Conception et Programmation Orientées-Objet

INFOB234 – IHDCB237
2018 – 2019

Professeur : Patrick.Heymans@unamur.be
Assistants : Tony.Leclercq@unamur.be
Julie.Henry@unamur.be
Victor.Amaral@unamur.be

UNIVERSITÉ DE NAMUR

Préambule

Contrat pédagogique

- Prérequis
  - « programming in the small » (algorithmique, procédures, pointeurs, structures de données de base...)
  - comprendre et écrire des spécifications (pré/post)

- Ce que vous allez apprendre
  - concevoir (*design*) des logiciels de qualité (fiables, maintenables,...) selon les principes de l'orienté-objet
    - « programming in the large »
    - utilisation de mécanismes d'abstraction puissants et éprouvés
  - implémenter ces *designs* en Java
  - « apprendre à apprendre » Java et d'autres langages et techniques OO

Contrat pédagogique

- Ce que vous **n'**allez **pas** apprendre: le hacking
  - devenir des « codeurs » Java (pas besoin d'aller à l'unif pour ça !)
  - les moindres détails de la syntaxe Java (mais vous saurez où les trouver au cas où)

- Leitmotiv du cours
  - De-sign, not de-bug !

Contrat pédagogique

- Support *officiel* au niveau des **connaissances**
  - Slides (disponibles sur http://webcampus.unamur.be/)
  - le livre de Liskov et Guttag (les pages référencées dans les slides)
  - tout ce que le prof dit au cours !

Contrat pédagogique

- Attentes au niveau du **savoir-faire**
  - cf. les TPs
- Examen:
  - écrit, 4h
  - porte sur les connaissances (le cours) **et** le savoir-faire (les TPs)

Enseignement mixte

Ex-Cathedra + TPs

- Ch. 1 – Introduction
- Ch. 2 – Objets et Java
- Ch. 3 – Abstraction procédurales
- Ch. 5 – Abstraction de données
- Ch. 6 – Hiérarchie des données
- Ch. 9 – Specification-driven testings (en)

Classes inversées (TP)

- Ch. 4 – Exceptions
- Ch. 7 – Abstraction de l'itération
- Ch. 8 – Génériques

Classes inversées ?

- L'étudiant ne donnera pas cours… (rassurez-vous)
- Cours collégial guidé !
- Plus d'explications lors du premier TP !

Recommandations

- Ne vous laissez pas dépasser !
  - En programmation OO encore plus que dans d'autres matières, tout est dans tout
- Le prof est interruptible et souhaite savoir quand vous ne comprenez pas

De-sign, not de-bug

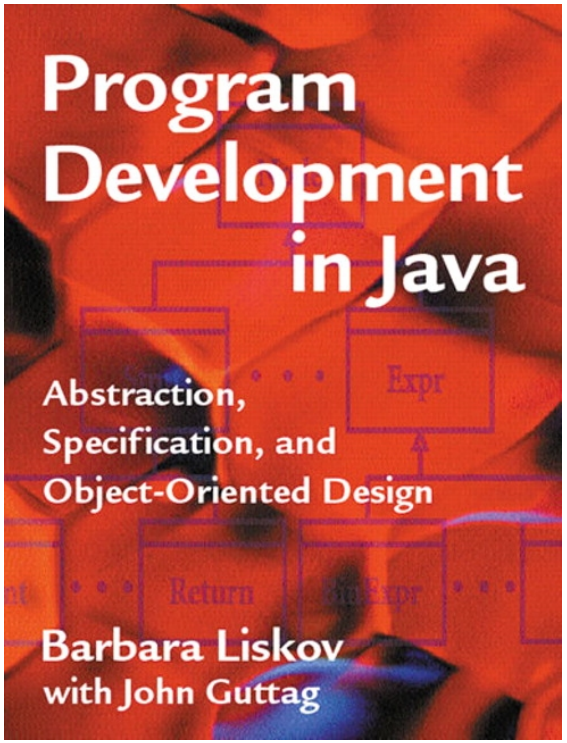
- Réfléchir d'abord, ensuite coder
- Bien plus efficace et bon marché que débugger !
- Rend possible la délégation et le travail en équipe
- Les faiblesses de design impactent directement les utilisateurs: incohérences, procédures trop rigides, manque d'ergonomie du logiciel
- Les faiblesse de design impactent également les développeurs: abondance de bugs, changements compliqués et coûteux…

Comment obtenir un bon design ?

- Utiliser des langages de programmation, des librairies, frameworks **appropriés**
  - apportent un certain support mais l'influence est limitée …
  - … et ils deviennent inévitablement obsolètes
- Suivre un design **approprié**
  - amène une différence réelle/notable
  - valable à long terme
  - repose sur les programmeurs !

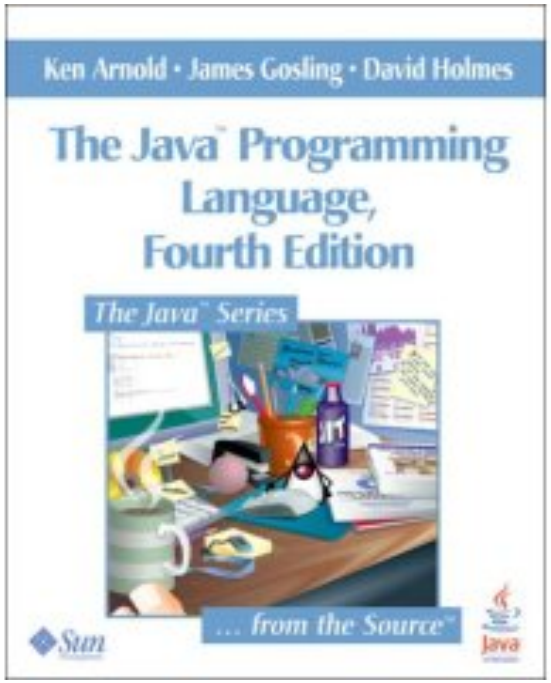
Bibliographie

- Le cours se base sur (i.e. il est fortement recommandé de se procurer) :
  - B. Liskov et J. Guttag « Program Development in Java : Abstraction, Specification and Object-Oriented Design », Addison-Wesley, 2001



Bibliographie

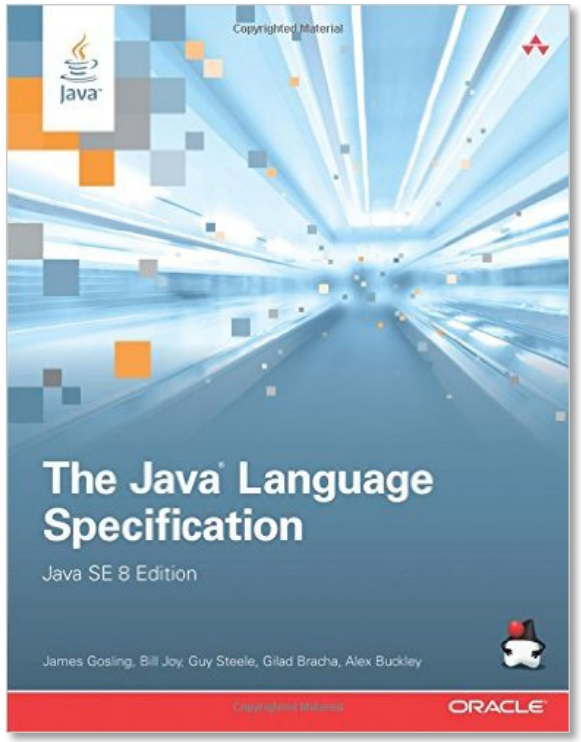
- Références sur le langage Java
  - K. Arnold, J. Gosling et D. Holmes, « The Java Programming Language », 4th edition, Prentice-Hall



Bibliographie

- Références sur le langage Java

  - J. Gosling, B. Joy, G. Steele,
    G. Bracha, et A. Buckley,
    « The Java Language Specification,
    Java SE 8 Edition »,
    Addison-Wesley
    (http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf)



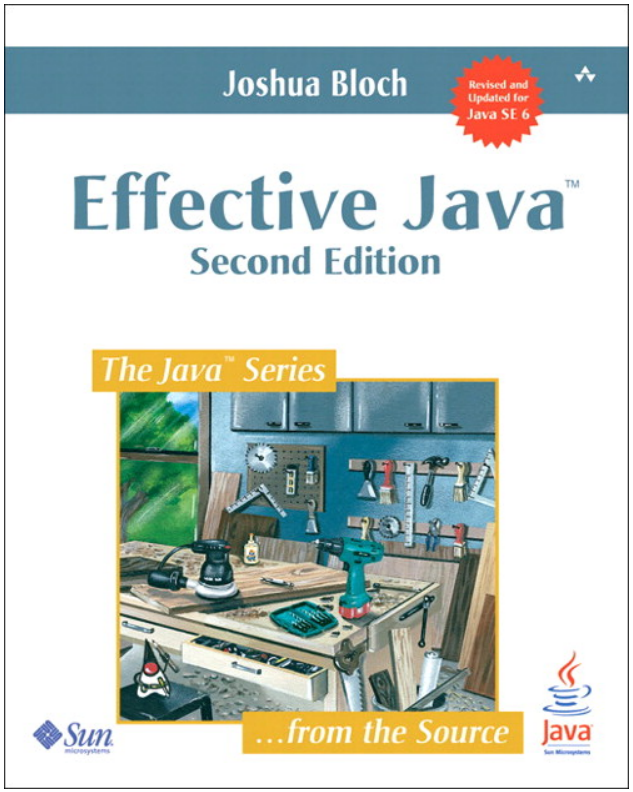
Bibliographie

- Références sur le langage Java

  - J. Bloch,

    « Effective Java», 2nd edition,

    Addison Wesley, 2008

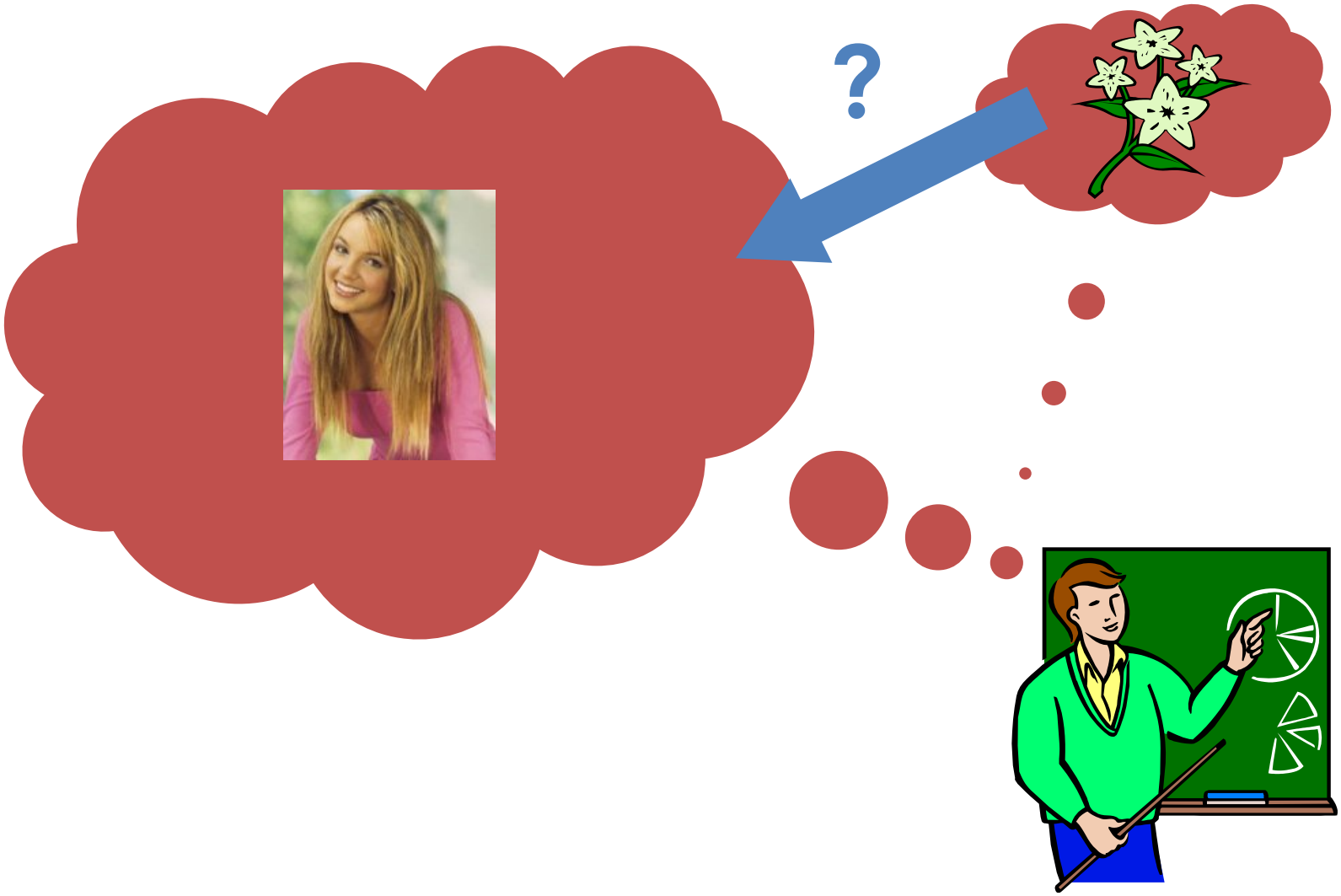


- Attention : les « bonnes pratiques » renseignées dans ces livres divergent parfois de celles vues au cours. C'est le cours qui prime !

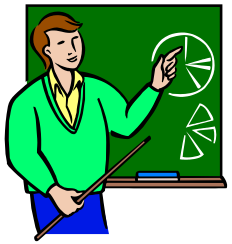
Chapitre 1
Introduction

Voir Liskov&Guttag,
ch. 1 « Introduction », pp 1-13

La modularité : une métaphore romantique et fleurie



La modularité : une métaphore romantique et fleurie



Jardiniers

Horticulteur

Grossiste

Gorille

Livreur

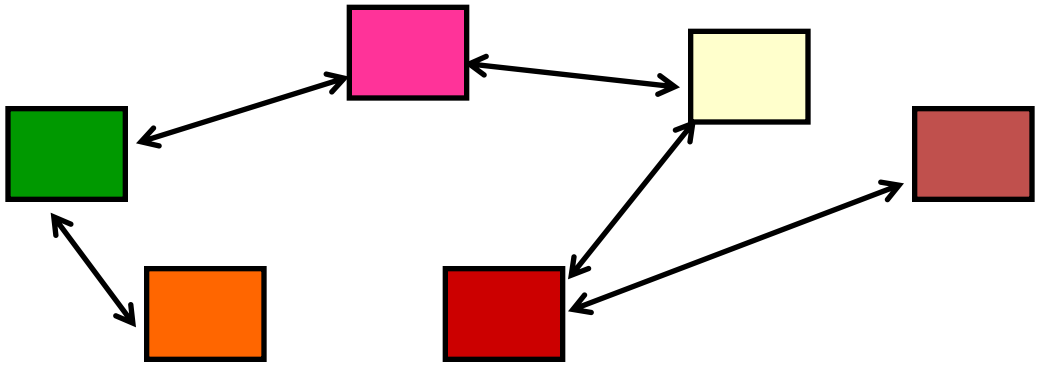
Marguerite
(employée du "Bia bouquet", fleuriste Namurois)

Interflora

Margaret
(employée du "Guns n' Roses" fleuriste de L.A.)

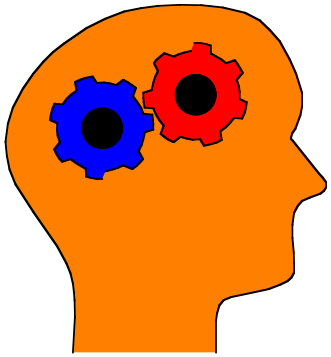
Modularité : définition

- La modularité est la caractéristique qu'ont les programmes décomposés en modules !
- Les modules ont des **responsabilités bien définies** et **coopèrent** pour honorer les obligations du programme.
- Un module peut **déléguer** une partie de ses responsabilités à un autre.

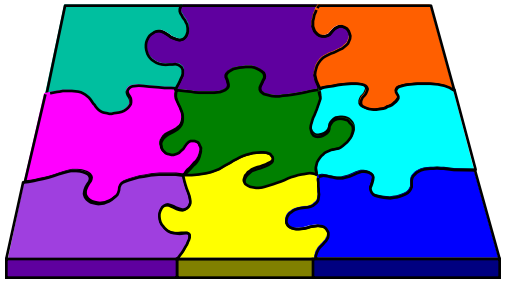


- La modularité est une forme particulière de **décomposition**

Modularité: analogie



L'intelligence d'un être humain ne se trouve pas dans **un** neurone, mais dans la collaboration des neurones du cerveau !



L'intelligence d'un système logiciel ne se trouve (idéalement) pas dans **un** module, mais dans la collaboration des modules du système.

Pourquoi modulariser?

- Soit un programme rédigé en un seul bloc monolithique
- Plus il grandira, plus il sera difficile
  - à lire,
  - à comprendre,
  - à modifier.
- Or, la plupart des programmes « réels » évoluent et sont sujets à une longue maintenance
- Dans un programme **modulaire**, les effets ci-dessus sont fortement atténués
- **Qualité essentielle d'un bon programmeur : savoir bien modulariser !**

Tailles des programmes

- **Petits programmes** : mini-traitement de texte, calcul des salaires d'une PME, gestion des prêts d'une vidéothèque ...
  - Développeurs : 1 à 5 (10)
  - Budget : x 10.000 EUROs
  - NLOC : x 1.000 - 10.000
  - Durée développement : 1 mois - 1 an
  - Durée de vie : 2 - 5 ans
  - Types d'utilisateurs : 1 à 5

- **Systèmes d'Information** (SI) **moyens** : vente par correspondance, gestion d'hôpital, gestion de grand magasin ...
  - Développeurs : 5 à 10 (50)
  - Budget : x 100.000 EUROs
  - NLOC : x 10.000 - 100.000
  - Durée développement : 6 mois - 2 ans
  - Durée de vie : 2 - 5 ans
  - Types d'utilisateurs : 5 à 10

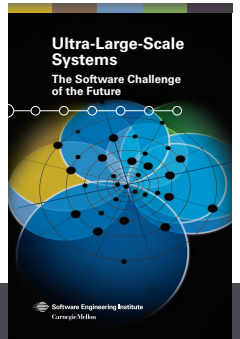
Tailles des programmes

- **Grands SI** : réservation dans une chaîne d'hôtels, bourse, amazon.com, contrôle du trafic aérien,...
  - Développeurs : x 100
  - Budget : x 1.000.000 EUROs
  - NLOC : x 1.000.000
  - Durée développement : 6 mois - 5 ans
  - Durée de vie : 2 - 5 - 10 ans et plus
  - Types d'utilisateurs : x 10

Les programmes d'aujourd'hui et de demain
(Extrait de "Ultra-Large-Scale Systems", SEI, 2006)

The scale and complexity of systems is increasing dramatically. Ultra-large-scale (ULS) systems are systems of unprecedented scale in some of these dimensions:

- lines of code
- amount of data stored, accessed, manipulated, and refined
- number of connections and interdependencies
- number of hardware elements
- number of computational elements
- number of system purposes and user perception of these purposes
- number of routine processes, interactions, and “emergent behaviors”
- number of (overlapping) policy domains and enforceable mechanisms
- number of people involved in some way

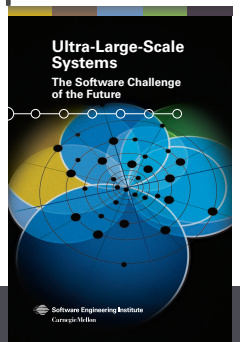


Les programmes d'aujourd'hui et de demain

(Extrait de "Ultra-Large-Scale Systems", SEI, 2006)

How are ULS systems different?

The sheer scale of ULS systems changes everything. ULS systems will necessarily be **decentralized** in a variety of ways, developed and used by a **wide variety of stakeholders** with conflicting needs, **evolving continuously**, and constructed from **heterogeneous parts**. People will not just be users of a ULS system; they will be elements of the system. Software and hardware **failures will be the norm** rather than the exception. The acquisition of a ULS system will be simultaneous with its operation and will require new methods for control. **These characteristics may appear in today's systems and systems of systems, but in ULS systems, they will dominate**. Consequently, ULS systems will place unprecedented demands on software acquisition, production, deployment, management, documentation, usage, and evolution practices.



Décomposer, c'est Machiavélique...

Nicolas Machiavel (1469-1527)

Divide ut imperes !
(Diviser pour régner !)

- Décomposer un problème = le diviser en sous-problèmes
  - de même niveau de détail
  - pouvant être résolus indépendamment
  - dont les solutions, combinées, résolvent le problème d'origine

Décomposition

- Une *bonne* décomposition (en modules) **accélère le développement** en :
  - **facilitant la compréhension** du code,
  - permettant de **travailler en parallèle**,
    - on peut modifier le code localement sans compromettre le fonctionnement global tant que les responsabilités restent inchangées
  - permettant de **réutiliser** des modules existants
- Une *mauvaise* décomposition est une source insidieuse de problèmes :
  - chaque module peut fonctionner comme demandé mais l'ensemble ne produit pas les résultats escomptés
  - très problématique car on ne s'en aperçoit souvent que très tard : au moment des tests d'**intégration** ou quand un bug surgit lors de l'utilisation du logiciel

Attention aux mauvaises décompositions

Scénario

Auteur 1

Auteur 2

Répliques de Darth Vader

Répliques de Luke Skywalker

Fiston, faudrait vraiment que tu deviennes un gros méchant. C'est trop top cool.

Allléééé-eeuuu, on irait casser de l'alien tous les deux dans ma soucoupe… Comment qu'on rigolerait, j'te raconte pas…

Oh, Père, vous voici donc. Comment allez vous ? Vos maux de gorge, tout ça… .

Tiens, il faut que je vous dise: j'ai réussi mes examens.

Je comprends pas. Exploser des grosses bêtes bleues à tentacules, tu trouves pas ça rigolo? Ah, t'es bien comme ta mère!

Attention aux mauvaises décompositions

Abstraction

- Une *bonne* décomposition est guidée par l'**abstraction**
- L'abstraction est la faculté d'**oublier** momentanément les *détails* d'un problème pour **se focaliser sur les aspects intéressants**
  - NB : La notion de détail est relative (contextuelle), pas absolue
- Deux formes d'abstraction que vous avez déjà utilisées :
  - Abstraction par **paramétrisation**
  - Abstraction par **spécification**

Abstraction par paramétrisation

```
…
// user enters a sequence of (max 20) characters read(a)
// check there are no special characters
found ← false
i ← 1
while (i≤20 and found= false)
    if (a(i)='.' or a(i)=';' or a(i)='_' or ... )
    then
      print('Forbidden character!')
      found← true
    end-if
    i ← i+1
end-while
```

(1)

```
…
// user enters a sequence of (max 10) characters read(w)
// check there are no special characters
found← false
j ← 1
while (j≤10 and found= false)
    if (w(j)='.' or w(j)=';' or w(j)='_' or ... )
    then
      print('Forbidden character!')
      found← true
    end-if
    j ←  j + 1
end-while
…
```

(2)

Abstraction par paramétrisation

- Abstraction de (1) et (2) en une routine* qui signale l'existence de caractères spéciaux dans *n'importe quelle chaîne de caractères* de *n'importe quelle longueur*

```
boolean verifieChaine(chaineCar a, int longueur)
begin
   int i
   for i:=1 to longueur
    if (a(i)='.' or a(i)=';' or a(i)='_' or ... )
    then
        print('caractère interdit!')
        return false
    end-if
   end-for
   return true
end
```

* Une routine (procédure, fonction,...) est une forme de module

Abstraction par paramétrisation

- Le programme devient

```
…
// l'utilisateur entre une chaîne de caractères (20 max)
read(a)
// appel à la routine de vérification
trouve ← verifieChaine(a,20)
...
// l'utilisateur entre une chaîne de caractères (10 max)
read(w)
// appel à la routine de vérification
trouve ← verifieChaine(w,10)
...

boolean verifieChaine(chaineCar  a, int longueur)
begin
...
end
```

paramètres réels ou effectifs (actual parameters)

paramètres formels (formal parameters)

Abstraction par paramétrisation

- Abstraction **par rapport à l'identité des données** traitées en les remplaçant par des paramètres
- Permet au même module d'être utilisé dans un plus grand nombre (une infinité) de situations
  - une infinité d'exécutions peuvent donc être représentées de manière très succincte
- Indispensable et ultra-répandu parmi les langages de programmation mais insuffisant

Abstraction par spécification

```
float myMethod (float coef)

begin
  float ans ← coef/2.0
  int i ← 1
  while (i<7)
   ans ← ans - ( (ans*ans - coef)/(2*ans) )
   i ← i+1
  end-while
  return ans
end
```

Abstraction par spécification

```
float sqrt (float coef)
// pre : coef > 0
// post : retourne une approximation de √coef
begin
  float ans ← coef/2.0
  int i ← 1
  while (i<7)
   ans ← ans - ( (ans*ans - coef)/(2*ans) )
   i ← i+1
  end-while
  return ans
end
```

Abstraction par spécification

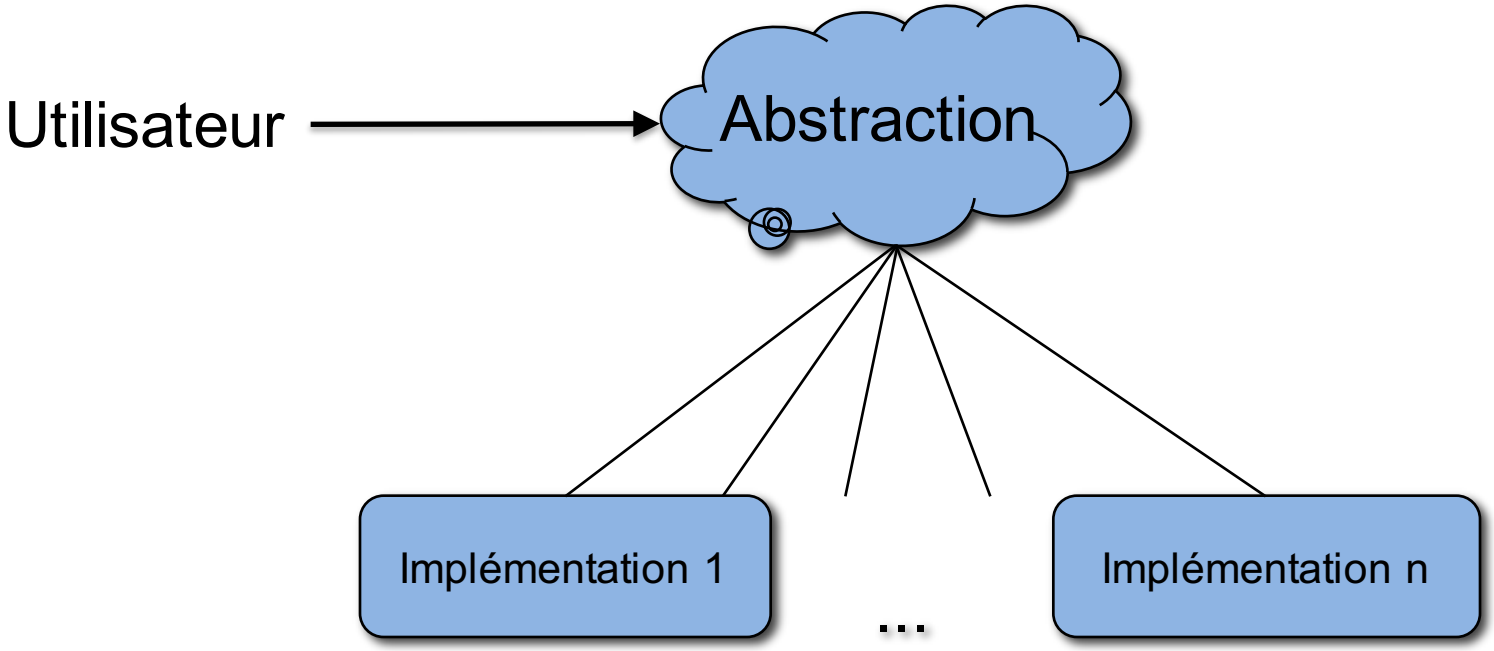
- Une bonne technique d'abstraction par spécification est l'utilisation de **pré-** et de **post-conditions**

- Rien qu'en lisant ces **assertions**, on déduit que
  - l'appel y ← `sqrt(x)` donnera à y la valeur de √x, pour peu que x>0
  - on ne peut rien attendre d'utile de cet appel si x=<0

Abstraction par spécification

- Abstraction **par rapport à l'implémentation**
  (= le ***comment***) d'une abstraction
- Focalisation sur les **responsabilités** (= le ***quoi***) assumées par l'abstraction
- Les spécifications d'une abstraction devraient :
  - dire à l'utilisateur (= le « client ») de l'abstraction ce qu'il peut en attendre
    (= ce que l'abstraction fait et ce qu'elle ne fait pas)
  - dire au programmeur ce que l'implémentation doit faire pour satisfaire l'utilisateur de l'abstraction
- Les spécifications sont un **contrat**
  - L'utilisateur et le programmeur se basent dessus pour accomplir leur travail

Abstraction par spécification



- Quand un utilisateur utilise une abstraction, cette dernière devrait fonctionner comme l'utilisateur s'y attend, quelque soit l'implémentation sous-jacente

Spécifications = un contrat

- L'utilisateur promet de satisfaire aux pré-conditions

```
f ()
      // pré-conditions
      // post-conditions
```

Si les pré-conditions sont vraies,
après l'appel à f (),
les post-conditions sont vraies.

- Le programmeur promet que, si l'utilisateur satisfait aux pré-conditions, la valeur et l'état de retour quand la fonction se terminera satisferont les post-conditions

Avantages des spécifications

- **Compréhension rapide** du rôle d'une abstraction (pas nécessaire de lire le code)
- **Indépendant des choix d'implémentation**
  - Algorithmes, types de variable… voire même le langage
  - Deux implémentations peuvent varier au niveau de leurs propriétés non-fonctionnelles (e.g. leur performance)
- Les spécifications fournissent deux propriétés fondamentales:
  - **Localité**
  - **Modifiabilité**

Localité and Modifiabilité (Découplage)

- **Localité** : l'implémentation d'une abstraction peut être lue/écrite sans avoir besoin de consulter **l'implémentation** d'aucune autre abstraction

- **Modifiabilité** : une abstraction peut être ré-implementée sans ré-implementer aucune autre partie du programme

Localité and Modifiabilité (Découplage)

- Les propriétés de **localité** et **modifiabilité** permettent l'**indépendance des tâches de programmation**, ce qui inclus :
  - les modules dépendants
  - Les cas de test (en tout cas les tests « black-box »)

- Ce **découplage**
  - permet une parallélisation sûre et performante de la programmation
  - Limite drastiquement l'impact des changements

- Globalement, il permet une **ENORME** réduction du temps nécessaire et des coûts liés à la programmation

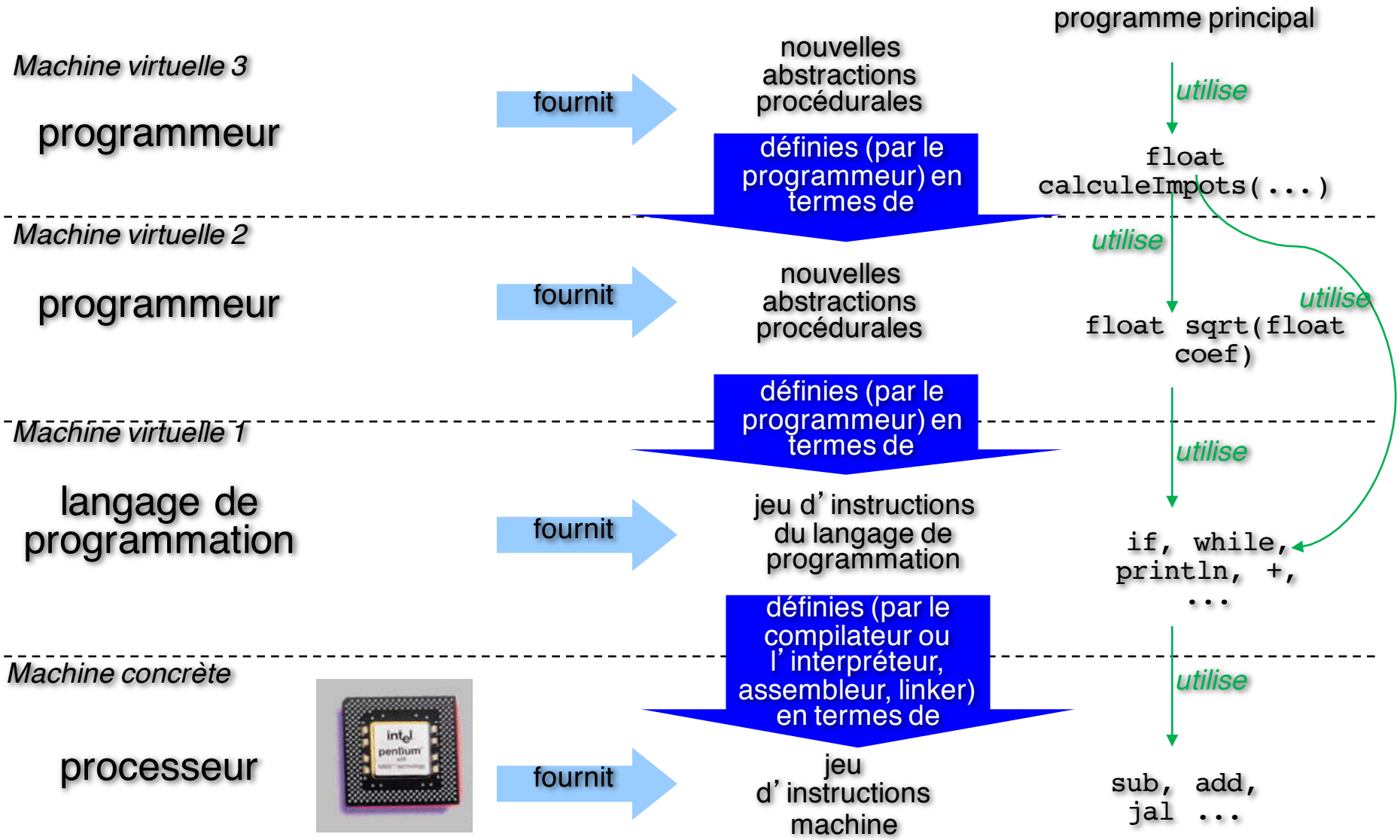
Types d'abstractions

- Le 2 techniques de base ...
  - abstraction par paramétrisation
  - abstraction par spécification
- ... permettent de définir 3 techniques avancées :
  - **abstraction procédurale** (procedural abstraction)
  - **abstraction de données** (data abstraction)
    + abstraction par hiérarchie de types
  - **abstraction de l'itération** (iteration abstraction)

Abstraction procédurale

- **Extension de l'ensemble des opérations/procédures/fonctions/routines** fournies par la *machine virtuelle/abstraite* définie par le langage de programmation
- Exemple : `float sqrt (float coef)`
- Utilise les 2 techniques d'abstraction de base
- Convient quand le problème se prête bien à une décomposition de type *fonctionnel*

Abstraction procédurale



Abstraction de données

- **Extension de l'ensemble des types de données** fournis par la machine virtuelle du langage
- Autour d'un type de données, on regroupe les opérations qui le concernent
  - **CRUD** (create, read, update, delete)
  - … et toutes les autres opérations « métiers » !
- On parle de **type de données abstrait** aka **Abstract Data Type** (**ADT**) aka Type Abstrait Algébrique (TAA)

Abstraction de données

- Exemple: un « multiset » (a.k.a. « bag »)
  - Les multisets généralisent la notion d'ensemble (set) mathématique en admettant plusieurs occurrences du même élément
  - Par exemple, m = {3,5,7,2,3,9,9} est un multiset d'entiers (mais pas un ensemble d'entiers)
  - Quelles opérations pourrait-on avoir envie d'effectuer sur un multiset ?

Abstraction de données

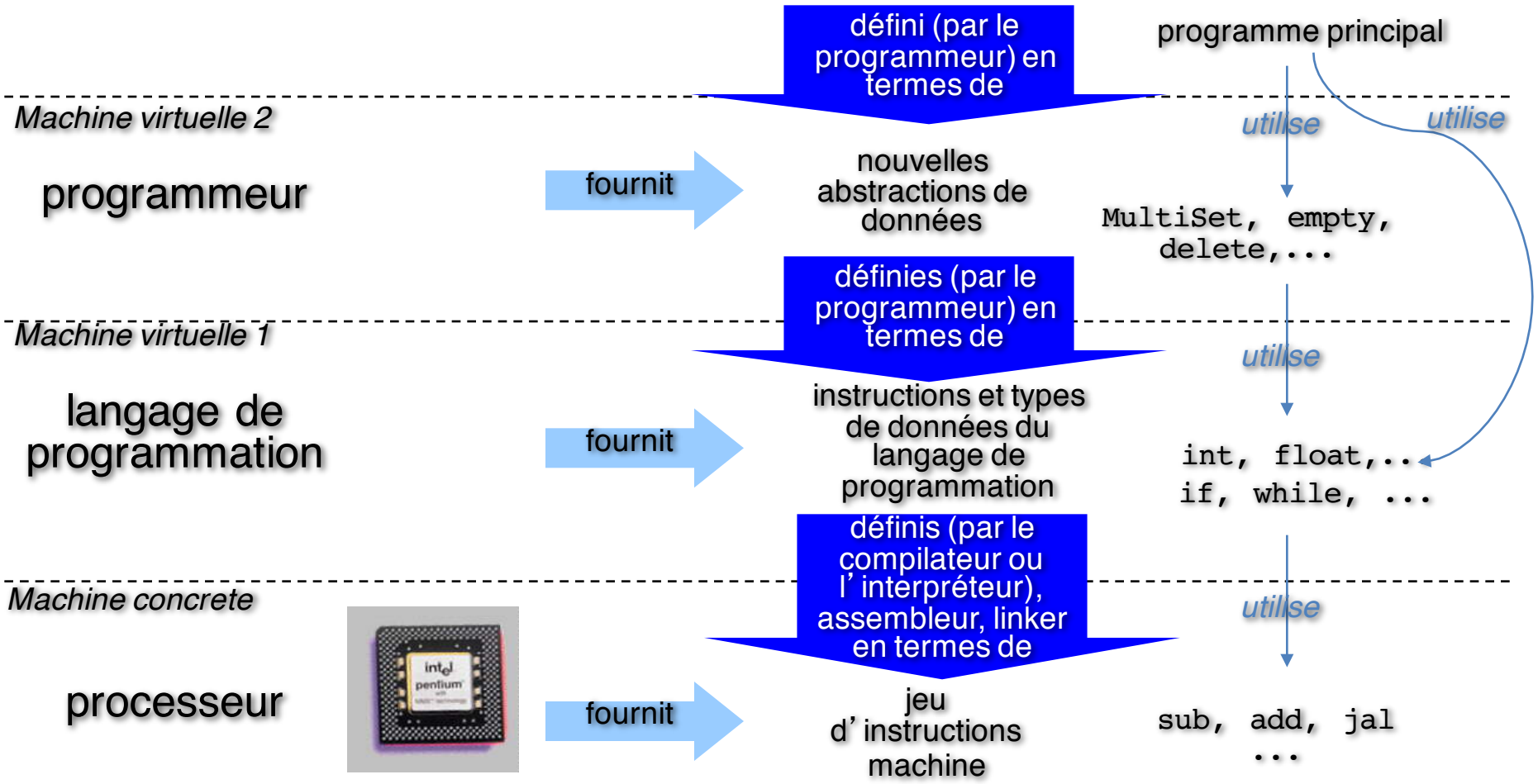
- Opérations :
  - créer un multiset vide
    - `empty()` retourne `{}`
  - ajouter un élément
    - `insert(m,5)` retourne `{3,5,7,2,3,9,9,5}`
  - éliminer une occurrence d'un élément
    - `delete(m,9)` retourne `{3,5,7,2,3,9}`
  - éliminer toutes les occurrences d'un élément
    - `deleteAll(m,9)` retourne `{3,5,7,2,3}`
  - retourner le nombre d'occurrences d'un élément
    - `numberOf(m,9)` retourne `2`
  - retourner le nombre total d'éléments
    - `size(m)` retourne `7`
  - retourner l'union de deux multisets
    - `union(m,n)` avec `n={2,3}` retourne `{3,5,7,2,3,9,9,2,3}`

  etc ...

Abstraction de données

- Dans l'abstraction de données, ces opérations sont « rassemblées » autour d'un nouveau type de données `MultiSet` :
  - `MultiSet empty()`
    `// retourne un MultiSet vide`
  - `MultiSet insert(MultiSet m, int e)`
    `// ajoute une occurrence de e à m`
  - `MultiSet delete(MultiSet m, int e)`
    `// élimine une occurrence de e dans m si e existe;`
    `// ne fait rien sinon`
  - `MultiSet deleteAll(MultiSet m, int e)`
    `// élimine toutes les occurrences de e dans m`
  - `int numberOf(MultiSet m, int e)`
    `// retourne le nombre d'occurrences de e dans m`
  - `int size(MultiSet m)`
    `// retourne le nombre d'éléments dans m`
  - `MultiSet union(MultiSet m, MultiSet n)`
    `// retourne l'union de m et n`

Abstraction de données



Abstraction de données

- Exercices :
  - Quelles seraient les opérations pour les types de données suivants ?
    - `Pile // (aka Stack)`
    - `String`
    - `CompteBancaire`
    - `...`
  - Qualifiez chaque opération selon le CRU(D)

Abstraction de données

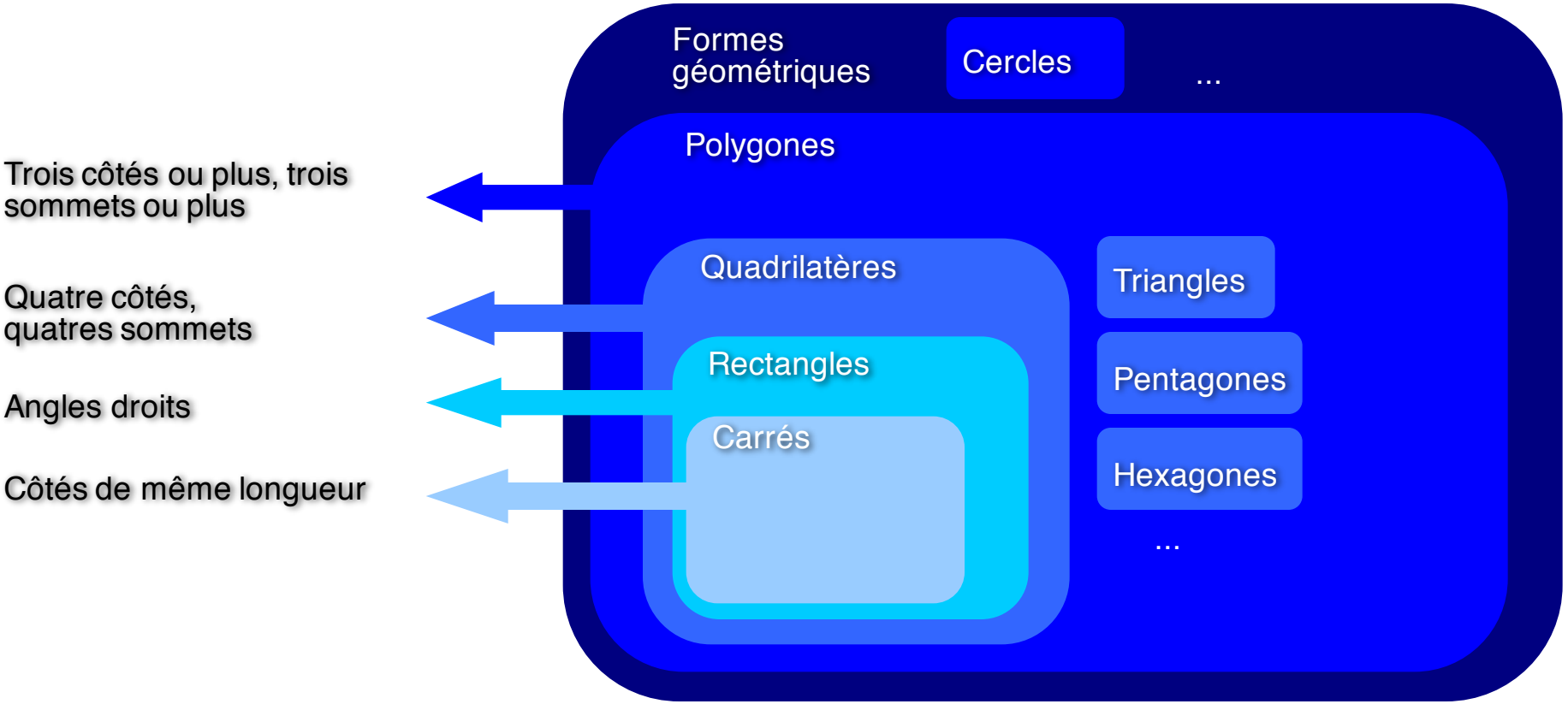
- Pour les autres programmeurs, le nouveau type de donnée peut être utilisé au travers des opérations :
  `empty`, `insert`, `delete` …
- Ces opérations doivent pouvoir être utilisées sans avoir à connaître leur implémentation
- Elles peuvent être définies **les unes par rapport aux autres** :
  - Par exemple, en spécifications algébriques, cela pourrait donner :
    - *`size(insert(s,e)) = size(s)+1`*
    - *`size(empty()) = 0`*
    - *`numberOf(insert(s,e),e) = numberOf(s,e) + 1`*
    - *`numberOf(empty(),e) = 0`*
    - *…*
- Cela garantit qu'un ADT est **plus qu'un simple assemblage de procédures (abstractions procédurales)**…

Abstraction de données

- Abstraction de données
  - la plus importante !
  - outil de base de la programmation orientée-objet
- Complétée par l'abstraction par hiérarchie de types

Abstraction par hiérarchie de types

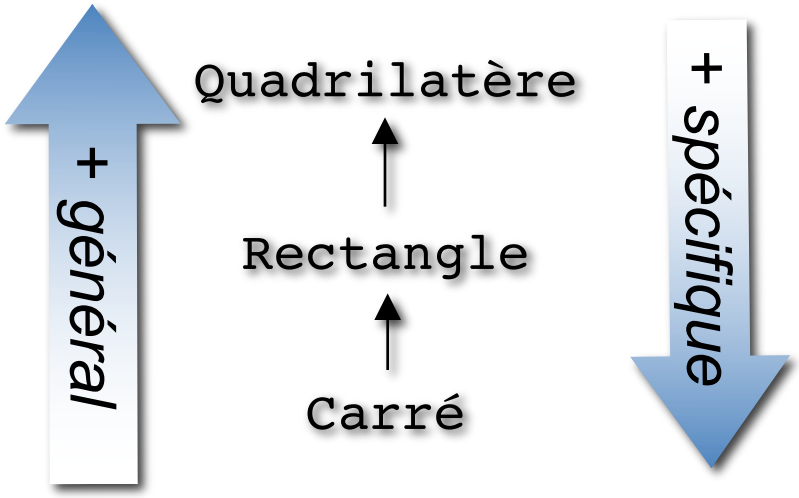
- Question: un polygone possédant les sommets (0,1), (1,1), (1,0) et (0,0) est-il un quadrilatère, un rectangle ou un carré ?



Abstraction par hiérarchie de types

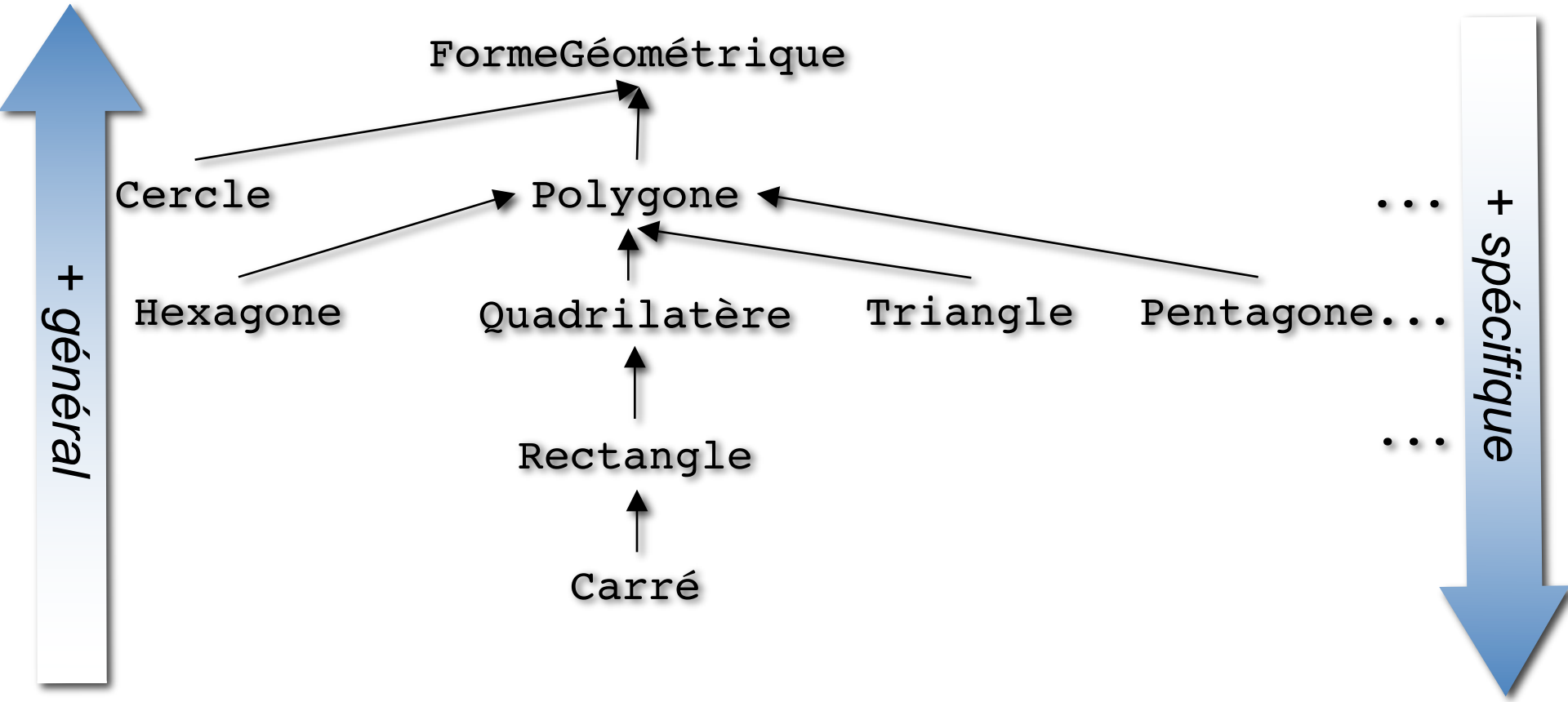
- Réponse: **les 3 à la fois !**
- Pourquoi ?
  - C'est un carré
  - Tous les carrés sont des rectangles
  - Tous les rectangles sont des quadrilatères

- Dans certains langages, il est possible de définir des hiérarchies de types
  - `Quadrilatère` un **supertype** de `Rectangle` et `Carré` qui en sont des **sous-types**
  - `Rectangle` est un **supertype** de `Carré`
  - `Rectangle` et `Carré` sont des **sous-types** de `Quadrilatère`



Abstraction par hiérarchie de types

- Hiérarchie (plus) complète

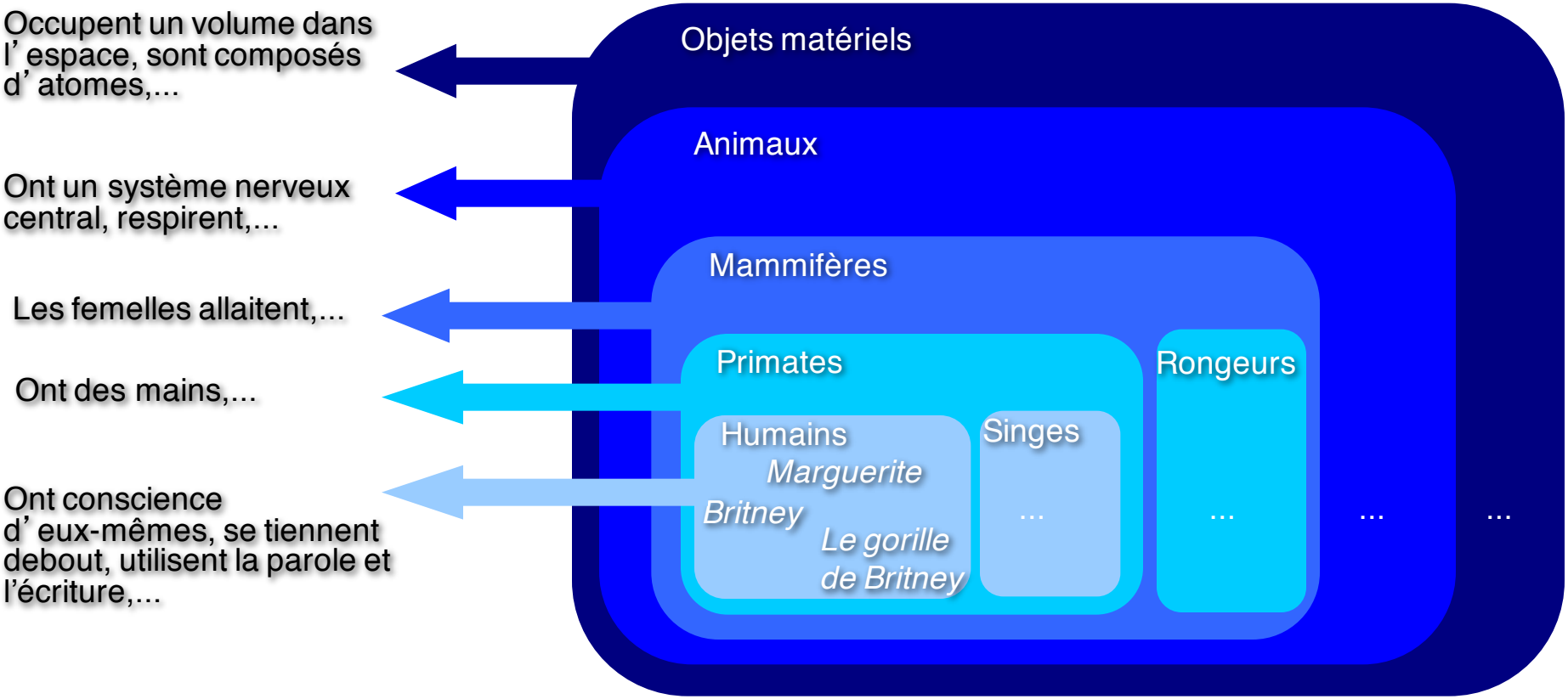


Abstraction par hiérarchie de types

- Toute propriété valable pour un type est également valable pour tous ses sous-types
  - exemple : les quadrilatères ont 4 sommets, donc les rectangles aussi et, transitivement, les carrés également
- En particulier, on verra que toute opération définie pour un type est également applicable à tous ses sous-types
  - exemple :
    - Le type `FormeGéométrique` a pour opération `FormeGéométrique translater(FormeGéométrique f, float x, float y)`
    - L'opération `float donneRayon(Cercle c)` est définie au niveau du type `Cercle`. Elle est plus spécifique que `translater`.

Abstraction par hiérarchie de types

- Permet de s'abstraire d'un type de donnée particulier pour **raisonner sur une famille de types**



Abstraction de l'itération

- Permet **l'itération sur un ensemble d'éléments en ignorant *comment* ces éléments sont obtenus**

- Exemple (pseudo-code):
  - L'itération sur les éléments d'un `MultiSet` ne nécessite pas de connaître l'ordre dans lequel les éléments sont parcourus

```
Element e
MultiSet m ← ...
Iterator i ← genIt(m) // génère i à partir de m
while hasNext(i)
   e ← getNext(i)
   ... // do stuff on e
end-while
```

Résumé du chapitre

- Le « secret » du développement efficace de logiciels de qualité est une bonne **décomposition**
- La décomposition doit être **guidée par l'abstraction**
- Deux formes d'abstraction de base : par **paramétrisation** et par **spécification**
- Formes plus élaborées : **procédurale**, **itération**, **données** (la plus importante !), **hiérarchie de types**
- Il est **plus important** de savoir bien **décomposer** un problème/ programme que de savoir « coder »
- Au prochain cours : au fait, c'est quoi l'«orienté-objet»?
  Et tiens, Java, ça ressemble à quoi ?