Correction TD2

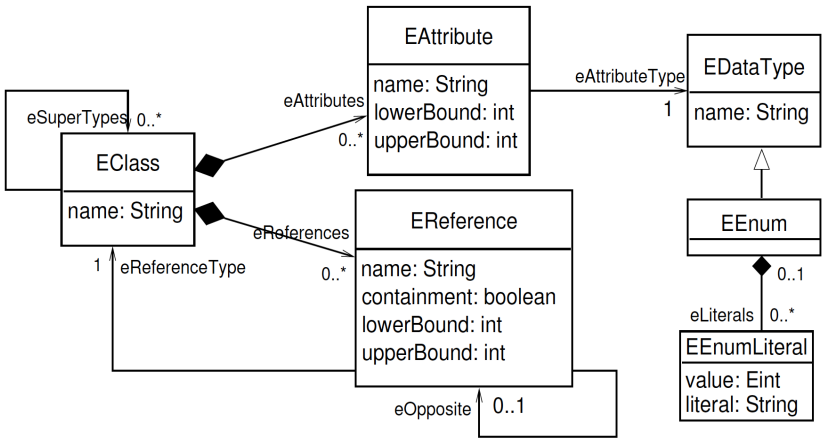
Exo 1 :

Exo 1.1 : diagramme d'objet qui vérifie la conformité entre M2/M3

Concepts Ecore : Pour commencer voici un rappel de la signification des principaux concepts Ecore présents dans la figure Figure 1 du sujet et exposée ci-dessous :

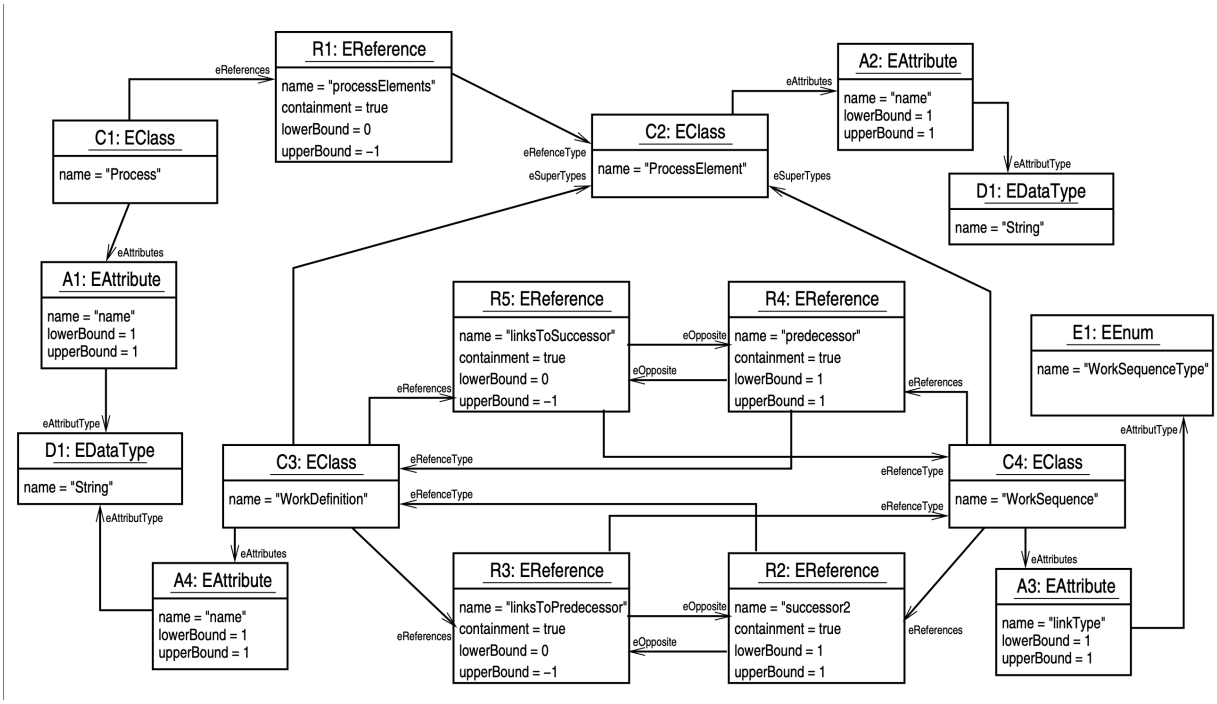
— Une EClass comme une classe.
— Un EAttribute comme un attribut de la classe qui le contient ;
— les EReference comme des relations orientées de l'EClass qui contient la référence vers la EClass référencée par eReferenceType,
— la référence eSuperTypes d'une EClasse comme une relation d'héritage,
— l'attribut containment d'une EReference à vrai comme une composition, etc.

FIGURE 1 – Version très simplifiée du méta-métamodèle Ecore (M3 [OMG])



==> Justifier la conformité par approche tabulaire entre M2/M3 : Ici, deux solutions sont possibles :

Solution 1 : On donne le diagramme d'objet qui montre que les éléments M2 sont des instances des éléments du diagramme de classe M3. Voir le diagramme ci-dessous :



Solution 2 : On donne la vision tabulaire. Ici, pour chaque concept (classe) au niveau (Mi+1) : je dessine une table/tableau où

EClass /* c'est un des concept au niveau Mi+1 (ici M3) */				
ID	**Name /* les noms d'instances au niveau Mi */**	eSuperType	eAttributes	eReferences
C1	Process		A1	R1
C2	ProcessElement		~~A4~~	
C3	WorkDefinition	C2	A2	R2, R4
C4	WorkSequence	C2	A3	R3, R5

EAttribute				
ID	name	lowerBound	upperBound	eAttributeType
A1	name	1	1	D1
A2	name	1	1	D1
A3	linkType	1	1	E1
~~A4~~	~~name~~			

EDataType	
ID	name

D1	EString

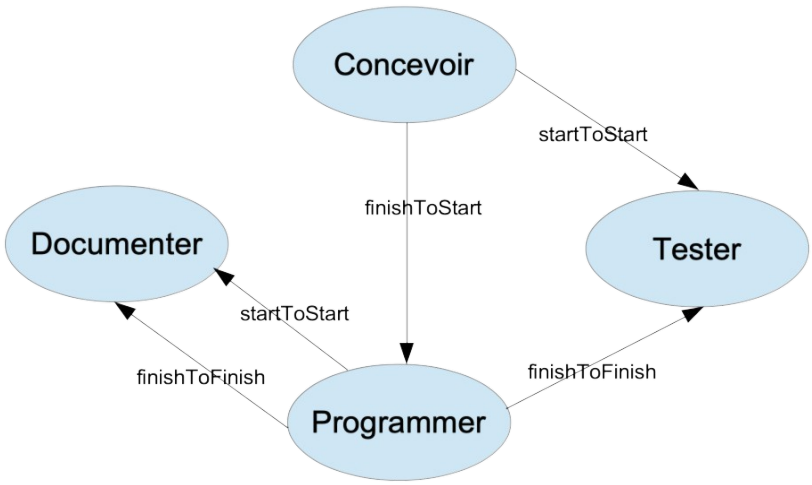
EEnum		
ID	name	eLiterals
E1	workSequenceType	L1, L2, L3, L4

EEnumLiteral	
ID	name
L1	startToStart
L2	finishToStart
L3	startToFinish
L4	finishToFinish

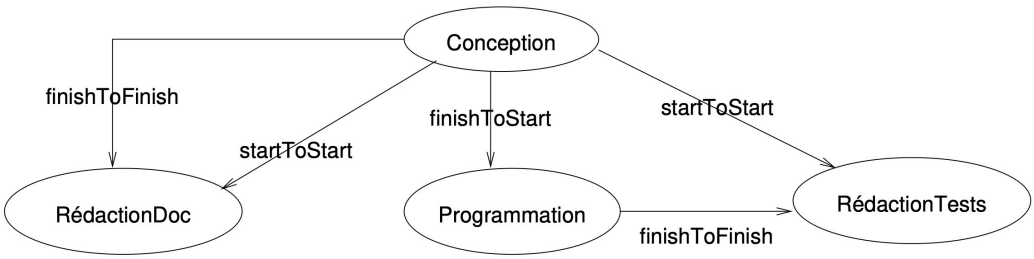
EReference					
ID	name	lowerBound	upperBound	eReferenceType	eOpposite
R1	processElements	0	* ou noté (- 1)	C2	
R2	successor	1	1	C3	R3
R3	linksToPredecessor	0	-1	C4	R2
R4	predecessor	1	1	C3	R5
R5	linksToSuccessor	0	-1	C4	R4

Exo1.2 :
Sémantique ? Sens des concepts ? On nomme les concepts (WorkDefinition, WorkSequence, etc.) mais au niveau méta modèle on n'explique pas grande chose ! Par exemple les contraintes ne sont pas capturées à ce niveau ==> nécessité de formaliser

Exo 1.3 :
Exo1.3.1 : Modèle de «processus », niveau M1 [OMG]



Exemple pas correcte par rapport aux contraintes décrites dans le sujet !



Modèle de «processus » KO, niveau M1 [OMG]

Exo 1.3.2 :
Conformité M1/M2. Il faudrait donner le diagramme d'objet (peu lisible), si non, donner les tableaux.

Exo 1.4 : OCL contraintes statiques

```
context ProcessElement
        def: process(): Process =
                Process.allInstances()
                ->select(p | p.processElements ->includes(self))
                ->asSequence()->first()
```

Objectif : retourner le Process d'un ProcessElement (workDefinition ou WorkSquence) ! On definit donc la référence opposite de ProcessElements !

Il s'agit d'une méthode process() définie sur l'élément ProcessElement :

- Cette méthode process() n'a pas des paramètres en entrée, et elle retourne par contre un objet de type Process.
- L'instruction Process.allInstances() permet d'exécuter la méthode allInstances() pour retourner tous les objets de type sur lequel on a fait l'appel, ici Process.
- « -> » est un sélecteur appelé sur la « collection » retournée par Process.allInstances()
- self est comme this en java, ici self fait référence à processElement

- select est un opérateur qui permet de conserver la collection cible que les éléments qui respectent la propriété qui est ici (p | p.processElements -> includes(self)) . Autrement dit : cette propriété décrit la collection des processElement p de tous les processElements de ce même Process sur lequel la méthode process() est appelée.

- Enfin, le retour de la méthode process() est déduit de asSequence() qui traduit la collection précédente en liste pour appliquer le méthode first() afin de retourner le Process en résultat

---

```
context WorkSequence
        inv previousWDinSameProcess: self.predecessor.process() = self.process()
        inv nextWDinSameProcess: self.successor.process() = self.process()
```

Ici on se focalise au niveau WorkSequence. Deux invariants sont mis en place afin d'exiger que l'activité précédente et aussi l'activité suivantes doivent appartenir au même processus.

Question : à quel niveau M on rajouter les contraintes OCL ?
Réponse : Les contraintes OCL peuvent être utiles pour définir des invariants sur une modèle M1 ou un métamodèle M2 pour capturer les contraintes qui n'ont pas pu l'être au niveau du métamodèle ou du méta-métamodèle.

Exo 1.5 :

1- une dépendance qui ne peut pas être réflexive :

On se focalise sur WorkSequence et on vérifie le prédécesseur vs le successeur

```
context WorkSequence
        inv notReflexive: self.predecessor <> self.successor;
```

Grace à cette contrainte OCL, je ne peux pas accepter l'exemple conception startToStart conception ==> KO l'outil ne l'accepte pas

2- deux sous-activités différentes d'un même processus ne peuvent pas avoir le même nom.

Ici, il y a deux possibilités. on se focalise sur un niveau plus générique (context Process) ou bien un autre plus précis (WorkDefinition) ?

```
context WorkDefinition
        inv uniqNames: self.process.processElements
        ->select(pe | pe.oclIsKindOf(WorkDefinition))
        ->collect(pe | pe.oclAsType(WorkDefinition))
        ->forAll(w | self = w or self.name <> w.name);
```

3- le nom d'une activité doit être composé au moins d'un caractère

```
context WorkDefinition
        inv nameIsDefined: -- tester si le nom est déjà définie si non il doit etre different d'une
```

chaine vide

```
if self.name.oclIsUndefined() then false
else
self.name <> ''
endif
```