1ère année Master GLSD

Département d'Informatique

Université de Biskra

Année 2020/2021

Ingénierie des Modèles
Transformations de modèles

Dr. Mohamed Lamine Kerdoudi

Email: l.kerdoudi@univ-biskra.dz

Transformation

Qu'est-ce une transformation?

Une transformation prend en entrée des modèles (**sources**) et fournit en sortie des modèles (**cibles**)

- **Transformation endogène**
  - ➢ Dans le même espace technologique
  - ➢ Les modèles source et cible sont conformes au même méta-modèle
    - ❑ Transformation d'un modèle UML en un autre modèle UML
- **Transformation exogène**
  - ➢ Entre 2 espaces technologiques différents
  - ➢ Les modèles source et cible sont conformes à des méta-modèles différents
    - ❑ Transformation d'un modèle UML en programme Java
    - ❑ Transformation d'un fichier XML en schéma de BDD

Transformation

Dans l'outil de transformation, il y a une **Définition de Transformation.**

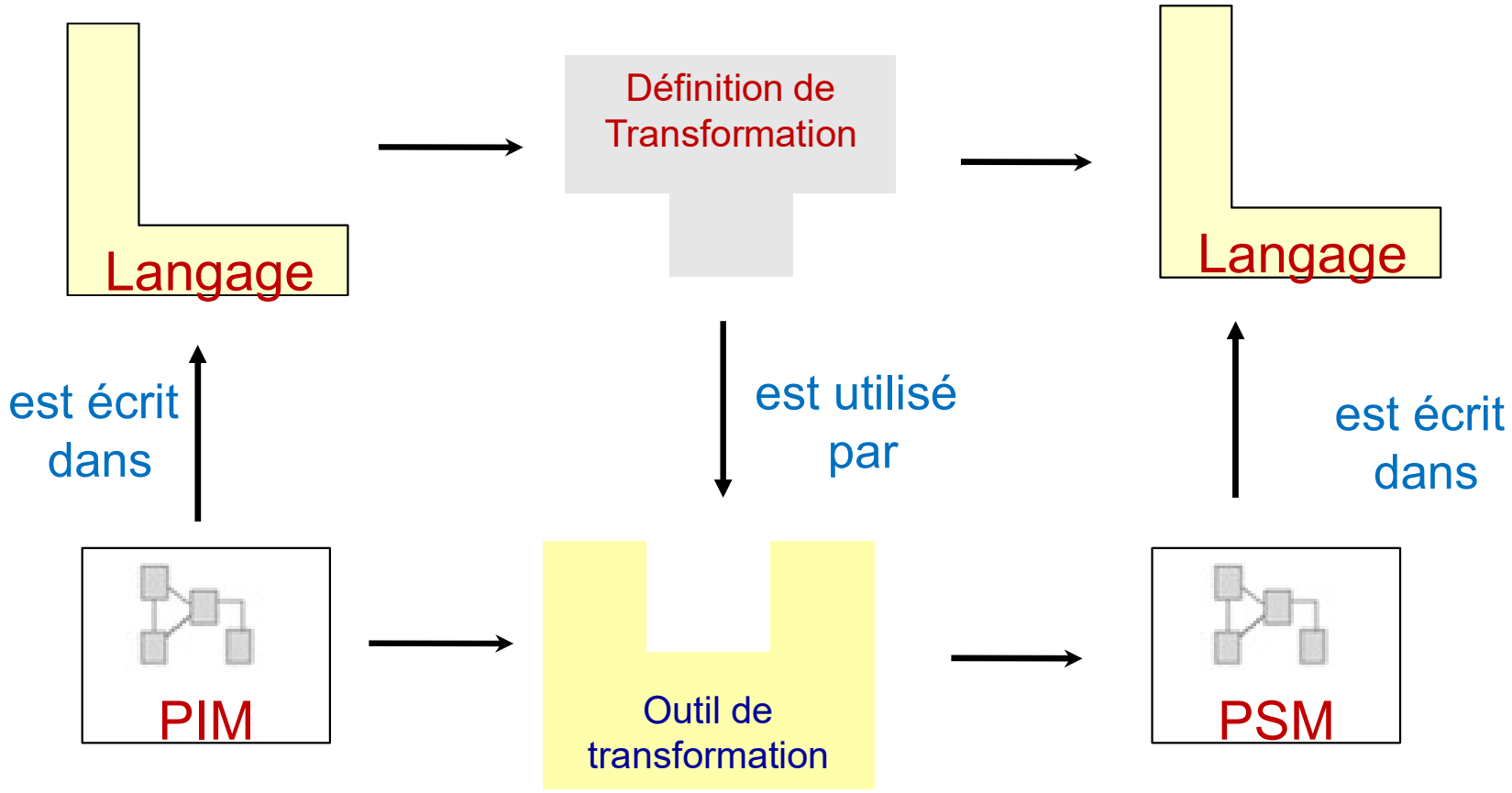
Qu'est-ce qu'une définition de la transformation ?

➔ Elle décrit comment le modèle devrait être transformé.

➔ Est un ensemble de règles de transformation.

❑ Une règle de transformation est une **description** de la manière dont une ou plusieurs **constructions** dans un **langage source** peuvent être transformés en une ou plusieurs **constructions** dans le **langage cible**.

Le Framework MDA de base



- Un développeur **focalise sur le développement d'un PIM**, décrivant le système de logiciel à un haut niveau d'abstraction.
- Il choisit un ou plusieurs **outils de transformation**.
- Les **PSMs résultats** peuvent être **transformé en code**

Autre vision des transformations

❑ Les processus de développement automatisé et basé sur les **transformations** sont utilisés depuis longtemps.

➔ Rien de totalement nouveau

- **Adaptation** à un nouveau contexte

➔ **Exemple** : compilation d'un programme C

- **Programme C** : modèle abstrait
- **Transformation de ce programme** sous une autre forme mais en restant à un niveau **abstrait**
  - ➢ **Modélisation**, représentation différente du programme C pour le manipuler : transformation en un modèle équivalent
  - ➢ **Exemple** : **arbres décorés** : arbre syntaxique dont les nœuds peuvent être décorés (informations complémentaires)
- **Génération du code** en langage machine
  - ➢ Avec optimisation pour une architecture de processeur donnée

Outils pour réaliser des transformations

- Outils de mise en œuvre
  - Exécution de transformations de modèles
    - Nécessité d'un **langage de transformation**
    - Qui pourra être défini via un **méta-modèle** de transformation
  - Les **modèles** doivent être manipulés, créés et enregistrés
    - Via un **repository** (dépôt, référentiel)
    - Doit pouvoir représenter la **structure des modèles**
      - Via des méta-modèles qui devront aussi être manipulés via les outils
      - On les stockera également dans un repository
- Il existe de nombreux outils ou qui sont en cours de développement (industriels et académiques)
  - Notamment plusieurs langages de transformation

Transformations : types d'outils

- Langage de programmation « standard »
  - **Ex : Java**
  - Pas forcément adapté pour tout
  - Sauf si interfaces spécifiques
    - Ex : JMI (Java Metadata Interface) ou framework Eclipse/EMF
- Langage dédié d'un atelier de génie logiciel
  - **Ex : J dans Objecteering**
  - Souvent propriétaire et inutilisable en dehors de l'AGL
- Langage lié à un domaine/espace technologique
  - **Ex: XSLT** dans le domaine XML, AWK pour fichiers texte ...
- Langage/outil dédié à la transformation de modèles
  - **Ex : standard QVT de l'OMG, ATL**
- Atelier de méta-modélisation avec langage d'action
  - **Ex : Kermeta**

Transformations : types d'outils

- 3 grandes familles de modèles et outils associés

1) Données sous forme de séquence

   - **Ex : fichiers textes** (AWK: le langage script)

2) Données sous forme d'arbre

   - **Ex : XML** (XSLT)

3) Données sous forme de graphe

   - **Ex : diagrammes** UML
   - **Outils**
     - Transformateurs de graphes déjà existants
     - Nouveaux outils du MDE et des Atelier de Génie Logiciel (**QVT, J, ATL, Kermeta** ...)

Techniques de transformations

3 grandes catégories de techniques de transformation

- **Approche déclarative**
  - **Recherche** de certains **patrons** (d'éléments et de leurs relations) dans le **modèle source**
  - Chaque patron trouvé est **remplacé** dans le **modèle cible** par une **nouvelle structure** d'élément
  - Ecriture de la transformation « assez » simple mais ne permet pas toujours d'exprimer toutes les transformations facilement
- **Approche impérative**
  - Proche des langages de programmation usuels
  - On **parcourt** le modèle source dans un certain ordre et on **génère** le modèle cible lors de ce parcours
  - Ecriture transformation peut être plus lourde
- **Approche hybride** : à la fois déclarative et impérative
  - La plupart des approches déclaratives offrent de l'impératif en complément car plus adapté dans certains cas

Transformations en EMF

- Plusieurs langages pour le framework EMF/Ecore
  - Directement via le framework **Java** EMF
    - Chaque élément du méta-modèle Ecore correspond à une classe Java avec ses interfaces d'accès aux éléments de modèles
    - **Pb** : reste de la progammation en Java, pas totalement adapté à la manipulation de modèles
- Langages de plus haut niveau,
  - **ATL** : approche déclarative
    - Une **règle de transformation** sélectionne un élément dans le source et génère un (ou plusieurs) élément(s) dans le cible
    - La sélection se fait par des **query OCL**
  - **Kermeta** : approche impérative
    - Langage de programmation **orienté-objet**
    - **Primitives dédiées** pour la manipulation de modèles
    - **Ajout** d'opérations/attributs sur méta-éléments par **aspect**
    - **Intègre** aussi **un langage de contraintes similaire à OCL**
      - **Invariant** sur méta-élément et pré/post sur leurs opérations

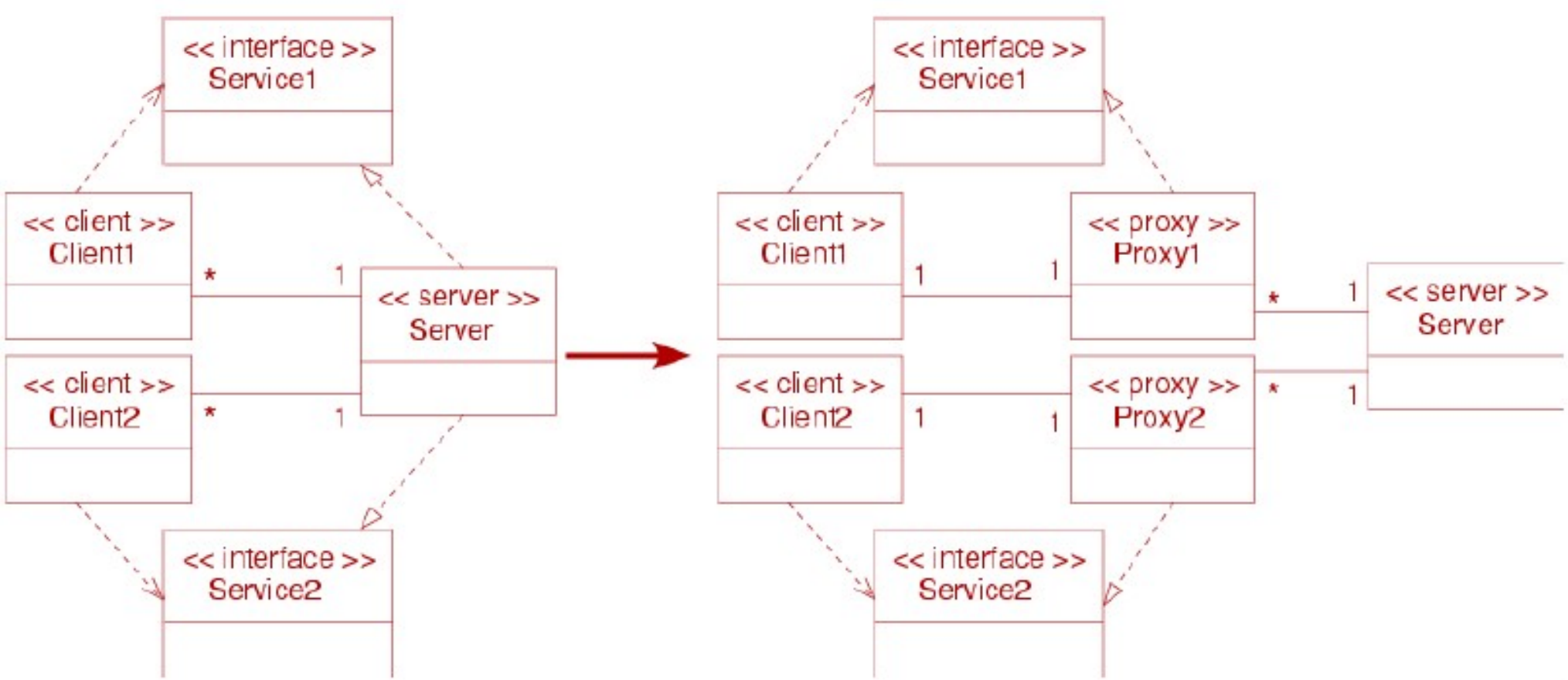
Transformation :
exemple de l'ajout de proxy

Ajout de proxy

- Modélisation très simplifiée d'architecture logicielle
  - **Modèles source** : **clients et serveurs**
    - Un type de client est connecté à un serveur
    - Chaque type de client requière une interface qui est implémentée par le serveur
  - **Modèles cible** : **clients, serveurs et proxy**
    - Entre un type de client et le serveur, on rajoute un proxy dédié
    - Le client est maintenant connecté à son proxy
    - Le proxy implémente l'interface de services requise par le client
  - **Méta-modèles source et cible**
    - Proches mais différents : **transformation exogène**
    - Contraintes OCL dont une qui exprime qu'il n'y a **qu'un seul serveur** par modèle

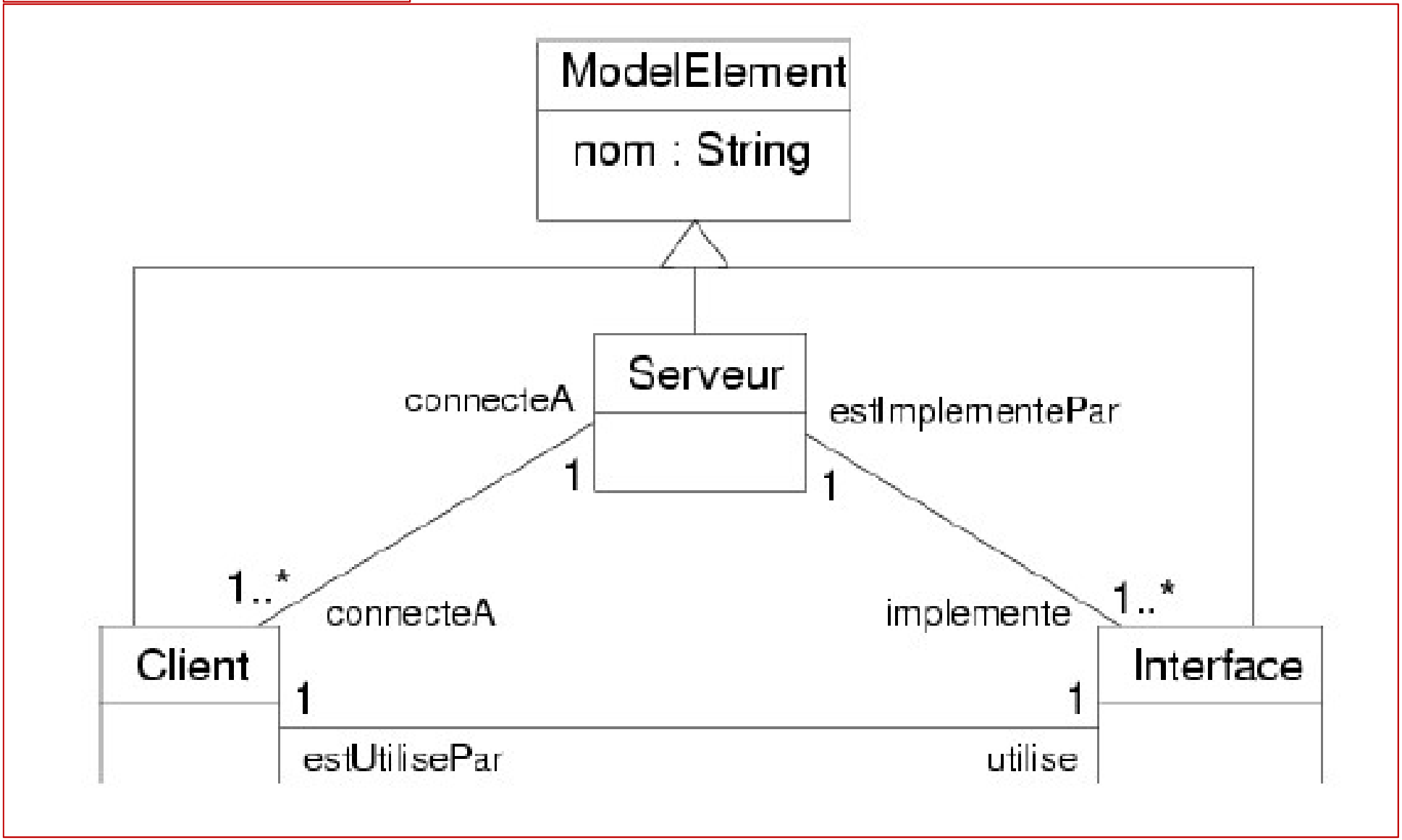
Ajout de proxy : exemple

- Exemple de transformation
  - En utilisant un diagramme de classes UML avec stéréotypes



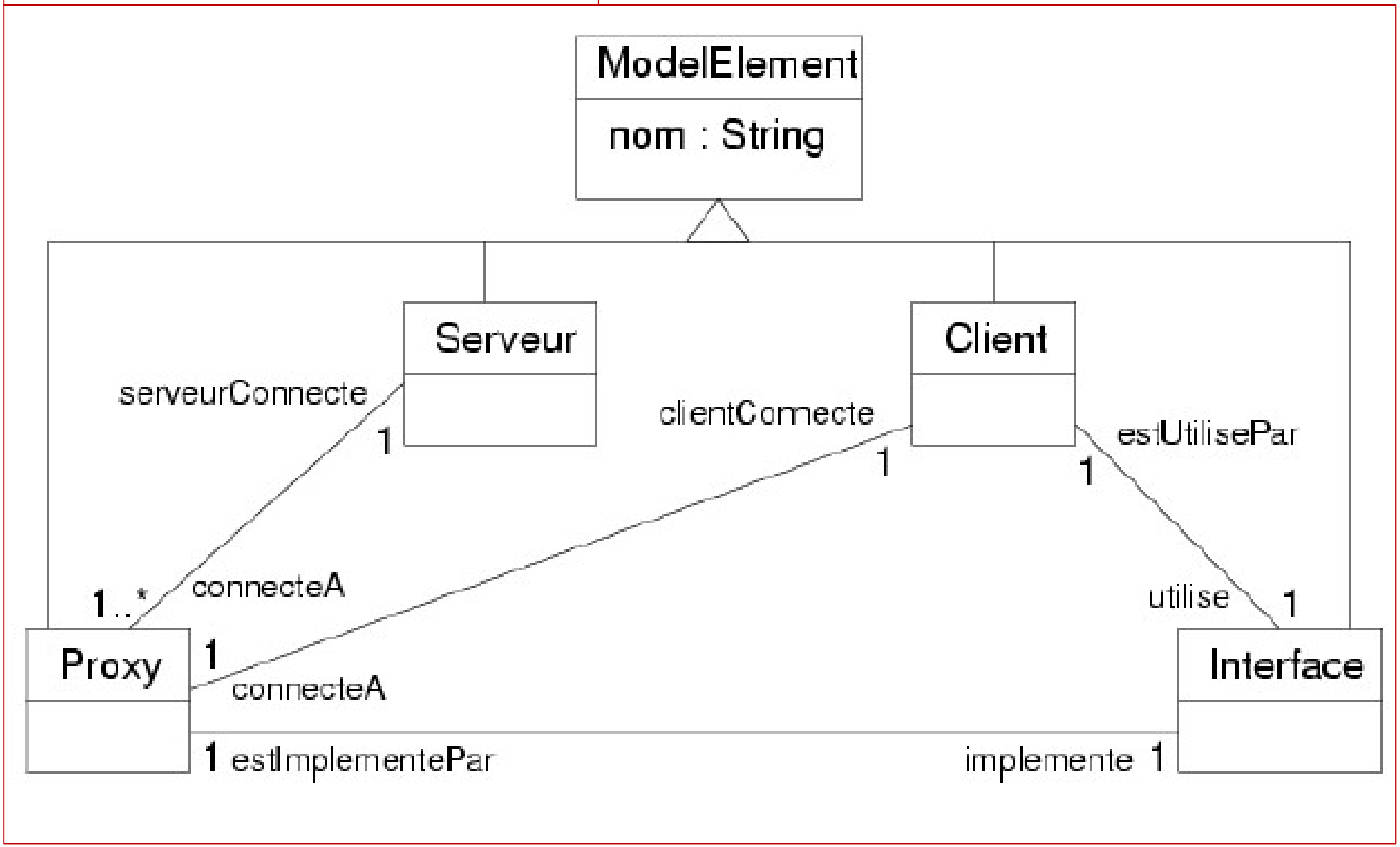
Ajout de proxy : MM source

ClientServeur



Ajout de proxy : MM cible

ClientProxyServeur



Ajout de proxy : transfo Kermeta

```
  operation transformerModele(serveur : ClientServeur::Serveur) :
ClientProxyServeur::Serveur is do
        // on cree un serveur cible avec le même nom
        var serveurCible : ClientProxyServeur::Serveur init
    ClientProxyServeur::Serveur.new
        serveurCible.nom := serveur.nom
        // pour chaque client auquel le serveur est connecté, on crèe
        // une interface, un client et un proxy cible avec les bonnes associations
        serveur.connecteA.each { client |
                        var clientCible : ClientProxyServeur::Client init
    ClientProxyServeur::Client.new
                        var interfaceCible : ClientProxyServeur::Interface init
    ClientProxyServeur::Interface.new
                         var proxy : ClientProxyServeur::Proxy init
    ClientProxyServeur::Proxy.new
                         clientCible.nom := client.nom
                        interfaceCible.nom := client.utilise.nom
                          proxy.nom := client.nom + serveur.nom
                          clientCible.connecteA := proxy
                          clientCible.utilise := interfaceCible
                          interfaceCible.estImplementeePar := proxy
                          interfaceCible.estUtiliseePar := clientCible
                          proxy.implemente := interfaceCible
                          proxy.clientConnecte := clientCible
                          proxy.serveurConnecte := serveurCible
              serveurCible.connecteA.add(proxy)
              }
        result := serveurCible
     end
```

Ajout de proxy : transfo ATL

- **Duplication du serveur**
  - On lui associe tous les proxys qui existent coté cible

```
rule LeServeur {
from
      serveurSource : ClientServeur!Serveur
to
      serveurCible : ClientProxyServeur!Serveur (
      nom <- serveurSource.nom,
      connecteA <- ClientProxyServeur!Proxy.allInstances()
      )
}
```

- **Duplication des interfaces**
  - On associe une interface au **proxy** qui a été associé à son client

```
rule Interfaces {
from
      interSource : ClientServeur!Interface
to
      interCible : ClientProxyServeur!Interface (
                nom <- interSource.nom,
                estUtiliseePar <- interSource.estUtiliseePar,
                estImplementeePar <- (ClientProxyServeur!Proxy.allInstances() ->
any (p |
                          p.implemente = interCible)) )
}
```

Ajout de proxy : transfo ATL

- **Duplication des clients**
  - On rajoute en plus un proxy associé à chaque client
  - Nom du proxy : concaténation des noms du client et du serveur

```
rule Clients {
from
   clientSource : ClientServeur!Client
To
        clientCible : ClientProxyServeur!Client (
           nom <- clientSource.nom,
           utilise <- clientSource.utilise,
           connecteA <- proxy
    ),
    proxy : ClientProxyServeur!Proxy (
           nom <- clientSource.nom +
           ClientServeur!Serveur.allInstances().first().nom,
           clientConnecte <- clientCible,
           serveurConnecte <-
ClientProxyServeur!Serveur.allInstances().first(),
           implemente <- clientCible.utilise
        )
}
```

Ajout de proxy : comparaison

- **ATL**
  - **Purement déclaratif**
  - A un type d'élément source, on associe un (ou deux) type(s) d'élément cible
    - **Ne se préoccupe pas de savoir comment les éléments sont retrouvés dans le modèle**
    - Accède à des éléments crées dans d'autres règles sans de préoccuper de l'ordre de leurs créations
- **Kermeta**
  - **Approche impérative**
  - **Doit explicitement naviguer sur les modèles**
    - **sources** : pour récupérer les éléments à dupliquer avec modifications requises
    - **cibles**: pour en créer les éléments un par un

Questions ?