# Génie logiciel avancé

Mihaela Sighireanu

UFR d'Informatique Paris Diderot, LIAFA, 175 rue Chevaleret, Bureau 6A7
http://www.liafa.jussieu.fr/∼sighirea/cours/genielog/

## Spécification formelle: Types de données abstraits (ADT)

# Résumé

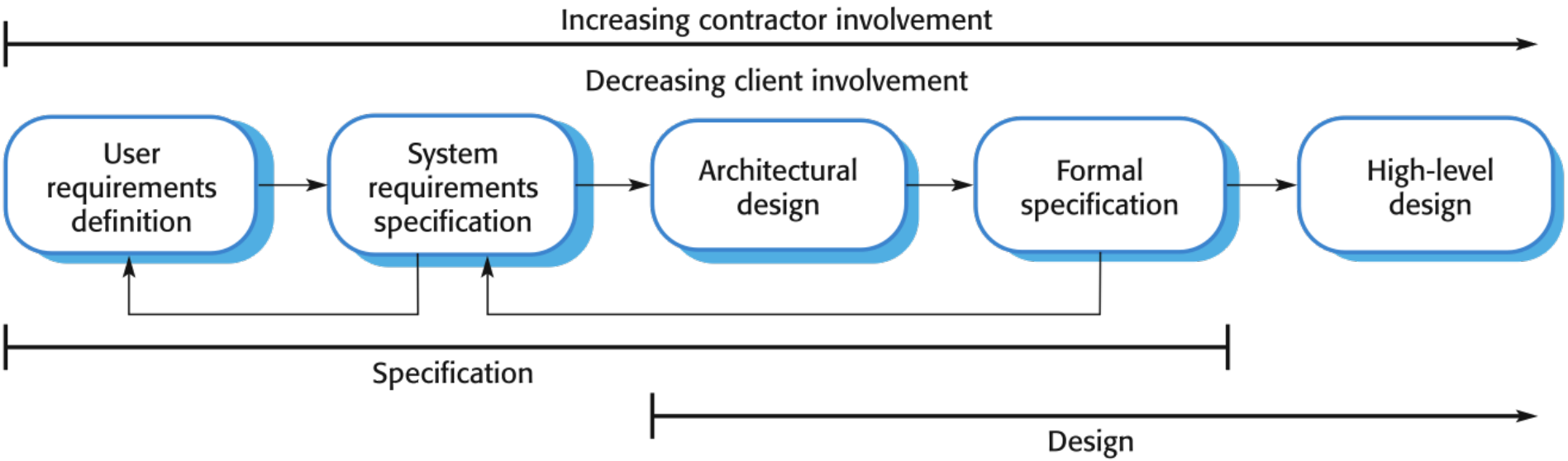
# Objectifs

- Montrer comment les techniques de spécification formelles aident à découvrir des problèmes dans la spécification du système.
- Définir et utiliser les techniques algébriques de spécification (ADT) pour spécifier les interfaces.
- (Cours suivants :) Définir et utiliser des techniques basées sur les modèles pour la spécification des comportements.

# Méthodes formelles

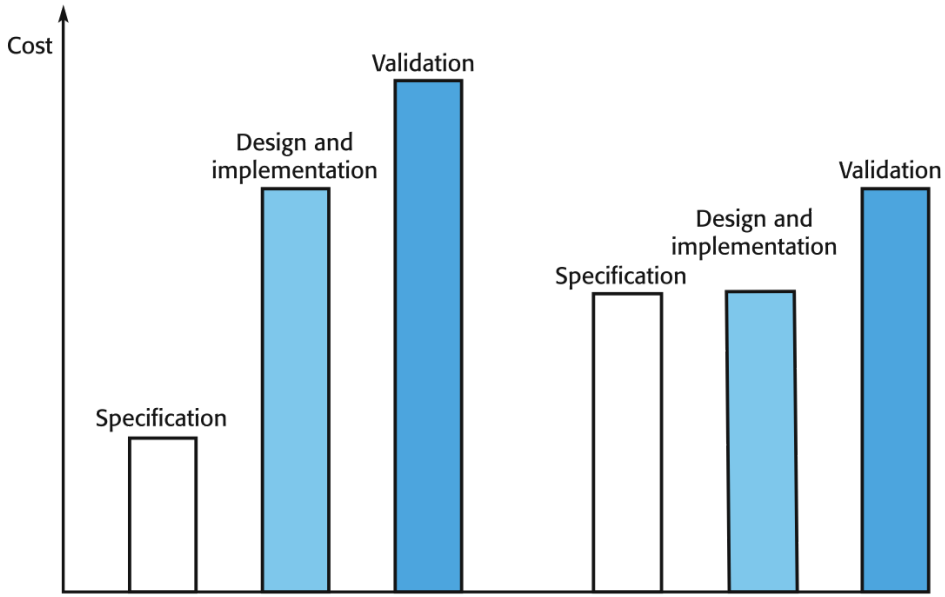
- Les spécifications formelles font partie d'une collection de techniques connues sur le nom de "méthodes formelles".
- Les méthodes formelles ont à la base des représentations mathématiques du logiciel ou du matériel.
- Les méthodes formelles contiennent :
  - Spécifications formelles,
  - Analyse et preuve de spécifications,
  - Développement par raffinement des spécifications,
  - Vérification de programmes.
- Prévues représenter LA technique de développement, elles sont utilisées que dans des domaines "critiques" à cause de leur coût (matériel, en temps et humain).
- Le principal bénéfice de leur utilisation est la réduction du nombre d'erreurs dans le logiciel.

# Processus de spécification et de design



# Coût de la spécification formelle

- La spécification formelle demande plus d'effort dans les phases avant du projet.
- Elle réduit les erreurs (incomplétude ou inconsistance) de la spécification des charges.
- Ainsi, le nombre de changements du projet à cause d'un problème de spécification de charges est réduit.



# Classes de spécifications formelles

- Algébriques : le système est spécifié en termes d'ensembles, d'opérations et de leur relation.
  Ex. séquentiel : Act One, Larch, OBJ.
  Ex. concurrent : Lotos.

- Basés sur les modèles : le système est spécifié en termes de modèle à états et utilisent des opérations qui changent l'état du sytème.
  Ex. séquentiel : Z, VDM, B.
  Ex. concurrent : CSP, réseaux de Pétri, automates hiérarchiques (*statecharts*).

# Motivation

Spécification formelle d'interfaces :

- Les grands systèmes sont décomposés en sous-systèmes qui se composent à travers d'interfaces bien définies.
- La spécification des interfaces des sous-systèmes permet leur développement independant.
- Les interfaces peuvent être définies comme des types de données abstraits ou des interfaces de classes.
- L'approche algébrique pour la spécification formelle des interfaces est souhaitable car elle permet pour les opérations de l'interface :
  - de les définir formellement,
  - d'analyser formellement (preuve) leur comportement,
  - de dériver l'implémentation d'un objet ou d'un type en partant de sa définition formelle.

# Structure d'une spécification d'ADT

4 parties :

**Introduction :** définit la *sorte* (nom du type) et déclare les autres spécifications utilisées.

**Description :** (optionnelle) décrit de manière informelle les opérations de la sorte.

**Opérations :** définit la syntaxe des opérations dans l'interface et leurs paramètres.

**Axiomes :** définit la sémantique des opérations sous forme d'équations.

Spécification (sort + opérations) = signature.

*Exemple : spécification ADT des booléens*

**type** BOOLEAN **is**

**sorts** bool

(* Type boolean avec les opérations usuelles *)

**opns**

```
false, true : -> bool
not         : bool -> bool
_and_, _or_, _xor_, _implies_, _iff_, _eq_, _ne_
            : bool, bool -> bool
```

**eqns**

**forall** `x, y : bool`

**ofsort** `bool`

```
not (true)   =  false;
not (false)  =  true;
x and true   =  x;
x and false  =  false;
x or true    =  true;
x or false   =  x;
x xor y      =  (x and not (y)) or (y and not (x));
x implies y  =  y or not (x);
x iff y      =  (x implies y) and (y implies x);
x eq y       =  x iff y;
x ne y       =  x xor y;
```

**endtype**

# Signature

**Sort** : le(s) type(s) à définir.

**Opérations** : plusieurs classes par rapport au type défini

- constructeur : définit les valeurs du type ; exemples : `true`, `false`.
- inspecteur : renvoie les composantes du type ; exemple : tête de la liste.
- observateur : décrit la relation (les propriétés) du type avec les autres types ; exemple : longueur liste.

Les axiomes doivent définir complètement et correctement les opérations (voir une méthode plus loin).

**type** STRING **is imports** CHARACTER, NATURAL, BOOLEAN

**sorts** string

**opns**

```
new      : -> string
append   : string, string -> string
add      : char, string -> string
size     : string -> nat
isEmpty  : string -> bool
eq       : string, string -> bool
first    : string -> char
```

**eqns**

**forall** x, y : string, c,d : char

**ofsort** bool

```
isEmpty(new)              = true;
isEmpty(add (c, x))       = false;
eq(new, new)              = true;
eq(add(c, x), new)        = false;
eq(new, add(c, x))        = false;
eq(add(c, x), add(d, y)) = eq(c, d) and eq(x, y);
```

**ofsort** nat

```
size(new)      = 0;
size(add(c, x)) = succ(size(x));
```

**ofsort** string

```
append(new, x)       = x;
append(add(c, x), y) = add(c, append(x,y));
```

# Exemple : ADT chaînes de caractères

- Opérations nécessaires :
  - Chaîne vide (`new`)
  - Concaténation de deux chaînes (`append`)
  - Concaténation d'un caractère et d'une chaîne (`add`)
  - Calcul de la longueur (`size`)
  - Test de chaîne vide (`isEmpty`)
  - Egalité de chaînes (`eq`)
  - Sélection du premier caractère (`first`)
- Types nécessaires pour définir l'ADT :
  - `char` : le type caractère
  - `nat` : le type entier naturel
  - `bool` : le type booléen

# Mathématiques : signature

Algèbre hétérogène (Birkhoff).

**Definition (Ensemble $S$-typé)**

Soit $S \subseteq \mathbf{S}$ un ensemble fini de sortes. Un ensemble $S$-typé $A$ est l'union disjointe d'une famille d'ensembles indexée par $S$ : $A = \left(\bigcup_{s \in S} A_s\right)$.

Exemple : l'ensemble des valeurs défini par la spécification STRING est un ensemble $S$-typé avec $S = \{\texttt{bool}, \texttt{char}, \texttt{nat}, \texttt{string}\}$.

**Definition (Signature)**

Une signature est un couple $\Sigma = \langle S, F \rangle$, avec $S \subseteq \mathbf{S}$ un ensemble fini de sortes et $F = (F_{w,s})_{w \in S^*, s \in S}$ est un ensemble $(S^* \times S)$-typé de noms d'opérations en $\mathbf{F}$. Les $f \in F_{\epsilon,s}$ sont appelées des constantes.

# Mathématiques : termes

**Definition (Termes d'une signature)**

Soit $\Sigma = \langle S, F \rangle$ une signature et $X$ un ensemble $S$-typé de variables. L'ensemble de termes de $\Sigma$ utilisant les variables de $X$ est un ensemble $S$-typé $T_{\Sigma,X}$ avec chaque ensemble $(T_{\Sigma,X})_s$ définit inductivement par :

- chaque variable $x \in X_s$ est un terme de sort $s$,
- chaque constante $f \in F_{\epsilon,s}$ est une terme de sort $s$,
- pour toute opération non constante $f \in F_{w,s}$ avec $w = s_1 \cdots s_n$ et pour tout $n$-uple de termes $(t_1, \ldots, t_n)$ tel que $t_i \in (T_{\Sigma,X})_{s_i}$ $(1 \leq i \leq n)$ alors $f(t_1, \ldots, t_n)$ est un terme de sort $s$.

Exemples : `append(new, x)`, `x and (y xor z)`, ...

# Mathématiques : axiomes

**Definition (Axiome simple)**

Soit $\Sigma = \langle S, F \rangle$ une signature et $X$ un ensemble $S$-typé de variables. Les axiomes sur les variables de $X$ sont des égalités de termes $t = t'$ tel que $t, t' \in (T_{\Sigma,X})_s$.

Remarque : les variables de $X$ sont quantifiées universellement.
Exemple : `first(add(c, new)) = c ;`

**Definition (Axiome conditionnelle)**

Soit $\Sigma = \langle S, F \rangle$ une signature et $X$ un ensemble $S$-typé de variables. Les axiomes conditionnelles sur les variables de $X$ sont $t_0 = t'_0 \wedge \ldots t_n = t'_n \Rightarrow t = t'$ tel que $t, t' \in (T_{\Sigma,X})_s$, $t_0, t'_0 \in (T_{\Sigma,X})_{s_0}$, $\ldots$, $t_n, t'_n \in (T_{\Sigma,X})_{s_n}$.

Exemple : ajout d'un élément dans un arbre binaire de recherche

```
lt(d,data(t))=true => add(d,t) = node(add(d,left(t)),data(t),right(t)) ;
```

# Ecriture d'axiomes

Attention : ne pas écrire des axiomes contradictoires ou oublier des cas !

Méthode pour obtenir la complétude et la correction hiérarchique :
Pour chaque opération non-constructeur :

1. écrire une axiome avec la partie gauche un terme qui commence avec le nom de l'opération ;
2. pour chaque paramètre de l'opération (de gauche à droite) appliquer le principe suivant :
   - utiliser une variable pour ce paramètre ;
   - si une équation est difficile à écrire avec une variable, écrire les équations en décomposant cette variable à l'aide de constructeurs.
   - si un constructeur n'est pas suffisant pour écrire l'équation, utiliser les conditions qui décomposent en cas d'utilisation.

# Exemple : spécification ADT NATURAL

Equations pour l'opération `_ + _ : nat, nat -> nat` :
Décomposition du premier paramètre :

```
eqns forall x, y : nat
    ofsort nat
    0 + y = y ;
    succ(x) + y = succ(x + y) ;
```

Que fait-on pour la commutativité : `x + y = y + x ;`

**Exercice :** Ecrire les équations pour `_ > _ : nat, nat -> bool`

# Mathématiques : spécification algébrique

**Definition (Spécification algébrique)**

Une spécification algébrique multi-sortes $Spec = (S, F, X, AX)$ est une signature $\Sigma = (S, F)$ et un ensemble d'axiomes $AX$ sur un ensemble de variables $X$.

On utilisera aussi la notation $Spec = (\Sigma, X, AX)$.

La sémantique d'une spécification algébrique est donnée par un modèle, c'est-à-dire une implémentation possible de la spécification.

# Mathématiques : spécification algébrique

## Definition (Modèle)

Soit $Spec = (\Sigma, X, AX)$ une spécification algébrique. L'ensemble de ses modèles $Mod(Spec)$ est l'ensemble de $\Sigma$-algèbres $M$ tel que $\forall ax \in AX, \forall X, M \models ax$.

## Definition ($\Sigma$-algèbre)

Une $\Sigma$-algèbre est un couple $A = \langle D, O \rangle$, avec $D$ un ensemble $S$-typé de valeurs $(D = D_{s_1} \cup \ldots \cup D_{s_n}))$ et $O$ est un ensemble de fonctions, tel que pour tout nom d'opération $f \in F_{w,s}$ $(w = s_1 \ldots s_n)$ il existe une fonction $f^A \in O$ tel que $f^A : D_{s_1} \times \ldots \times D_{s_n} \rightarrow D_s$.

La relation $\models$ utilise :

- un morphisme $eval : T_\Sigma \rightarrow M$ et
- une interprétation des variables $I : X \rightarrow M$.

Alors $M \models t_1 = t_2$ ssi $\forall I, eval(t_1[I]) = eval(t_2[I])$.

# Spécification incrémentale

Pour minimiser l'effort d'écriture de spécifications, trois mécanismes sont disponibles :

1. Importation de spécifications existantes pour re-utilisation de sortes ou d'opérations.
   Ex. : importer BOOLEAN dans NATURAL.
2. Spécification générique (paramétrée) et sa concrétisation.
   Ex. : pile d'entiers obtenue à partir d'une pile générique d'éléménts.
3. Héritage et extension d'une spécification.
   Ex. : QUEUE est obtenue par héritage de LIST.

# Importation de spécifications

- Importation multiple par liste de spécifications importées après **imports**.
- Correspond à une union disjointe de spécifications algébriques.
  Attention : à la surcharge de noms d'opérations !
- Peut amener des problèmes :
  - d'inconsistance : des classes d'équivalences de termes sont confondues et
  - d'incomplétude : des classes d'équivalences de termes sont introduites.
- Toutefois, il faut utiliser l'importation sans retenue car c'est le moyen le plus simple pour avoir des spécification modulaires.

# Importation et consistance

**Definition (Consistance de la composition)**

Soient $Spec, Spec'$ deux spécifications algébriques. Leur composition disjointe $Spec + Spec'$ est consistante ssi $\forall t_1, t_2 \in T_{\Sigma \cup \Sigma'}$, $\forall M \in Mod(Spec), M' \in Mod(Spec'), M'' \in Mod(Spec + Spec'), M'' \models t_1 = t_2 \Rightarrow M \models t_1 = t_2$.

Exemple : pile de naturels avec les équations :

```
top(empty) = 0 ;
top(push(n,s)) = n ;
push(n,s) = s ;
```

permet de démontrer que `0=succ(0)`, donc tous les naturels seront dans la même classe d'équivalence !

# Importation et complétude

**Definition (Complétude de la composition)**

Soient $Spec, Spec'$ deux spécifications algébriques. Leur composition disjointe $Spec + Spec'$ est complète ssi $\forall s \in S$, $\forall t \in (T_{\Sigma \cup \Sigma'})_s$, $\exists t_1 \in (T_\Sigma)_s$ tel que $\forall M'' \in Mod(Spec + Spec'), M'' \models t_1 = t$.

Exemple : pile de naturels avec l'unique équation :

```
top(push(n,s)) = n ;
```

permet d'introduire des classes d'équivalence de termes pour la sort `nat`, par exemple `top(empty)`, `succ(top(empty))`, ... !

# Spécification générique : définition

Par exemple : Arbre binaire de recherche générique.

```
type BTREE is
  imports BOOLEAN
  formalsorts elem
  formalopns
    Undef : -> elem
    eq, lt : elem, elem -> bool
  sorts btree
  opns
    nil : -> btree
    node : btree, elem, btree -> btree
    add : elem, btree -> btree
    left, right : btree -> btree
    data : btree -> elem
    isEmpty : btree -> bool
    isin : elem, btree -> bool
  eqns
    forall t,l,r : btree, d,e : elem
    ofsort btree
    add(d,nil) = node(nil,d,nil) ;
    lt(d,data(t))=true =>
      add(d,t) = node(add(d,left(t)),data(t),right(t)) ;
    lt(d,data(t))=false =>
      add(d,t) = node(left(t),data(t),add(d,right(t))) ;
    left(nil) = nil ;
    left(node(l,d,r)) = l ;
    right(nil) = nil ;
    right(node(l,d,r)) = r ;
    data(nil) = Undef ;
    data(node(l,d,r)) = d ;
    ofsort bool
    isEmpty(nil) = true ;
    isEmpty(node(l,d,r)) = false ;
    isin(nil,e) = false ;
    eq(d,e)=true =>
      isin(node(l,d,r),e) = true ;
    lt(e,d)=true =>
      isin(node(l,d,r),e) = isin(l,e) ;
    lt(e,d)=false and eq(e,d)=false =>
      isin(node(l,d,r),e) = isin(r,e) ;
endtype
```

# Spécification générique : concrétisation

La concrétisation des **formal**s est faite avec une spécification d'ADT qui contient des sortes et des opérations (éventuellement renommés) qui satisfont les mêmes contraintes de profile.

Exemple : Arbre binaire de recherche contenant des naturels.

```
type BTREE-NAT is BTREE
  actualizedby NATURAL using
    sortnames nat for elem
              btreeNat for btree
    opnames _=_ for eq
            _<_ for lt
            NaN for Undef
endtype
```

Une opération formelle n'est pas renommé si l'opération concrète a le même nom !

Le renommage peut concerner les sortes/opérations génériques (voir `btreeNat`) !

# *Rappel : spécification algébrique NATURAL*

```
type NATURAL is
  imports BOOLEAN
  sorts nat
  opns
    0 : -> nat
    succ : nat -> nat
    NaN : -> nat
    _+_, _-_  : nat, nat -> nat
    _*_, _**_ : nat, nat -> nat
    _eq_, _ne_, _lt_, _le_,
     _gt_, _ge_ : nat, nat -> bool
    _mod_, _div_ : nat, nat -> nat
    min, max, gcd, scm : nat, nat -> nat
  eqns
  forall x, y : nat
  ofsort nat
  succ(NaN) = NaN;
    0 + x = x;
    succ(x) + y = succ(x + y);
    NaN + x = NaN;
    x + y = y + x;
  ofsort nat
    x - 0 = x;
    0 - succ(y) = NaN;
    succ(x) - succ(y) = x - y;
    x - NaN = NaN;
    NaN - y = NaN;
  ofsort nat
    0 * y = 0;
    succ(x) * y = (x * y) + y;
    NaN * y = NaN;
    x * y = y * x;
  ofsort nat
    x ** 0 = succ(0);
    x ** succ(y) = x * (x ** y);
    x ** NaN = NaN;
    NaN ** y = NaN;
  ofsort bool
    0 eq 0 = true;
    0 eq succ(y) = false;
    0 eq NaN = false;
    succ(x) eq 0 = false;
    succ(x) eq succ(y) = x eq y;
    succ(x) eq NaN = false;
  ofsort bool
    0 lt 0 = false;
    0 lt succ(y) = true;
    (* NaN is a negative integer *)
```

```
  0 lt NaN = false;
  succ(x) lt 0 = false;
  succ(x) lt succ(y) = x lt y;
  succ(x) lt NaN = false;
  NaN lt y = true;
ofsort bool
  x le y = (x lt y) or (x eq y);
ofsort bool
  x gt y = not(x le y);
ofsort bool
  x ge y = not(x lt y);
ofsort nat
  y ne 0, x lt y => x div y = 0;
  y ne 0, x ge y =>
    x div y = 1 + ((x - y) div y);
  y eq 0 => x div y = NaN;
  NaN div y = NaN;
  x div NaN = NaN;
ofsort nat
  y ne 0, x lt y => x mod y = x;
  y ne 0, x ge y =>
    x mod y = ((x - y) mod y);
  y eq 0 => x mod y = NaN;
  NaN mod y = NaN;
  x mod NaN = NaN;
ofsort nat
  x le y => min (x, y) = x;
  x gt y => min (x, y) = y;
ofsort nat
  x ge y => max (x, y) = x;
  x lt y => max (x, y) = x;
ofsort nat
  x eq y, x ne 0 => gcd (x, y) = x;
  x lt y, x ne 0 =>
    gcd (x, y) = gcd (x, x - y);
  x gt y, y ne 0 =>
    gcd (x, y) = gcd (x - y, y);
  gcd (x, y) = gcd (y, x);
  gcd (NaN, y) = NaN;
ofsort nat
  scm (x, y) = (x * y) div gcd (x, y);
endtype
```

# Spécification héritée : définition

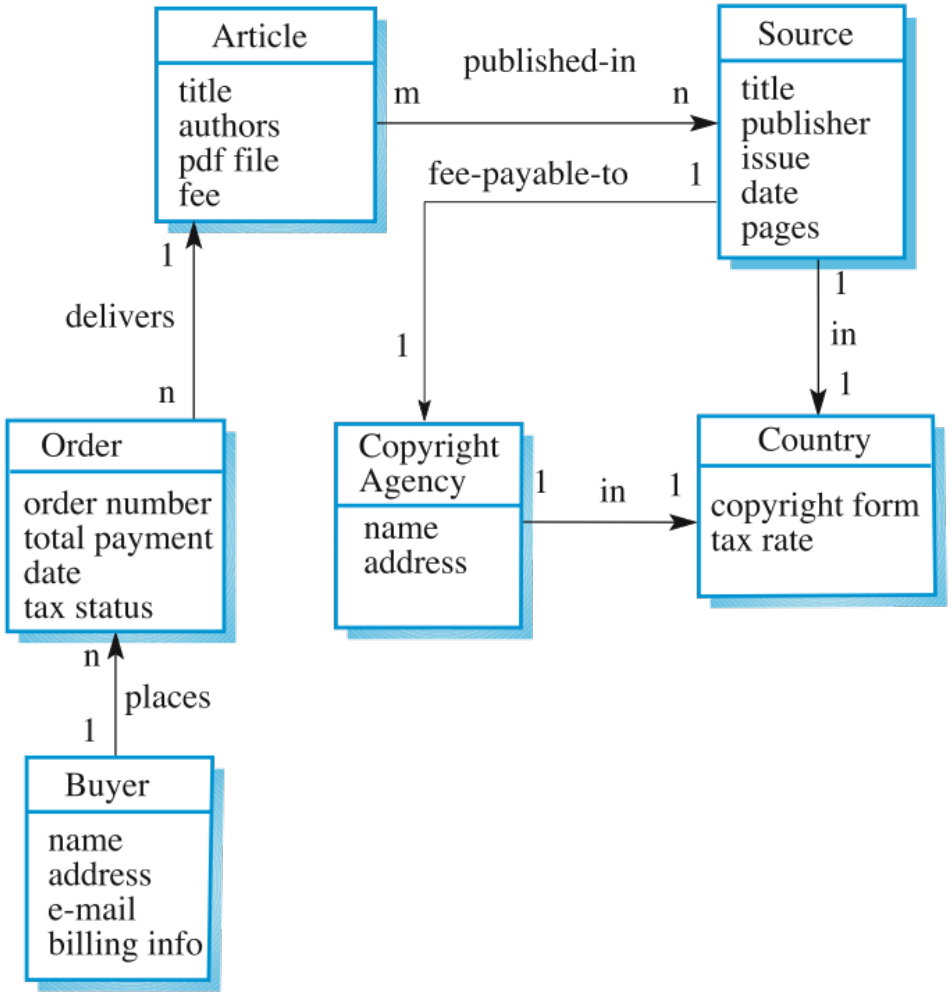
- Des sortes, opérations ou équations sont ajoutées à une liste de spécifications.
- L'héritage est multiple !
- Les même problèmes de consistance et complétude que pour l'importation.

Exemple :

```
type ABELIAN-GROUP is
  extends GROUP
    eqns
      x . y = y . x
endtype
```

# Exemple : formalisation des données LIBSYS

Modèle relationel :



# Exemple : formalisation des données LIBSYS

Spécifications ADT :

- Standard : `BOOLEAN`, `NATURAL`, `STRING`
- Utilitaires : `DATE`, `PAGES`, `EMAIL`, `ADDRESS`, `FILE`, `LIST`
- Principales : `COUNTRY`, `AGENCY`, `SOURCE`, `ARTICLE`, `ORDER`, `BUYER`, `AUTHOR`

Une partie de ces spécifications sont disponibles sur le site du projet.

# Utilisation des spécifications ADT

- Preuve des propriétés des spécifications :
  - obtenir des nouvelles théorèmes (équations) sur l'ADT.
    Ex. : Prouvez que `succ(0)+succ(succ(0))=succ(succ(succ(0)))`.
  - prouver l'inconsistance = impossibilité à obtenir une implémentation,
  - prouver l'incomplétude = non déterminisme à résoudre dans l'implémentation,
  - prouver une relation hiérarchique/d'inclusion entre les spécifications (voir cours GL),

- Obtenir du code correct par construction : si la spécification des opérations est faite dans un style fonctionnel, du code peut être généré ! Exemple : CAESAR.ADT de CADP.
  - nécessite de l'aide du spécificateur (quels sont les constructeurs ?)
  - certaines parties peuvent être externes pour obtenir l'efficacité
  - rejet de spécifications correctes