Ingénierie Dirigée par les Modèles

Méta-modélisation et transformations de modèles

Marc Pantel, Xavier Crégut, Benoît Combemale, Arnaud Dieumegard

IRIT–ENSEEIHT
2, rue Charles Camichel - BP 7122
F-31071 Toulouse Cedex 7
{prenom.nom}@enseeiht.fr

Sommaire

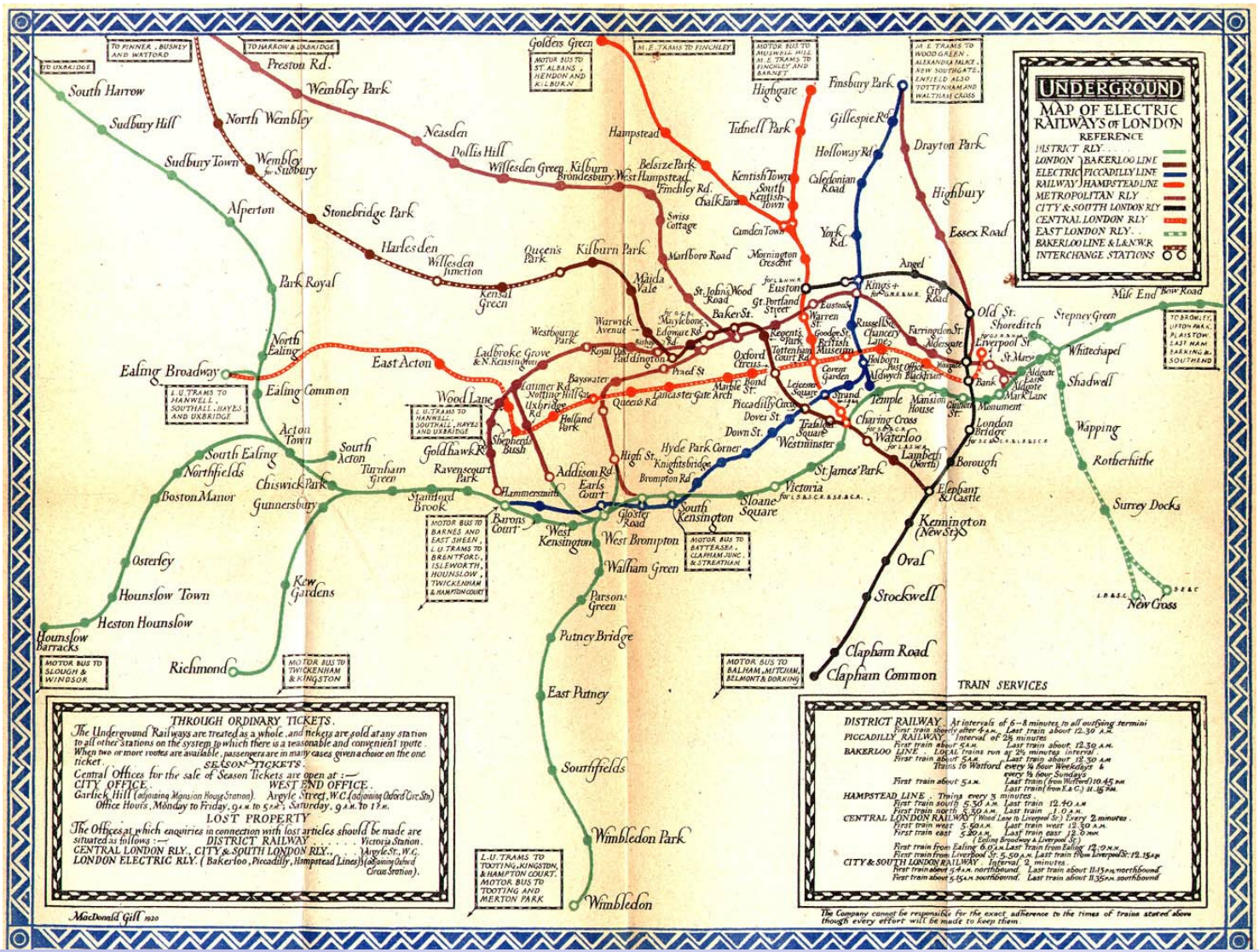
Sommaire

Rôle d'un modèle

- On utilise des modèles pour mieux comprendre un système.
  Pour un observateur A, M est un modèle de l'objet O, si M aide A à répondre aux questions qu'il se pose sur O. (Minsky)
- Un modèle est une simplification, une abstraction du système.
- Exemples :
  - une carte routière
  - une partition de musique
  - un plan d'architecte
  - un diagramme UML
  - ...
- Un modèle permet :
  - de comprendre,
  - de communiquer,
  - de construire

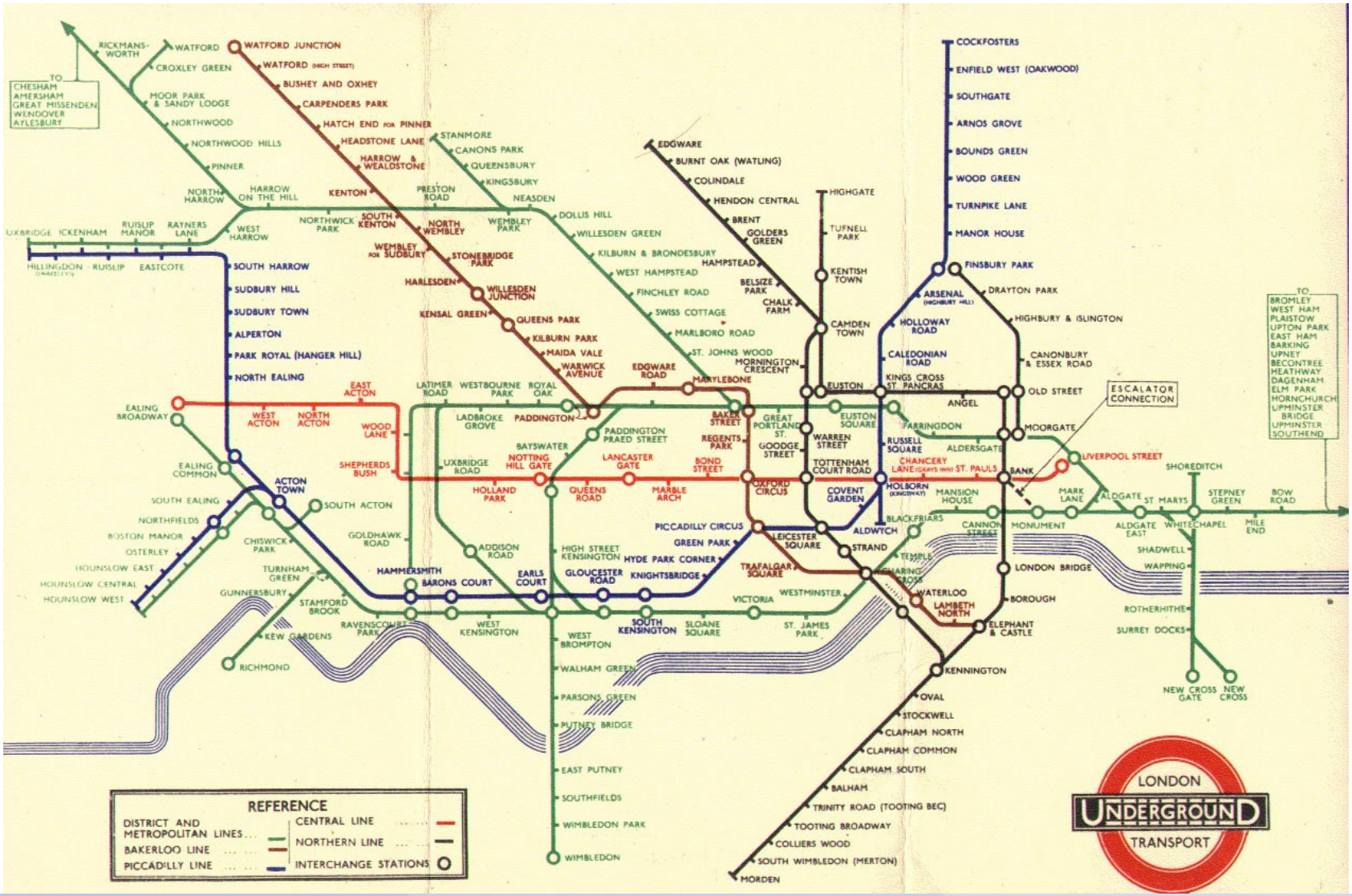
Exemple : plan géographique du métro de Londres

Version de 1921 (`http://www.clarksbury.com/cdl/maps.html`)



Exemple : plan schématique du métro de Londres

Version schématique — Harry Beck — de 1938 (`http://www.clarksbury.com/cdl/maps.html`)



Pourquoi modéliser ?

- Mieux comprendre les systèmes complexes
- Séparation des préoccupations/aspects
- Abstraction des plateformes :
  - Architecture matérielle, Réseau
  - Architecture logicielle, Système d'exploitation
  - Langages
- Abstraction des domaines applicatifs
- Réutilisation
- Formalisation

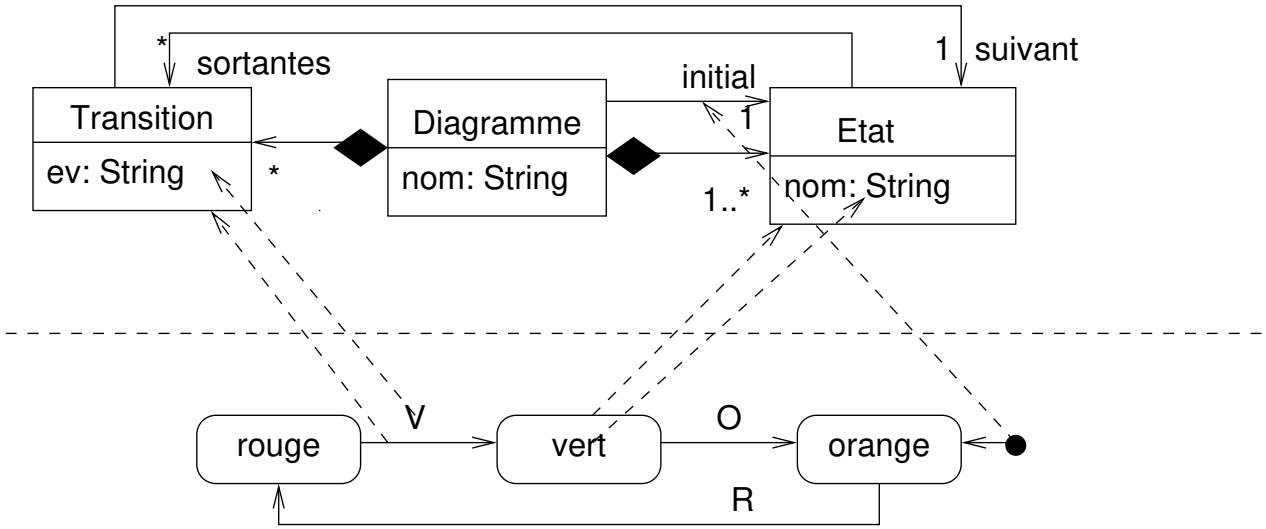
Pourquoi de nombreux modèles ?

- Le long du cycle de vie :
  - Analyse des besoins (indépendant solution)
  - Architecture, Conception détaillée (indépendant plateforme)
  - Réalisation, Déploiement (dépendant plateforme)
- Différentes étapes de raffinement dans une même phase
- Séparation des préoccupations
  - Nombreux domaines applicatifs
  - Nombreuses plateformes (matériel, logiciel, technologique)
  - Nombreuses contraintes (service et qualité de service)

Modèles et méta-modèles

Définition : Méta-modèle = modèle du modèle.
$\Longrightarrow$ Il s'agit de décrire la structure du modèle.

Exemple : Structure d'un diagramme à état (diagramme de classe UML)



Conformité : Un modèle est conforme à un méta-modèle si :

- tous les éléments du modèle sont instance d'un élément du méta-modèle ;
- et les contraintes exprimées sur le méta-modèle sont respectées.

Conformité (vision tabulaire) M1/M2

Etat

ID	nom	sortantes
E1	"orange"	T2
E2	"rouge"	T3
E3	"vert"	T1

Transition

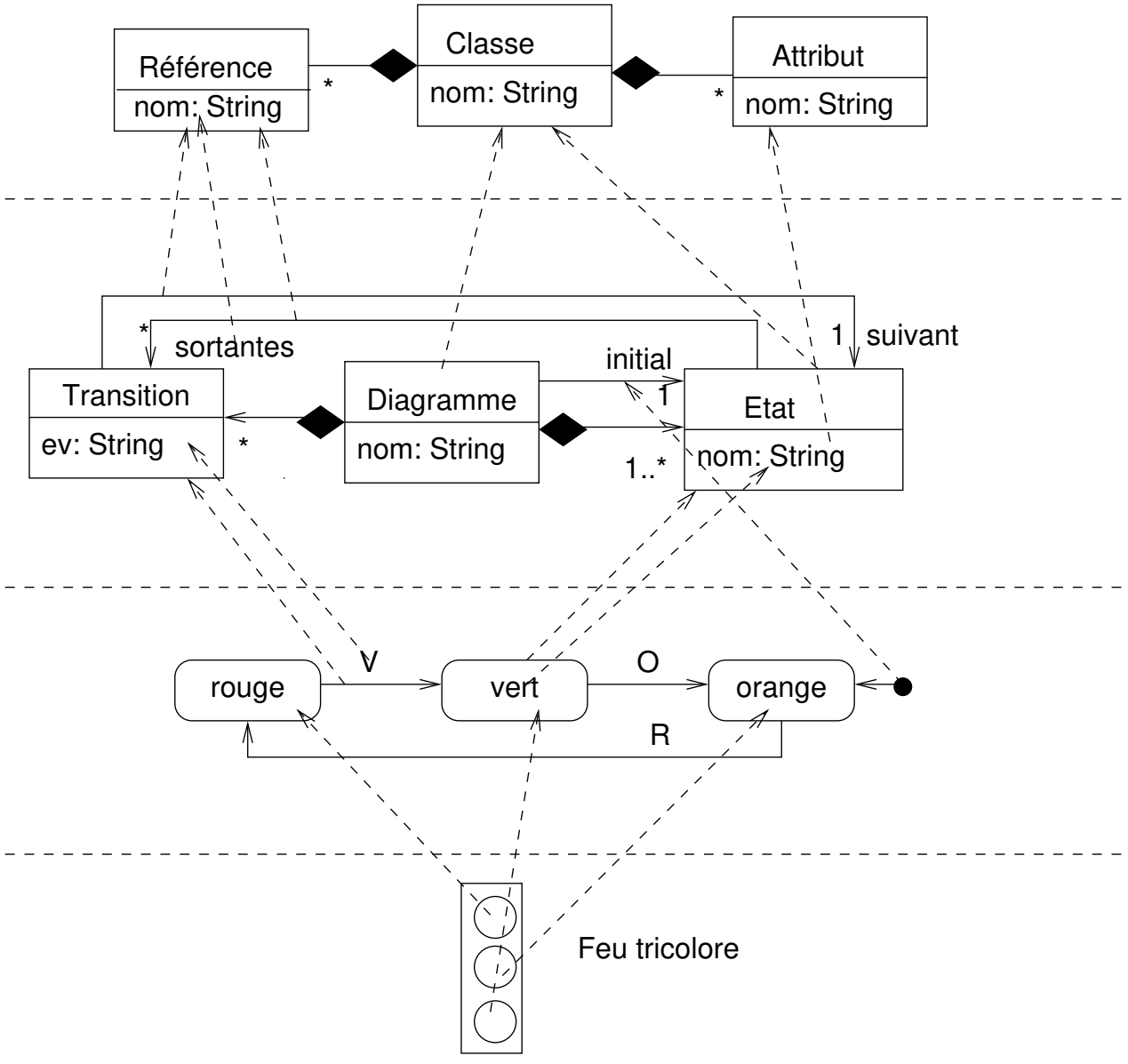
ID	ev	suivant
T1	"O"	E1
T2	"R"	E2
T3	"V"	E3

Diagramme

ID	nom	etats	initial	transitions
D1	"Feu Tricolore"	E1, E2, E3	E1	T1, T2, T3

Suite : Où est décrit le méta-modèle ?

Exemple : le monde réel est un feu tricolore



Conformité (vision tabulaire) M2/M3

Classe

ID	nom	attributs	references
C1	"Diagramme"	A1	R1, R2, R3
C2	"Etat"	A2	R4
C3	"Transition"	A3	R5

Attribut

ID	nom	type
A1	"nom"	"String"
A2	"nom"	"String"
A3	"ev"	"String"

Reference

ID	nom	cible	min	max	composition
R1	"etats"	C2	1	*	true
R2	"transitions"	C3	0	*	true
R3	"initial"	C2	1	1	false
R4	"sortantes"	C3	0	*	false
R5	"suivant"	C2	1	1	false

Suite : Où est décrit le méta-méta-modèle ?

Conformité (vision tabulaire) M3/M3

Classe

ID	nom	attributs	references
C1	"Classe"	A1	R1, R2
C2	"Attribut"	A2, A3	
C3	"Reference"	A4, A5, A6, A7	R3

Attribut

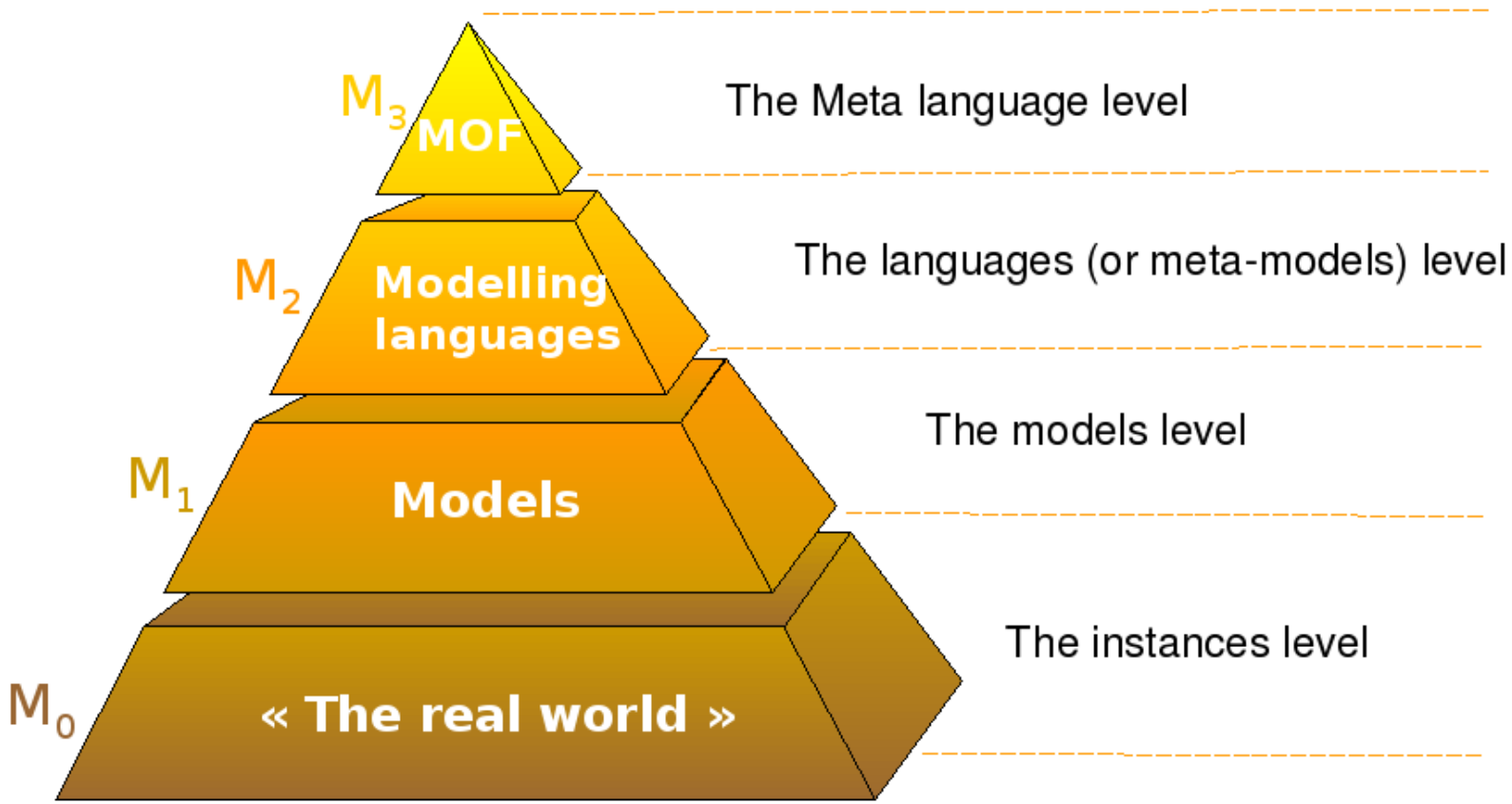
ID	nom	type
A1	"nom"	"String"
A2	"nom"	"String"
A3	"type"	"String"
A4	"nom"	"String"
A5	"min"	"int"
A6	"max"	"int"
A7	"composition"	"boolean"

Reference

ID	nom	cible	min	max	composition
R1	"attributs"	C2	0	-1 (*)	true
R2	"references"	C3	0	-1 (*)	true
R3	"cible"	C1	1	1	false

Suite : Où est décrit le méta-méta-méta-modèle ? **Réponse :** Il n'y en pas : M4 = M3

Pyramide de l'OMG



Pyramide de l'OMG

Explications

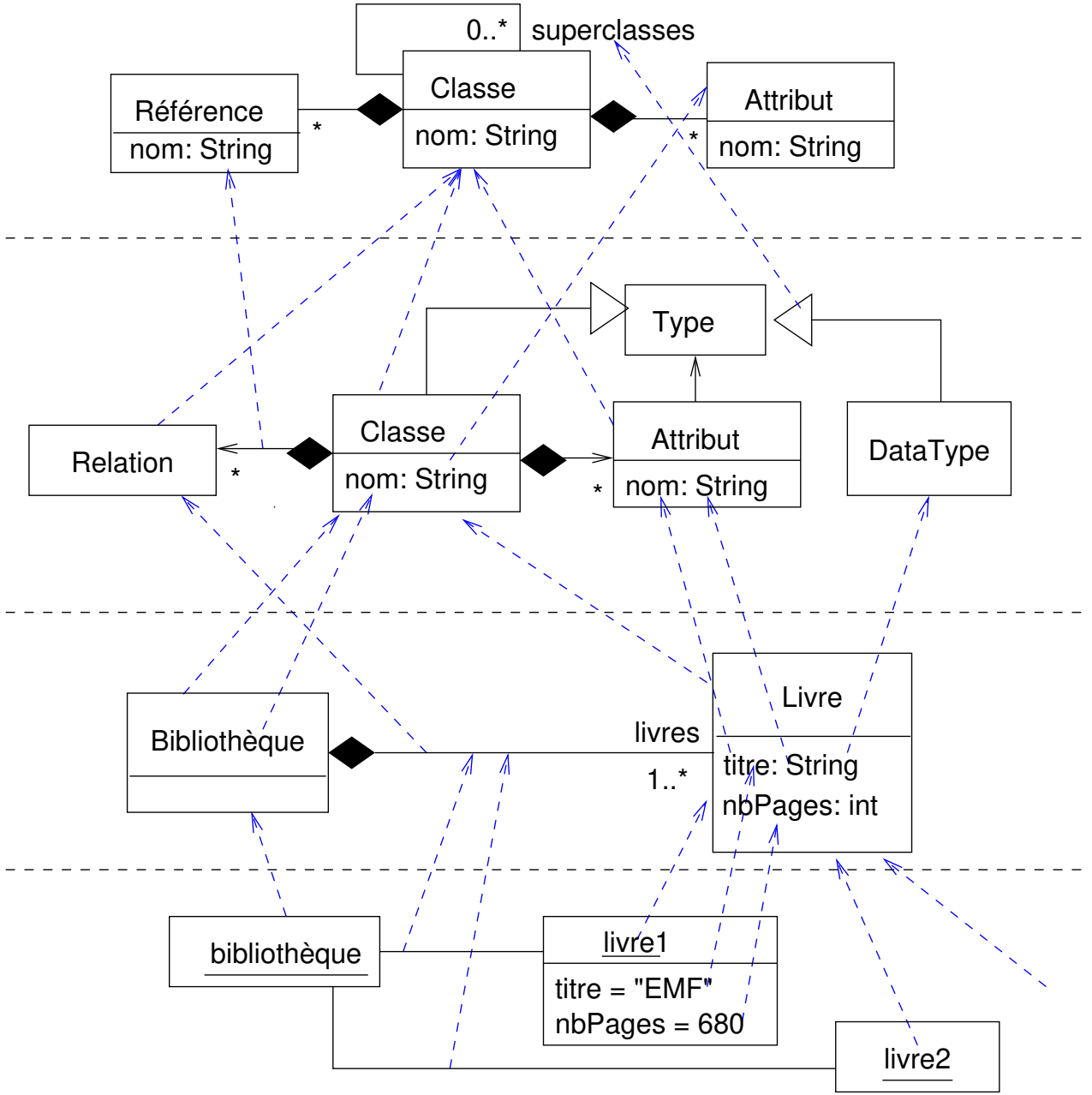
- M3 : méta-méta-modèle :
  - réflexif : se décrit en lui-même
  - pour définir des méta-modèles, langages (exemple : UML)
  - exemple MOF de l'OMG
- M2 : méta-modèle : langage de modélisation pour un domaine métier
  - Exemples : UML2, SPEM...
- M1 : modèle : un modèle du monde réel
  - Exemples : un modèle de feu tricolore, un modèle de bibliothèque...
- M0 : le monde réel
  - Exemples : un feu tricolore, une bibliothèque...

Remarque : Le numéro permet de préciser l'objectif du « modèle ». Dans la suite, les notions de modèle et méta-modèle sont suffisantes.

Pas nouveau :

- Grammarware : EBNF, syntaxe de Java, Programme Java, exécution
- XMLware : XML+DTD, MathML, document valide, données réelles

Exemple : le monde réel est une bibliothèque



Intérêt des méta-modèles

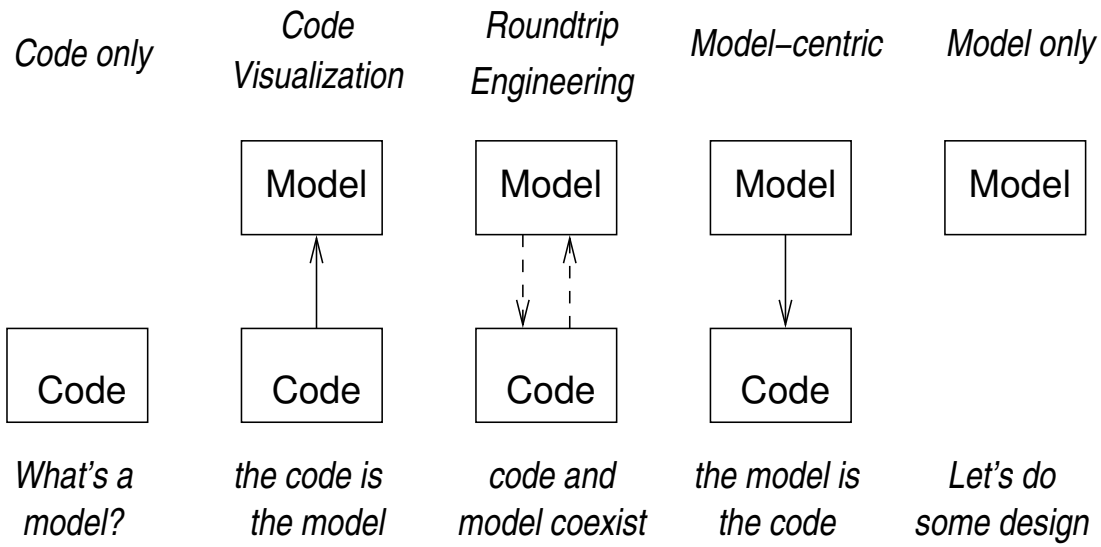
- définir les **propriétés structurelles** d'une famille de modèles :
  - capturées par la structure du méta-modèle (multiplicité, références, etc.)
  - exprimées dans un langage de contrainte.
    Exemple : Exprimer que le nb de pages d'un livre est positif en OCL :

    context Livre **inv**: nbPages > 0

- décider de la **conformité** d'un modèle par rapport à un métamodèle
- **transformer** le modèle (restructuration, raffinement, traduction vers un autre MM, syntaxes concrètes...)
- permettre l'**interopérabilité** entre outils grâce à une description connue (le MM) des données échangées
- plus généralement, **raisonner** et **travailler** sur les modèles
- ...
- Mais a-t-on défini la sémantique du MM ?

Remarque : La sémantique d'un langage de programmation est définie sur le langage (M2), pas sur le programme (M1).

Modèle et code : différentes perspectives



(http://www.ibm.com/developerworks/rational/library/3100.html)

Remarque : L'évolution est à aller vers le tout modèle :

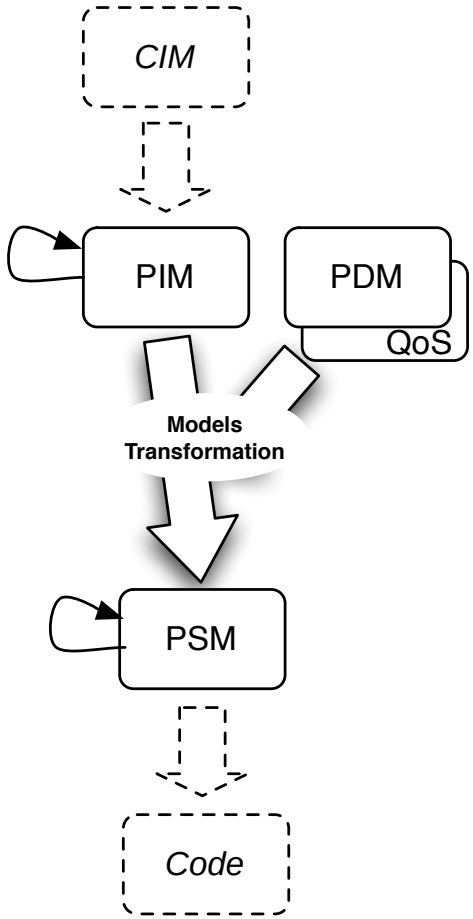
- modèles nécessaires car au début le système est trop compliqué
- besoin de vérifier/valider les modèles (coût si erreurs identifiées tardivement)
- raffiner les modèles et aller vers le code

Le modèle au centre du développement

Objectif : Tenter une interopérabilité par les modèles

- Partir de CIM (Computer Independant Model) :
  - aucune considération informatique n'apparaît
- Faire des modèles indépendants des plateformes (PIM)
  - rattaché à un paradigme informatique ;
  - indépendant d'une plateforme de réalisation précise
- Spécifier des règles de passage (transformation) vers ...
- ... les modèles dépendants des plateformes (PSM)
  - version modélisée du code ;
  - souvent plus facile à lire.
- Automatiser au mieux la production vers le code
  PIM $\rightarrow$ PSM $\rightarrow$ Code

$\Longrightarrow$ Processus en Y

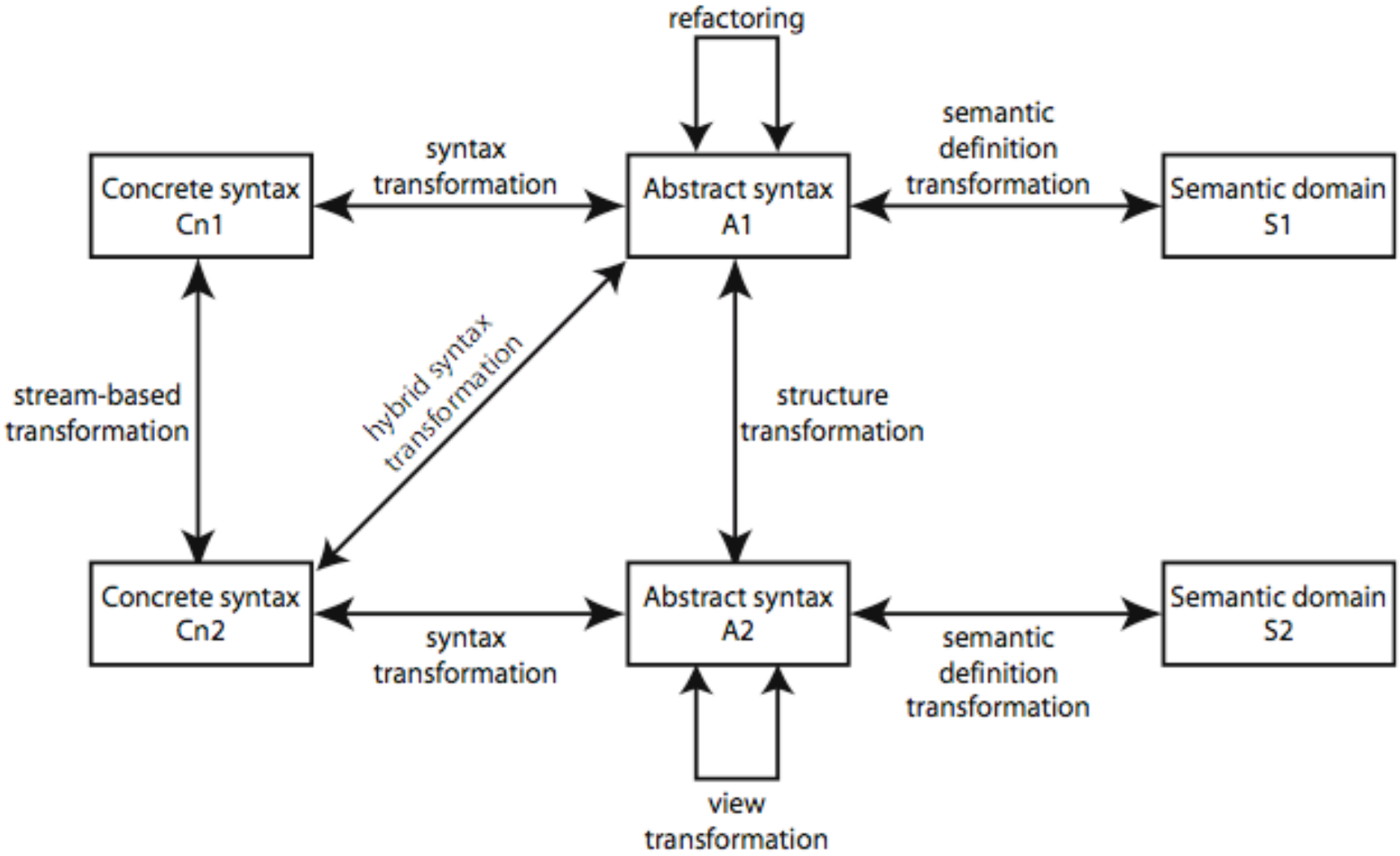


Exemples de transformation

- PIM $\longrightarrow$ PIM :
  - privatiser les attributs
  - réorganiser le code (refactoring) : introduction de patron de conception...
- PIM $\longrightarrow$ PSM
  - génération semi-automatique grâce à des marqueurs :
    - classe marquée active $\Rightarrow$ hérite de Thread...
    - persistance
  - motif de passage d'une classe UML à une classe Java
  - prise en compte de l'héritage multiple (C++, Eiffel, Java)
  - prise en compte des technologies disponibles
- PSM $\longrightarrow$ PIM :
  - adaptation pour gérer l'interopérabilité entre outils
  - rétroconception, abstraction, analyse statique...

Conséquence : L'Ingénierie Dirigée par les Modèles repose sur la **méta-modélisation** ET les **transformations**.

Types de transformation

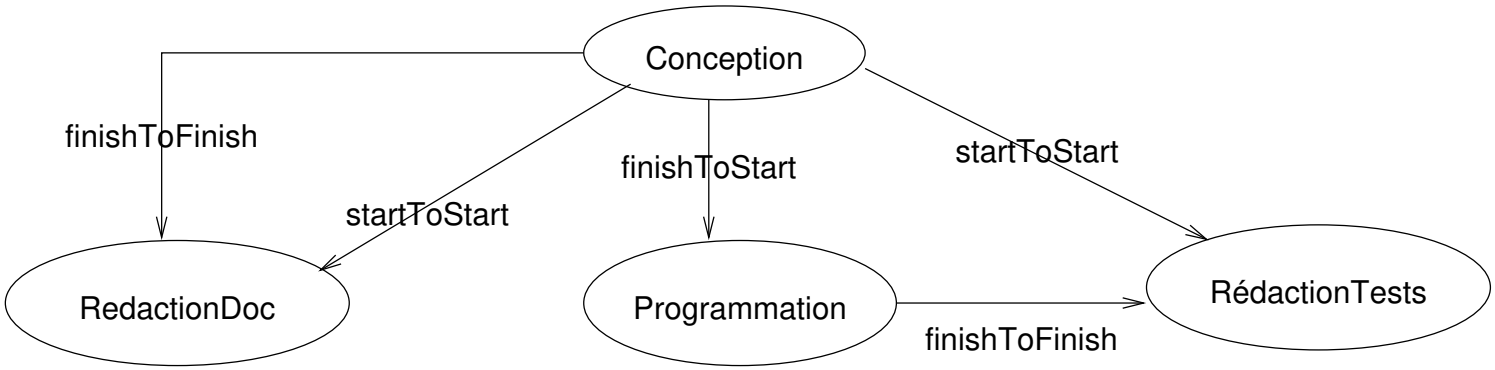


(from Anneke Kleppe)

Sommaire

Étude de cas : Vérifier la terminaison de processus

- **Définition :** Un processus (Process) est composé de plusieurs éléments :
  - activités (WorkDefinition)
  - dépendances (WorkSequence) entre activités
  - ressources (Resource)
  - et des notes (Guidance)
- **Exemple** de processus (sans resources)



- **Question :** Est-ce qu'un processus (quelconque) peut se terminer ?

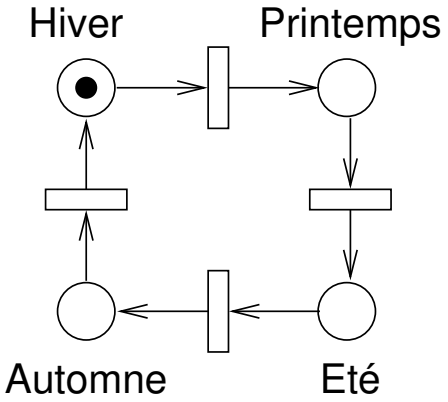
Problèmes posés

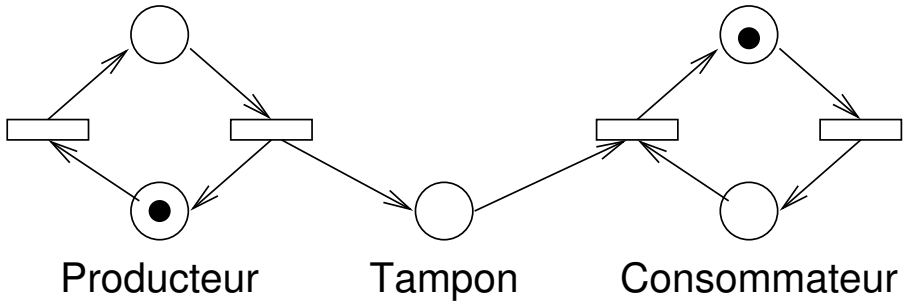
- **Pour répondre à la question, il faut :**
  - savoir comment sera exécuté un procédé (sémantique d'exécution)
  - en particulier, tenir compte des contraintes :
    - dépendances (*WorkSequence*) : vérifier l'état des activités
    - ressources (*Resource*) : il faut gérer les allocations et les libérations
  - examiner (toutes) les exécutions possibles pour voir si une au moins termine
  - être efficace, etc.

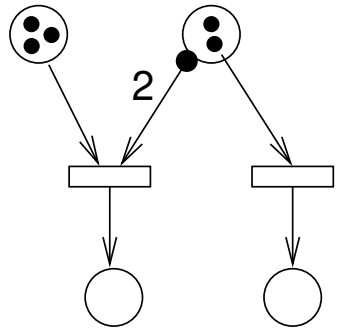
  $\Longrightarrow$ **Ceci est difficile !**
- **Solution choisie :**
  - Définir une **sémantique par traduction**
    - exprimer la sémantique de SimplePDL en s'appuyant sur un langage formel (ex. les réseaux de Petri).
  - et s'**appuyer sur les outils existants** (ex. le *model-checker de Tina*)

Les réseaux de Petri

- **Quelques exemples :**

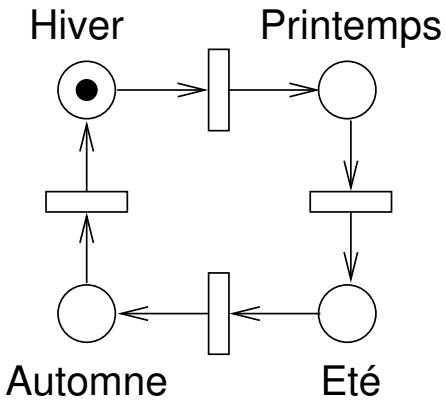






- **Vocabulaire :** place, transition, arc, read_arc, jeton
- Une transition est **franchissable** si toutes les places entrantes contiennent au moins le nombre de jetons indiqué sur l'arc
- **Tirer une transition :** enlever des places entrantes le nombre de jetons correspondant au poids de l'arc (sauf read_arc) et placer dans les places de sorties le nombre de jetons indiqué sur les arcs sortants

Syntaxe concrète des réseaux de Petri pour Tina




```
pl Hiver (1)
tr h2p Hiver -> Printemps
tr p2e Printemps -> Ete
tr e2a Ete -> Automne
tr a2h Automne -> Hiver
```

Expression de propriétés

- Propriétés exprimées en LTL (Logique Temporelle Linéaire) :

```
[] <> Ete;      # Toujours il y aura un été
− <> Ete;       # Il n'y aura pas d'été
```

- Pour vérifier ces propriétés, il suffit de taper :

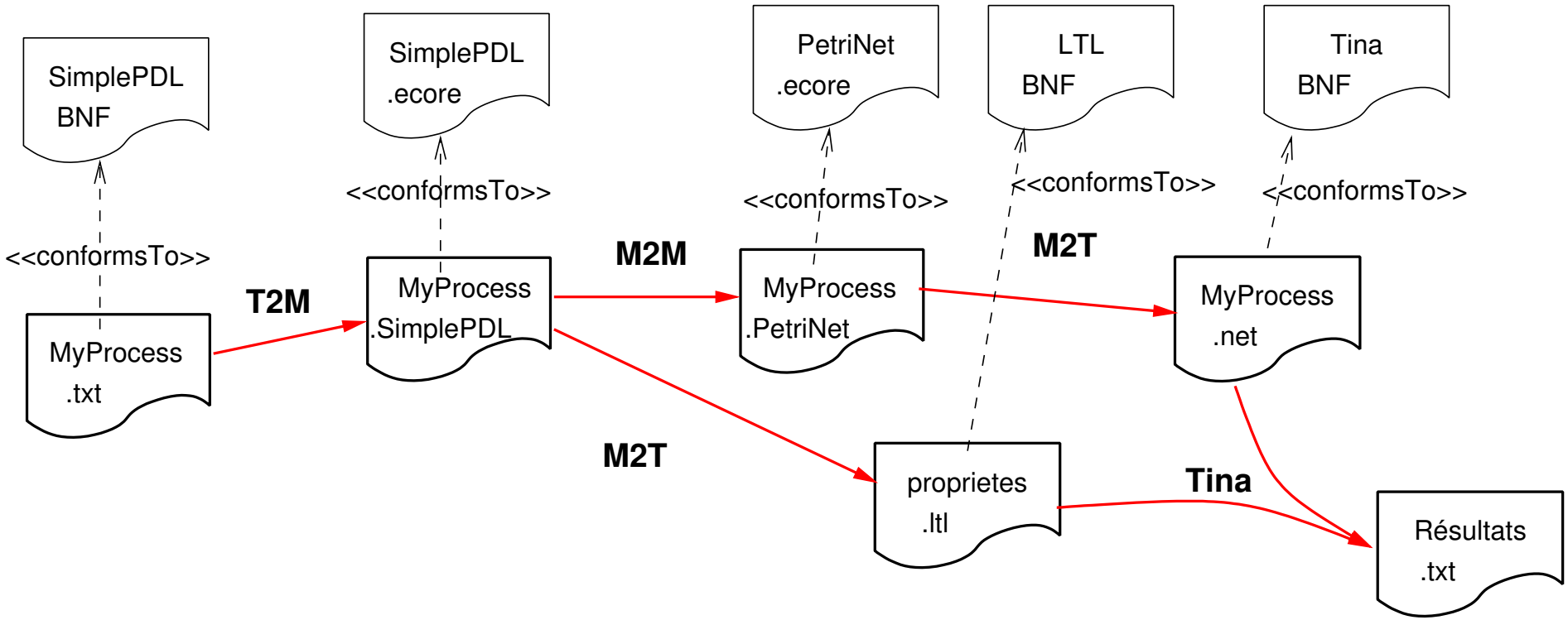
```
tina −s 3 saisons.net saisons.ktz
selt −S saisons.scn saisons.ktz −prelude saisons.ltl
```

- Le résultat est :

```
1  Selt version 2.9.4 −− 11/15/08 −− LAAS/CNRS
2  ktz loaded, 4 states, 4 transitions
3  0.000s
4
5  − source saisons.ltl;
6  TRUE
7  FALSE
8    state 0: Hiver
9    −h2p ... (preserving T)−>
10   state 2: Ete
11   −e2a ... (preserving Ete)−>
12   state 4: Automne
13   [accepting all]
14 0.000s
```

Propriété 1 vraie
Propriété 2 fausse
un contre-exemple fourni

Schéma général



Méta-modèles et transformations

Deux méta-modèles :

- SimplePDL
- PetriNet

Trois types de transformations :

- Transformations **texte à modèle** pour définir des syntaxes concrètes :
  - textuelles : par exemple avec Xtext
  - graphique : par exemple avec GMF ou Sirius
- Transformations de **modèle à modèle** :
  - traduire un modèle SimplePDL en un modèle PetriNet
- Transformations **modèle à texte** :
  - transformer un modèle PetriNet dans la syntaxe concrète de Tina
  - engendrer la propriété LTL de terminaison à partir d'un modèle de SimplePDL

Sommaire

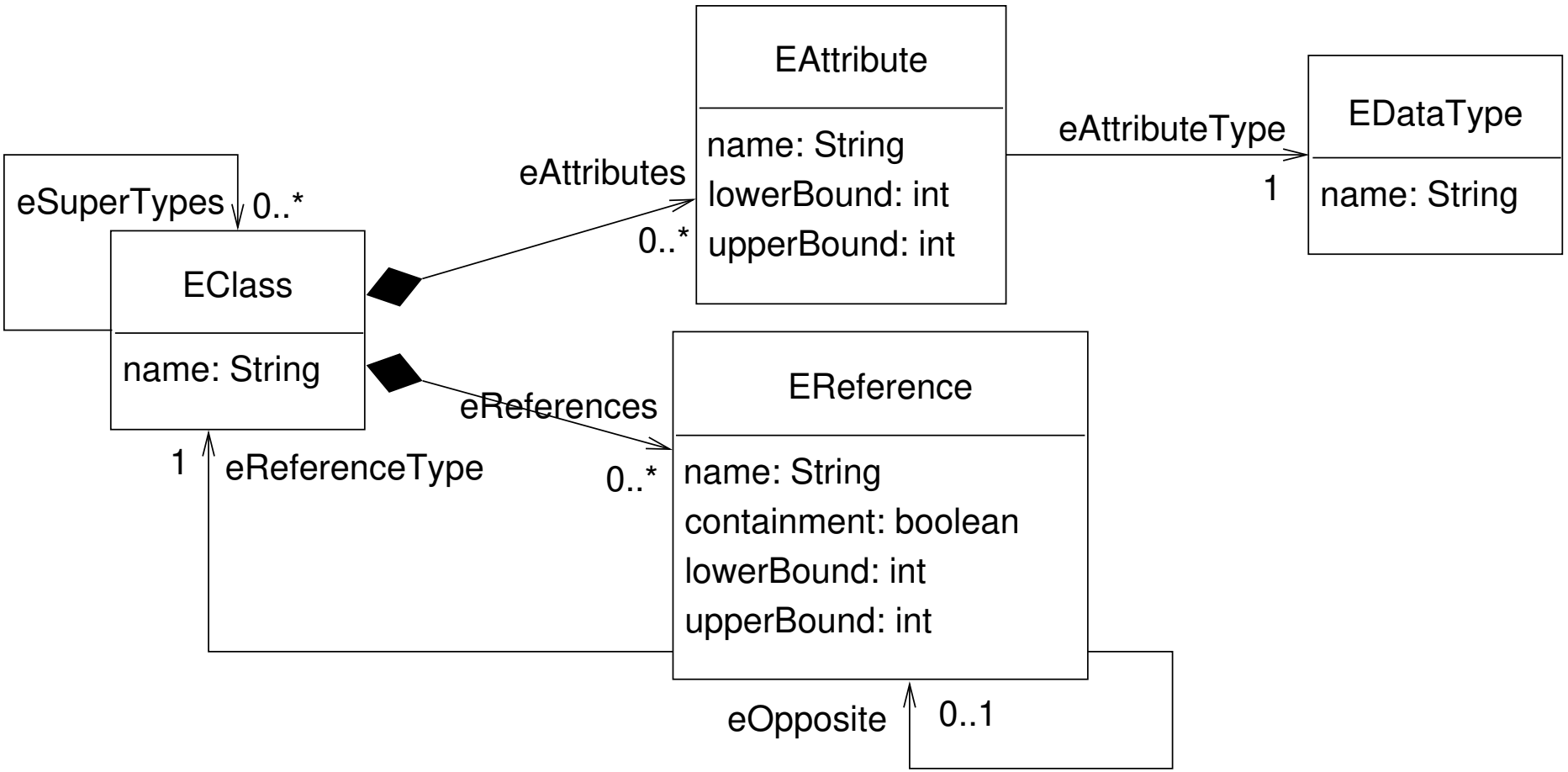
Les langages de méta-modélisation

Plusieurs langages proposés :

- MOF (Meta-Object Facility) proposé par l'OMG, variantes EMOF et CMOF
  Au départ description de UML en UML
  Extraction du minimum d'UML pour décrire UML $\Longrightarrow$ MOF
- **Ecore** : Eclipse/EMF (Eclipse Modelling Framework)
  Implantation de EMOF (équivalent)
- KM3 (Kernel MetaMetaModel) : Meta-modèle de AMMA/ATL, (LINA, Nantes)
- Kermeta : (IRISA, Rennes) extension de EMOF/Ecore pour permettre de décrire le comportement d'un méta-modèle (méta-programmation).
- GME (The Generic Modeling Environment), Vanderbilt.
  `http://www.isis.vanderbilt.edu/projects/gme/`
- ...

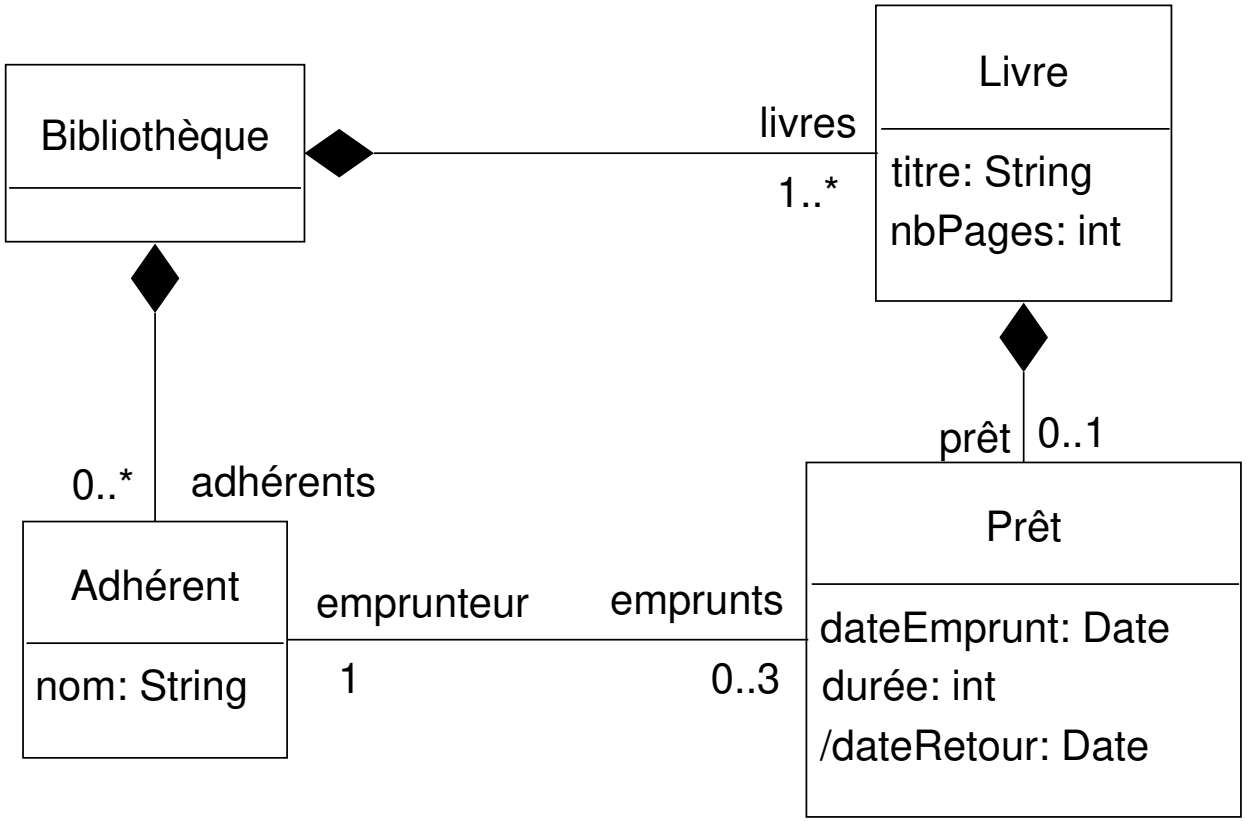
Le langage de méta-modélisation ECore (Eclipse/EMF)

Extrait du méta-modèle ECore : principales notions



Le langage de méta-modélisation ECore (Eclipse/EMF)

Exemple de modèle ECore : bibliothèque



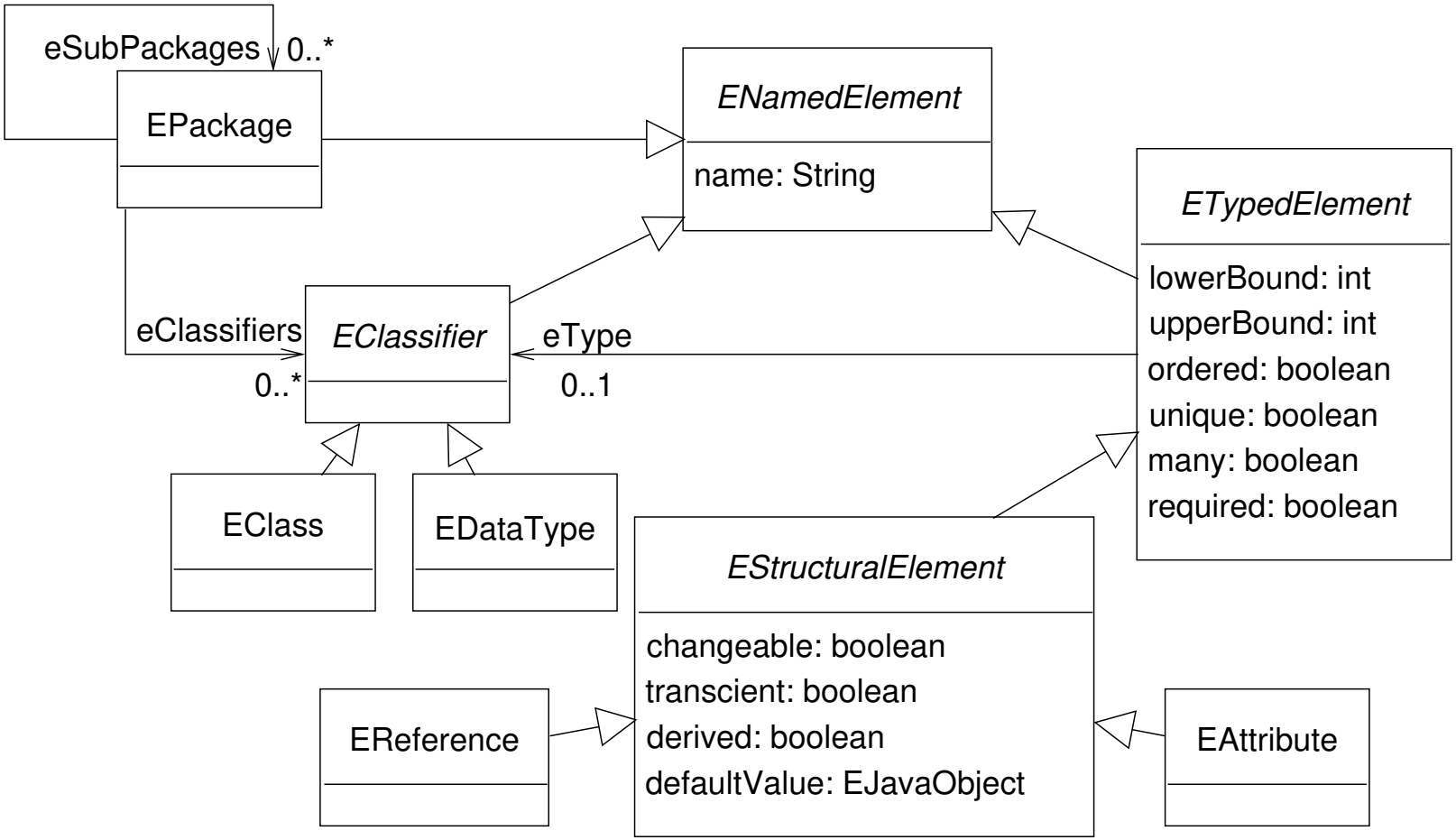
Le langage de méta-modélisation ECore (Eclipse/EMF)

Principaux constituants

- EClass : Description d'un concept
  caractérisé par des attributs et des références
- EAttribute : une propriété de l'objet dont le type est « élémentaire »
- EReference : une référence vers un autre concept (EClass)
  équivalent à une association UML avec sens de navigation
- La propriété *containment* indique s'il y a *composition* :
  - vrai : l'objet référencé est contenu (durées de vie liées)
  - faux : c'est une référence vers un objet contenu par un autre élément.
- multiplicité définie sur les attributs et les références (idem UML).
  Convention : on note -1 pour indiquer $*$ pour *upperBound*
- Héritage multiple : *eSuperTypes*
- Référence *eOpposite* pour indiquer que deux références opposées sont liées (équivalent association UML).

Le langage de méta-modélisation ECore (Eclipse/EMF)

Extrait méta-modèle ECore : propriétés structurelles



Le langage de méta-modélisation ECore (Eclipse/EMF)

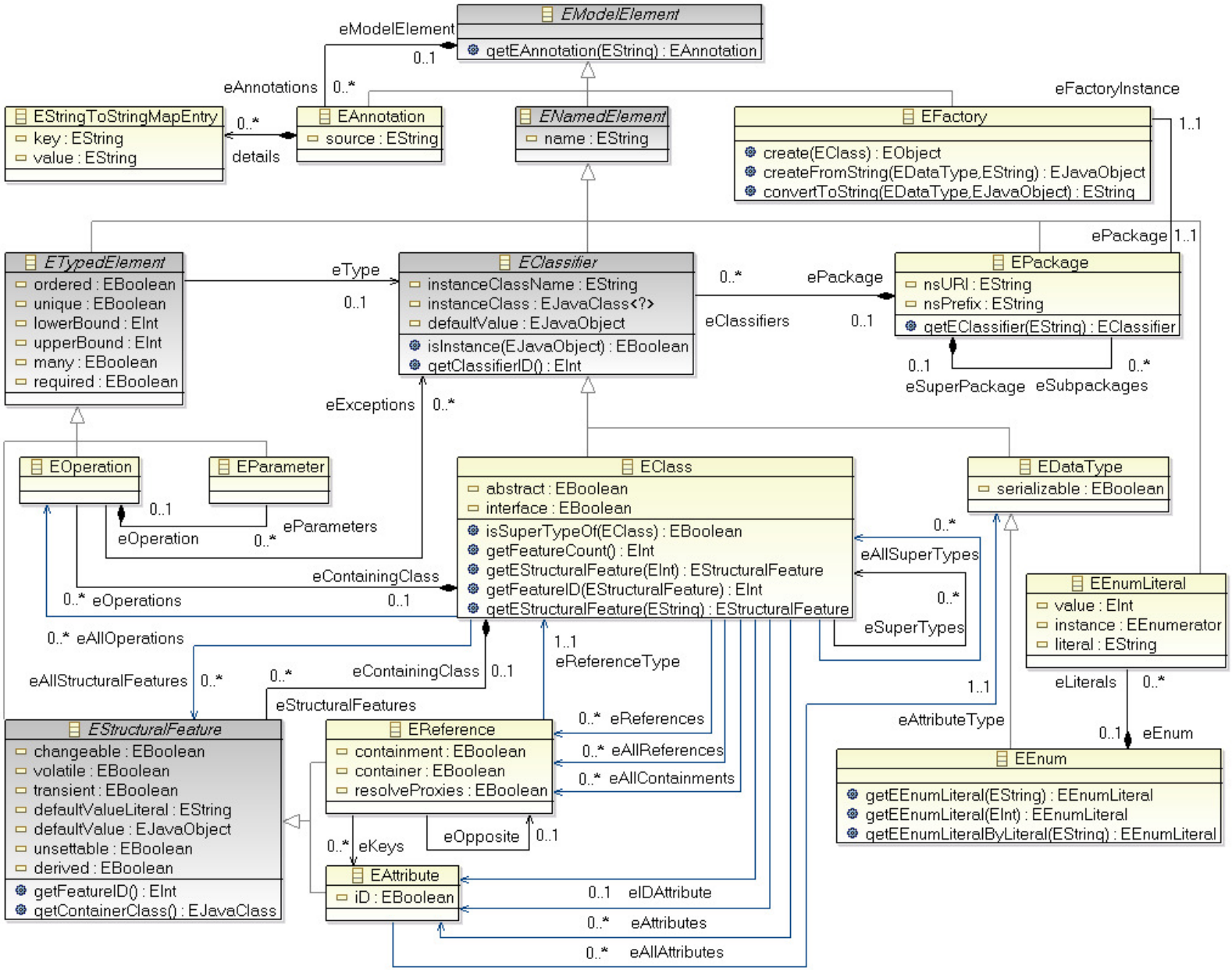
Autres caractéristiques de Ecore

- Méta-modèle : plus riche que le premier présenté.
- Éléments abstraits : ENamedElement, ETypedElement, etc.
- Paquetage : ensemble de classes et paquetages
- Caractéristiques liées à la multiplicité : ordered, unique...
- EEnum : énumération : lister les valeurs possibles d'un EDataType.
- Opération (non présentées) : décrit la signature des opérations, pas le code.

Remarque : Héritage multiple et classes abstraites favorisent la factorisation et la réutilisation (ex : ENamedElement, ETypedElement).

Métamodèle d'Ecore

refcardz.dzone.com/refcardz/essential-emf

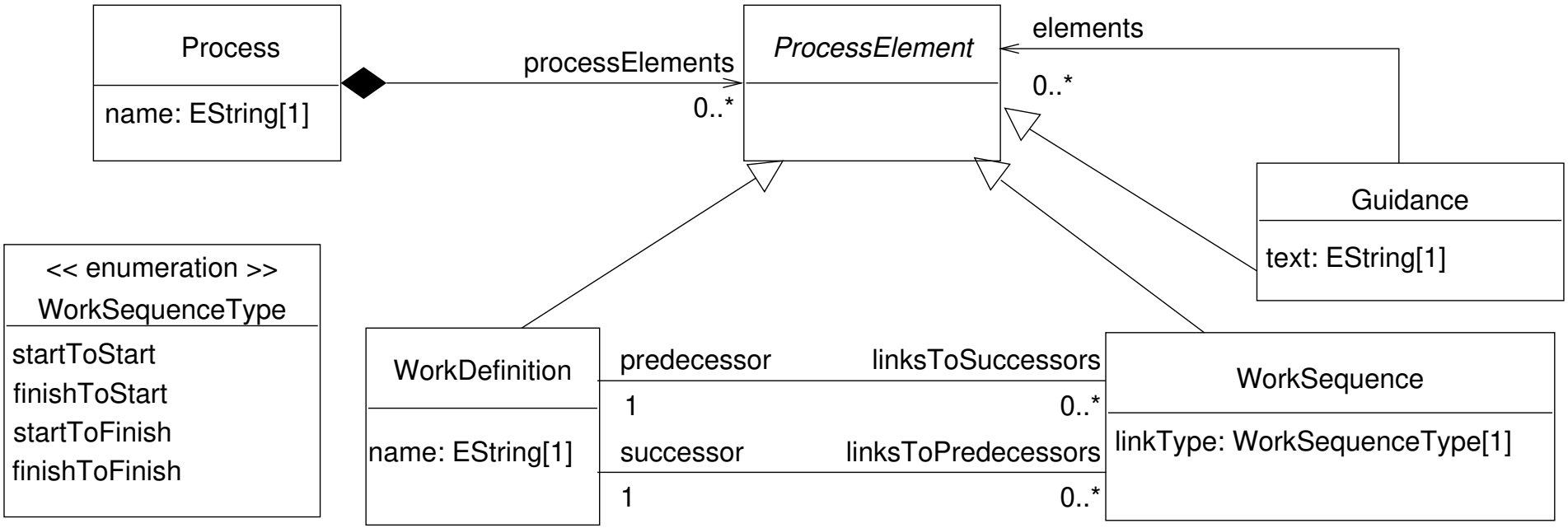


Intérêt de définir un modèle ECore

EMF permet d'engendrer :

- Le modèle Java correspondant :
  - chaque EClass donne une interface et une réalisation. Justification :
    - Bonne pratique que de définir des interfaces !
    - Permet de gérer l'héritage multiple
  - équipée d'observateurs (changement d'attribut ou de référence).
- Un schéma XML correspondant et les opérations de sérialisation/désérialisation associées.
- Un éditeur arborescent pour saisir un modèle.

Le métamodèle de SimplePDL



Attention : Toutes les propriétés ne sont pas capturées par le MM.

$\Longrightarrow$ Il faut donc le compléter : **sémantique statique**

Par exemple avec des propriétés OCL.

Contraintes OCL

```
1  context ProcessElement
2  def: process(): Process =
3          Process.allInstances()->select(p | p.processElements->includes(self))
4                  ->asSequence()->first()
5
6  context WorkSequence
7  inv previousWDinSameProcess: self.predecessor.process = self.process
8  inv nextWDinSameProcess: self.successor.process = self.process
```

Même métamodèle dans avec une syntaxe textuelle (OCLinECore) I

```
1   package simplepdl : simplepdl = 'http://simplepdl' {
2     enum WorkSequenceType { serializable } {
3       literal startToStart;
4       literal finishToStart = 1;
5       literal startToFinish = 2;
6       literal finishToFinish = 3;
7     }
8
9     class Process {
10      attribute name : String;
11      property processElements : ProcessElement[*] { ordered composes };
12      invariant nameForbidden: name <> 'Process';
13    }
14
15    abstract class ProcessElement {
16      property process : Process { derived readonly transient volatile !resolve } {
17        derivation: Process.allInstances()
18            ->select(p | p.processElements->includes(self))
19            ->asSequence()->first();
20      }
21    }
22
23    class WorkDefinition extends ProcessElement {
24      property linksToPredecessors#successor : WorkSequence[*] { ordered };
25      property linksToSuccessors#predecessor : WorkSequence[*] { ordered };
26      property suivantes : WorkDefinition[*] { derived readonly transient volatile !resolve
          }
27      {
```

Même métamodèle dans avec une syntaxe textuelle (OCLinECore) II

```
28          derivation: self.linksToSuccessors->select(successor);
29        }
30        attribute name : String;
31        invariant previousWSinSameProcess:
32          self.process.processElements->includesAll(self.linksToPredecessors);
33        invariant nextWSinSameProcess:
34          self.process.processElements->includesAll(self.linksToSuccessors);
35      }
36
37      class WorkSequence extends ProcessElement {
38        attribute linkType : WorkSequenceType;
39        property predecessor#linksToSuccessors : WorkDefinition;
40        property successor#linksToPredecessors : WorkDefinition;
41        invariant previousWDinSameProcess: self.process = self.predecessor.process;
42        invariant nextWDinSameProcess: self.process = self.successor.process;
43      }
44
45      class Guidance extends ProcessElement {
46        property element : ProcessElement[*] { ordered };
47        attribute text : String[?];
48      }
49    }
```

Le métamodèle PetriNet

Exercice : Proposer un métamodèle en langage ECore pour représenter un réseau de Petri.

Exercice : Lister les contraintes OCL à définir pour garantir que les modèles conformes correspondent à un réseau de Petri valide.

Sommaire

Objectif général

Objectif : OCL est avant tout un **langage de requête** pour calculer une *expression sur un modèle en s'appuyant sur sa syntaxe* (son méta-modèle).
$\Longrightarrow$ Une expression exprimée une fois, pourra être évaluée sur tout modèle conforme au méta-modèle correspondant.
Exemple : pour une bibliothèque particulière on peut vouloir demander :

- Livres possédés par la bibliothèque ? Combien y en a-t-il ?
- Auteurs dont au moins un titre est possédé par la bibliothèque ?
- Titres dans la bibliothèque écrits par Martin Fowler ?
- Nombre de pages du plus petit ouvrage ?
- Nombre moyen de pages des ouvrages ?
- Ouvrages de plus 100 pages écrits par au moins trois auteurs ?
- ...

Programmation par contrat

Principe : Établir formellement les responsabilités d'une classe et de ses méthodes.

Moyen : définition de propriétés (expressions booléennes) appelées :

- **invariant :** propriété définie sur une **classe** qui doit toujours être vraie, de la création à la disparition d'un objet.
  Un invariant lie les requêtes d'une classe (état externe).
- **précondition :** propriété sur une **méthode** qui :
  - doit être vérifiée par l'appelant pour que l'appel à cette méthode soit possible ;
  - peut donc être supposée vraie dans le code de la méthode.

  postconditions : propriété sur une **méthode** qui définit l'effet de la méthode, c'est-à-dire :
  - spécification de ce que doit écrire le programmeur de la méthode ;
  - caractérisation du résultat que l'appelant obtiendra.

Exercice : Invariant pour une Fraction (état = numérateur et dénominateur) ?

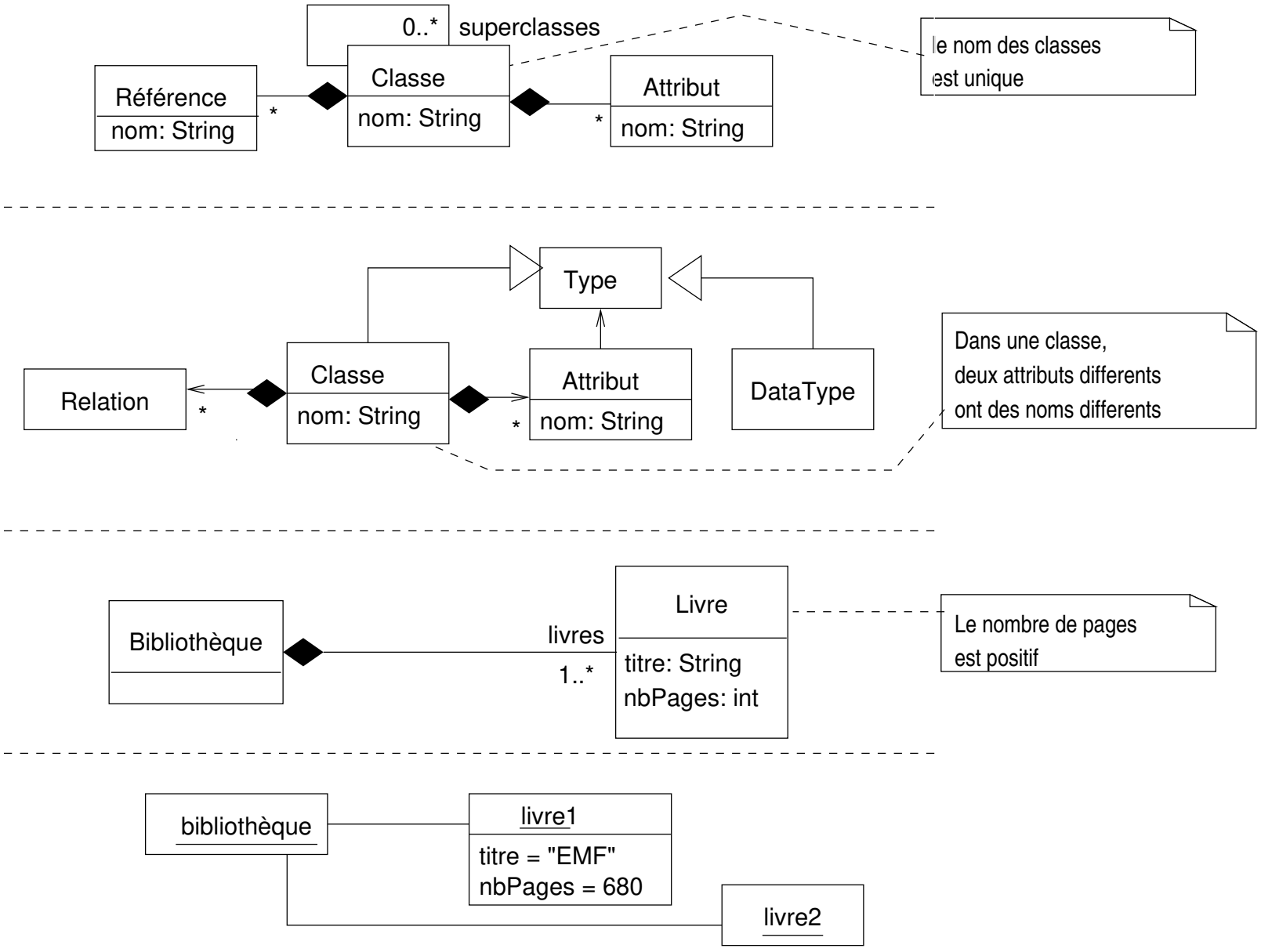
Exercice : Pré- et postconditions de racine carrée et de pgcd ?

OCL et Diagrammes d'UML

OCL peut être utilisé sur différents diagrammes d'UML :

- diagramme de classe :
  - définir des préconditions, postconditions et invariants :
    Stéréotypes prédéfinis : «precondition», «postcondition» et «invariant»
  - caractérisation d'un **attribut dérivé** (p.ex. le salaire est fonction de l'âge)
  - spécifier la **valeur initiale** d'un attribut (p.ex. l'attribut *salaire* d'un employé)
  - spécifier le **code d'une opération** (p.ex. le salaire annuel est 12 fois le salaire mensuel)
- diagramme d'état :
  - spécifier une garde sur une transition
  - exprimer une expression dans une activité (affectation, etc.)
  - ...
- diagramme de séquence :
  - spécifier une garde sur un envoi de message
- ...

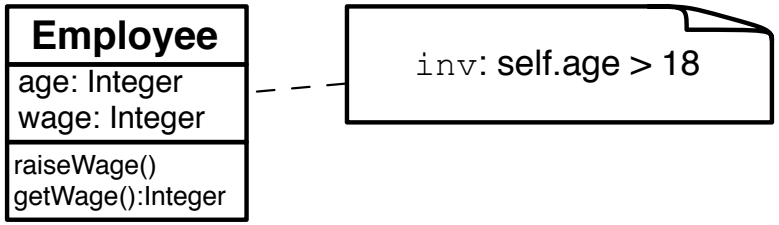
OCL et Méta-modélisation : préciser la sémantique statique d'un modèle



The *Object Constraint Language*

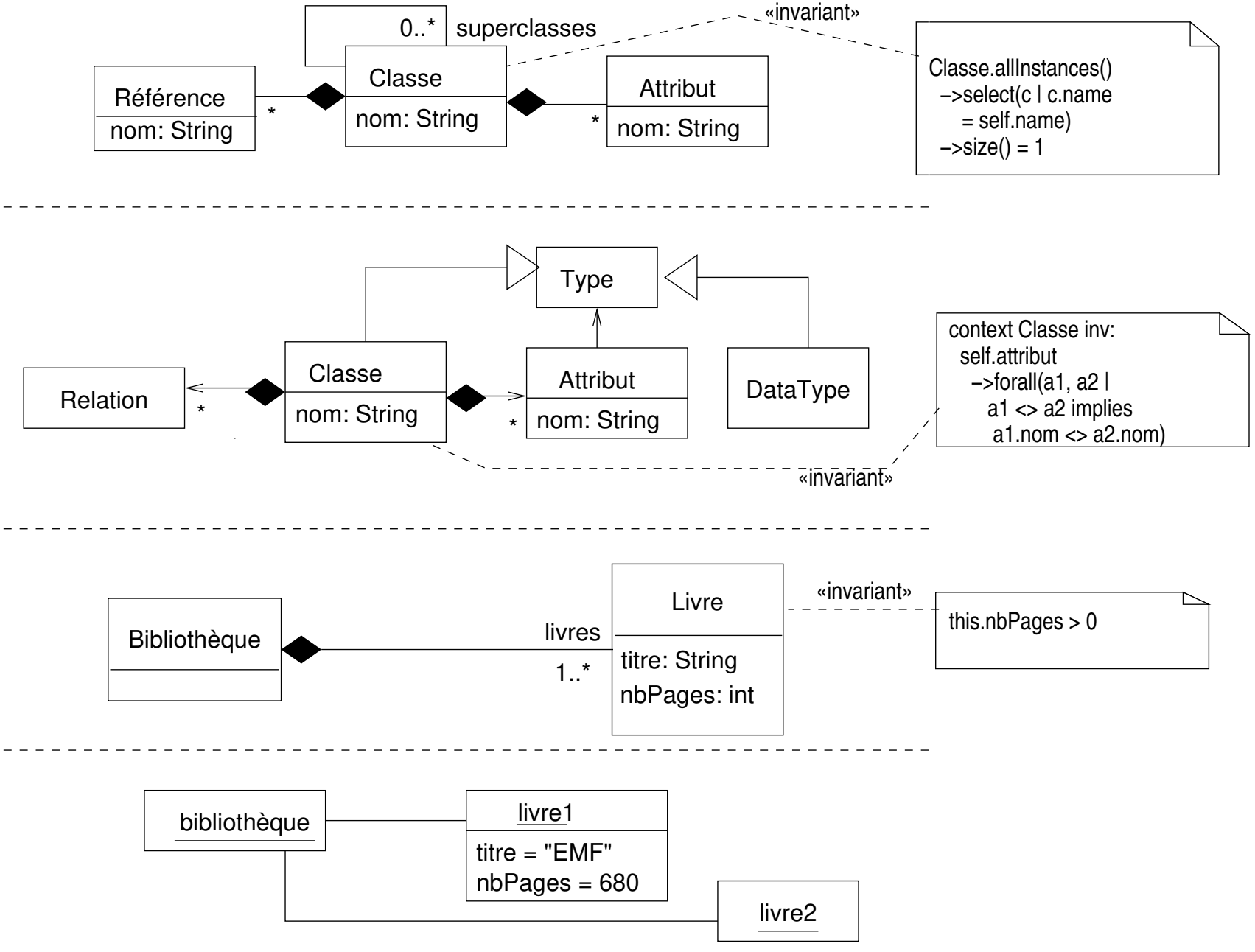
Objectifs initiaux

- Les langages formels traditionnels (e.g. Z) requièrent de la part des utilisateurs une bonne compréhension des fondements mathématiques.
- *Object Constraint Language (OCL)* a été développé dans le but d'être :
  - formel, précis et non ambigu,



  - utilisable par un large nombre d'utilisateurs,
  - un langage de spécification (et non de programmation !),
  - supporté par des outils.

Préciser la sémantique statique d'un modèle



The *Object Constraint Language*

Historique

- Développé en 1995 par IBM,
- Inclu dans le standard UML jusqu'à la version 1.1 (1997),
- OCL 2.0 *Final Adopted Specification* (ptc/06-05-01), May 2006.
- VErsion actuelle : OCL 2.3.1 (ptc/2009-05-03), January 2012.

The *Object Constraint Language*

Propriétés du langage

- **Langage de spécification sans effet de bord**
  - une expression OCL calcule une valeur... **et** laisse le modèle inchangé !
    - ⇒ l'état d'un objet ne peut pas être modifié **par** l'évaluation d'une expression OCL
  - l'évaluation d'une expression OCL est instantanée
    - ⇒ l'état des objets ne peut donc pas être modifié **pendant** l'évaluation d'une expression OCL
  - OCL n'est pas un langage de programmation !
- OCL est un **langage typé** :
  - Chaque expression OCL a un type
  - OCL définit des types primitifs : **Boolean**, **Integer**, **Real** et **String**
  - Chaque *Classifier* du modèle est un nouveau type OCL
  - *Intérêt :* vérifier la cohérence des expressions
    exemple : il est interdit de comparer un String et un Integer

Les types OCL de base

Les types de base (*Primitive*) sont **Integer**, **Real**, **Boolean** et **String**. Les opérateurs suivants s'appliquent sur ces types :

Opérateurs relationnels	$=, <>, >, <, >=, <=$
Opérateurs logiques	*and*, *or*, *xor*, *not*, *if* . . . *then* . . . *else* . . . *endif*
Opérateurs mathématiques	$+, -, /, *, min(), max()$. . .
Opérateurs pour les chaînes de caractères	*concat*, *toUpper*, *substring*...

Attention : Concernant l'opérateur **if** ... **then** ... **else** ... **endif** :

- la clause **else** est nécessaire et,
- les expressions du **then** et du **else** doivent être de même type.

Attention : and, **or**, **xor** ne sont pas évalués en court-circuit !

Priorité des opérateurs

Liste des opérateurs dans l'ordre de priorité décroissante :

```
1  @pre
2  .    ->                  -- notation pointée et fléchée
3   not  -                   -- opérateurs unaires
4  *    /
5  +    -                   -- opérateurs binaires
6   if- then- else- endif
7  <    >    <=    >=
8  =    <>
9   and   or    xor
10 implies                   -- implication
```

Remarque : Les parenthèses peuvent être utilisées pour changer la priorité.

Les autres types OCL

- Tous les éléments du modèle sont des types (*OclModelElementType*),
  - y compris les énumérations : *Gender* :: *male*,
- Type *Tuple* : enregistrement (produit cartésien de plusieurs types)
  Tuple {a : Collection(Integer) = Set{1, 3, 4}, b : String = 'foo'}
- *OclMessageType* :
  - utilisé pour accéder aux messages d'une opération ou d'un signal,
  - offre un rapport sur la possibilité d'envoyer/recevoir une opération/un signal.
- *VoidType* :
  - a seulement une instance *oclUndefined*,
  - est conforme à tous les types.

Contexte d'une expression OCL

Une expression est définie sur un **contexte** qui identifie :

- une **cible** : l'élément du modèle sur lequel porte l'expression OCL

T	Type (Classifier : Interface, Classe...)	**context** Employee
M	Opération/Méthode	**context** Employee::raiseWage(inc:Int)
A	Attribut ou extrémité d'association	**context** Employee::job : Job

- le **rôle** : indique la **signification** de cette expression (pré, post, invariant...) et donc contraint sa **cible** et son **évaluation**.

rôle	**cible**	**signification**	**évaluation**
inv	T	invariant	toujours vraie
pre	M	précondition	avant tout appel de M
post	M	postcondition	après tout appel de M
body	M	résultat d'une requête	appel de M
init	A	valeur initiale de A	création
derive	A	valeur de A	utilisation de A
def	T	définir une méthode ou un attribut	

Syntaxe d'OCL

inv (invariant) doit toujours être vrai (avant et après chaque appel de méthode)

context Employee
inv: self.age > 18

context e : Employee
inv age_18: e.age > 18

pre (precondition) doit être vraie avant l'exécution d'une opération

post (postcondition) doit être vraie après l'exécution d'une opération

context Employee::raiseWage(increment : **Integer**)
pre: increment > 0
post my_post: self.wage = self.wage**@pre** + increment

context Employee::getWage() : **Integer**
post: result = self.wage

Remarques : result et **@pre** : utilisables seulement dans une postcondition

- **exp@pre** correspond à la valeur de expr avant l'appel de l'opération.
- result est une variable prédéfinie qui désigne le résultat de l'opération.

Syntaxe d'OCL

- **body** spécifie le résultat d'une opération

  context Employee::getWage() : **Integer**
  body: self.wage

- **init** spécifie la valeur initiale d'un attribut ou d'une association

  context Employee::wage : **Integer**
  init: 900

- **derive** spécifie la règle de dérivation d'un attribut ou d'une association

  context Employee::wage : **Integer**
  derive: self.age $*$ 50

- **def** définition d'opérations (ou variables) qui pourront être (ré)utilisées dans des expressions OCL.

  context Employee
  def: annualIncome : **Integer** = 12 $*$ wage

La navigation dans le modèle

Accès aux informations de la classe

- Une expression OCL est définie dans le contexte d'une classe
  - en fait : un type, une interface, une classe, etc.
- Elle s'applique sur un objet, instance de cette classe :

  $\Longrightarrow$ cet objet est désigné par le mot-clé **self**.
- Étant donné un accès à un objet (p.ex. self), une expression OCL peut :
  - accéder à la valeur des attributs :
    - self.nbPages
    - unLivre.nbPages
  - appeler toute requête définie sur l'objet :
    - self.getNbPages()
    - unLivre.getNbPages()

Rappel : Une requête (notée {isQuery} en UML) est une opération :

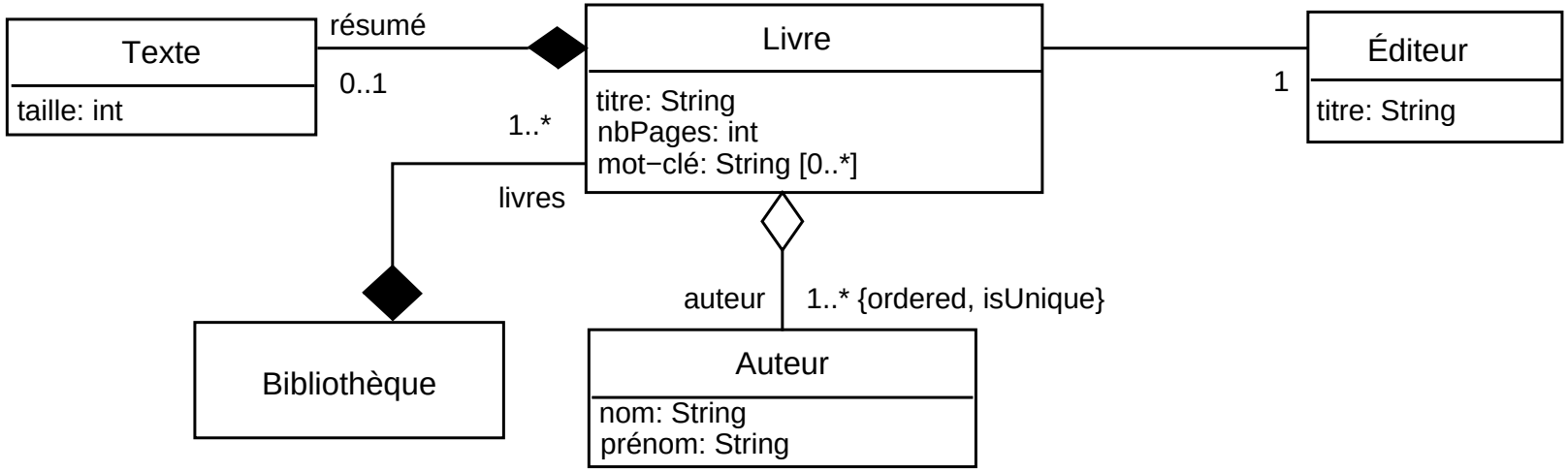
- *qui a un type de retour (calcule une expression) ;*
- *et n'a pas d'effet de bord (ne modifie pas l'état du système).*

Attention : une opération avec effet de bord est proscrite en OCL !

  - parcourir les associations...

Correspondance entre association et OCL

- Exemple de diagramme de classe



- pour atteindre l'autre extrémité d'une association, on utilise :
  - le rôle, p.ex. : unLivre.résumé
  - à défaut le nom de la classe en minuscule : unLivre.éditeur
- La manière dont une association est vue en OCL dépend :
  - de sa multiplicité : un exactement (1), optionnel (0..1), ≥ 2
  - de ses qualificatifs : { **isUnique**}, { **isOrdered**}

Correspondance entre association et OCL

association avec multiplicité ≤ 1

- multiplicité 1 : nécessairement un objet à l'extrémité (invariant implicite)
  - unLivre.éditeur
- multiplicité 0..1 (optionnel) :
  - utiliser l'opération **oclIsUndefined**()
  - unLivre.résumé. **oclIsUndefined**() est :
    - vraie si pas de résumé,
    - faux sinon
  - Exemple d'utilisation :

```
if unLivre.résumé.oclIsUndefined() then
    true
else
    unLivre.résumé.taille >= 60
endif
```

Correspondance entre association et OCL

association avec multiplicité ≥ 2

- les éléments à l'extrémité d'une association sont accessibles par une collection
- OCL définit quatre types de collection :
  - **Set** : pas de double, pas d'ordre
  - **Bag** : doubles possibles, pas d'ordre
  - **OrderedSet** : pas de double, ordre
  - **Sequence** : doubles possibles, ordre
- Lien entre associations UML et collections OCL

UML	Ecore	OCL
		Bag
isUnique	Unique	Set
isOrdered	Ordered	Sequence
isUnique, isOrdered	Unique, Ordered	OrderedSet

- Exemple : unLivre.auteur : la collection des auteurs de unLivre

Les collections OCL

- *Set* : ensemble d'éléments *sans* doublon et *sans* ordre

  Set {7, 54, 22, 98, 9, 54, 20..25}
  —— *Set{7,54,22,98,9,20,21,23,24,25} : Set(Integer)*
  —— *ou Set{7,9,20,21,22,23,24,25,54,98} : Set(Integer), ou...*

- *OrderedSet* : ensemble d'éléments *sans* doublon et *avec* ordre

  OrderedSet {7, 54, 22, 98, 9, 54, 20..25}
  —— *OrderedSet{7,9,20,21,22,23,24,25,54,98} : OrderedSet(Integer)*

- *Bag* : ensemble d'éléments *avec* doublons possibles et *sans* ordre

  Bag {7, 54, 22, 98, 9, 54, 20..25}
  —— *p.ex. : Bag{7,9,20,21,22,22,23,24,25,54,54,98} : Bag(Integer)*

- *Sequence* : ensemble d'éléments *avec* doublons possibles et *avec* ordre

  Sequence{7, 54, 22, 98, 9, 54, 20..25}
  —— *Sequence{7,54,22,98,9,54,20,21,22,23,24,25} : Sequence(Integer)*

Les collections sont génériques : Bag(Integer), Set(String), Bag(Set(Livre))

Opérations sur les collections (bibliothèque standard)

Pour tous les types de Collection

size(): **Integer** —— *nombre d'éléments dans la collection self*
includes(object: T): **Boolean** —— *est−ce que object est dans sefl ?*
excludes(object: T): **Boolean** —— *est−ce que object n'est pas dans self ?*
count(object: T): **Integer** —— *nombre d'occurrences de object dans self*
includesAll(c2: **Collection**(T)): **Boolean**
—— *est−ce que self contient tous les éléments de c2 ?*
excludesAll(c2: **Collection**(T)): **Boolean**
—— *est−ce que self ne contient aucun des éléments de c2 ?*
isEmpty(): **Boolean** —— *est−ce que self est vide ?*
notEmpty(): **Boolean** —— *est−ce que self est non vide ?*
sum(): T —— *la somme (+) des éléments de self*
—— *l'opérateur + doit être défini sur le type des éléments de self*
product(c2: **Collection**(T2)): **Set**(**Tuple**(premier: T, second: T2))
—— *le produit (∗) des éléments de self*

Opérations de la bibliothèque standard pour les collections

En fonction du sous-type de *Collection*, d'autres opérations sont disponibles :

- union
- intersection
- append
- flatten
- =
- ...

Une liste exhaustive des opérations de la bibliothèque standard pour les collections est disponible dans [OMG OCL 2.3.1, §11.7].

Opérations de la bibliothèque standard pour tous les objets

OCL définit des opérations qui peuvent être appliquées à tous les objets

- *oclIsTypeOf*(*t* : *OclType*) : *Boolean*
  Le résultat est vrai si le type de *self* et *t* sont identiques.

  context Employee
  inv: self. **oclIsTypeOf**(Employee) −− *is true*
  inv: self. **oclIsTypeOf**(Company) −− *is false*

- *oclIsKindOf*(*t* : *OclType*) : *Boolean*
  vrai si *t* est le type de *self* ou un super-type de de *self*.
- *oclIsNew*() : *Boolean*
  Uniquement dans les post-conditions
  vrai si le récepteur a été créé au cours de l'exécution de l'opération.
- *oclIsInState*(*t* : *OclState*) : *Boolean*
  Le résultat est vrai si l'objet est dans l'état *t*.

Opérations de la bibliothèque standard pour tous les objets

OCL définit des opérations qui peuvent être appliquées à tous les objets

- $oclAsType(t : OclType) : T$
  Retourne le même objet mais du type t
  Nécessite que $oclIsKindOf(t) = true$
- $allInstances()$
  - prédéfinie pour les classes, les interfaces et les énumérations,
  - le résultat est la collection de toutes les instances du type au moment de l'évaluation.

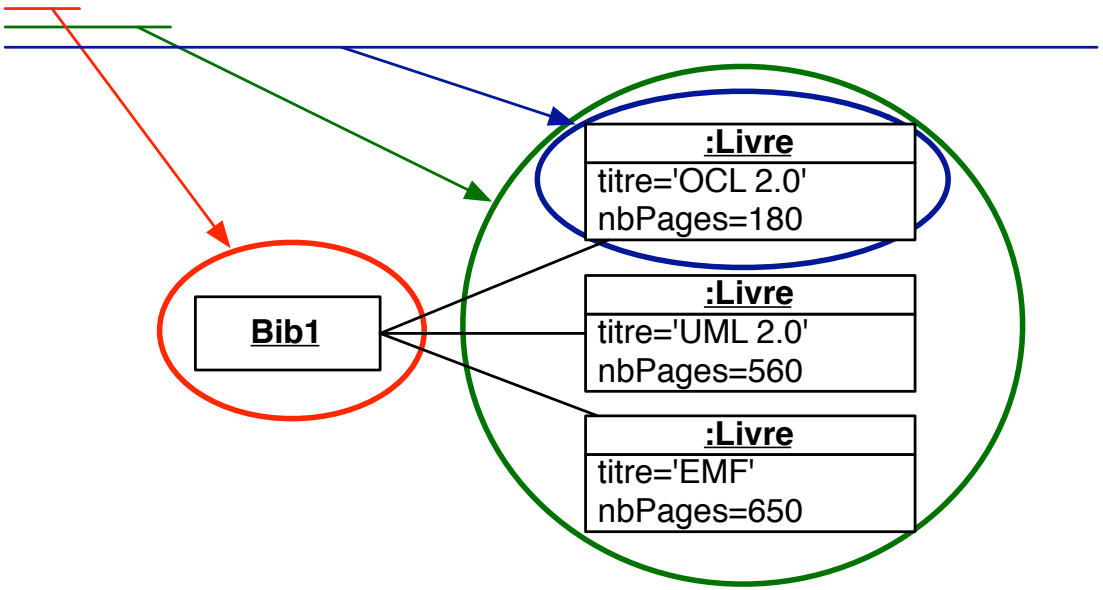
```
context Employee
    inv: Employee. allInstances()−> forAll(p1, p2
        | p1 <> p2 implies p1.name <> p2.name)
```

Opérateur *select* (resp. *reject*)

Permet de spécifier le sous-ensemble de tous les éléments de *collection* pour lesquels l'expression est vraie (resp. fausse pour *reject*).

- $collection \rightarrow select(elem : T|expr)$
- $collection \rightarrow select(elem|expr)$
- $collection \rightarrow select(expr)$

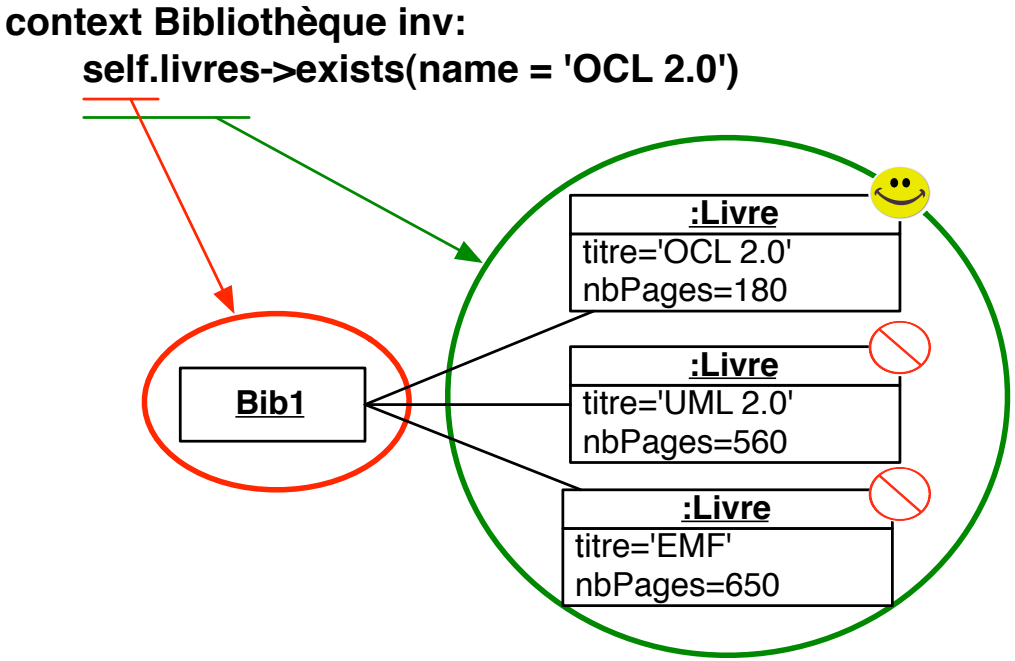
context Bibliothèque inv:
self.livres->select(name = 'OCL 2.0')->notEmpty()



Opérateur *exists*

Retourne vrai si l'expression est vraie pour au moins un élément de la collection.

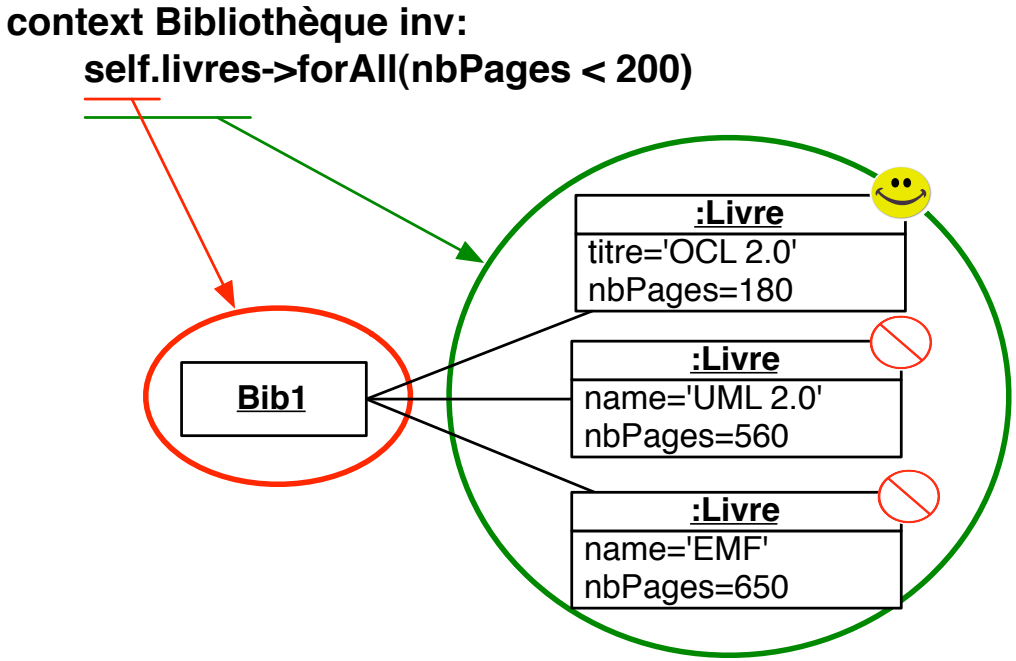
- $collection \rightarrow exists(elem : T|expr)$
- $collection \rightarrow exists(elem|expr)$
- $collection \rightarrow exists(expr)$



Opérateur *forAll*

Retourne vrai si l'expression est vraie pour tous les éléments de la collection.

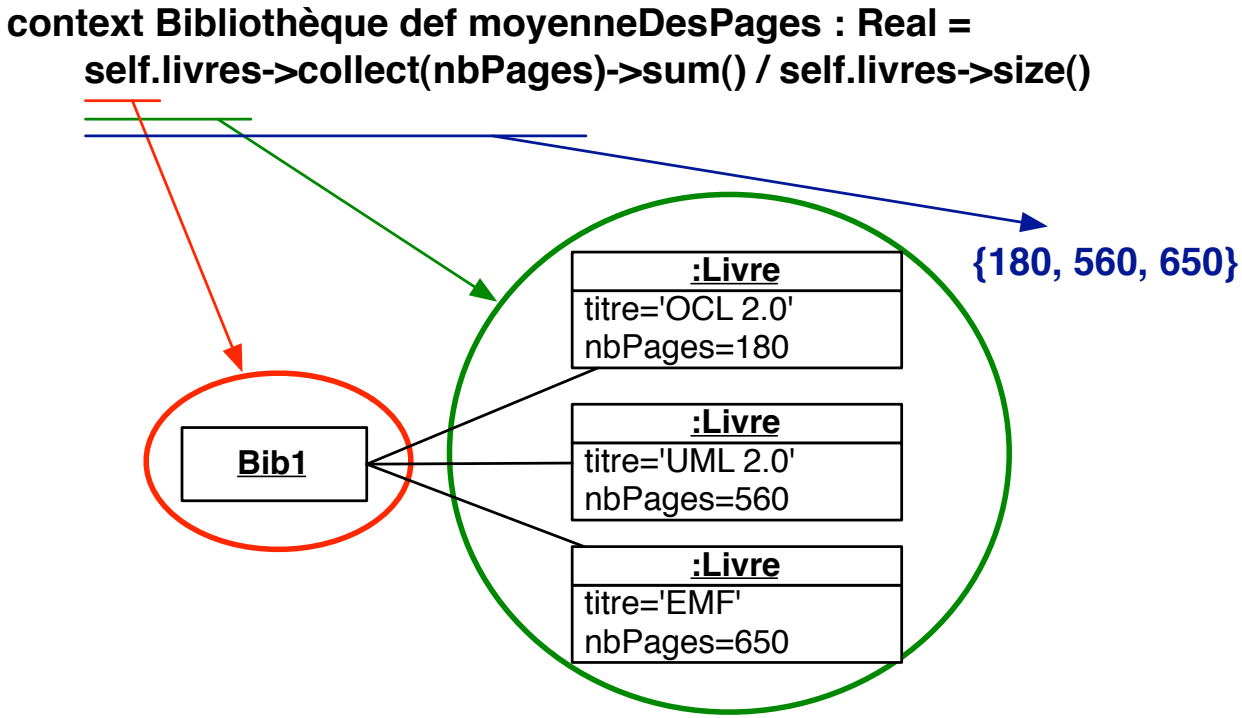
- $collection \rightarrow forAll(elem : T|expr)$
- $collection \rightarrow forAll(elem|expr)$
- $collection \rightarrow forAll(expr)$



Opérateur *collect*

Retourne la collection des valeurs (*Bag*) résultant de l'évaluation de l'expression appliquée à tous les éléments de collection.

- $collection \rightarrow collect(elem : T|expr)$
- $collection \rightarrow collect(elem|expr)$
- $collection \rightarrow collect(expr)$



Opérateur *iterate*

Forme générale d'une itération sur une collection et permet de redéfinir les précédents opérateurs.

```
collection−> iterate(elem : Type;
        answer : Type = <value>
        | <expression_with_elem_and_answer>)
```

```
context Bibliothèque def moyenneDesPages : Real =
    self.livres−> collect(nbPages)−> sum() / self.livres−> size()

−− est identique à :

context Bibliothèque def moyenneDesPages : Real =
    self.livres−> iterate(l : Livre;
            lesPages : Bag{Integer} = Bag{}
            | lesPages−> including(l.nbPages))
        −> sum() / self.livres−> size()
```

Plusieurs itérateurs pour un même opérateur

Remarque : les opérateurs *forAll*, *exist* et *iterate* acceptent plusieurs itérateurs :

```
Auteur. allInstances()−> forAll(a1, a2 |
        a1 <> a2 implies
            a1.nom <> a2.nom  or a1.prénom <> a2.prénom)
```

Bien sûr, dans ce cas il faut nommer tous les itérateurs !

Conseils

- **OCL ne remplace pas les explications en langage naturel.**
  - Les deux sont *complémentaires* !
  - *Comprendre* (informel)
  - *Lever les ambiguités* (OCL)
- **Éviter les expressions OCL trop compliquées**
  - éviter les navigations complexes (utiliser **let** ou **def**)
  - bien choisir le contexte (associer l'invariant au bon type !)
  - éviter d'utiliser **allInstances**() :
    - rend souvent les invariants plus complexes
    - souvent difficile d'obtenir toutes les instances dans un système (sauf BD !)
  - décomposer une conjonction de containtes en plusieurs (inv, post, pre)
  - Toujours nommer les extrémités des associations (rôle des objets)