Utiliser Kermeta pour transformer des modèles

Génie Logiciel & Ingénierie Des Modèles

Sébastien Mosser
mosser@polytech.unice.fr

Université de Nice Sophia – Antipolis
Équipe Rainbow, Laboratoire I3S, CNRS
http://rainbow.i3s.unice.fr/~mosser

Année universitaire 2008 – 2009
Sciences Informatique 5ème année
EPU Polytech'Nice – Sophia Antipolis

Agenda

1. Étude de cas : *Network* ↔ *Graph*
2. Introduction aux transformations de modèle
3. Kermeta : Un Kernel de Métamodélisation
4. Tissages d'aspects & Contraintes dans Kermeta
5. Rendre un modèle transformable ...
6. Implémentation de transformations : *Graph* → *X*
7. Conclusions
8. Exemple : Opérationalisation dans Eclipse

Agenda

1 Étude de cas : *Network* ↔ *Graph*

2 Introduction aux transformations de modèle

3 Kermeta : Un Kernel de Métamodélisation

4 Tissages d'aspects & Contraintes dans Kermeta

5 Rendre un modèle transformable . . .

6 Implémentation de transformations : *Graph* → *X*

7 Conclusions

8 Exemple : Opérationalisation dans Eclipse

Des réseaux aux graphes

Un métamodèle de graphe

- Un graphe est un ensemble de noeuds et d'arcs
- Un arc relie deux noeuds

$\Rightarrow$ Modélisation "horizontale"

Un métamodèle de réseau

- Un réseau contient des éléments (ordinateurs, routeurs)
- Un routeur définit et contient un sous–réseau

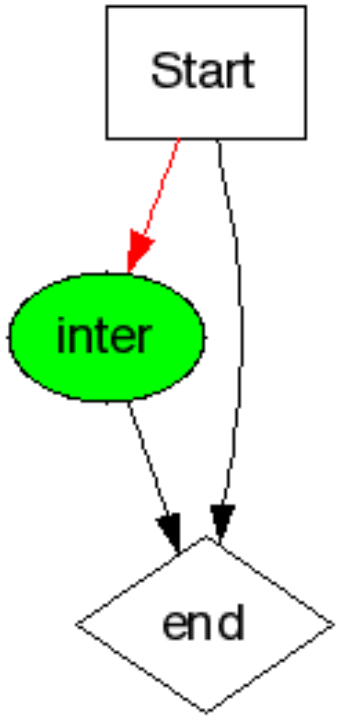
$\Rightarrow$ Modélisation "hiérarchique"

- Idée : Valider un réseau en validant des propriétés de graphe
  - Acircuicité du réseau construit, validation des liens, ...

Langage de description de graphe : GraphViz

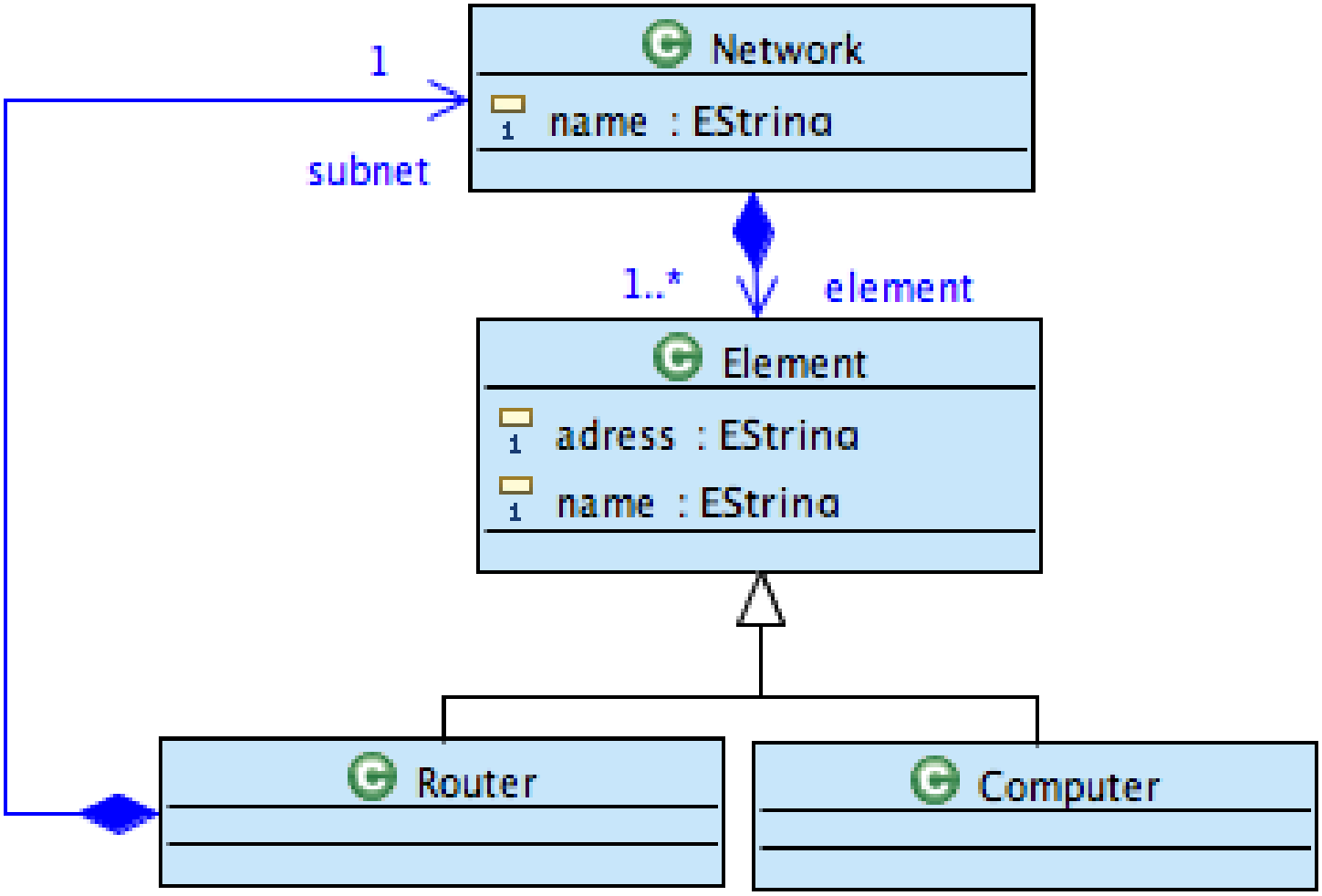
- Domain Specific Language
  - Description de noeuds et d'arcs
- Commande Unix : dot
  - dot -Tpng -o g.png g.dot

```
digraph g {
   start [shape=box,label="Start"] ;
   inter [style=filled, fillcolor="#00ff005f"] ;
   end   [shape=diamond] ;

   start -> inter [color="red"] ;
   inter -> end ;
   start -> end ;
}
```

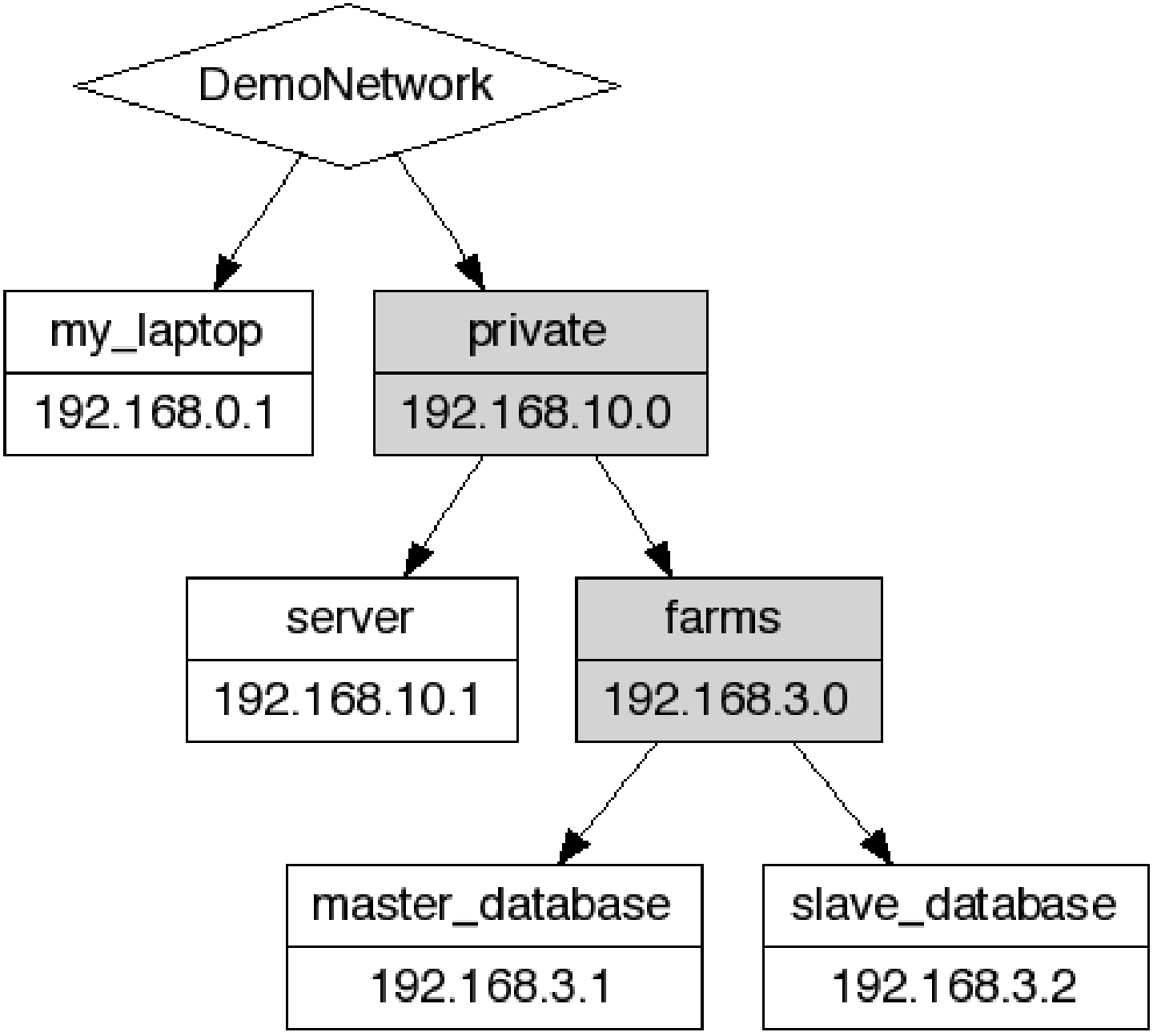


http://www.graphviz.org/Gallery.php

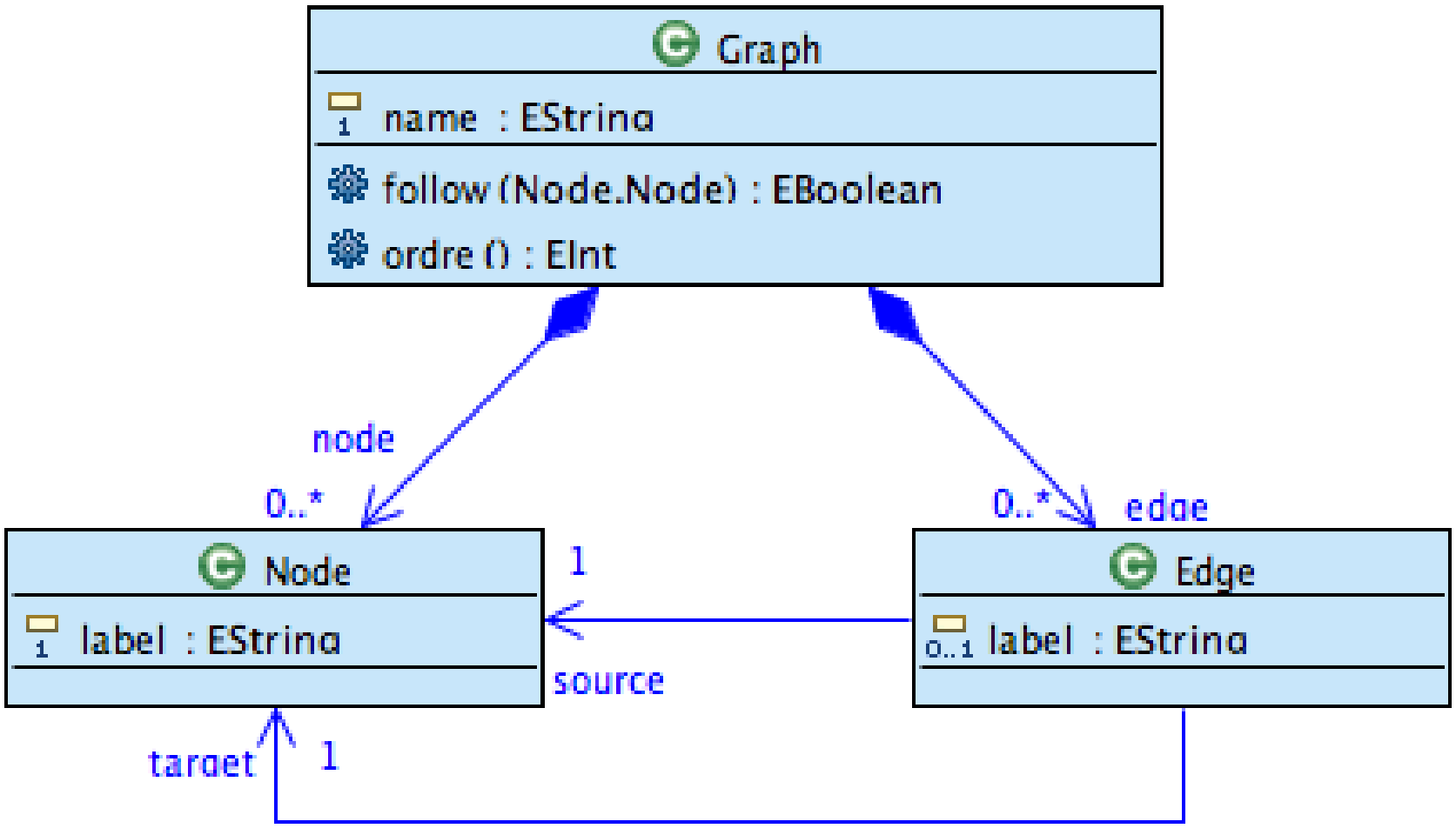
Métamodèle de réseaux [network.ecore]



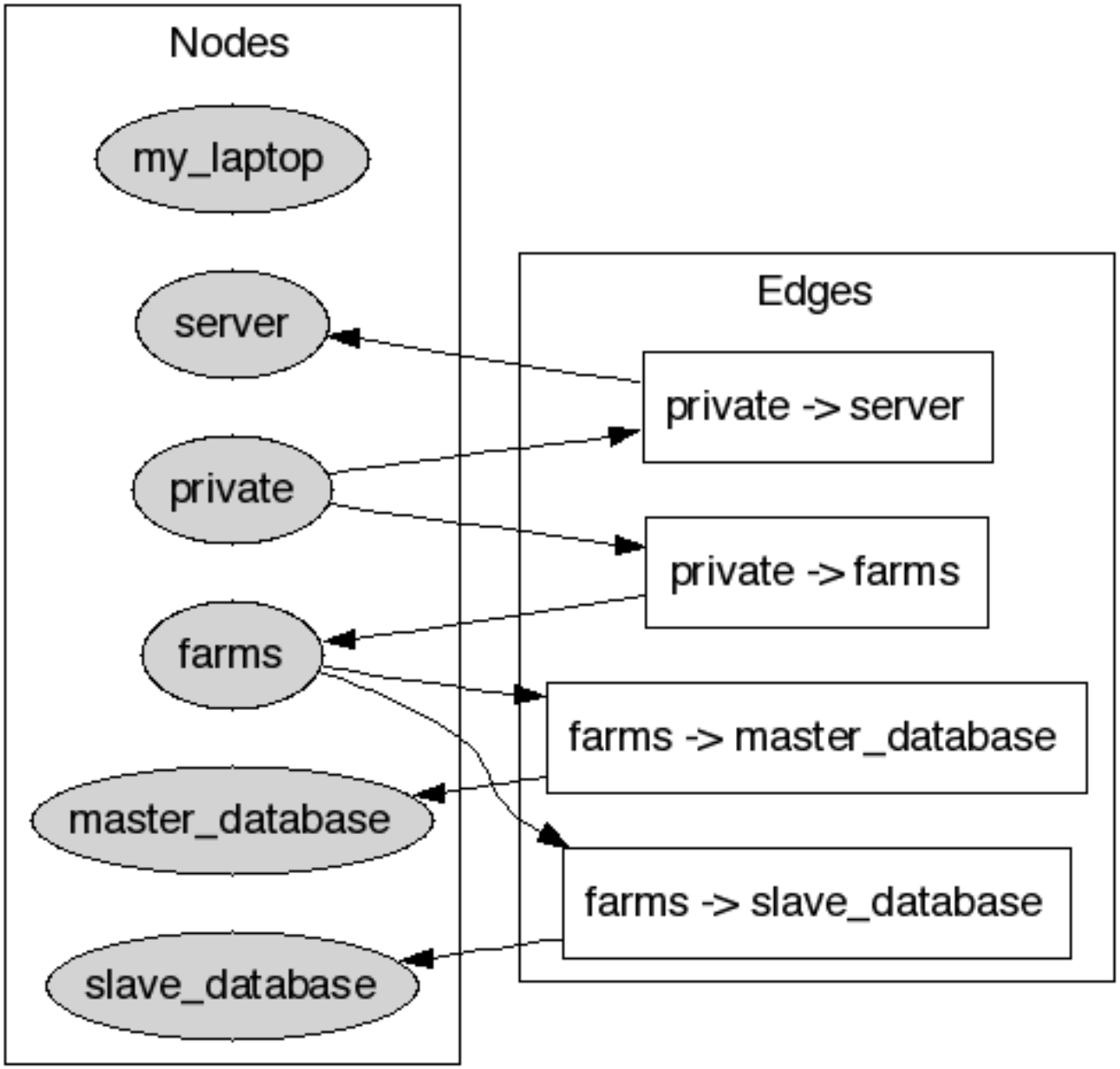
Visualisation GraphViz d'un réseau



Métamodèle de graphes [graph.ecore]



Visualisation GraphViz d'un graphe



Agenda

1 Étude de cas : *Network* ↔ *Graph*

2 Introduction aux transformations de modèle

3 Kermeta : Un Kernel de Métamodélisation

4 Tissages d'aspects & Contraintes dans Kermeta

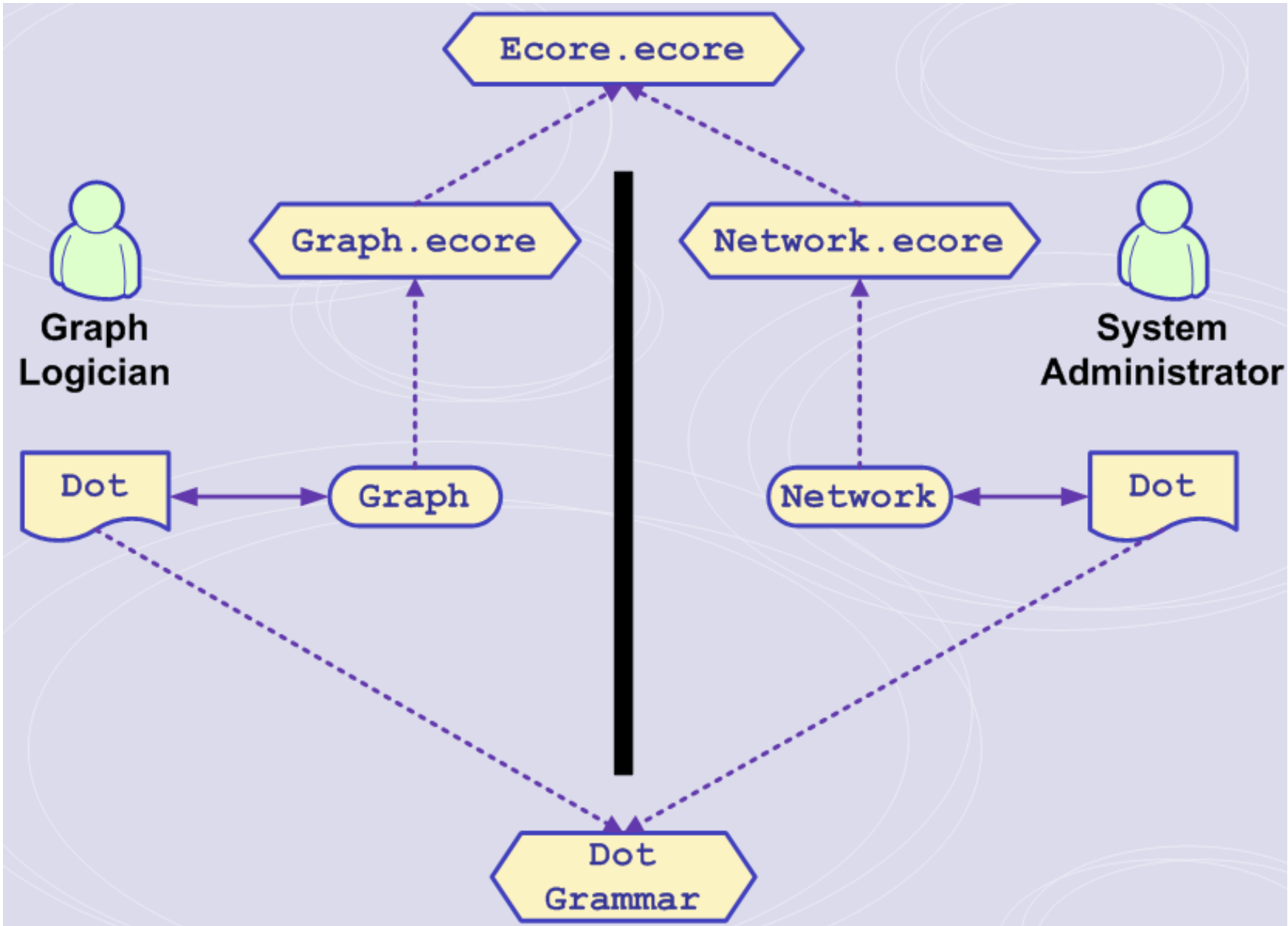
5 Rendre un modèle transformable . . .

6 Implémentation de transformations : *Graph* → *X*

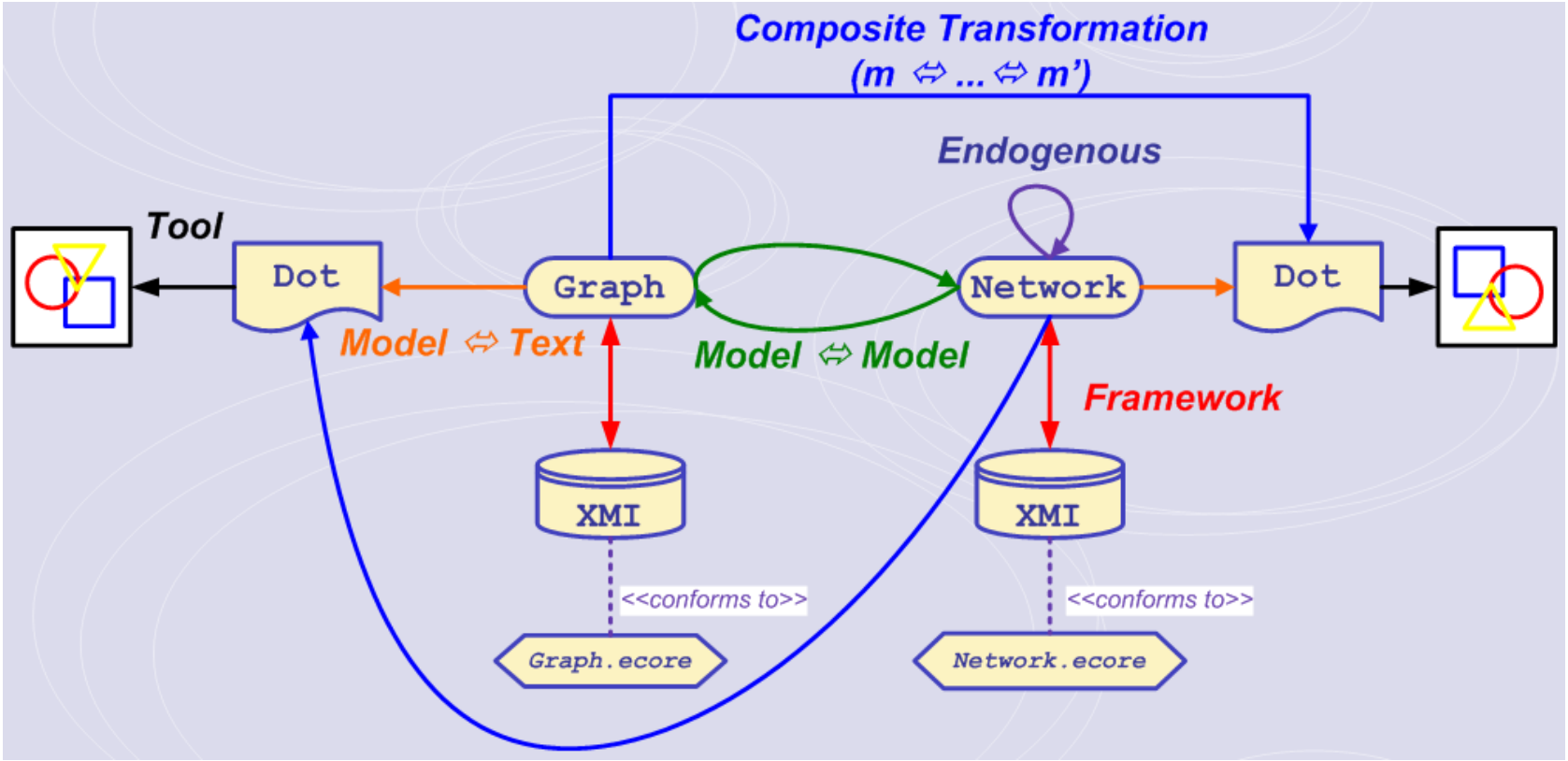
7 Conclusions

8 Exemple : Opérationalisation dans Eclipse

Graph ↔ *Network* : État des lieux

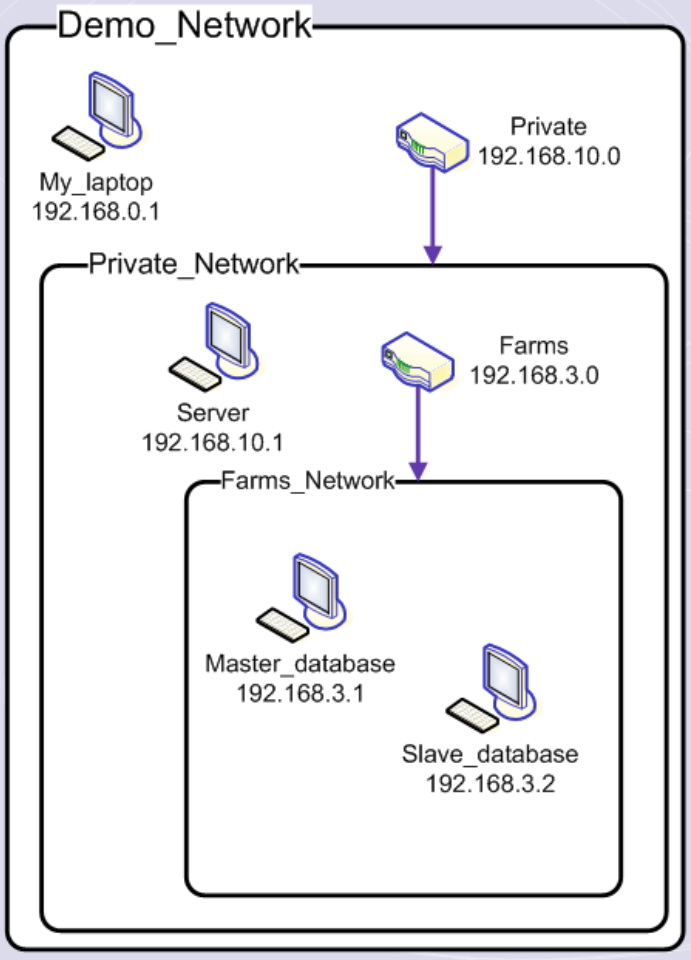


Différents types de transformations

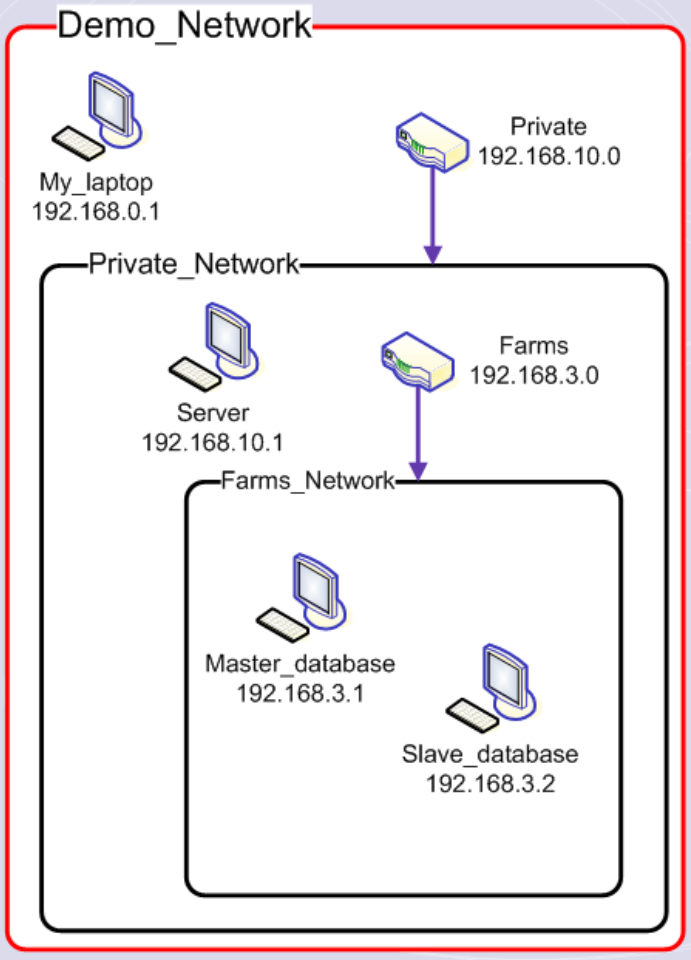


1. Outillage externe : `dot -Tpng -o g.png g.dot`
2. Canevas logiciel : Persistance XMI via EMF
3. *Model* $\longrightarrow$ *Text* : D'un réseau à un code `GraphViz`
4. *Model* $\longrightarrow$ *Model* : D'un réseau à un graphe
5. Endomorphisme : Calcul d'adressage sur un réseau
6. Composite : *graph* $\longrightarrow$ *network* $\longrightarrow$ *dot*

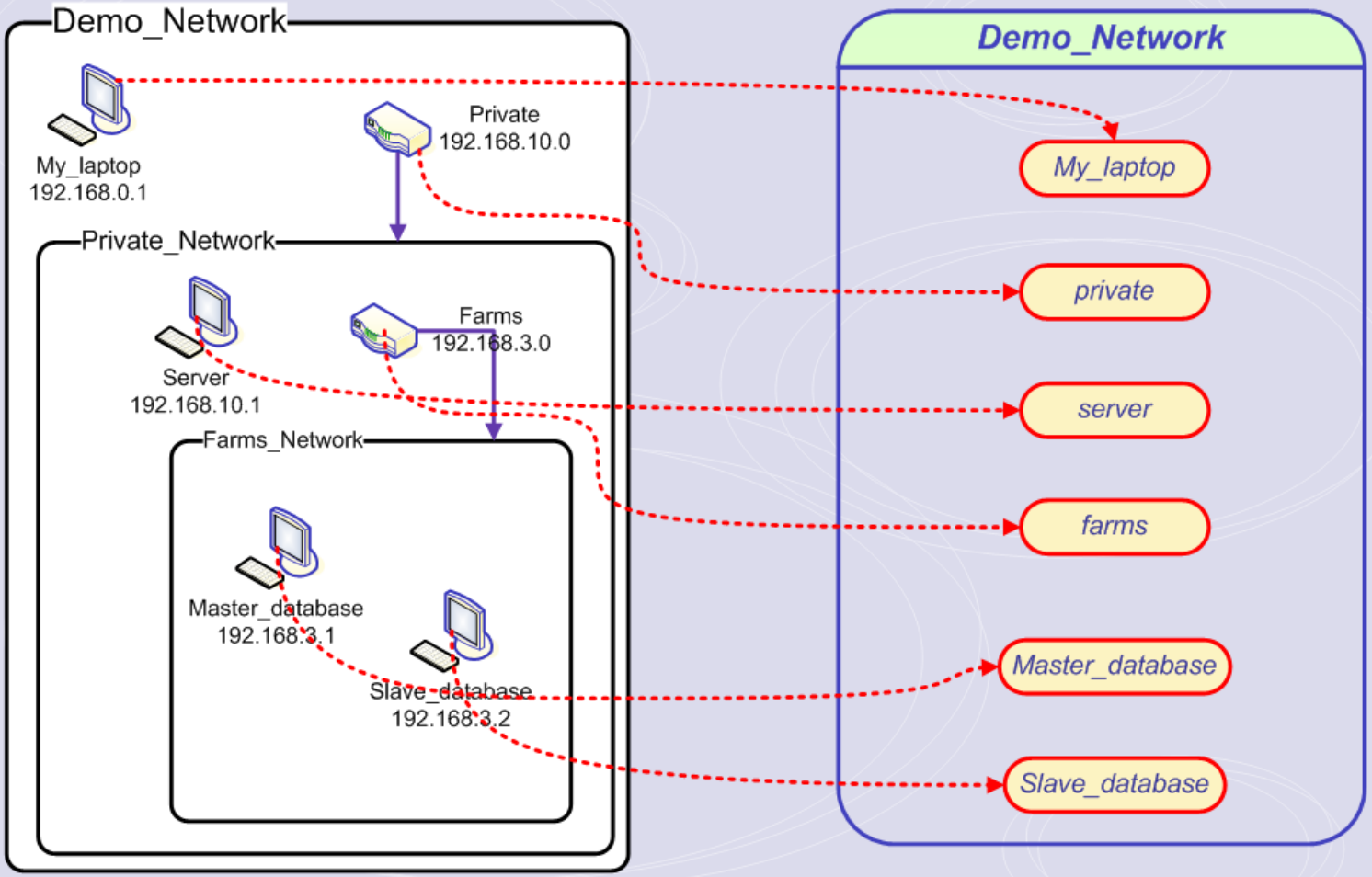
Modèle d'entrée : `demo_network.xmi`



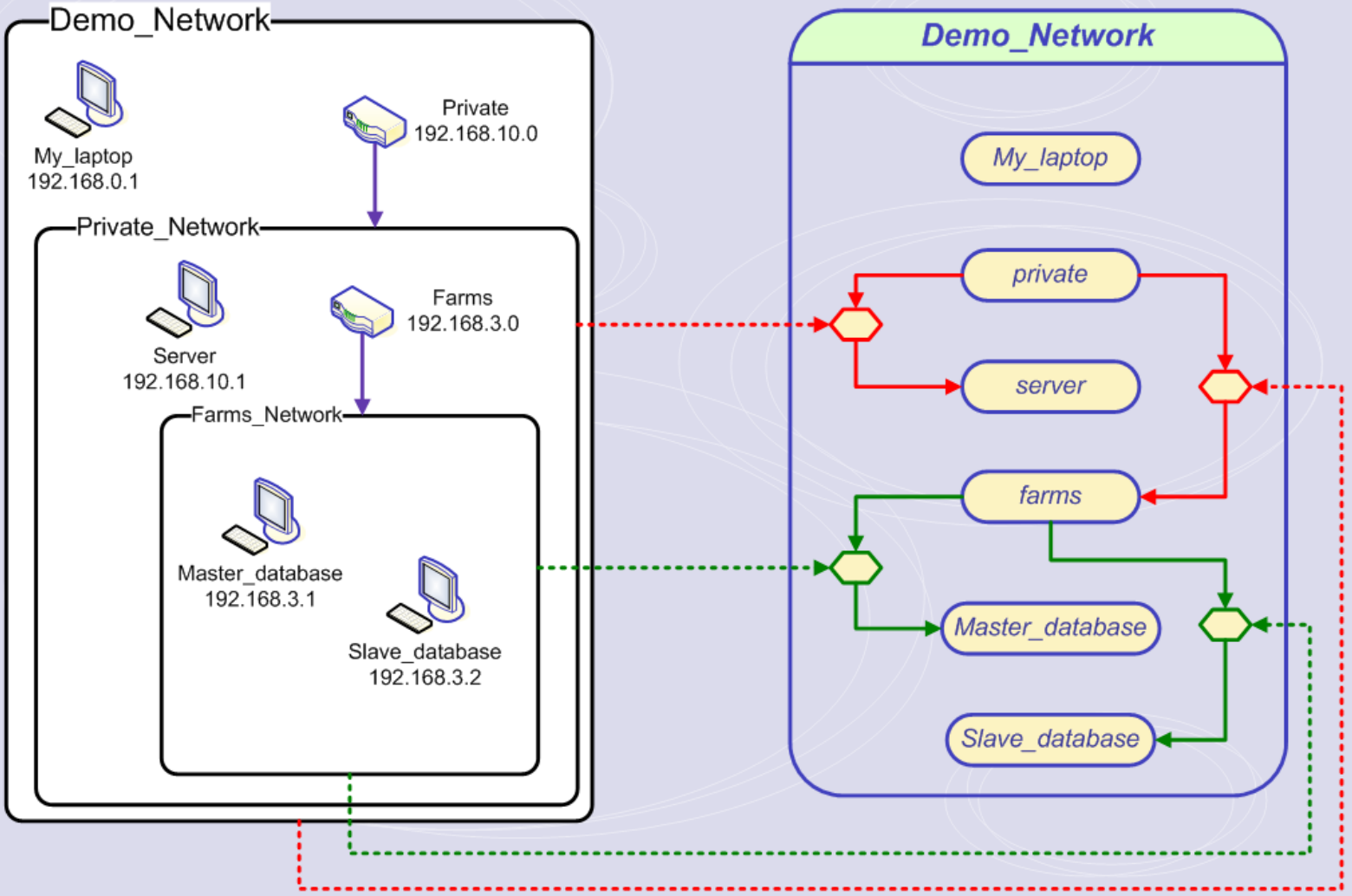
Network → Graph



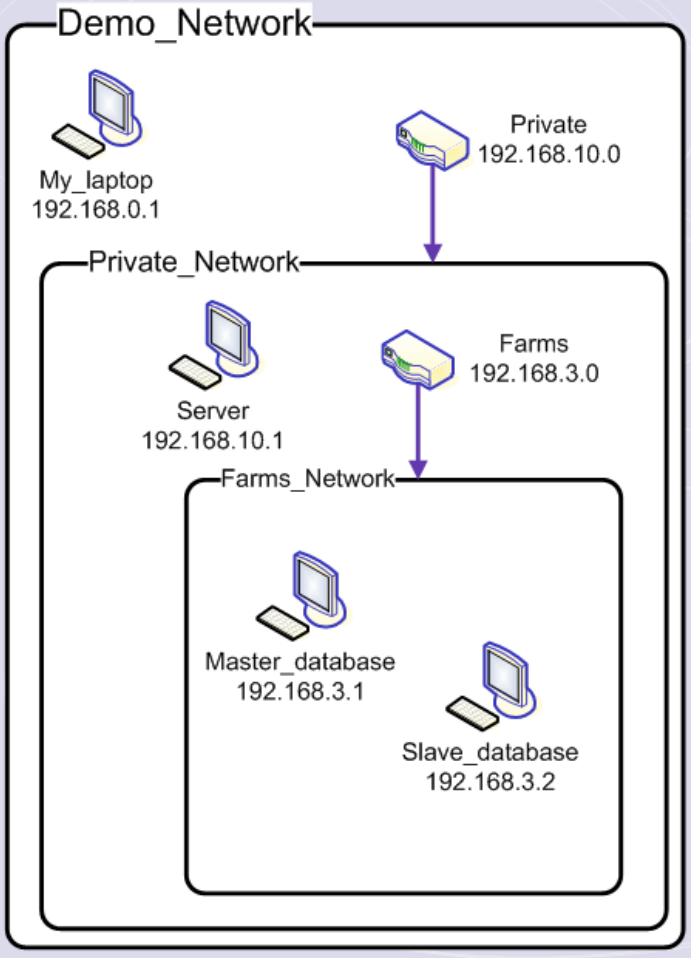
Phase de construction : *Element* → *Node*

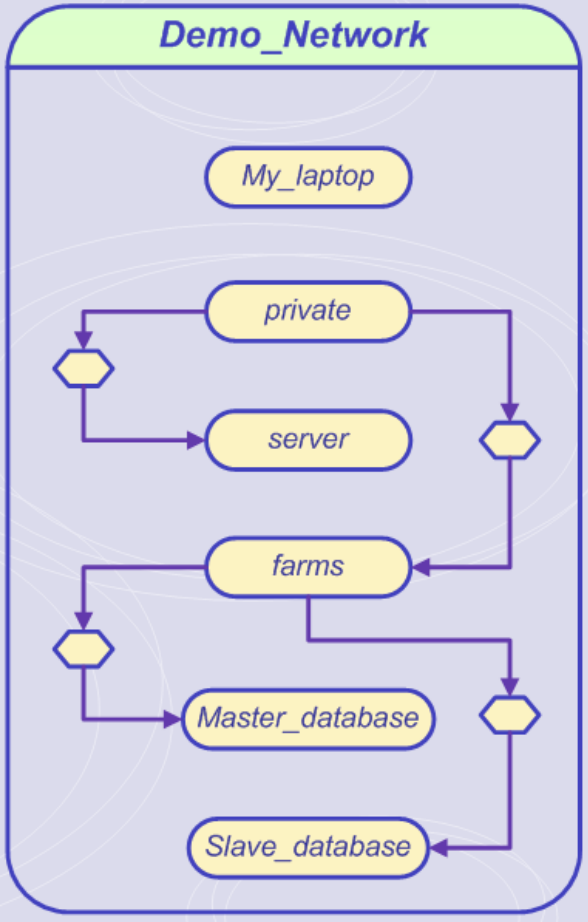


Phase de liaison : Création des *Edges*



Résultat final : *Network* → *Graph*





Agenda

1 Étude de cas : *Network* ↔ *Graph*

2 Introduction aux transformations de modèle

3 Kermeta : Un Kernel de Métamodélisation

4 Tissages d'aspects & Contraintes dans Kermeta

5 Rendre un modèle transformable . . .

6 Implémentation de transformations : *Graph* → *X*

7 Conclusions

8 Exemple : Opérationalisation dans Eclipse

- Modèles, Métamodèles, métamétamodèles, DSLs, ...
  - "Méta–bla–bla" : trop complexe pour l'ingénieur λ
- Par contre, les concepts suivants lui sont familiers :
  - Programmation orientée objets (Java, C#, C++, ...)
  - UML (au moins les diagrammes de classes)
  - Notion de *design by contract* (pré,post, invariants)
- Kermeta utilise ces points pour supporter la métamodélisation

"Kermeta - Breathe life into your metamodels"

Un langage pour la Métamodélisation [J.M. Jézéquel]

- Une extension d'EMOF
- Typage statique :
  - Génériques, Type Fonction (λ–itérateurs "à la" OCL)
- Orientation "objets" :
  - Héritage multiple, liaison dynamique, refléxivité
- Orientation "modèles" :
  - Association / Composition
  - Les modèles sont des entités de première classe
- Orientation "aspects" :
  - Enrichissement des modèles par ajout de fonctionnalités
  - Une syntaxe simplifiée
- Un "kernel" : fournir le strict nécessaire ...

Types & opérateurs usuels [kermeta::standard]

- Types scalaires très restreints
  - `Integer`, `String`, `Boolean`
- Opérateurs :
  - Affectation : := (naïve), ?= (cast)
  - Arithmétique : +,-,/,*
  - Comparaison : ==, !=, <, <=,>,>=
  - Logique : `and`, `or`, `not`
- Collections : fondées sur la définition d'OCL

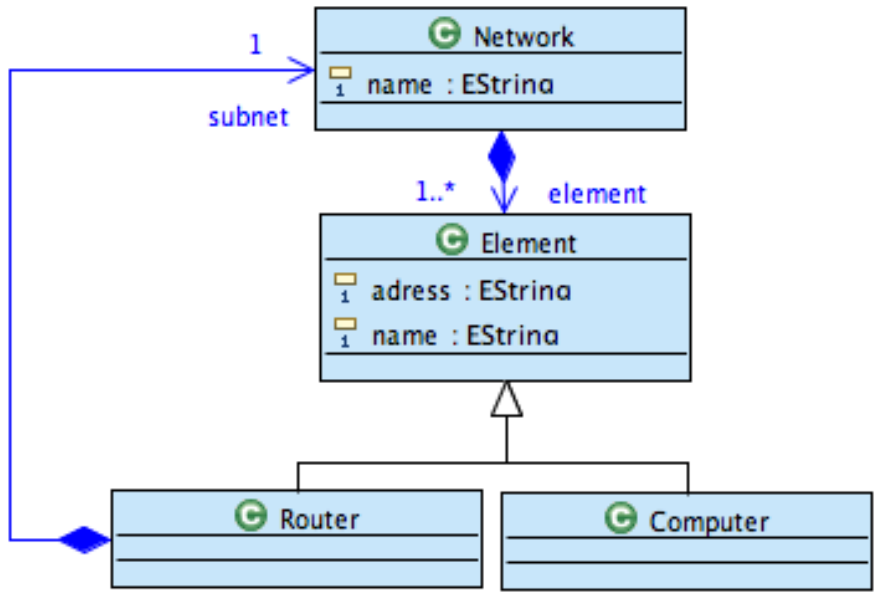
Mot–clé	Classe générique	Unicité	Ordre
`set`	`Set< T >`	Oui	Non
`oset`	`OrderedSet< T >`	Oui	Oui
`bag`	`Bag< T >`	Non	Non
`seq`	`Sequence< T >`	Non	Oui

Définition de classes, opérations, méthodes

- Déclaration de classes *à la* Java (`class C { }`)
  - Classes abstraites (`abstract`), généricité (`class A<T>`)
  - Héritage (`inherits`), multiple ou non
- Constructions : pas de constructeur ! (`MyClass.new`)
- Variables de classes : Attributs & Référence
  - `attribute a: String` $\Rightarrow$ *a* est contenue par composition ($\diamond$)
  - `reference r: String` $\Rightarrow$ *r* est référencée
  - `self` représente l'instance courante
  - Absence de visibilité : tout est *public*
- Méthode d'instance : Opérations & Méthodes
  - `operation name(arg1: T): OutputType is do ... end`
  - Redéfinition par héritage : `operation` $\longrightarrow$ `method`
  - Variable locale : `var tmp: String init String.new`
  - Retour : pas de `return` ! On utilise la variable `result`
  - Pas de surcharge dans le langage (simplification)

Mon premier métamodèle avec Kermeta

```
package network;

require kermeta
using kermeta::standard

abstract class Element{
  attribute adress: String
  attribute name: String
}

class Computer inherits Element {
}

class Router inherits Element {
  attribute subnet: Network
}

class Network {
  attribute name: String
  attribute element: set Element[1..*]
}
```

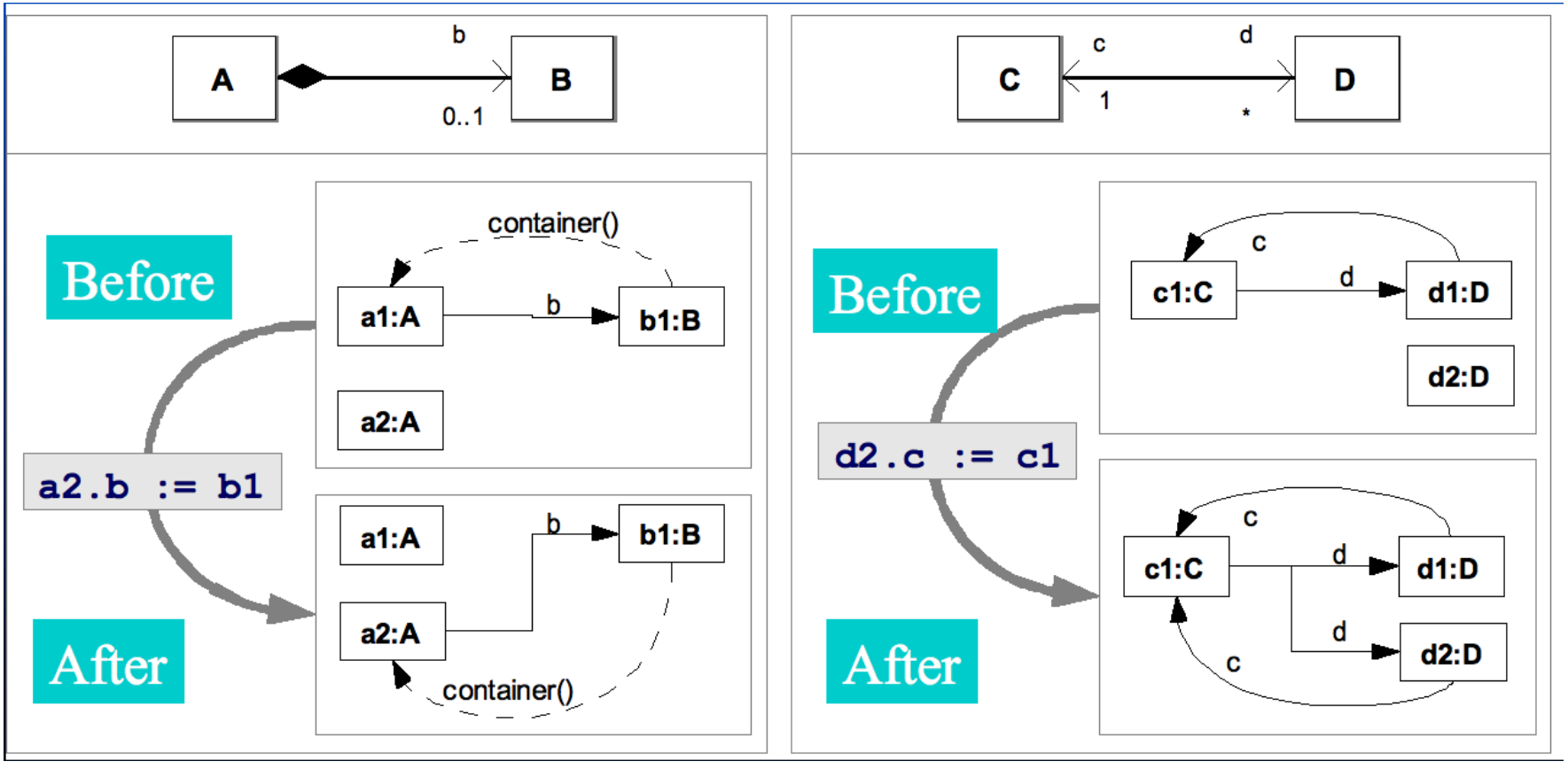


Mon premier exécutable avec Kermeta

```
@mainClass "root_package::Main"
@mainOperation "main"
package root_package;

require kermeta
require "network.kmt"

using network
using kermeta::standard

class Main
{
  operation main() : Void is do
    stdio.writeln("Creating␣a␣network")
    var aNetwork: Network init Network.new
    aNetwork.name := "DemoNetwork"
    stdio.writeln("aNetwork's␣name␣:␣" + aNetwork.name)
  end
}
```

Affectation : `attribute` ≠ `reference`

[J.M. Jézéquel]



Effet de l'affectation sur un exemple

```
class Main {
  operation main() : Void is do
    // init vars ...

    a1.b_att := b1
    a1.b_ref := b2
    stdio.writeln("## BEFORE ##")
    a1.display()
    a2.display()

    a2.b_att := a1.b_att
    a2.b_ref := a1.b_ref
    stdio.writeln("## AFTER ##")
    a1.display()
    a2.display()
  end
}
class A {
  attribute b_att: B
  reference b_ref: B
}
class B {}
```

```
## BEFORE ##
a1 :
  - b_att : b1
  - b_ref : b2
a2 :
  - b_att : <void>
  - b_ref : <void>
## AFTER  ##
a1 :
  - b_att : <void>
  - b_ref : b2
a2 :
  - b_att : b1
  - b_ref : b2
```

Manipulation de collections : Suite de Fibonacci

$$\mathcal{F}(n) = \begin{cases} n = 0 & \Rightarrow \quad \mathcal{F}(0) = 1 \\ n = 1 & \Rightarrow \quad \mathcal{F}(1) = 1 \\ n \geq 2 & \Rightarrow \quad \mathcal{F}(n) = \mathcal{F}(n-1) + \mathcal{F}(n-2) \end{cases}$$

```
operation fibo(n: Integer): Sequence<Integer> is do
  var preds : Sequence<Integer> init Sequence<Integer>.new
  if n == 1 then
    preds.add(1) preds.add(1)
  else
    preds.addAll(fibo(n-1))
    var current: Integer init preds.elementAt(preds.size -2)
    current := current + preds.elementAt(preds.size -1)
    preds.add(current)
  end
  result := preds
end
```

Fermetures & λ–fonctions pour l'itération

- $\mathcal{F}(12) = \{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233\}$
  - `var f: seq Integer[12] init fibo(12)`
- Effectuer une action $\forall e \in C$ — `each`
  - `f.each{ n | stdio.write( n.toString + "␣") }`
- Vérifier une condition $\forall e \in C$ — `forAll`
  - `var b: Boolean init f.forAll{ n | n < 250 }`
- Sélection d'un sous–ensemble (`filter`) — `select`
  - `var f2: Sequence<Integer> init f.select{n | n < 100}`
- Exclusion d'un sous–ensemble — `reject`
  - `var f3: Sequence<Integer> init f.reject{n | n < 100}`
- Mapping de fonction — `collect`
  - `var f4: Sequence<Integer> init fib.collect{n | n + 1}`
- Detection d'un élément — `detect`
  - `var x: Integer init fib.detect{n | n > 47}`
- Existence d'un élément — `exists`
  - `var b2: Boolean init fib.exists{n | n > 47 }`

Intégration avec EMF : manipulation d'entités ECore

- `require` : Importe un métamodèle (Kermeta ou ECore)
- Dans le package `kermeta::persistence` :
  - `EMFRepository` → `Repository`, `Resource`
- Sérialisation / déserialisation XMI (quasi) immédiate

```
require kermeta
require "platform:/resource/Graphs/metamodels/graph.ecore"
using kermeta::persistence
using kermeta::standard
using graph
class Main {
  operation main() : Void is do
    var file: String init "platform:/resource/Graphs/hand/g.xmi"
    var repository: Repository init EMFRepository.new
    var resource: Resource init repository.getResource(file)
    var loaded: Graph init resource.one.asType(Graph)
    stdio.writeln(loaded.name)
  end
}
```

Agenda

1 Étude de cas : *Network* ↔ *Graph*

2 Introduction aux transformations de modèle

3 Kermeta : Un Kernel de Métamodélisation

4 Tissages d'aspects & Contraintes dans Kermeta

5 Rendre un modèle transformable . . .

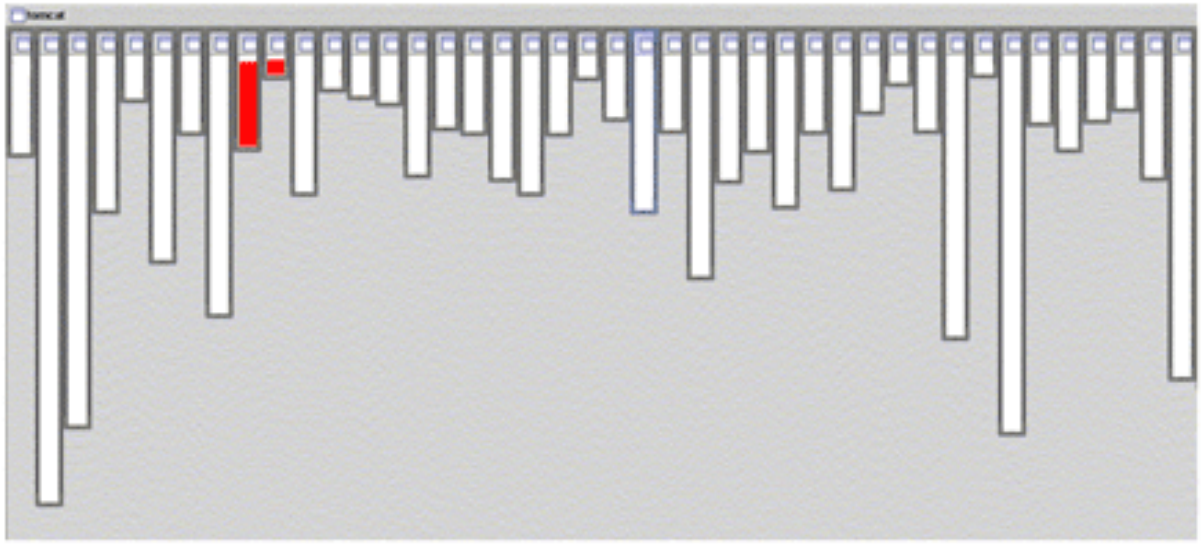
6 Implémentation de transformations : *Graph* → *X*

7 Conclusions

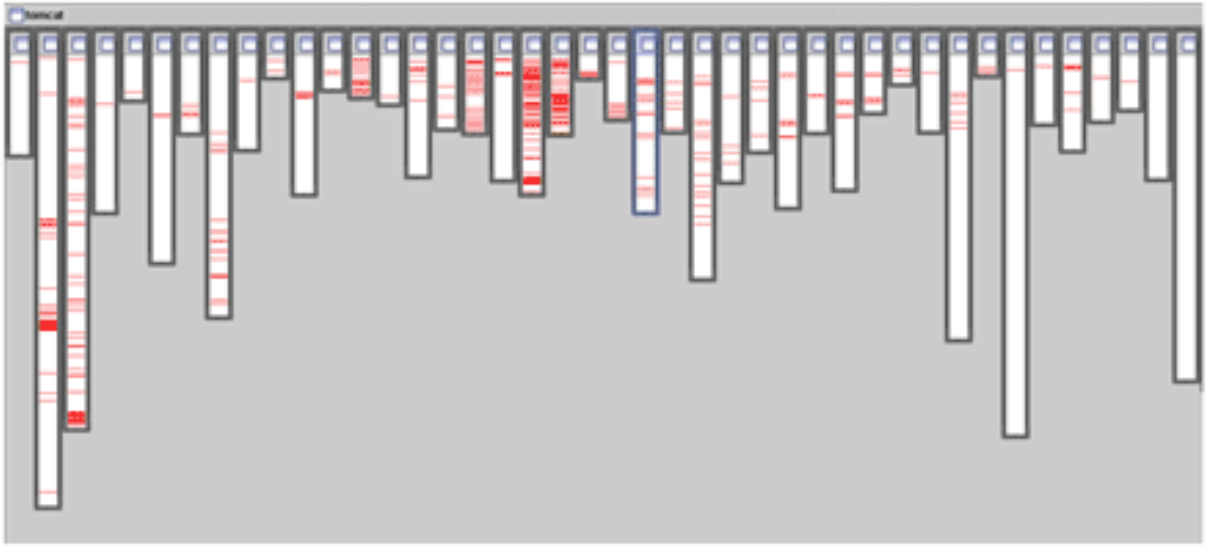
8 Exemple : Opérationalisation dans Eclipse

Aspects & Préoccupations transverses

- *Url pattern–matching* dans Tomcat : Couplage faible



- *Loging* dans Tomcat : Couplage (très) fort !



Programmation Orientée Aspects

Idée : Certaines préoccupations ne sont pas modularisables

- Ecriture du programme de base
- Isertion *a posteriori* des préoccupations transverses

Vocabulaire orienté aspects

- Endroit où l'on peut modifier le programme
  - Points de jonctions (*joinpoint*)
- Expression de la préoccupation transverse (code)
  - greffon (*advice*)
- Où appliquer la greffe ?
  - Point de coupe (*pointcut*), $\in$ *joinpoints*
- Appliquer le greffon sur le (les) points de coupe
  - Tissage d'aspect (*aspect weaving*)
- Expression d'un aspect :
  - $Aspect \equiv (pointcut, advice)$

Différents types d'aspects

Aspects comportementaux

- Enrichit le comportement d'un objet
- Exemple typique : *logger l'accès à une variable*
  - Point de coupe : Appel de la méthode `getMyVariable()`
  - Greffon : `before : System.out.println("log")`

Aspects structurels

- Enrichir la structure du modèle
- Ajout d'attributs, de classes, de méthodes, ...

- Canevas de référence Java : AspectJ
- Il existe des tisseurs pour de nombreux langages
- Les aspects sont partout ! ! (EJB, Persistance, Tomcat, ...)
- Kermeta accepte un sous ensemble des aspects structurels

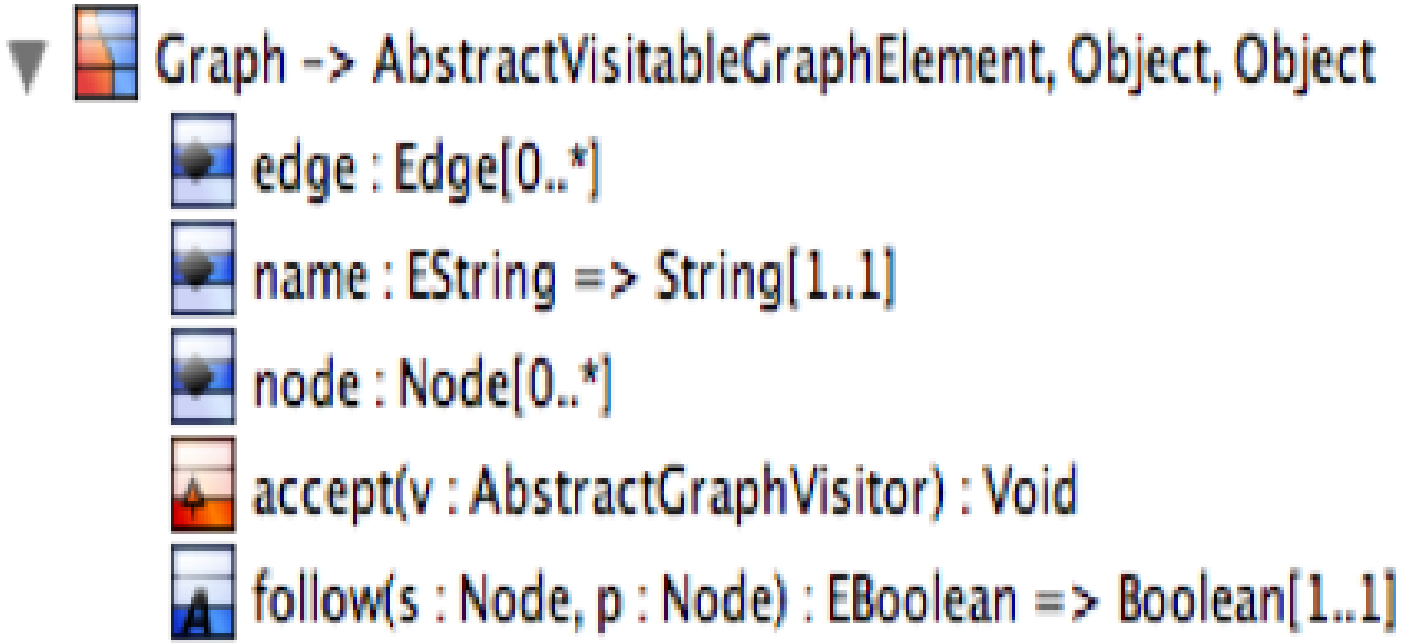
  $\Rightarrow$ Mais cela nous suffira amplement !

Tissage d'aspects avec Kermeta

- Où appliquer l'aspect ? (point de coupe, "pointcut")
  - On se positionne dans un `package`
  - Et on nomme la `class` à enrichir
- Comment enrichir ? (greffon, "advice")
  - On ajoute tout simplement du code Kermeta
  - Attention aux conflits de nommage !

```
package graph;
require "platform:/resource/Graphs/metamodels/graph.ecore"

aspect class Graph {

  attribute pKey: Integer

  operation isSource(n: Node): Boolean is do
    result := not self.edge.exists{e | e.target == n }
  end
}
```

Truc : Code couleur *outline*



- En bleu : Concept provenant du métamodèle original
- En rouge : Ajout provenant de l'aspect
- Bicolore : concept impacté par l'aspect

Design–by–contract : Des réseaux “contraints”

- Invariant de classe :
  - Vérifié sur demande par `checkAllInvariants()`
  - Ici, `self.name` ne peut être vide
- Contractualisation des opérations :
  - Approche pré–condition & post–condition
  - `rename` : préfixe non vide et résultat plus court que 255

```
aspect class Network {

  inv nameValidity is do self.name != "" end

  operation rename(prefix: String): Void
    pre prefixNotEmpty is do prefix != ""  end
    post nameTooLong is do self.name.size() <= 255 end
    is do self.name := prefix + self.name end
}
```

Mise en œuvre des contraintes

```
operation main() : Void is do
    var aNetwork: Network init Network.new
    aNetwork.name := ""
    do
      aNetwork.checkAllInvariants()
      rescue(e: Exception)
        stdio.writeln("Exception : [" + e.message +"]")
    end
    do
      aNetwork.rename("")
      rescue(e: Exception)
        stdio.writeln("Exception : [" + e.message +"]")
    end
  end
```

```
Exception : [Inv nameValidity of class Network violated]
Exception : [pre prefixNotEmpty of operation rename of
             class Network violated]
```

Agenda

1 Étude de cas : *Network* ↔ *Graph*

2 Introduction aux transformations de modèle

3 Kermeta : Un Kernel de Métamodélisation

4 Tissages d'aspects & Contraintes dans Kermeta

5 Rendre un modèle transformable . . .

6 Implémentation de transformations : *Graph* → *X*

7 Conclusions

8 Exemple : Opérationalisation dans Eclipse

Problématique générale

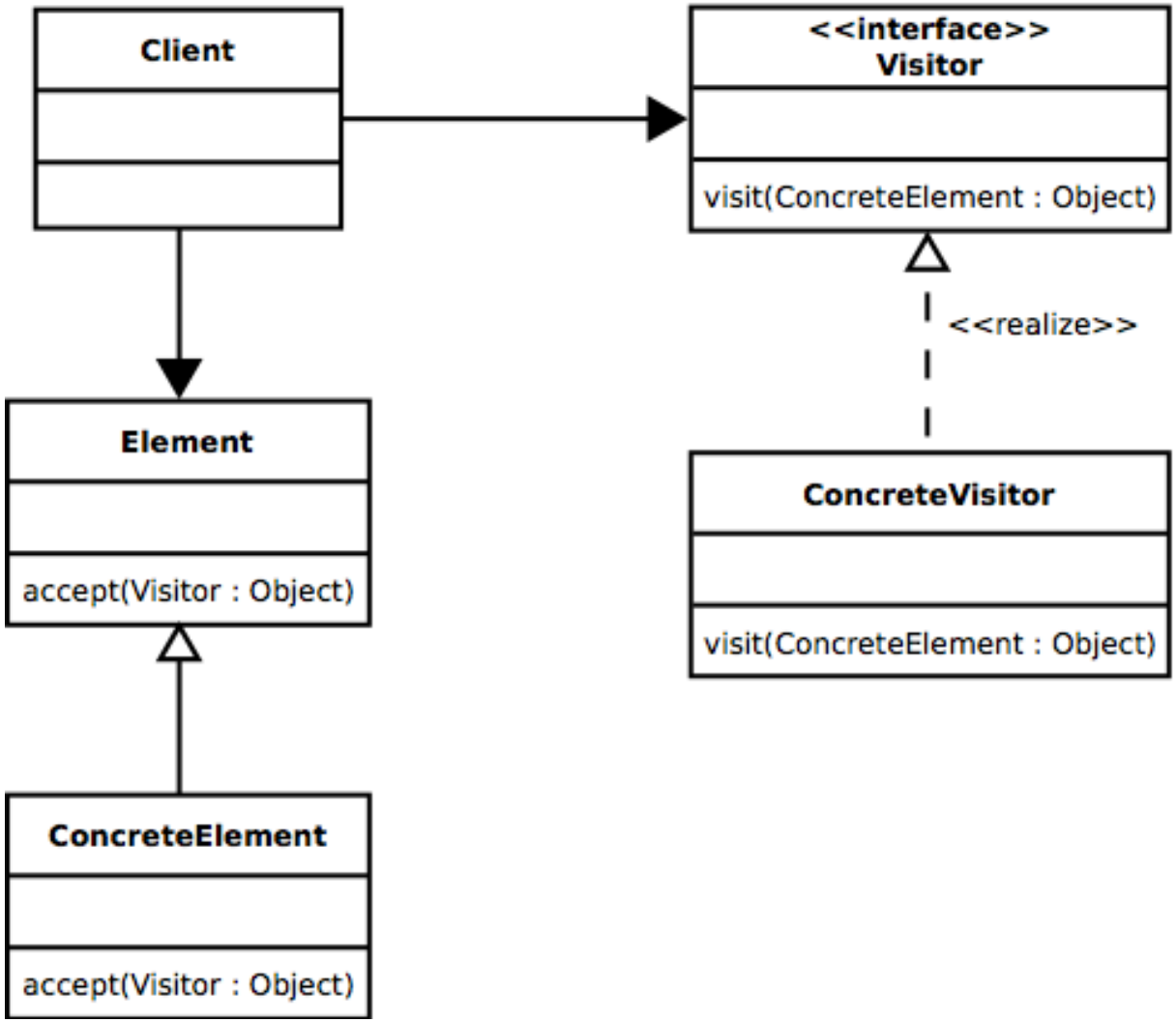
Problème

- La transformation prend en entrée un modèle m_1
- Et doit produire en sortie un autre modèle m_2
- Approche structurelle :
  - Les métamodèles définissent les structures
  - Mais aucun support à la transformation !

Solution

- Schéma de conception visiteur
- Visite des entités composant le modèle m_1 :
  1. Construction des entités de m_2 correspondantes
  2. Liaison des nouvelles entités entre elles

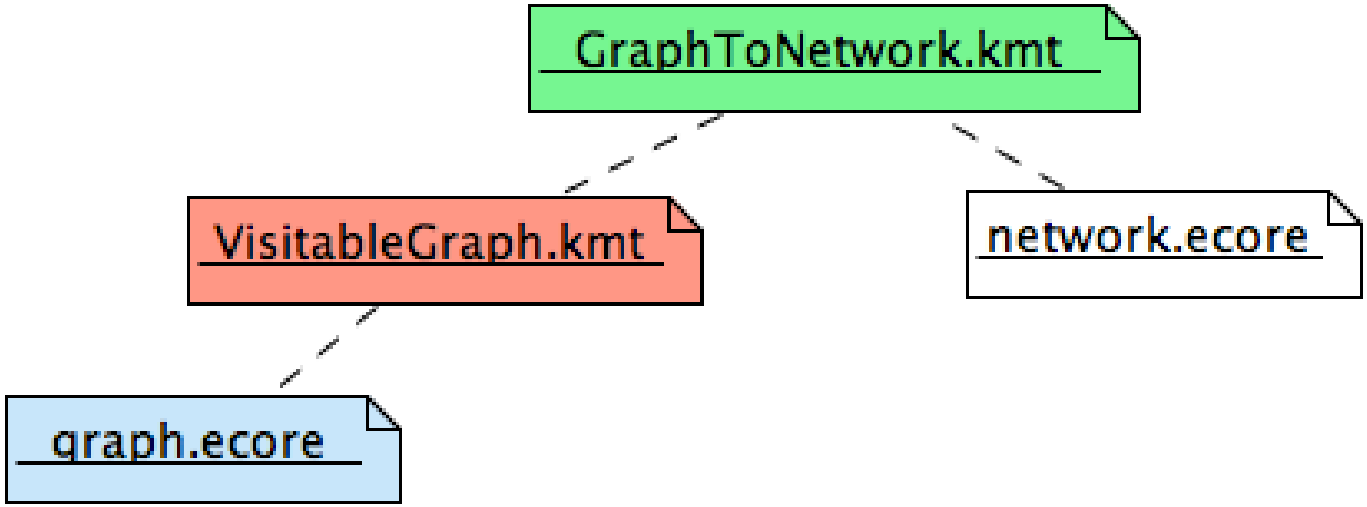
Rappel : Schéma de conception Visiteur



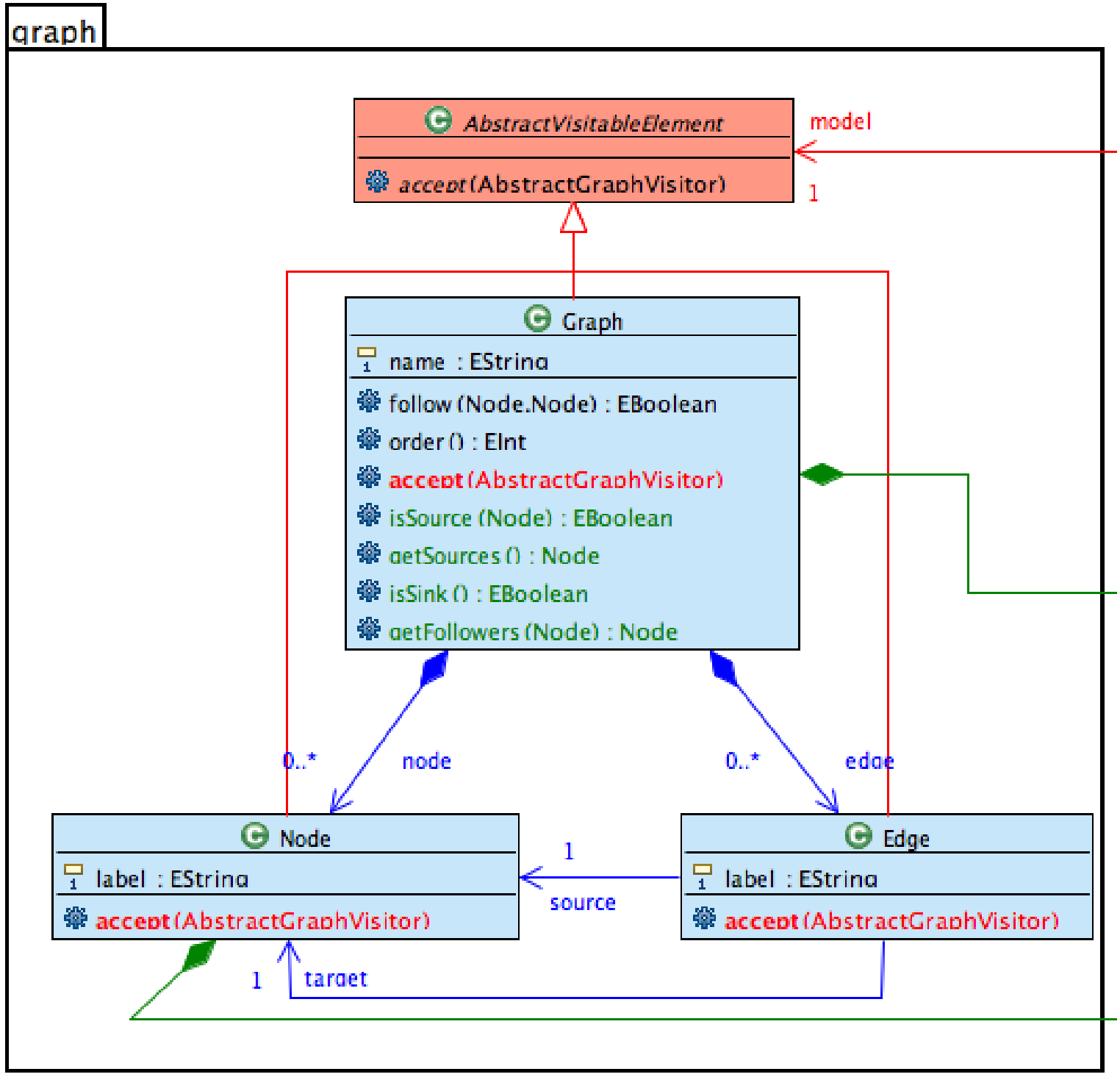
"Aspectisation" des transformation

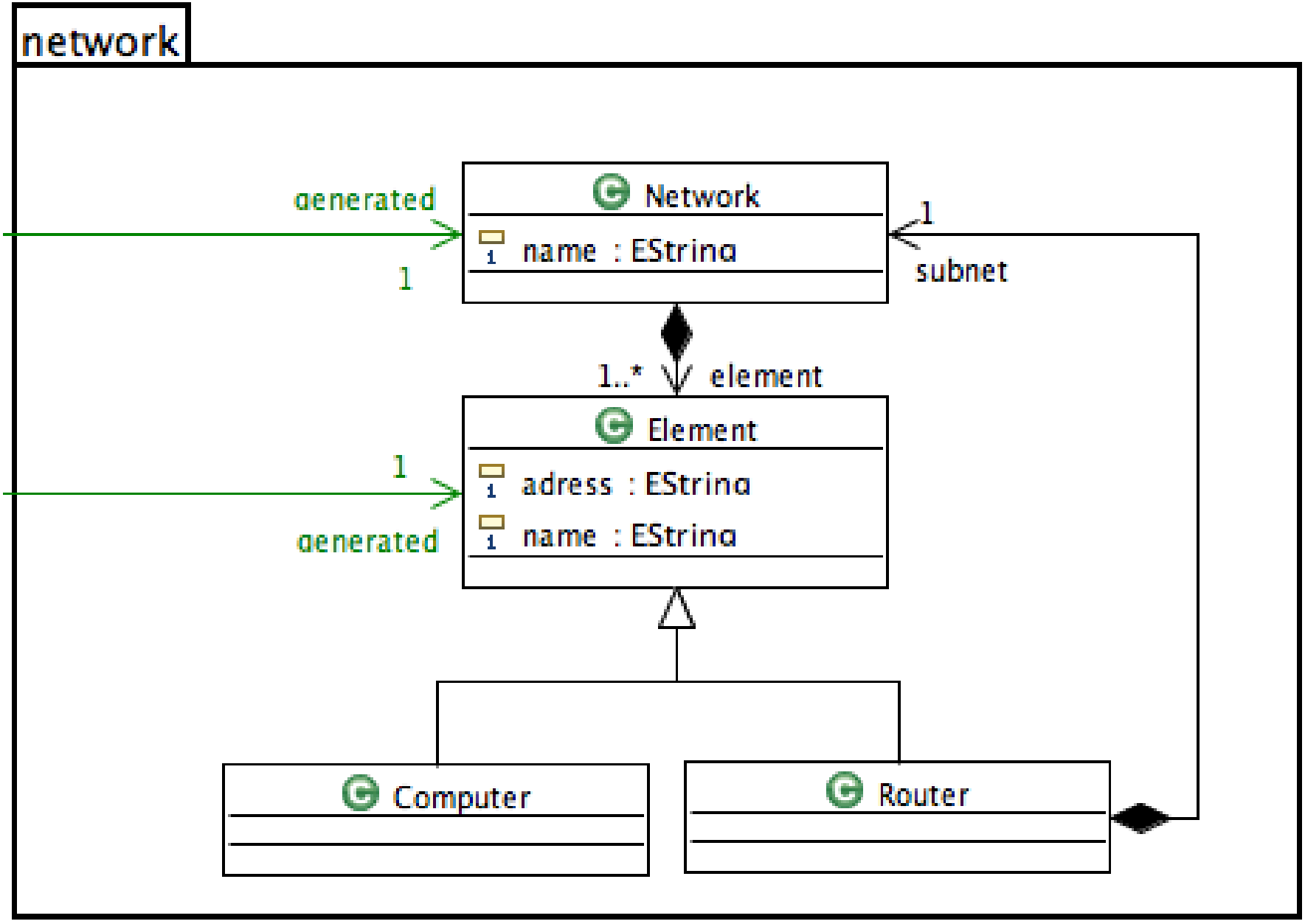
- Le métamodèle originel n'implante pas un schéma Visiteur
  - $\Rightarrow$ Il suffit de l'enrichir (tissage d'aspect)
- Quelle que soit la transformation :
  - On requiert le métamodèle voulu
  - On y tisse :
    - La méthode `accept` sur toutes ses entités
    - Une classe abstraite décrivant les visiteurs de ce modèle
    - Une classe abstraite représentant la transformation
- Pour une transformation donnée
  - On requiert le métamodèle précédemment enrichi
  - On y ajoute :
    - Un (ou plusieurs) visiteurs
    - Une classe de transformation

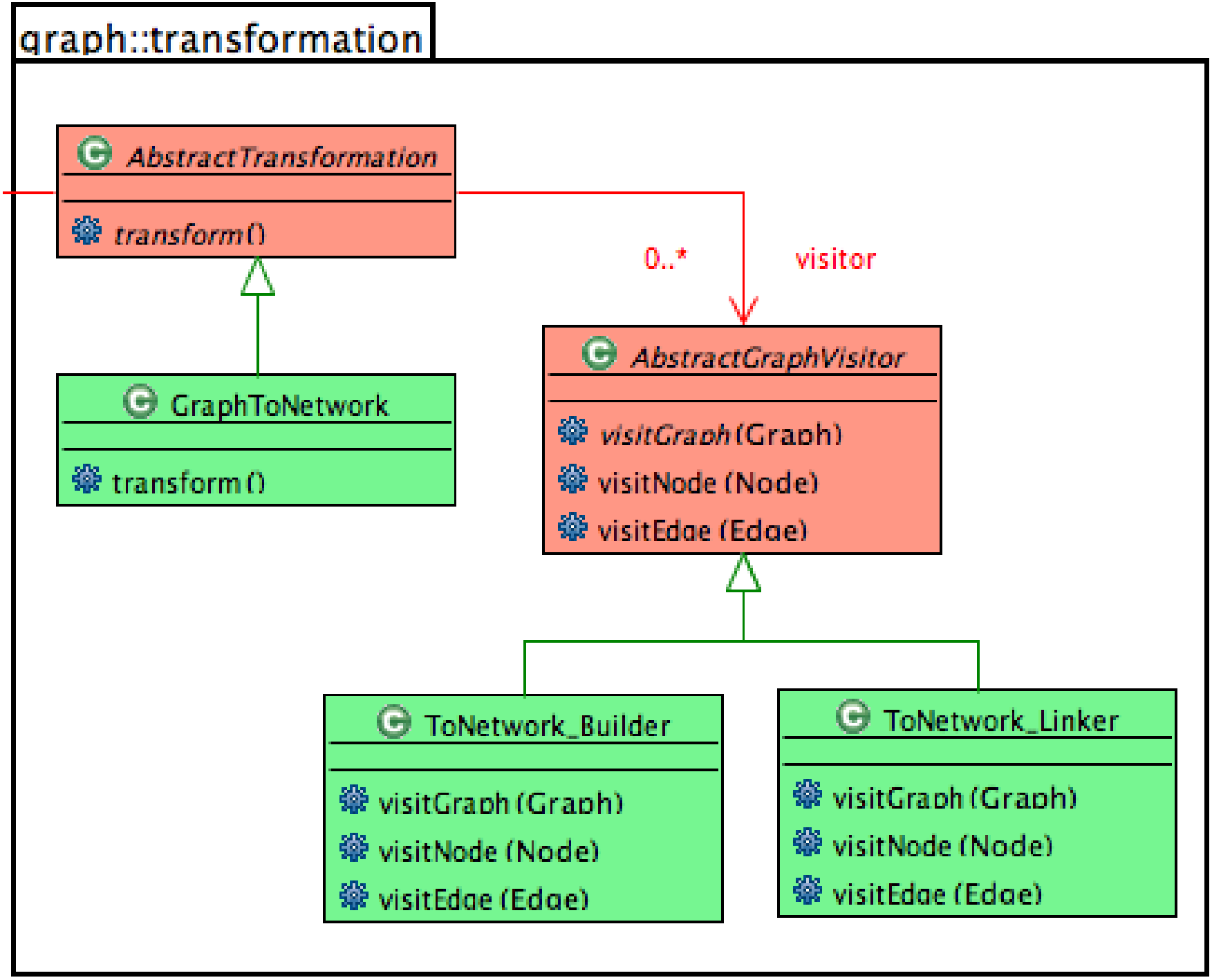
Mise en œuvre sur l'exemple : *Graph* → *Network*

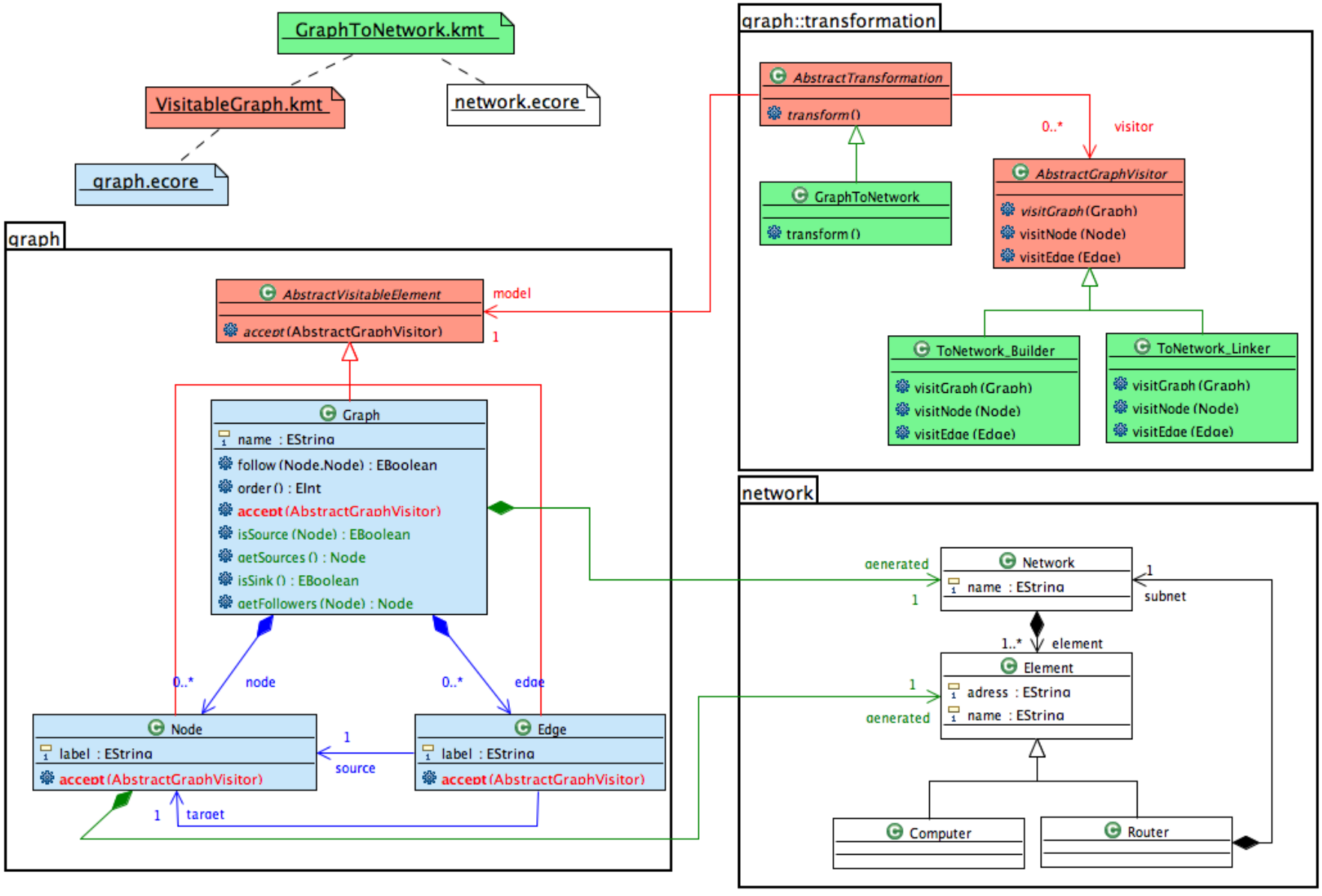


1. `VisitableGraph.kmt` : Abstraction (visite, transfo)
   - `AbstractVisitableElement`, `AbstractVisitor`, `AbstractTransformation`
2. `GraphToNetwork.kmt` : Implémentation (*Graph* → *Network*)
   - `ToNetwork_Linker`, `ToNetwork_Builder`, `Graph2Network`


graph
AbstractVisitableElement
accept(AbstractGraphVisitor)
model
1
Graph
name : EString
follow(Node.Node) : EBoolean
order() : EInt
accept(AbstractGraphVisitor)
isSource(Node) : EBoolean
getSources() : Node
isSink() : EBoolean
getFollowers(Node) : Node
0..*
node
0..*
edge
Node
label : EString
accept(AbstractGraphVisitor)
1
source
Edge
label : EString
accept(AbstractGraphVisitor)
1
target


network
generated
1
Network
1
name : EString
1
subnet
1..*
element
Element
1
adress : EString
1
name : EString
1
generated
Computer
Router


graph::transformation
AbstractTransformation
transform()
0..*
visitor
GraphToNetwork
transform()
AbstractGraphVisitor
visitGraph(Graph)
visitNode(Node)
visitEdge(Edge)
ToNetwork_Builder
visitGraph(Graph)
visitNode(Node)
visitEdge(Edge)
ToNetwork_Linker
visitGraph(Graph)
visitNode(Node)
visitEdge(Edge)


GraphToNetwork.kmt
VisitableGraph.kmt
network.ecore
graph.ecore
graph
AbstractVisitableElement
accept(AbstractGraphVisitor)
model
1
Graph
name : EString
follow(Node,Node) : EBoolean
order() : EInt
accept(AbstractGraphVisitor)
isSource(Node) : EBoolean
getSources() : Node
isSink() : EBoolean
getFollowers(Node) : Node
0..*
node
0..*
edge
Node
label : EString
accept(AbstractGraphVisitor)
1
source
Edge
label : EString
accept(AbstractGraphVisitor)
1
target
graph::transformation
AbstractTransformation
transform()
0..*
visitor
GraphToNetwork
transform()
AbstractGraphVisitor
visitGraph(Graph)
visitNode(Node)
visitEdge(Edge)
ToNetwork_Builder
visitGraph(Graph)
visitNode(Node)
visitEdge(Edge)
ToNetwork_Linker
visitGraph(Graph)
visitNode(Node)
visitEdge(Edge)
network
generated
1
Network
name : EString
1
subnet
1..*
element
Element
adress : EString
name : EString
1
generated
Computer
Router

Agenda

1 Étude de cas : *Network* ↔ *Graph*

2 Introduction aux transformations de modèle

3 Kermeta : Un Kernel de Métamodélisation

4 Tissages d'aspects & Contraintes dans Kermeta

5 Rendre un modèle transformable . . .

6 Implémentation de transformations : *Graph* → *X*

7 Conclusions

8 Exemple : Opérationalisation dans Eclipse

Rendre un *Graphe* visitable — `VisitableGraph.kmt`

- Ajout d'une *super–classe* abstraite `AbstractVisitableGraphElement`
- Ajout d'un héritage sur `Graph`, `Node` et `Edge`
- Implémentation de la méthode `accept` dans ces 3 classes

```
package graph;

require kermeta
require "platform:/resource/Graphs/metamodels/graph.ecore"

abstract class AbstractVisitableGraphElement {
  operation accept(v: AbstractGraphVisitor): Void is abstract
}

aspect class Graph inherits AbstractVisitableGraphElement {
  method accept(v: AbstractGraphVisitor): Void is do
    v.visitGraph(self)
  end
}
```

Package graph::transformation VisitableGraph.kmt

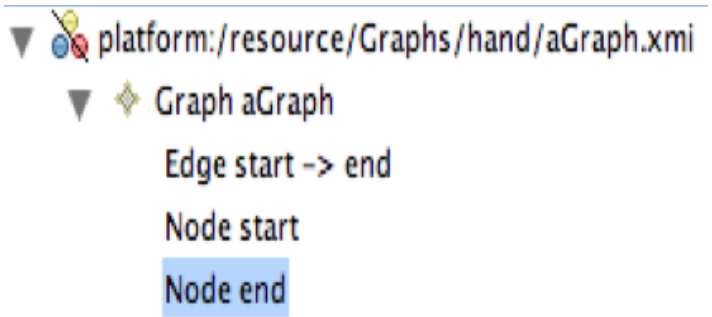
- Déclaration d'un Visiteur abstrait : `AbstractGraphVisitor`
- Déclaration d'une transformation abstraite :

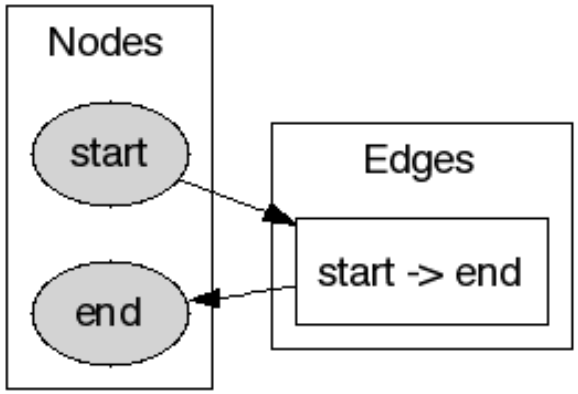
  `AbstractGraphTranasformation`

```
package transformation {
  abstract class AbstractGraphTransformation {

    operation transform(input: AbstractVisitableGraphElement) :
      Object is abstract
  }

  abstract class AbstractGraphVisitor {
    operation visitGraph(g: Graph): Void is abstract
    operation visitNode(n: Node): Void is abstract
    operation visitEdge(e: Edge): Void is abstract
  }
}
```

Transformation *Model* → *Text* : *Graph* → *Graphviz*

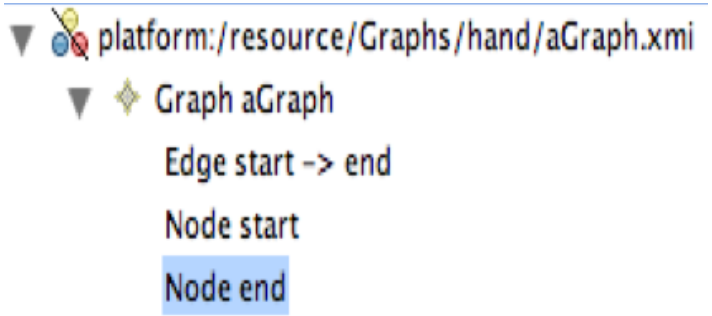
- $\tau_{graphviz}(graph) \mapsto sourceCode$
  - Les nœuds sont mis dans un sous–graphe
  - ⇒ Chaque noeud *n* du graphe est transformé en une boite
  - Les arcs sont mis dans un autre sous–graphe
  - ⇒ Chaque arc est transformé en une boite carrée
  - ⇒ On génère 2 flèches pour relier l'arc aux 2 boites

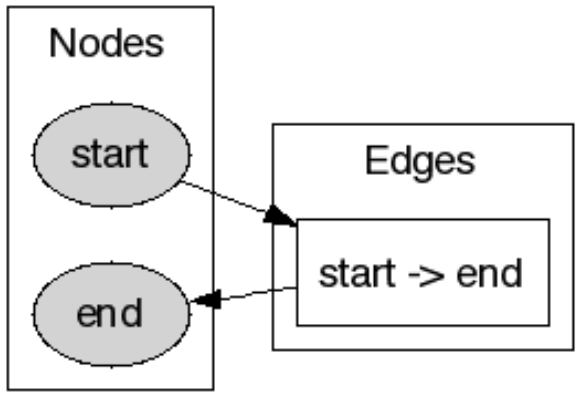




Code Graphviz associé

```
digraph aGraph {
  rankdir=LR ;

  subgraph cluster_nodes {
    node [style=filled] ;
    label="Nodes" ;
    fontname="Helvetica" ;
    start ;
    end ;
  }

  subgraph cluster_edge {
    label="Edges" ;
    fontname="Helvetica" ;
    edge1 [shape=box,
           label="start -> end"] ;
    start -> edge1 ;
    edge1 -> end ;
  }

}
```





Implémentation du visiteur — Graph2Dot.kmt

```
class ToDotVisitor inherits AbstractGraphVisitor {

  attribute res: String
  attribute cpt: Integer

  operation store(s: String): Void is do
    res.append(s+"\n")
  end

  method visitGraph(g: Graph): Void is do
    res := String.new
    cpt := 0
    store("digraph " + g.name + " {")
    store("  subgraph cluster_nodes {")
    g.node.each{n | n.accept(self) }
    store("  }")
    store("  subgraph cluster_edge {")
    g.edge.each{e | e.accept(self) }
    store("  }")
    store("}")
  end
```

Implémentation du visiteur — Graph2Dot.kmt

```
    operation escape(t: String): String is do
      result := t.split("\\.").elementAt(0)
    end

    method visitNode(n: Node): Void is do
      store("    " + escape(n.label) + " ;")
    end

    method visitEdge(e: Edge): Void is do
      cpt := cpt+1
      store("    edge" + cpt.toString +" [shape=box, label=\""
            + e.label + "\"] ;")
      store("    " + escape(e.source.label)+" -> edge"
            + cpt.toString +" ;")
      store("    edge" + cpt.toString + " -> "
            + escape(e.target.label) + " ;")
    end
}
```

Exécution de la transformation

- Implémentation de la transformation (`VisitableGraph.kmt`) :

```
class GraphToDot inherits AbstractGraphTransformation {

  method transform(input: AbstractVisitableGraphElement):
    Object is do
    var visitor: ToDotVisitor init ToDotVisitor.new
    input.asType(Graph).accept(visitor)
    result ?= visitor.res
  end
}
```

- Utilisation de la transformation :

```
var t: GraphToDot init GraphToDot.new
var s: String init t.transform(myGraph).asType(String)
stdio.writeln(s)
```

Transformation *Model* → *Model* : *Graph* → *Network*

- $\tau_{network}(graph) \mapsto network$
  - Les sources du *Graphe* sont les *éléments* du premier réseau
  - Chaque puits du *Graphe* est transformé en *Computer*
  - Tout *noeud* ayant des suivants est transformé en *Router*

  ⇒ Sources, Puits, ... : On doit tisser ces méthodes dans `Graph` !
- Implémentation en 2 visiteurs (double passe) :
  1. `Builder` : instancie un concept réseau pour chaque noeud
  2. `Linker` : Décide des liaisons entre éléments réseaux

  ⇒ Approche classique issue des techniques de compilation
- Construction de la transformation par assemblage
  - La transformation lance le `Builder`
  - Puis envoie le `Linker`
  - Et retourne le résultat

Enrichissement des *graphes*

```
aspect class Node { attribute generated : Element }

aspect class Graph {

  attribute generated: Network

  operation isSource(n: Node): Boolean is do
    result := not self.edge.exists{e | e.target == n }
  end

  operation getSources(): Sequence<Node> is do
    result := self.node.select{n| isSource(n) }
  end

  operation getFollowers(n: Node): Sequence<Node> is do
    var e: Sequence<Edge> init self.edge.select{e | e.source == n}
    var res: Sequence<Node> init Sequence<Node>.new
    e.each{ edge | res.add(edge.target) }
    result := res
  end
}
```

Implémentation de la transformation

```
class GraphToNetwork inherits AbstractGraphTransformation {

  method transform(input: AbstractVisitableGraphElement):
    Object is do

    var builder : ToNetworkVisitor_Builder
      init ToNetworkVisitor_Builder.new
    input.asType(Graph).accept(builder)

    var linker : ToNetworkVisitor_Linker
      init ToNetworkVisitor_Linker.new
    input.asType(Graph).accept(linker)

    result := input.asType(Graph).generated
  end

}
```

Agenda

1 Étude de cas : *Network* ↔ *Graph*

2 Introduction aux transformations de modèle

3 Kermeta : Un Kernel de Métamodélisation

4 Tissages d'aspects & Contraintes dans Kermeta

5 Rendre un modèle transformable . . .

6 Implémentation de transformations : *Graph* → *X*

7 Conclusions

8 Exemple : Opérationalisation dans Eclipse

Kermeta . . .

- Un *kernel* de métamodélisation
  - Langage de définition de métamodèles
  - Itérateurs fonctionnels, contraintes OCL, . . .

  $\Rightarrow$ *Boite à outils* du métamodélisateur
- Un tisseur d'aspects orienté modèles
  - Moins puissant que les tisseurs d'aspects usuels (AspectJ)
  - Mais a t'on réellement besoin de tant de puissance ?

  $\Rightarrow$ Surtout s'il en coûte l'utilisabilité !
- Un outil intégré dans EMF
  - Chargement de métamodèles ECore, navigabilité immédiate
  - Sérialisation / Desérialisation XMI

  $\Rightarrow$ Pourquoi s'en priver !

Kermeta n'est pas LA solution de transformation

- C'est UNE solution pour transformer des modèles
  - L'approche aspect évite la pollution de modèles
  - Mais l'approche reste impérative
- Il existe d'autres solutions de transformations :
  - Approche déclarative : ATL (Bézivin et al)
    - `http://www.eclipse.org/m2m/atl/`
    - Déclaration de règles de transformations
    - ⇒ On peut atteindre automatiquement la bi–directionalité
    - Utilisé entre autre par le projet Eclipse STP (Adrian Mos)
  - Transformations par styles `XSL`
    - Après tout, sur le disque, tout es XML, ...
    - ⇒ Bon courage ...
- Implémentation manuelle :
  - Modification des codes Java générés par EMF
  - Utilisations d'outils exotiques (`Prolog`, `Scheme`, ...)

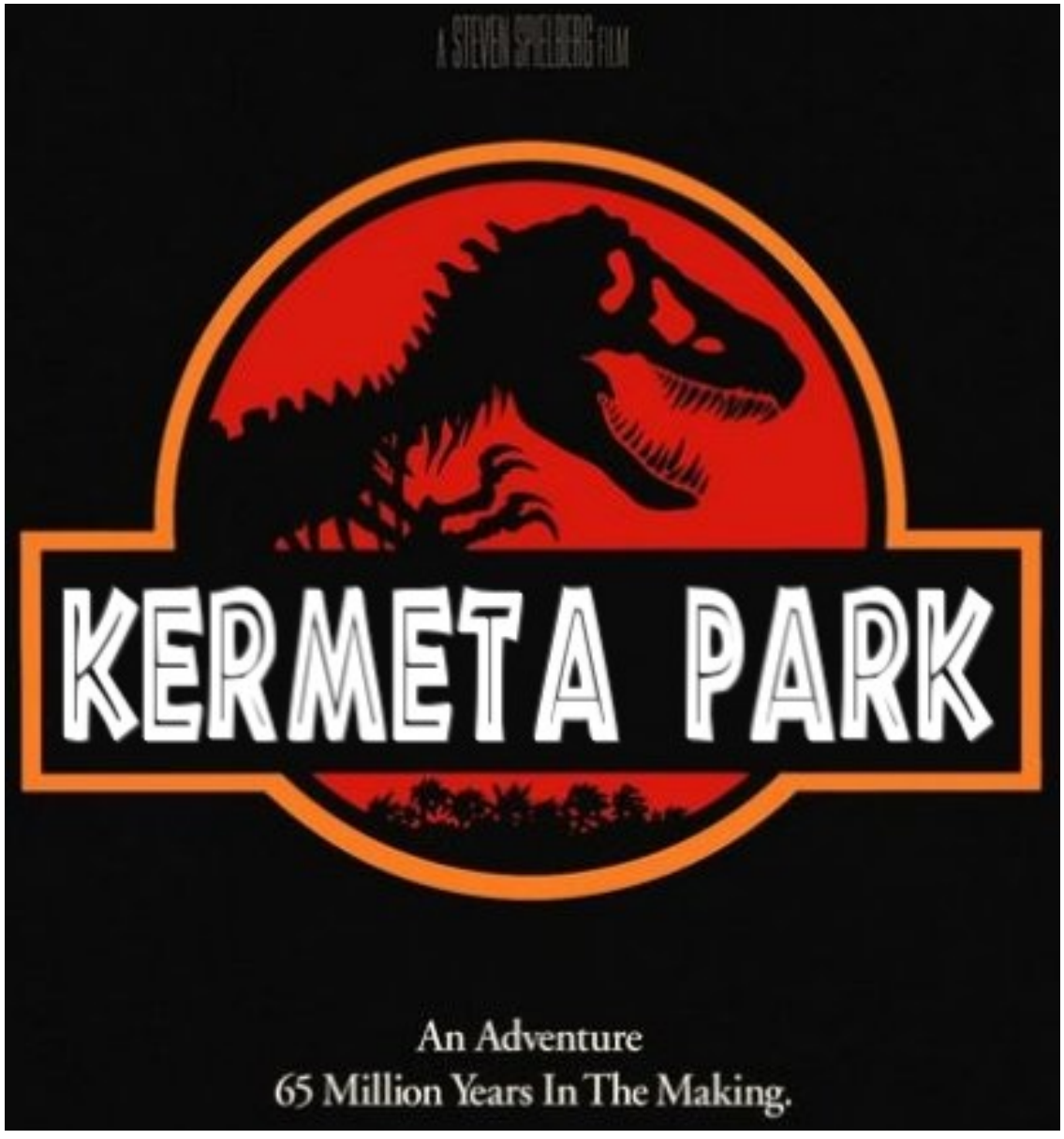
Remerciements non exhaustifs

- Franck Chauvel
- François Tanguy
- Vincent Mahé
- Gilles Perrouin
- Pierre – Alain Muller, aka *pam*
- Jean – Marc Jézéquel

Bibliographie (non exhaustive aussi)

- Site de référence :
  - `http://www.kermeta.org/`
- Manuel de référence :
  - `http://www.kermeta.org/documents/manual/`
- Model Development Kits :
  - `http://www.kermeta.org/mdk/`
- Article de recherche :
  - Weaving Executability into Object-Oriented Meta-Languages
  - Pierre-Alain Muller, Franck Fleurey & Jean-Marc Jézéquel [MODELS'05]

Avez vous des questions ?



Agenda

1 Étude de cas : *Network* ↔ *Graph*

2 Introduction aux transformations de modèle

3 Kermeta : Un Kernel de Métamodélisation

4 Tissages d'aspects & Contraintes dans Kermeta

5 Rendre un modèle transformable . . .

6 Implémentation de transformations : *Graph* → *X*

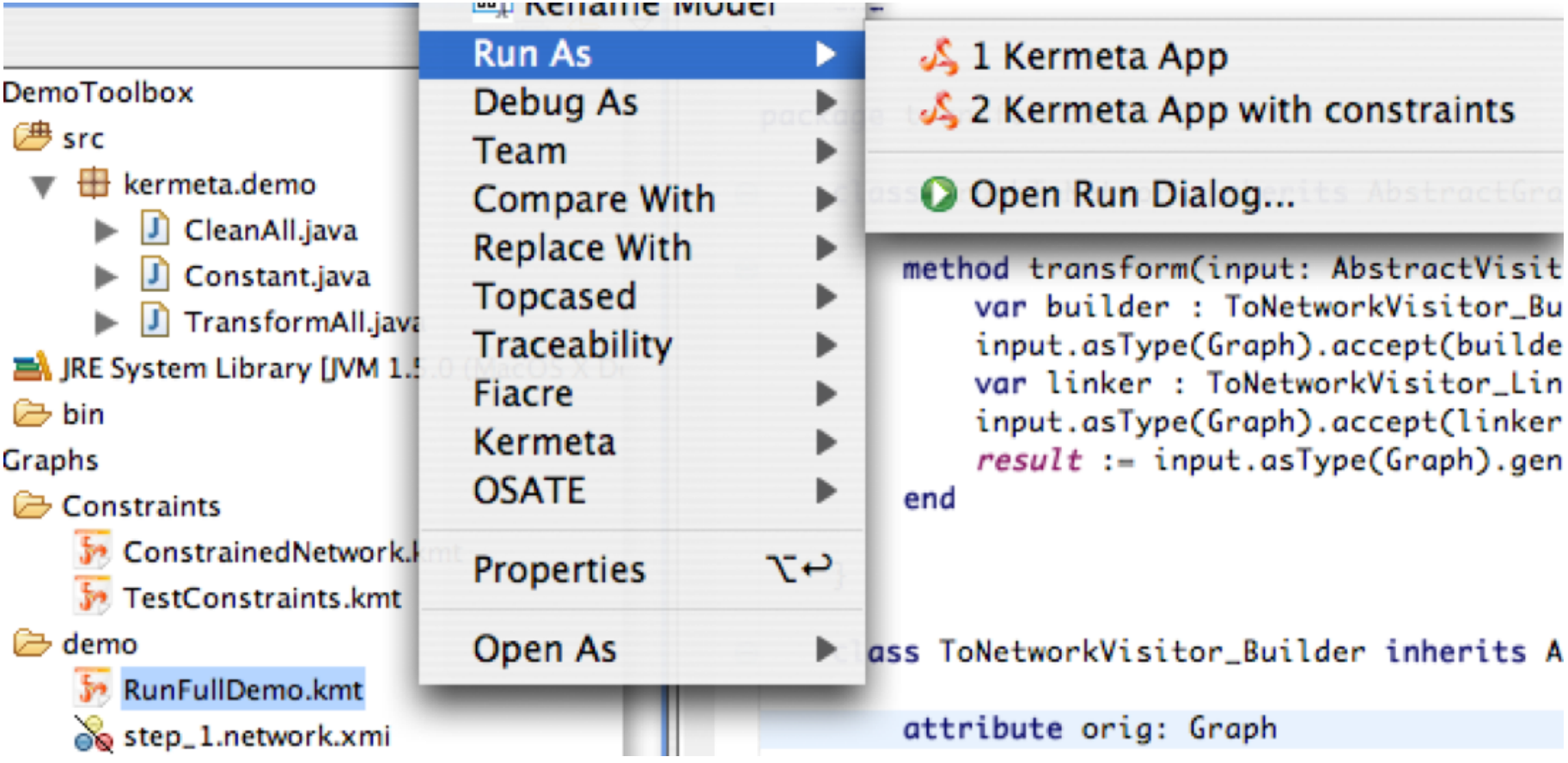
7 Conclusions

8 Exemple : Opérationalisation dans Eclipse

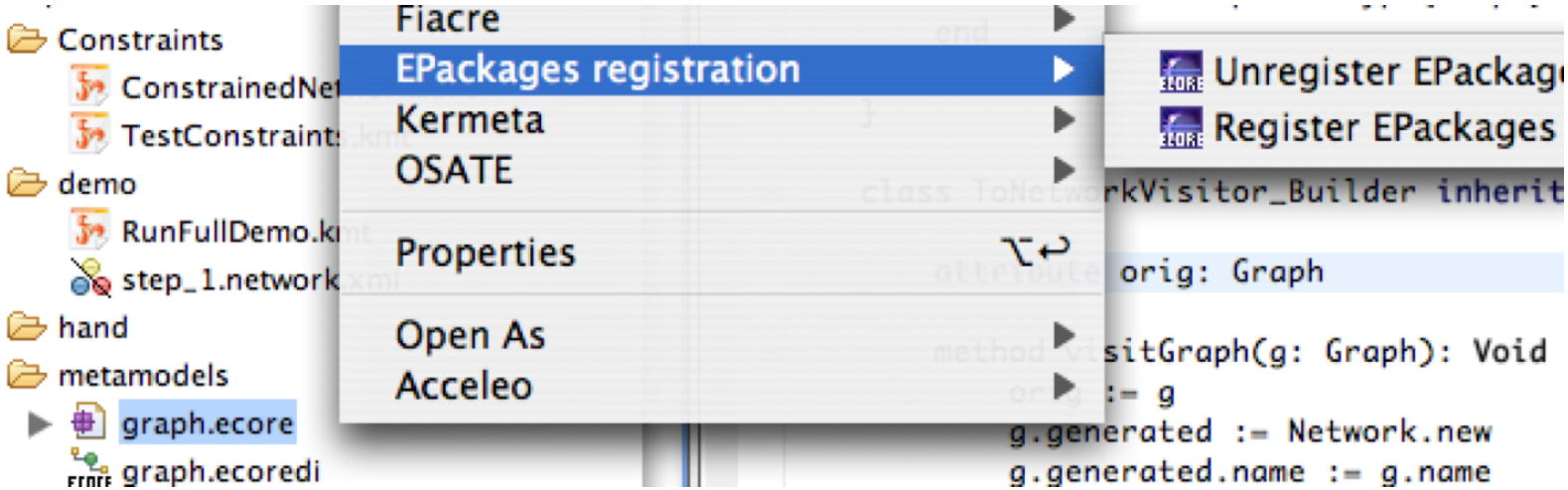
ModuleIDM : Workspace de transformation

- Récupérez le workspace compressé depuis le site web du cours
- Effectuez l'importation dans Eclipse
- Résultat de l'importation :
  - DemoToolBox : codes java aidant la démonstration
  - Graphs : Projet Kermeta utilisé
    - Constraints : *design–by–contract*
    - demo : Démonstration de bout en bout
    - hand : quelques codes "à la main"
    - metamodels : métamodèles ECore, aspects visitables
    - Templiers : exemples sur le réseau du bâtiment
    - transformations : implémentation des transformations
- Éditez Constants.java dans le projet DemoToolBox

Lancement d'un exécutable Kermeta



Enregistrement des métamodèles dans le registre global

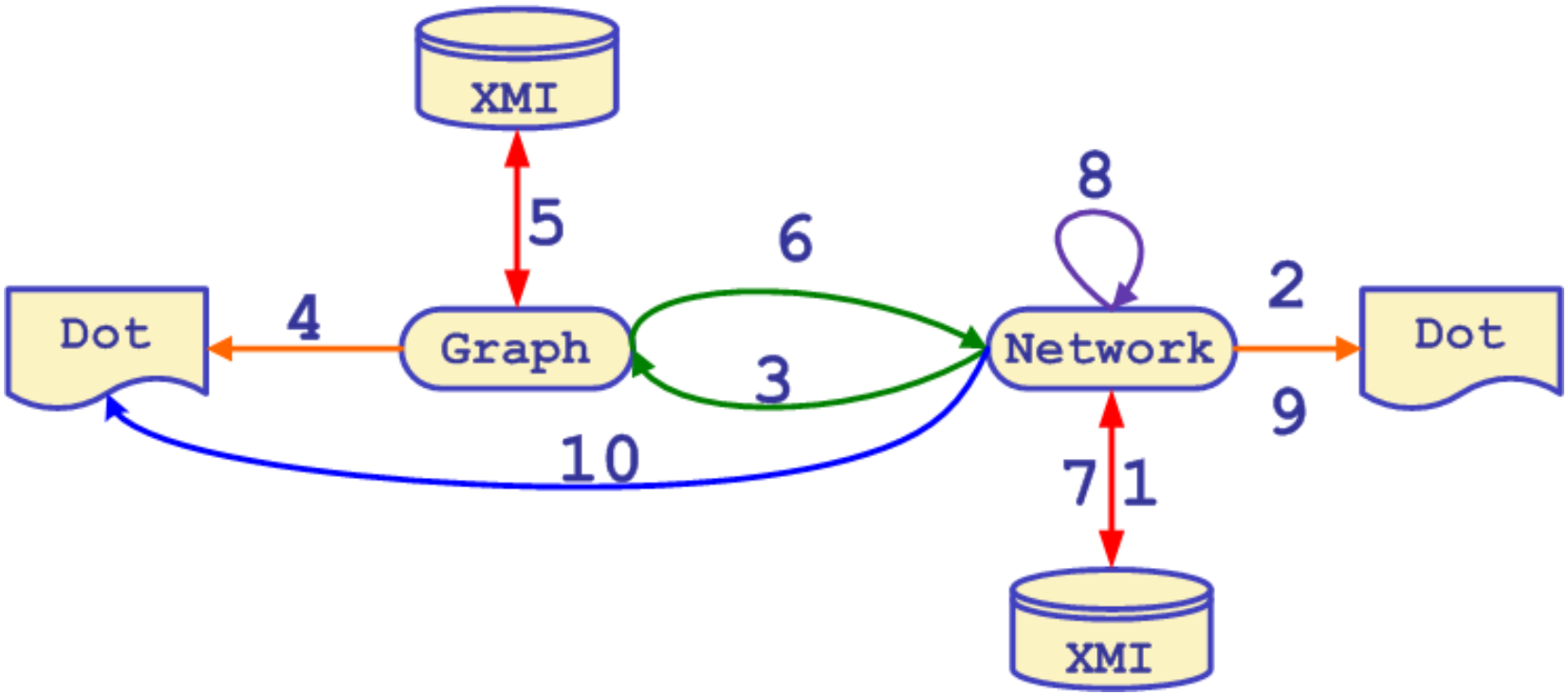


- Cette action déclare globalement le métamodèle
- Les fichiers XMI produits ne sont pas utilisables sinon
- Attention aux conflits de noms entre métamodèles ! !

Déroulement de la démonstration en 10 étapes

1. Chargement d'un modèle de réseau sérialisé XMI
2. Transformation du réseau en code graphviz
3. Transformation du réseau en graphe
4. Transformation du graphe en code graphviz
5. Sérialisation XMI du graph
6. Transformation du graph en un nouveau réseau
7. Transformation du nouveau réseau en code graphviz
8. Calcul d'adressage sur ce nouveau réseau
9. Transformation en code graphviz du nouveau réseau
10. Transformation composite vers du Graphviz via un graphe

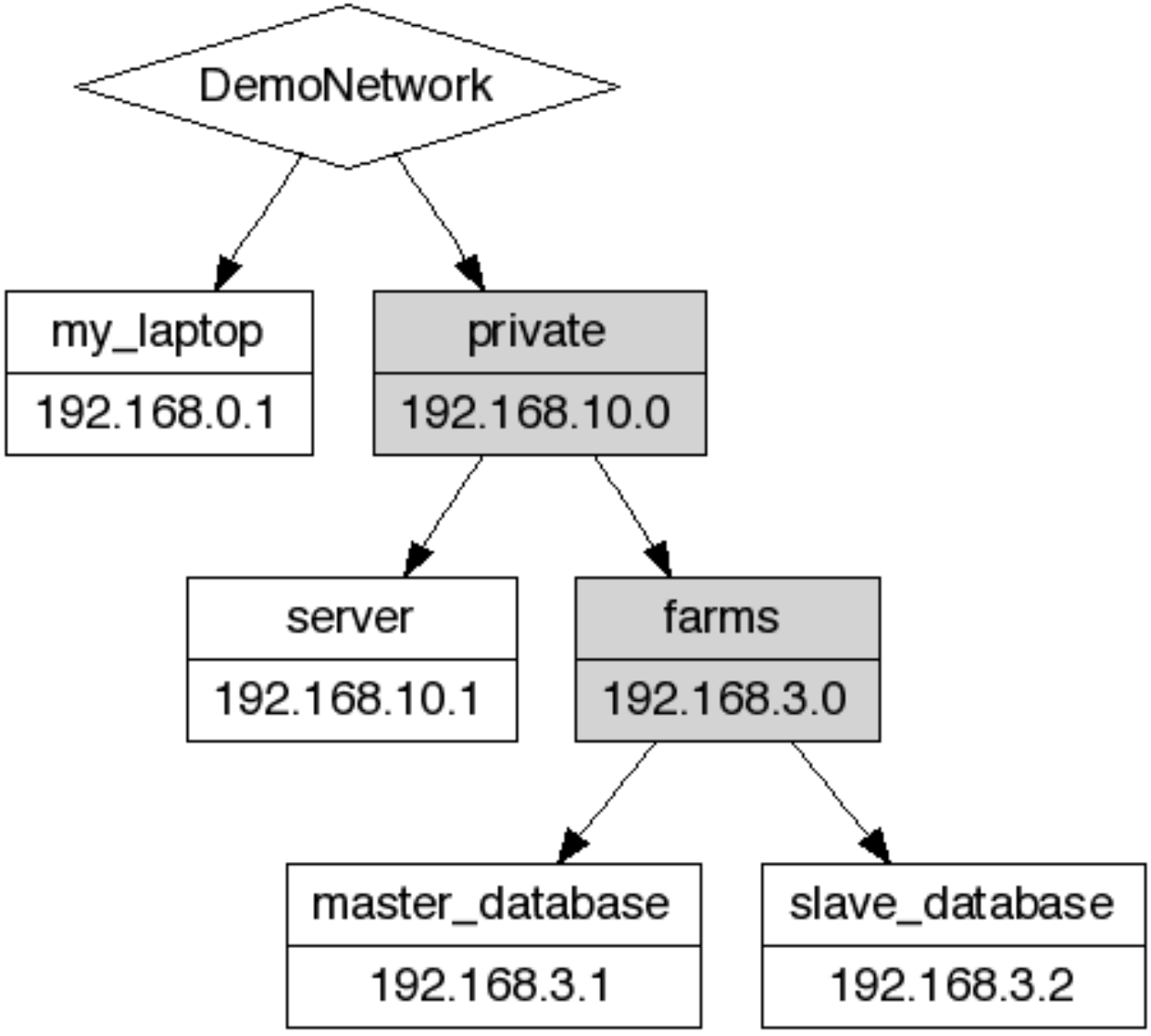
Version graphique



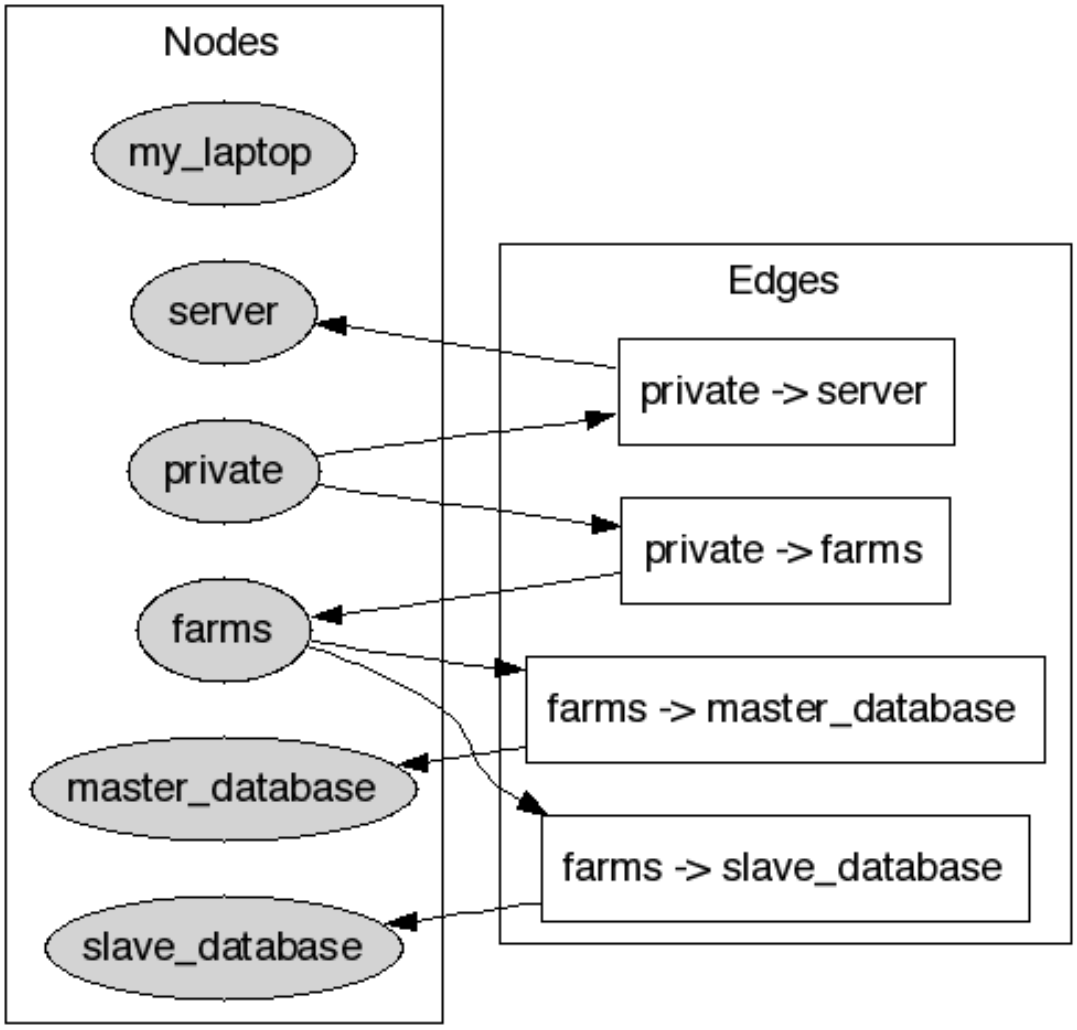
```
>>>> Running Graph & Network Kermeta demo
>>>> Contacts : Sebastien Mosser <mosser@polytech.unice.fr>
>>>> Transformation Set :
>> Step 1 : From XMI to Network
  - Input  : 'platform:/.../demo/step_1.network.xmi'
  - Output : network [network::Network:20238]
>> Step 2 : From Network to Graphviz dot code
  - Input  : network [network::Network:20238]
  - Output : 'platform:/.../demo/step_02.network.dot'
>> Step 3 : From Network to Graph
  - Input  : network [network::Network:20238]
  - Output : graph   [graph::Graph:20888]
>> Step 4 : From Graph to Graphviz dot code
  - Input  : graph [graph::Graph:20888]
  - Output : 'platform:/.../demo/step_04.graph.dot'
>> Step 5 : From Graph to XMI
  - Input  : graph [graph::Graph:20888]
  - Output : 'platform:/.../demo/step_05.graph.xmi'
```

```
>> Step 6 : From Graph to Network
  - Input  : graph   [graph::Graph:20888]
  - Output : network [network::Network:21639]
>> Step 7 : From Network to Graphviz dot code
  - Input  : network    [network::Network:21639]
  - Output : 'platform:/.../demo/step_07.network.dot'
>> Step 8 : From Network to Network [adress computation]
  - Input  : network    [network::Network:21639]
  - Output : network    [network::Network:21639]
>> Step 9 : From Network to Graphviz dot code
  - Input  : network    [network::Network:22284]
  - Output : 'platform:/.../demo/step_09.network.dot'
>> Step 10 : From Network to Graphviz dot code
  - Input  : network    [network::Network:22284]
  - Output : 'platform:/.../demo/step_10.graph.dot'
>>>> Ending transformations examples
```

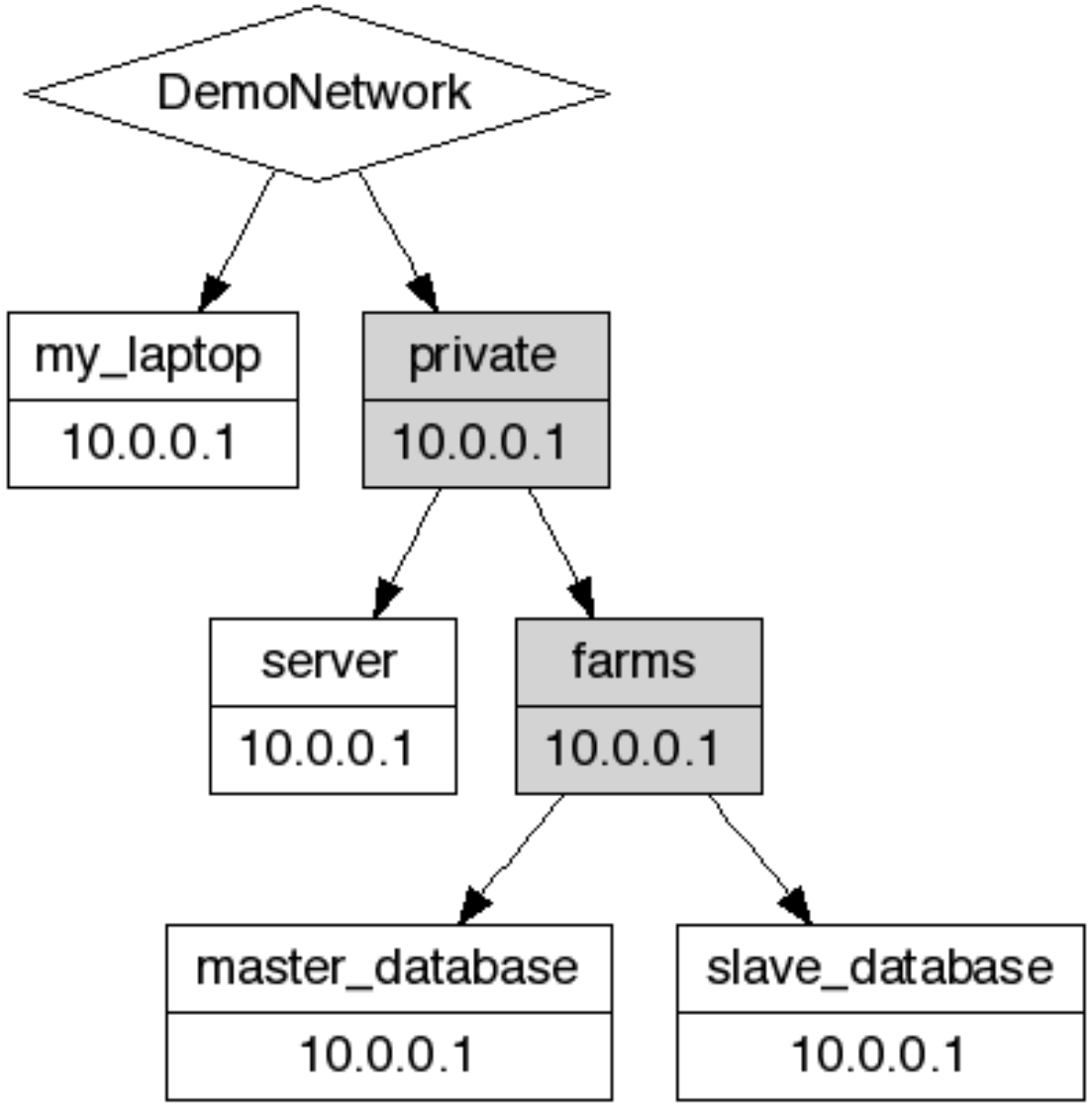
Réseau de départ : `step_2.network.png`



Graphe obtenu par transformation : `step_4.graph.png`



Retour dans le domaine réseau : `step_7.network.png`



Résultat après calcul d'adressage : `step_9.network.png`