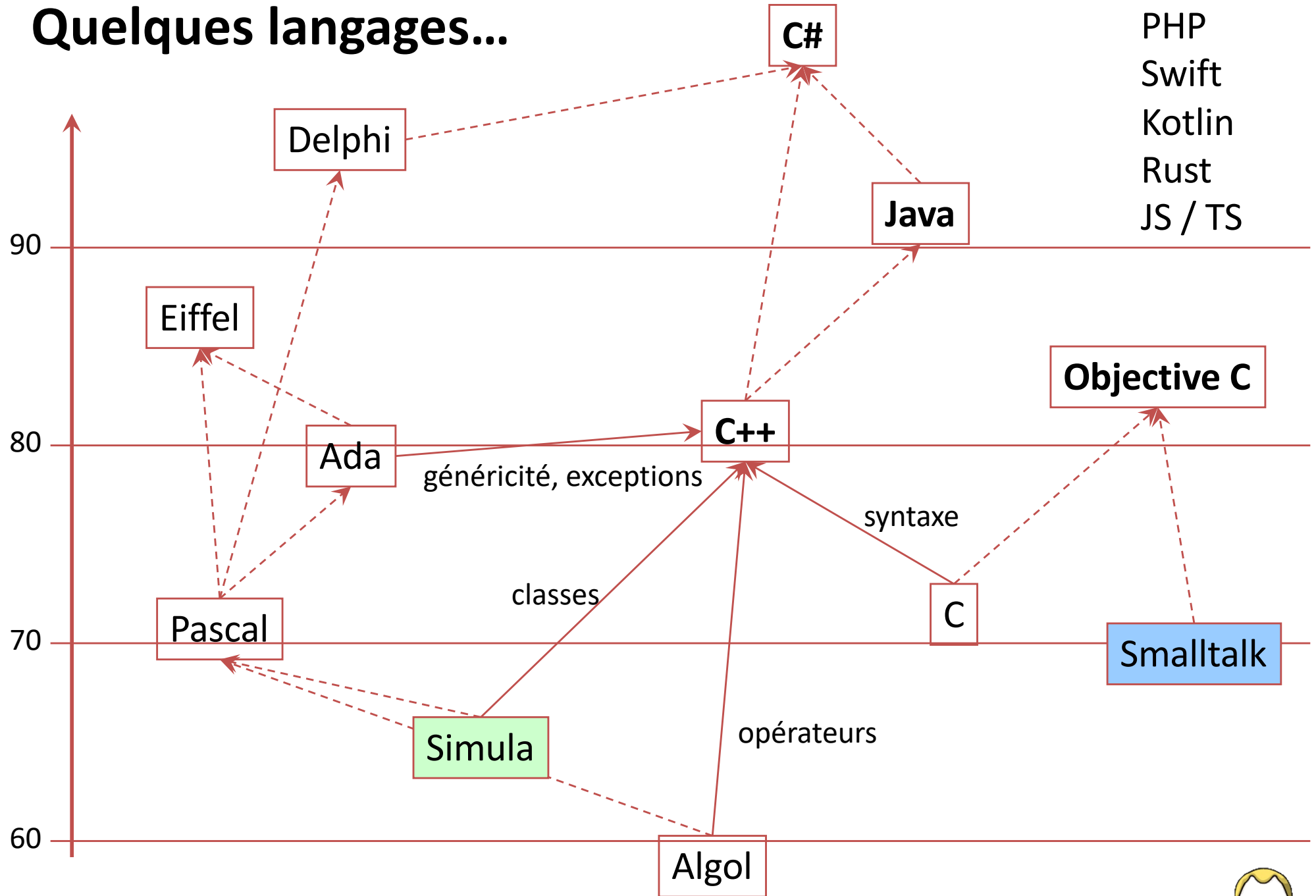
ZOO – V1



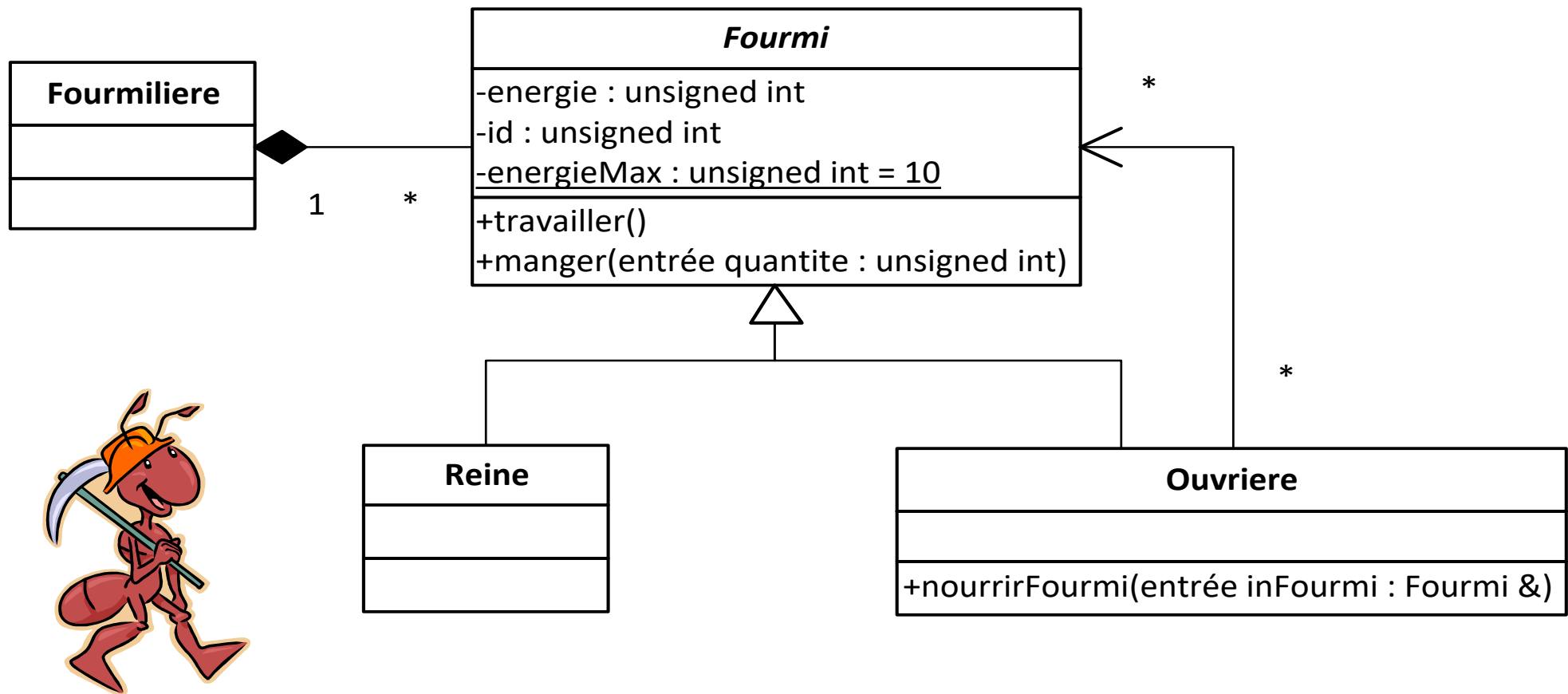
ZOO – V2



Quelques langages...



Fourmi ZZ ?



Et maintenant ?

- Modélisation et UML
- Mettre en pratique la POO
 - Langage ?
- *Design patterns*
 - *Problèmes / solutions “classiques”*