Introduction à l'orienté objet

Philippe d'Anfray GIP RENATER

2006-2007

Philippe.d-Anfray@renater.fr

Formation d'Ingénieurs de l'Institut Galilée MACS 2



Plan de l'exposé

- **Génie logiciel**:
  - Abstraction de données; Cycle de vie; Spécifier; Modéliser;
- **Programmation objet**:
  - Structurer, Héritage;
- **Langages à objets**:
  - Instanciation; Privé/Public; Généricité;
- **Conclusions (optimistes)**.



1. Génie Logiciel

Le premier problème du programmeur est bien (ou devrait être. . .):

"[. . .] quel est le langage le mieux adapté à mon application ? [. . .]"

- spécialisé ? généraliste ?
- compilé ? interprété ?
- . . .

dans le domaine de la simulation numérique, les langages les plus utilisés C, FORTRAN sont basés sur des concepts anciens (1950-1960) et conditionnent encore la pratique quotidienne.
Les langages à objets émergent (lentement!). . . , le contexte de l'informatique scientifique évolue (très vite!).



Introduction (1)

On observe un écalage entre les **outils** utilisés et les **techniques** à mettre en œuvre:

- progrès du génie logiciel (encapsulation des données. . .)
  - langages à objets dans le calcul scientifique (C++, Java, . . .)
- nouveaux modèles de programmation (liés aux architectures des calculateurs)
  - *vectoriel*, *parallélisme*, . . .
- prise en compte des réseaux (distribution, hétérogénéité)
  - architecture *client-serveur*, *grilles de calcul et de données*. . .



Introduction (2)

Une conséquence **fâcheuse**:

"[. . .] À défaut de changer de langage, on multiplie les outils [. . .]"

(et les problèmes) FORTRAN, C, voire C++ sont rarement utilisés seuls:

- précompilateurs, générateurs de code;
- outils de vérification;
- outils de validation;
- (. . . bibliothèques. . .).

Un nouveau soucis l'**interopérabilité**.

Abstraction de donnée (1)

Le fil conducteur de cette présentation est l'aspect:

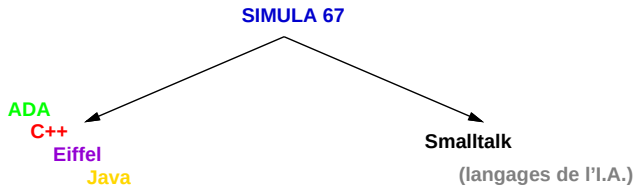
Abstraction de données

- l'*objet* n'est pas une "fin en soi", mais un outil de modélisation
  Attention: modèle de programmation *versus* modèle d'exécution ;
- cet outil permet le découpage de l'application en "modules homogènes" (et si possible indépendants);
- aide à maîtriser les différentes phases de conception, de développement et de suivi du logiciel (*cycle de vie*);
- séparer spécification "ce que l'on veut faire" et réalisation "comment on le fait".



Abstraction de donnée (2)

Ces notions, introduites dans `Simula67` (probablement le premier "vrai" langage à objets) se retrouvent dans de nombreux langages de programmation:



Les deux branches se distinguent par le modèle d'exécution.
L'orienté objet est devenu un environnement de travail courant.
"[. . .] même Fortran 90 se veut vaguement orienté objet [. . .]"

Abstraction de donnée (3)

"[. . .] Concevoir un logiciel c'est d'abord choisir une méthode pour le diviser en entités homogènes [. . .]"

Ce choix conditionne:

- les différentes étapes de conception / développement / maintenance / etc. . . du logiciel (*cycle de vie*);
- mais aussi la façon de travailler (e.g. ordonnancement et répartition des tâches de développement, . . .).

Il interfère aussi avec un autre choix déterminant: le langage de programmation qui sera utilisé et qui implique souvent aussi une façon de travailler.



Cycle de vie

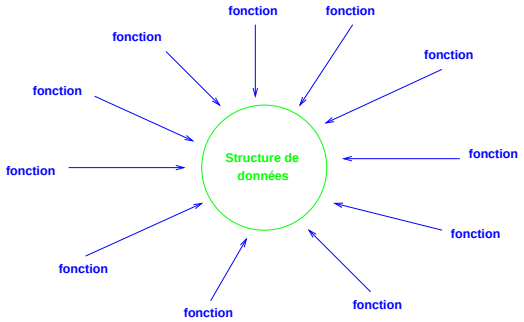
On distingue (au moins!) les étapes suivantes:

- spécification (à partir d'un cahier des charges);
- maquette (teste la cohérence);
- prototype (teste la "réalisabilité");
- réalisation (codage);
- validation (tests, . . .), certification, documentation, dossier qualité;
  - optimisations (souvent nécessaires), évolutions (rarement prévisibles);
  - maintenance (corrections d'erreurs, . . .), portage (suivi des OS, des architectures).

sur lesquelles il est possible (ou plutôt nécessaire) de "boucler".

Approche "Procédurale" (1)

Langage Procédural (FORTRAN, C) ==> découpage en tâches élémentaires et séparation *a priori* entre les données et les fonctions



Avec cette technique, la conception d'un code commence par la définition de structures de données (*a jamais ?*) figées!



Approche "Procédurale" (2)

Conséquence(s):

- difficulté à maîtriser des logiciels de grande taille
  (la cohérence entre modules n'est pas assurée au niveau programmation);
- multiplication des outils.

Pourquoi on ne change pas:

- poids de l'existant (un code=plusieurs années de travail);
- la normalisation, l'efficacité (points faibles des nouveaux langages);
- méthodes et organisation du travail.

"[. . .] quel gain à court ou long terme ? [. . .]"

Orienté Objets

L' approche **orientée objet** (abstraction de données):

"[. . .] s'intéresser en priorité aux entités manipulées par le programme plutôt qu'aux fonctionnalités [. . .]"

- FORTRAN, C, . . . possèdent un nombre limité de types de données (*entier, réel, . . .*);
- les langages à objets possèdent des mécanismes pour construire et manipuler des nouveaux types de données.

exemples d'objets: écran, fénêtres, . . . , points, arêtes, figures géométriques, . . . , vecteurs, matrices, . . .



Spécifier (1)

Il s'agit de mettre en évidence ces entités ou *objets* puis de les définir: les **spécifier**.

Spécifier un nouveau type de données, c'est finalement répondre à la question:

"Que veut-on faire avec ?"

- à quelles informations je veux accéder ?
- quelles opérations ou transformations spécifiques je vais effectuer ?
- quelles type utilisations ?: "brique de base", ... (quelle visibilité ?)



Spécifier (2)

Exemples (triviaux):

Un Vecteur	Une Pile
-sa dimension ? -accéder aux éléments -opérateurs: $+$, $*$, etc. . .	-nombres de données empilées ? -tester si la pile est vide -empiler ou depiler une donnée

"[. . .] si `t` est de type `T`, il possède certaines propriétés indépendantes de tout système de représentation et de toute convention liée au codage. [. . .]"



Spécifier (3)

Approche orientée objet:

- **le type** de l'objet est quelque chose d'**abstrait** car ce qui importe c'est ce que l'on fait avec l'objet et non pas sa structure ou sa réalisation;
- **les instances** (exemplaires) de l'objet, sont des choses bien **concrètes**, l'analogue des variables utilisées habituellement en programmation.

L'objet a un type, on manipule ses instances.



Encapsulation

Contrairement à l'approche procédurale, les nouveaux types de données vont intégrer à la fois des éléments statiques et dynamiques du problème à traiter:

des fonctions et **des données**

Le plus souvent, les fonctions sont appelées **méthodes**. Il n'y a plus de structure de données globale. La démarche est bien très différente de l'approche procédurale.

*"[. . .] les données sont **encapsulées** dans les objets [. . .]".*

exemple, chaque instance d'un type vecteur, "contient" parmi ses données encapsulées, ses éléments.



Modéliser

L' approche orientée objet (abstraction de données) est un puissant outil de modélisation:

- construire un modèle pour un système (physique, économique, . . .), c'est décrire:
  - des entités;
  - les relations entre ces entités.
- pour un domaine d'application donné:
  - les entités sont *figées*;
  - les relations sont *variables*.



Simuler

L'approche orientée objet permet de construite une base d'outils pour la modélisation.

- définir les **entités**, c'est définir les types de données manipulés par le programme, *[. . .] c'est disposer de toute la "partie permanente" du logiciel [. . .]* ;
- effectuer une simulation c'est coder les relations entre entités qui définissent le problème à traiter.

Difficile?? dépend de la structure de l'objet, de la théorie sous jacente.



Conclusions

L'approche orientée objets consiste à exhiber et spécifier les entités (types de données) manipulées dans le cadre d'un processus de modélisation. Cette approche permet de développer des logiciels de qualité.

Elle est "duale" de l' approche procédurale qui consiste à s'intéresser en priorité aux fonctionnalités utiles que le modèle doit fournir.

Plus générale, l'approche orientée objets s'adresse à un domaine d'activité plutôt qu'à une simulation particulière.

Enfin on appellera **objet** (dans le cadre [limité] de cet exposé), tout exemplaire d'un type de donnée **spécifié** et construit par le programmeur dans un langage [de programmation] à objets.



2. Programmation Objet

Il s'agit de développer un logiciel en s'appuyant sur les types de données. Exemples de spécifications fonctionnelles:

TYPE: Vecteur –– famille d'éléments	**TYPE:** Pile ––famille d'objets empilables
FONCTIONS:	**FONCTIONS**
dimension: Vecteur → Entier	**pile_vide:** Pile → Booléen
élement: Vecteur x Entier → Elément	**profondeur:** Pile → Entier
addition: Vecteur x Vecteur → Vecteur	**empiler:** Pile x Objet_empilable → Pile
...	**dépiler:** Pile → Objet_empilable

Les spécifications des types PILE et VECTEUR font apparaître d'autres types de données existants ou non.



Structurer (1)

Conception ascendante/descendante

la spécification d'un nouveau type de donnée:

- peut s'appuyer sur des types existants (ascendante);
- peut conduire à spécifier des types "plus simples" qui n'existent pas encore (descendante).

On fait apparaître des "niveaux d'abstraction".

Les fonctions portant sur des types "plus simples" permettent de décrire en terme algorithmiques celles concernant les types "plus complexes".



Structurer (2)

Une spécification	Un Algorithme (réalisation)

...
Addition
`Vecteur x Vecteur`$\longrightarrow$`Vecteur`
notation `U=V+W`
variable(s) de sortie: `U`
variable(s) d'entrée: `V, W`
propriétés:...
	...
	`pour i=1 a dim faire`
	`{`
	`...`
	`u[i]=V[i]+W[I]`
	`}`
	...

Clairement, deux problèmes bien distincts !
L'algorithme utilise les spécifications des entiers et des éléments de vecteurs.



Structurer (3)

Séparation spécification/réalisation

En pratique, la vue "utilisateur" d'un type (e.g. vecteur, Pile) ne doit pas dépendre (et doit même cacher!) la représentation interne des données. Exemple:

- si dans la spécification, la syntaxe `V[i]` est utilisée pour l'opérateur "accès au i ^{eme} élement de V" cela ne signifie pas que "V" est implanté en mémoire comme un tableau;
- à l'inverse, si l'on se sert d'un tableau pour implanter en mémoire les éléments de `V`, cela ne doit pas imposer pour l'accès aux éléments, la syntaxe `V[i]`.



Structurer (4)

Pour la définition et l'utilisation d'un type, on peut distinguer **3** tâches logicielles:

Les spécifications:
C'est à dire la description abstraite, fonctionnelle du type.

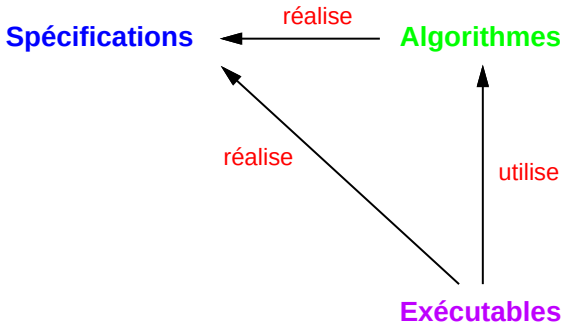
Les algorithmes:
Il s'agit ici de la réalisation "abstraite" des opérateurs par des algorithmes (utilisant les spécifications des types de plus bas niveau.)

(comment fabriquer) L'"exécutable":
Le code exécutable -module objet, bibliothèque, application-, c'est la réalisation concrète, sur une machine donnée (cf. `makefile`).



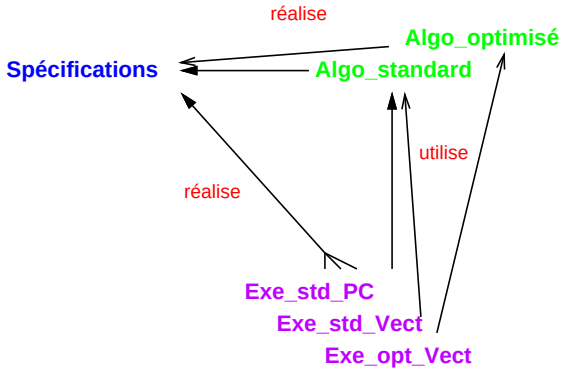
Structurer (5)

On schématise cela en faisant apparaître des “composants” logiciels reliés par des “relations”:



Structurer (6)

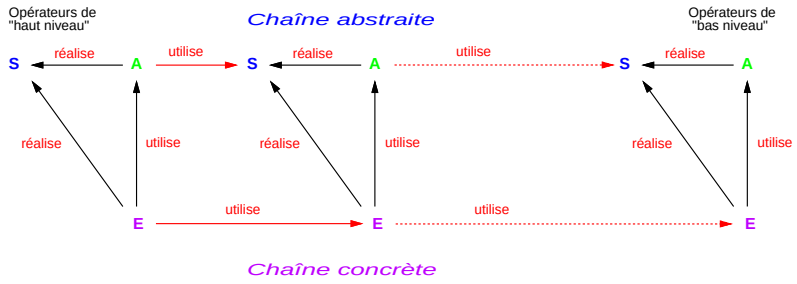
Tout l’édifice repose sur les spécifications.



Ici, tous les “exécutables” réalisent la même spécification.

Structurer (7)

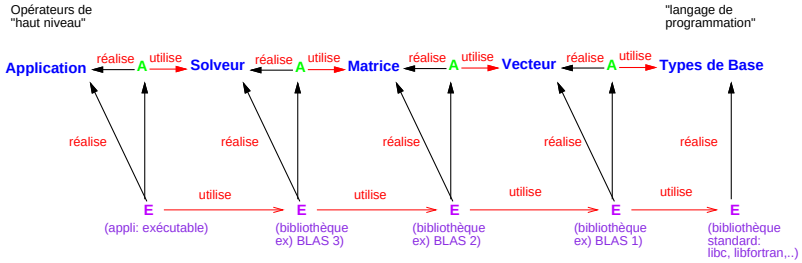
Les niveaux d’abstractions successifs se traduisent par de nouvelles relations:



Ce schéma représente la structure du logiciel et sert de support au “cycle de vie”.

Structurer (8)

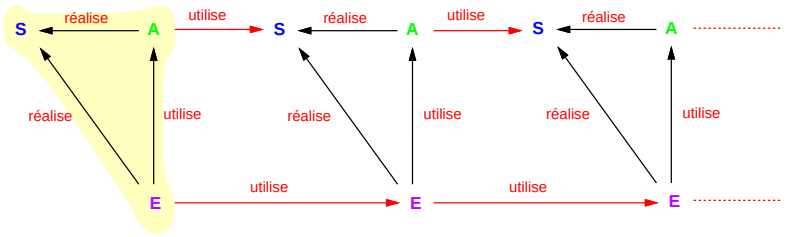
Un exemple plus concret:



La “chaîne abstraite” doit être indépendante de l’implémentation.
La “chaîne concrète” correspond aux codes exécutables sur une machine donnée.

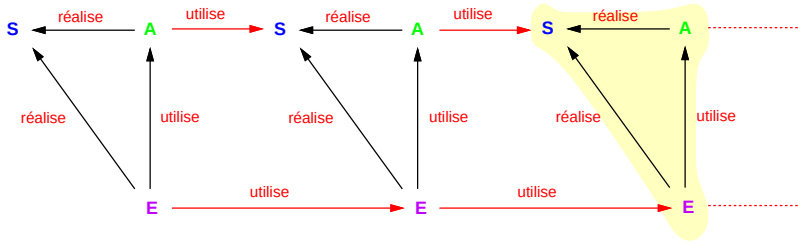
Structurer (9)

Cycle de vie: **la maquette**. Tester la cohérence.



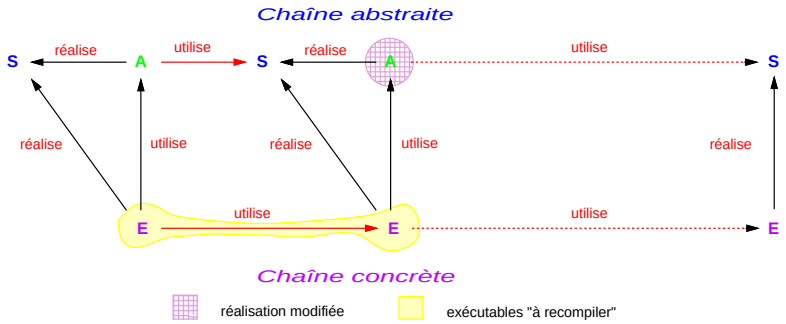
Structurer (10)

Cycle de vie: **le prototype**. Tester des choix de réalisation.



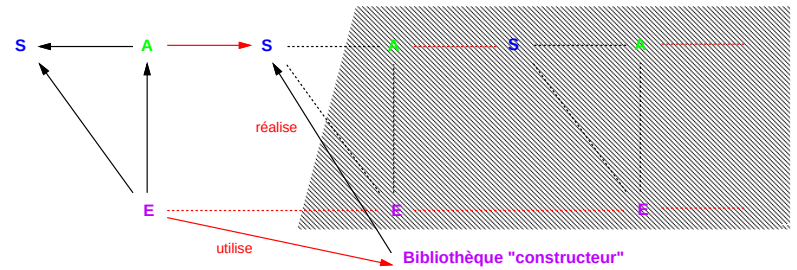
Structurer (11)

Cycle de vie: **la maintenance**. Evaluer l'impact d'une modification.



Structurer (12)

Cycle de vie: **l'optimisation**. Obtenir des performances (par exemple appel "direct" des BLAS "constructeur", . . .)



Structurer (13)

L'approche abstraction de données conduit à une structuration très forte du logiciel utile tout au long du cycle de vie.

Si "le même outil" (méthodologie, langage, etc. . .) est utilisé tout au long du cycle de vie, cela facilite les transmissions et rétroactions entre les différentes phases.

Tout cela a un impact très fort sur l'organisation du travail au sein des équipes en charge du développement logiciel.



Héritage (1)

"[. . .] Les entitès modélisées correspondent rarement à des concepts isolés. [. . .]"

Spécifier c'est aussi se demander:

"**en quoi le nouveau type est-il apparenté aux types existants?**".

Il peut en être "fonctionnellement" très proche.

On parlera de conception descendante par affinages successifs appliqués aux données.



Héritage (2)

La notion d'**héritage** est indispensable pour la réalisation de logiciels de grande taille: c'est de la **réutilisation**.

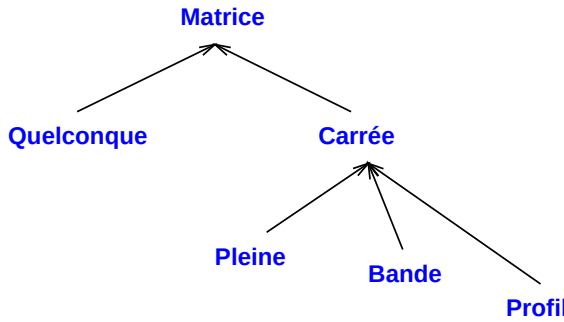
- première idée: ajouter de nouvelles fonctionnalités pour spécialiser les types existants;
- plus généralement: définir des **archétypes** (qui ne recevront aucune réalisation concrête) mais serviront de "modèles".

On aura une structure hiérarchique de types.



Héritage (3)

Cette structure se représente par un graphe; si l'héritage est **simple**, ce graphe est un arbre:

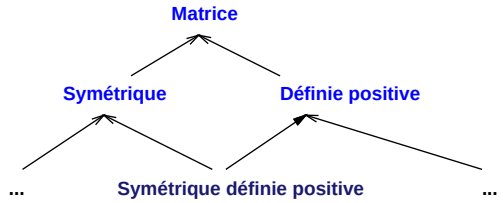


Cette hiérarchie simple **semble** souvent insuffisante (Simula67, Java).



Héritage (4)

Si l'héritage est **multiple**, le graphe est quelconque mais **acyclique**:

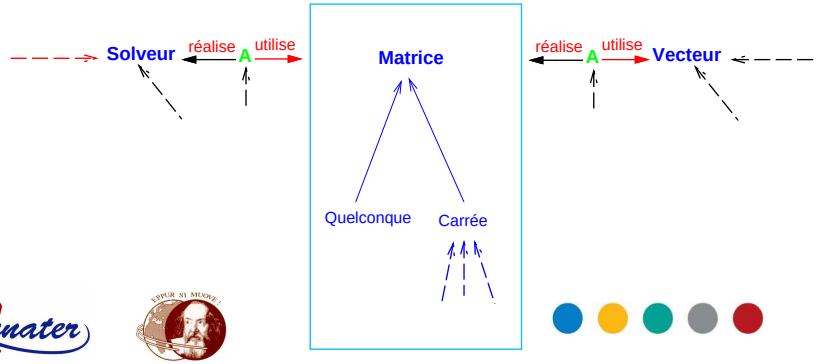


mais cela pose de *multiples* problèmes . . . (C++)

Java permet l'héritage multiple de spécifications.

Héritage (5)

La structure hiérarchique induite par l'héritage est en quelque sorte "orthogonale" aux niveaux d'abstraction étudiés précédemment:



Héritage (6)

L'héritage donne d'autres outils pour gérer la "complexité du logiciel":

- réutiliser les spécifications;
- organiser les types "apparentés";
- exprimer leurs relations mutuelles.

Difficultés: choix des archétypes; privilégier les "bonnes" hiérarchies.

Conclusions

Le génie logiciel c'est l'ensemble de toutes ces techniques:

- bien comprises (hum. . . peu fréquent),
- bien appliquées (hélas rare).

Les buts, en assurant une meilleure maîtrise de la complexité des programmes:

- **augmenter la fiabilité**, la réutilisabilité, les performances;
- **diminuer les coûts** (notamment la maintenance).



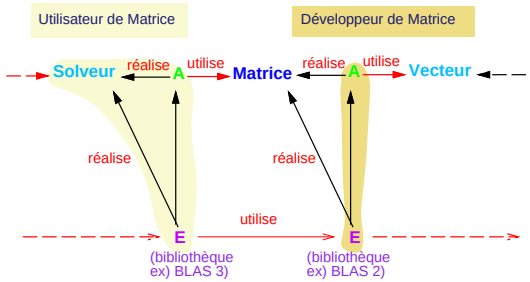
3. Mécanismes des langages à objets

Mécanismes mis en œuvre dans les langages de programmation qui permettent une programmation orientée objets.

Si l'on s'intéresse à un type de données, On distinguera deux points de vues:

- le programmeur *développeur* (construit les types "de base")
- le programmeur *utilisateur* (bâti l'application au dessus des types)

utilisateur *vs* développeur



Tous les "acteurs" partagent la spécification.

Les Types (1)

. . . cf Composants logiciels:

- le *concepteur* crée un nouveau type de donnée, il décrit un modèle (spécifications) le *développeur* le réalise:

```
Type Vecteur          Type Pile
     {                     {
     // données            // données
     ...                   ...
     //méthodes            //méthodes
     ..                    ..
     }                     }
```


Les Types (2)

- l' *utilisateur* se sert du mécanisme d' *instanciation*:

```
..main (...)              ..main (...)
  {                         {
    Vecteur  V1, V2;          Pile  P;
    ...                       ...
    ...                       ...
  }                         }
```

Il utilise les variables (`Vecteur`, `Pile`, `...`) pour résoudre un problème ou créer des objets "plus complexes".

Les Types (3)

Du point de vue “informatique” le type est une notion dynamique. Un type définit des attributs (*cf.* données) et des méthodes (*cf.* fonctions):

- les données **encapsulées** sont propres à chaque instance;
- les méthodes (fonctions membre) n’existent qu’en un seul exemplaire.

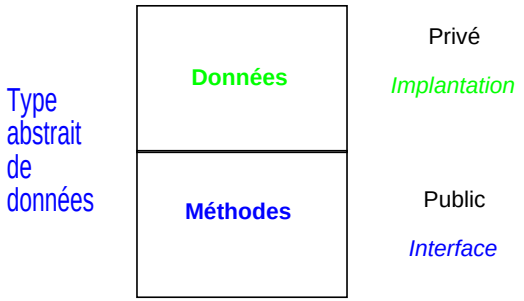
L’instanciation est un mécanisme complexe pris en charge par le “moniteur d’exécution”:

- gestion de la mémoire: allocation/désallocation, ramasse-miettes;
- problèmes de pointeurs (problème de la “référence folle”);
- stratégie de liaison instance/méthode (pour l’exécutable), . . .

Tout cela peut se révéler coûteux . . .

Les Types (4)

L’opposition **développeur-utilisateur** se retrouve dans la séparation **spécification/réalisation**.
La notion correspondante dans les langages est le **Privé-public**:



L’utilisateur ne peut pas tirer parti des détails de l’implémentation.

Les Types (5)

Les méthodes, partie publique, c’est ce que voit l’utilisateur (on dit aussi l’“*interface*”). Ces méthodes permettent:

- d’ accéder aux données encapsulées;
- de représenter les propriétés du type;
- de gérer la vie d’une instance: initialisation, etc. . .
- de décrire les opérateurs qui le manipulent;

Les Types (6)

“[. . .] peut on vraiment séparer spécification et réalisation ?? [. . .]”

Les possibilités varient selon les langages.
Il est rarement possible de décrire fonctionnellement (et donc indépendamment de la réalisation!):

- l’accès aux données - **sélection**-
- le parcours des données -**itération**-

Notions particulièrement importantes en simulation numérique.
Exemple typique qu’implique l’utilisation de la **notation** `A[i,j]`.

Généricité (1)

La **généricité**, elle, sera à mettre en parallèle avec [entre autres] la notion d'héritage. De quoi s'agit-il, d'un concept mathématique:

"[. . .] appliquer des résultats identiques à des objets concrètement différents mais caractérisés par des structures ou des propriétés identiques[. . .] "

La généricité peut être:

- incrémentale ou
- structurelle.



Généricité (2)

On parle de **généricité incrémentale** lorsque l'on a plusieurs implémentation d'un même concept.
Un premier exemple simple, la surcharge d'opérateurs:

```
...
Entier   i, j, k;
Vecteur  u, v, w;
...
k=i + j;
...
w=u + v;
...
écrire (k) ;
écrire (w) ;
...
```

Attention la syntaxe doit conserver sa sémantique "usuelle".



Généricité (3)

On utilisera (si c'est possible) la **généricité incrémentale** pour définir les accès aux données: "sélection" et "itération".

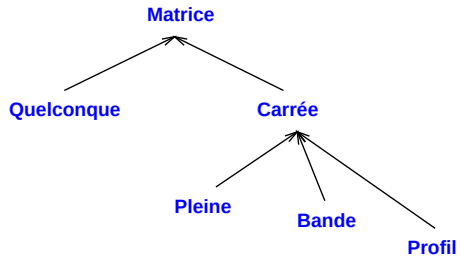
```
...
Matrice  M;
Entier   i, j, k, l;
Element  x;
...
x=  M [ i , j ] ;          // appel d'une méthode
M [ i , j ] = foo(...);    // Attention lvalue !!
...
pour  k in M.colonne[l]  faire  // itérateur
  ...
  ...
...
```



Généricité (4)

La généricité incrémentale c'est bien sûr l'héritage pour les langages à objets. On parle aussi de **dérivation**.

À partir d'un type de donnée, on construit un nouveau type ("dérivé") qui hérite des propriétés du type "parent" (ou "de base"):

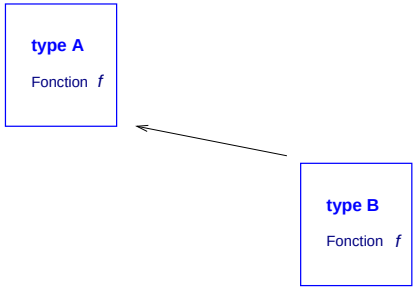


- spécialisation, . . .
- définition d'archetypes, . . .



Généricité (5)

Point clef (ou difficulté) **le polymorphisme**:



Le type `B` dérive du type `A`. Le comportement de `f`, définie dans `A`, peut être redéfini dans `B`. Mais toute *instance* de `B` peut être aussi considérée comme une *instance* de `A`.

“Que se passe-t-il si l’on invoque `f` ?”.

Généricité (6)

Les possibilités diffèrent selon les mécanismes présents dans les langages:

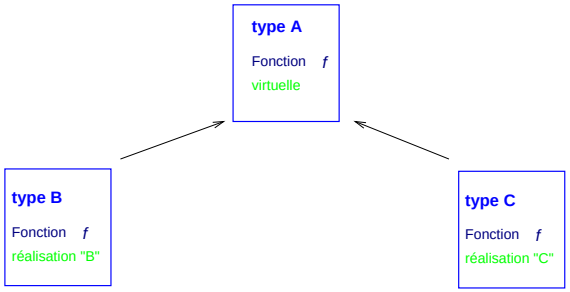
Le choix peut être effectué:

- à la compilation,
- à l’exécution.

Toujours se poser la question!!

Généricité (7)

Liaison Dynamique: le choix est effectué l’éxecution. On parle aussi de “fonctions virtuelles”.



Permet d’écrire des programmes travaillant sur `A` (“archetype”) qui utiliseront **à l’exécution** le code correspondant aux réalisations de la fonction pour les types dérivés `B,C,...`

Généricité (8)

La **liaison dynamique** permet d’écrire des programmes réutiisables (génériques) !

- le code est écrit en utilisant un archétype (e.g. `Matrice`);
- l’exécutable utilise “dynamiquement” les réalisations correspondant à un type dérivé “utile” (e.g. `Matrice_carrée_pleine`).

Tout ce travail est effectué par le moniteur d’exécution (et peut se révéler coûteux. . .)

Généricité (9)

Généricité structurelle:

Les propriétés d'un type de données ne dépendent pas du type des données qui sont encapsulées. Cas typique, les "conteneurs". Exemple: faut-il définir autant de Types `PILE` qu'il existe de types `OBJET_EMPILABLE`. . . candidats à être empilés.

On parle alors de "types paramètres" ou de "patrons":

"[. . .] le type de tout ou partie du contenu d'un type est un paramètre de type [. . .]".

Généricité (10)

Exemple de démarche, définition d'un type `PILE(OBJET_EMPILABLE)`:

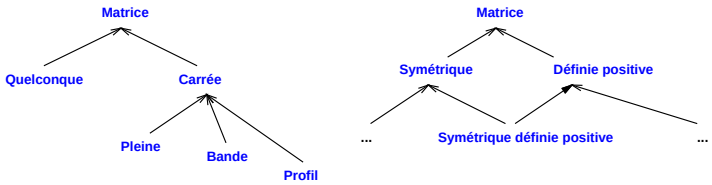
- spécifier un premier type, `OBJET_EMPILABLE`;
- spécifie ensuite un deuxième type, `PILE (OBJET_EMPILABLE)` où le type `OBJET_EMPILABLE` est un paramètre;
- écrire les algorithmes réalisant les méthodes de `PILE (OBJET_EMPILABLE)` en s'appuyant sur la spécification de `OBJET_EMPILABLE`;
- **c'est fait!**, tout type disposant des opérateur définis dans `OBJET_EMPILABLE` est candidat à être empilé.

Une seule spécification et un seul code pour toutes les piles.



Généricité (11)

Difficulté d'exhiber et de spécifier les types "utiles". *(de l'intérêt des méthodes d'analyse)* La réutilisation n'est pas évidente non plus...



Sur quoi se baser: caractéristiques structurelles, numériques etc. . . ?

Conclusions

Trois remarques finales:

- Utiliser un langage à objet c'est disposer du même "formalisme" tout au long du cycle de vie: spécification, maquelle, prototype, programmation, . . .
- C et Fortran proposent un seul type de composant logiciel tous indépendants et au même niveau. Le langage à objet et ses environnements de programmation et d'exécution doivent gérer des *contextes* très lourds;
- l'approche objet implique une nouvelle répartition des tâches dans les équipes de développement.