SE206: Domain-Specific Modeling Languages and Code Generation Techniques

Dominique Blouin

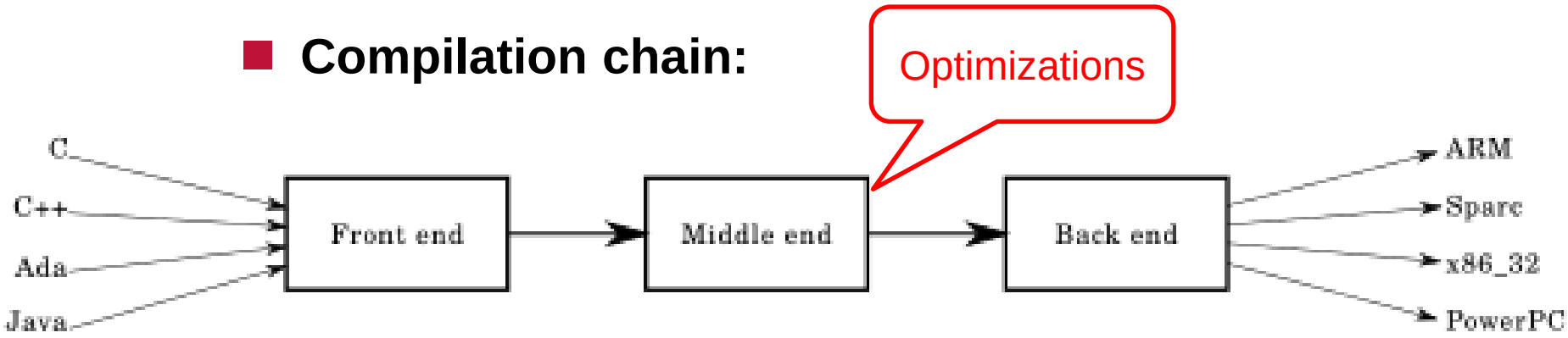
dominique.blouin@telecom-paris.fr

Content

- **Model-Based Engineering**
- Domain-Specific Languages in a Nutshell
- Overview of Eclipse Modeling Framework (EMF)
- Creating Ecore Metamodels
- Model Code Generation
- Exercise: Code Generation for Directed Acyclic Graph DSL

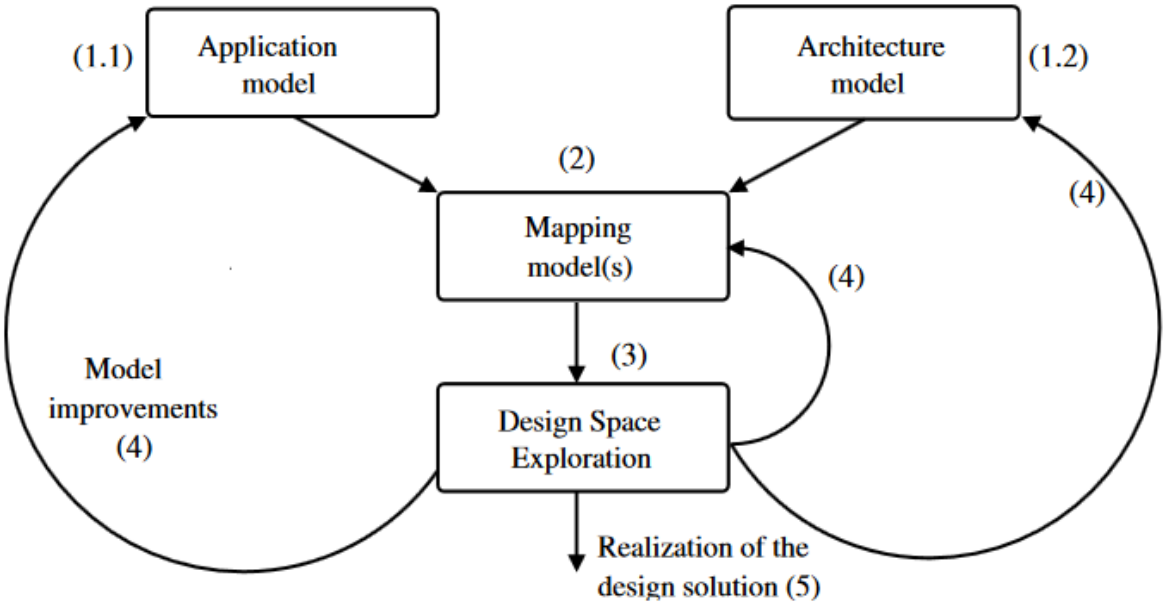
From Code Compilation to Model-Based Engineering (including Code Generation)

- **Compilation chain:**

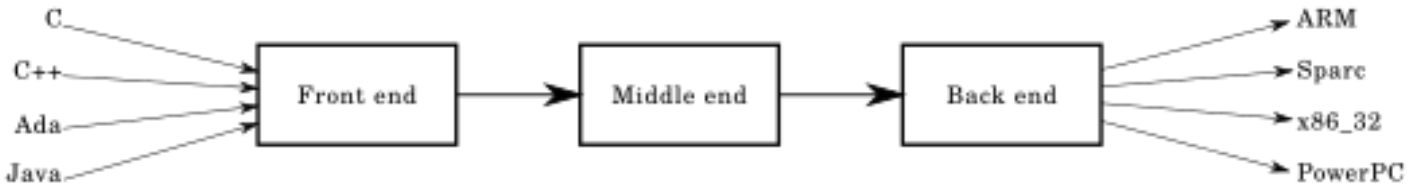


- **Key points:**
  - Increase level of abstraction
  - Execution platform independent

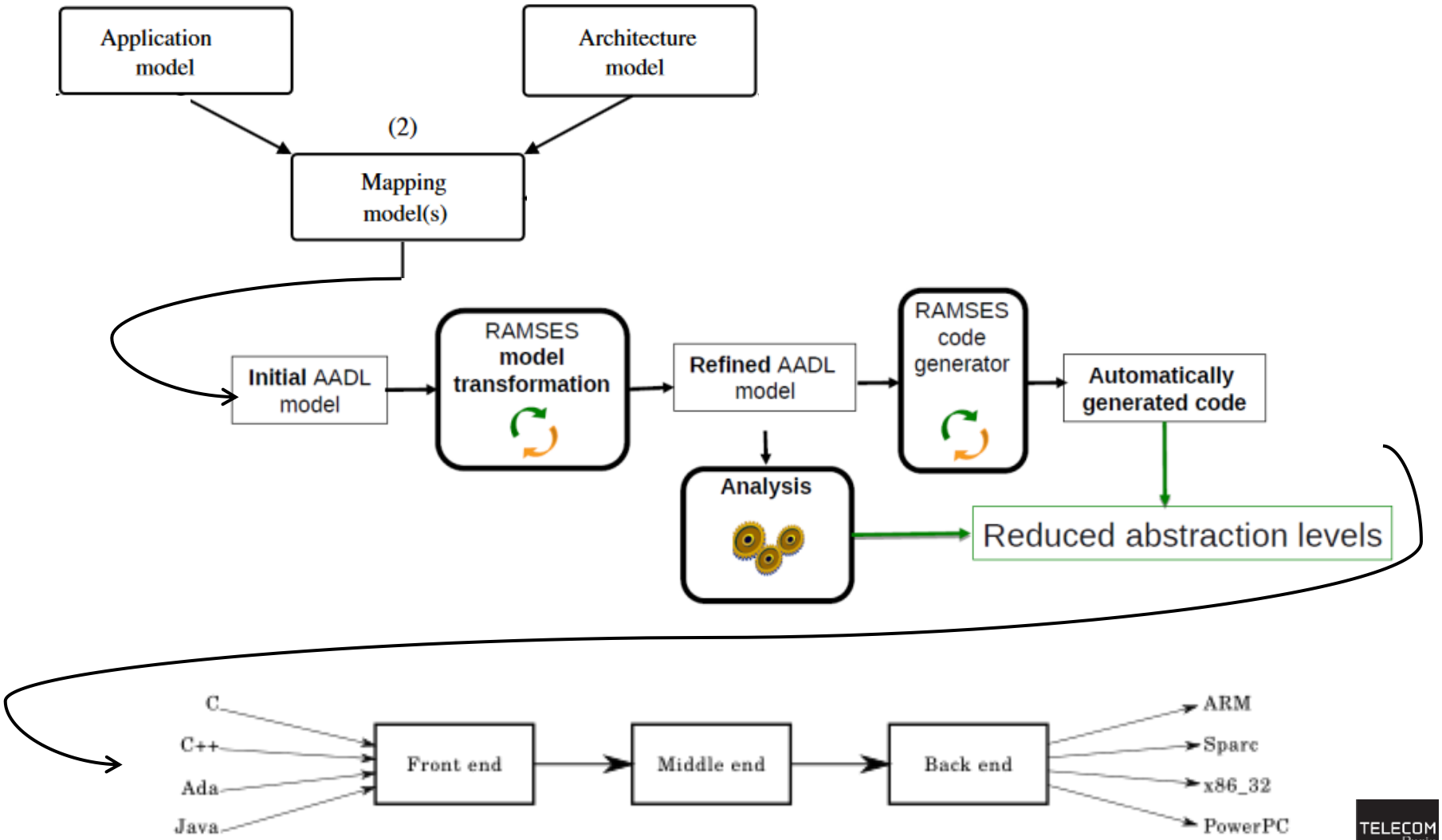
New Paradigm: Model-Based Engineering



- **Higher level of abstraction**
- **Input: Domain-Specific Modeling Languages (DSML)**
- **Output: Program code**



RAMSES (Refinement of AADL Models for the Synthesis of Embedded Systems)



Content

- Model-Based Engineering
- **Domain-Specific Languages in a Nutshell**
- Overview of Eclipse Modeling Framework (EMF)
- Creating Ecore Metamodels
- Model Code Generation
- Exercise: Code Generation for Directed Acyclic Graph DSL

Domain Specific Languages (DSL) in a Nutshell

- **Computer language specialized to a particular application domain**
- **In contrast, general-purpose languages (GPL) are broadly applicable across domains**
- **Language-oriented programming paradigm:**
  - Creation of special-purpose languages for expressing problems
  - Becomes part of the problem-solving process
  - Provides vocabulary specific to the domain
  - Allows a particular type of problem or solution to be expressed more clearly and easily than with a GPL

Kinds of GPLs

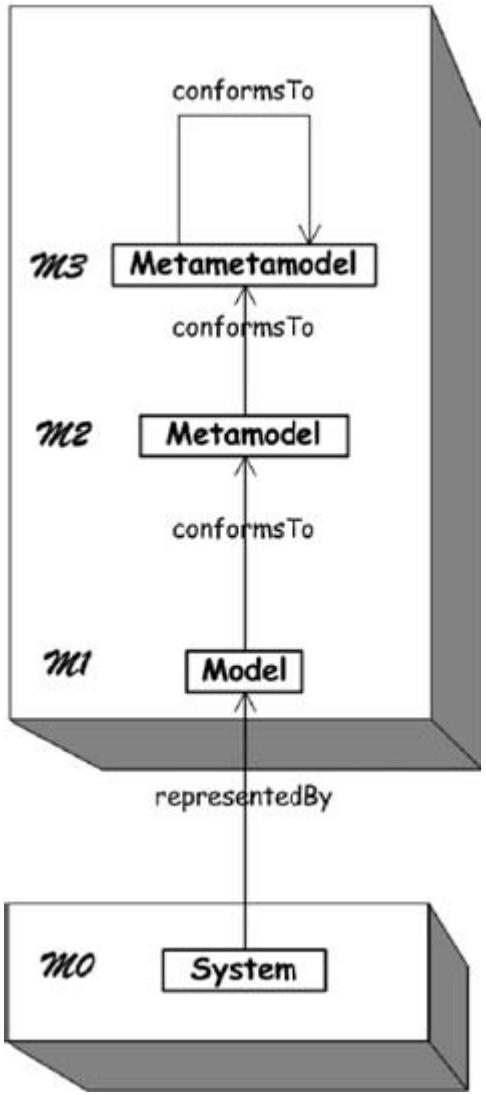
- **General-purpose programming languages**
  - C/C++, Java, C#, Python, etc.
- **General-purpose markup languages**
  - XML
- **General-purpose modeling languages**
  - Unified Modeling Language (UML)
  - Can be specialized with its built-in extension mechanism (profile)

Modeling Languages vs Programming Languages

- **Programming Languages**
  - Set of instructions to be executed by a computer
  - Used to create programs that implement specific functions / algorithms

- **Modeling languages:**
  - Used for the *specification* of a system
  - Describe a system at a much higher level of abstraction than a programming language (e.g., architecture, components; etc.)
  - Used to **specify**, **analyze** and **generate** executable code

OMG (Object Management Group) Model-Driven Architecture (MDA) Metamodeling Layers



- **Ecore**
- **Directed Acyclic Graph DSL**
- **Directed Acyclic Graph models**
- **Deployed running tasks**

Programs versus Models

- A program conforms to its language
- The language is defined by a grammar
- Program ➔ Model
- Programming Language ➔ Metamodel
- Grammar ➔ Meta-metamodel
- Compilation ➔Model transformation

Domain Specific Modeling Languages (DSML)

- **Specified by metamodels or grammars (textual language, e.g., AADL)**
- **Provide modeling and domain-specific capabilities**
- **Modeling:**
  - Specify and validate a design before implementation
  - **Generate** the executable code of a system
- **Domain-specific:**
  - Provide the vocabulary and concepts that closely matches the specific needs of users
  - Simplify usage and augment productivity
- **More and more used in industry:**
  - Simulink, Scade, AUTOSAR, SysML, AADL, etc.

Qualities of a DSL: *Requirements for Domain-Specific Languages*, Kolovos et al., 2006

- **Conformity: should reflect the domain as much as possible**
- **Orthogonality: each construct in the language should be used to represent exactly one distinct concept in the domain**
- **Supportability: should be feasible to provide DSL support via tools**
- **Integrability: DSL and its tools can be used in concert with other languages and tools**
- **Extensibility: DSL and its tools can be extended to support additional constructs and concepts**

Qualities of a DSL (continued)

- **Longevity: should be used and useful for a non-trivial period of time**
- **Simplicity: should be as simple as possible to express the concepts of interest and support users and stakeholders in their preferred ways of working**
- **Quality: provide general mechanisms for building high quality systems. E.g.: include constructs for improving reliability, security, safety, etc.**
- **Scalability: provide constructs to help manage large-scale descriptions**
- **Usability: e.g.: space economy, accessibility, understandability, etc.**

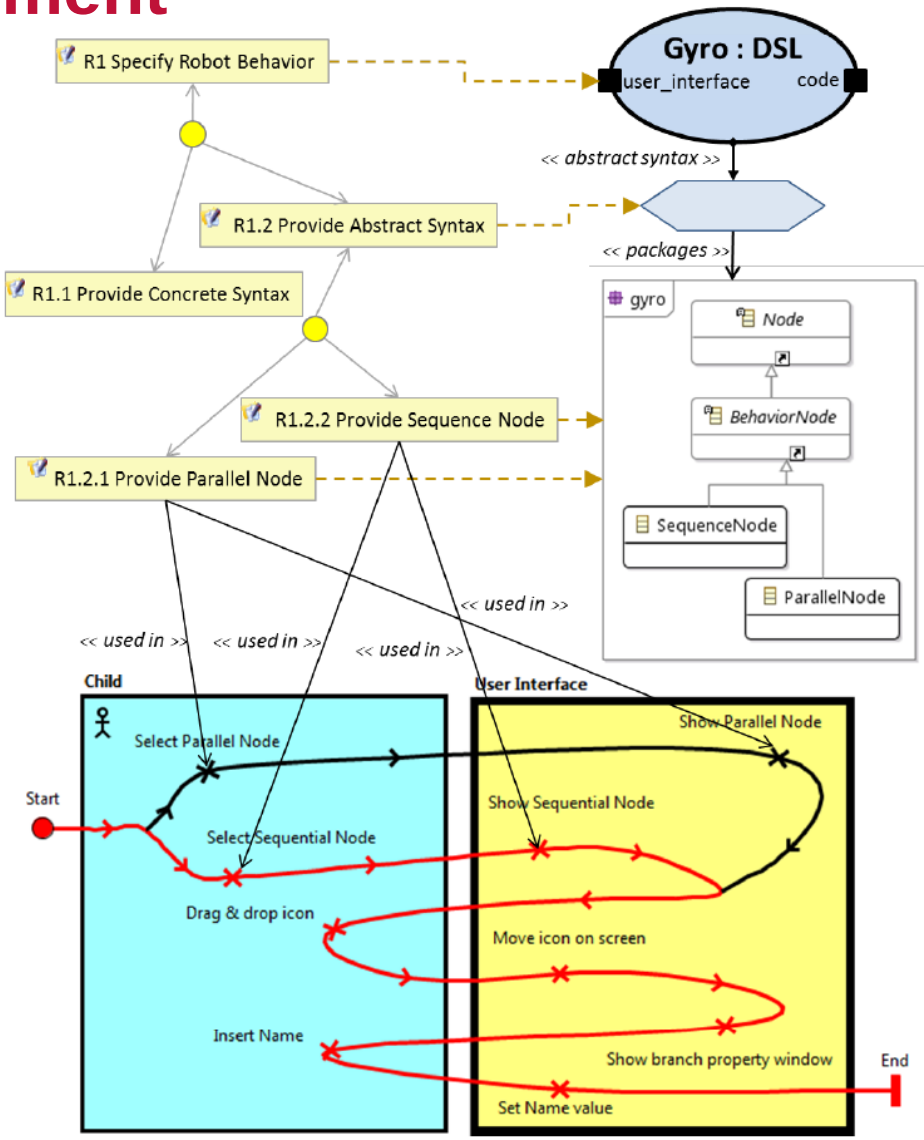
Requirements Engineering for Usability-Driven DSL Development

A Requirements Engineering Approach for Usability-Driven DSL Development

Ankica Barišić
NOVA-LINCS, FCT/UNL
Caparica, Portugal
a.barisic@campus.fct.unl.pt

Dominique Blouin
LTCI Lab, Telecom ParisTech
Paris, France
dominique.blouin@
telecom-paristech.fr

Vasco Amaral
Miguel Goulão
NOVA-LINCS, FCT/UNL
Caparica, Portugal
(vma,mgoul)@fct.unl.pt



Content

- Model-Based Engineering
- Domain Specific Languages in a Nutshell
- **Overview of Eclipse Modeling Framework (EMF)**
- Creating Ecore Metamodels
- Model Code Generation
- Exercise: Code Generation for Directed Acyclic Graph DSL

EMF in a Nutshell

- http://eclipse.org/modeling/emf/
- **A simple, pragmatic approach to meta-modeling**
- **Mature and proven modeling framework (since 2002)**
- **Supported by a very large and active open-source community**
  - De facto standard
- **Surrounded by a huge ecosystem of tools and frameworks**
- **EMF is not UML and not a modeling tool**
  - But modeling tools can be built using EMF
- **Used as the basis for hundreds of applications and DSLs**
  - Papyrus (UML), OSATE (AADL), etc.

Building Blocks of EMF

- **Modeling language and environment**
  - Develop your data/domain model using a metamodel (Ecore)
  - Can be imported from existing specifications (UML, XSD, annotated Java classes)
- **Code Generator**
  - Generation of high-quality Java API
  - Solving issues such as bidirectional references and containments
- **Framework for processing models**
  - Validation and model persistence
  - Generation of simple tree, graphical and form editors
  - Change notifications
  - Command-based manipulation
  - Generic API and reflection mechanisms

Key Characteristics of EMF

- **Modeling language and development environment**
  - Focusing on the essentials
  - Pragmatic and small simple meta-modeling language
- **Extensible and high-quality APIs**
  - Generated code is built to be extended
  - Separation of interfaces and implementation
  - Uses several proven design patterns (observer, adapter, factory, etc.)
- **Domain independent and generic**
  - Applicable to any domain
  - Support to process instances generically

Content

- Model-Based Engineering
- Domain-Specific Languages in a Nutshell
- Overview of Eclipse Modeling Framework (EMF)
- **Creating Ecore Metamodels**
- Model Code Generation
- Exercise: Code Generation for Directed Acyclic Graph DSL

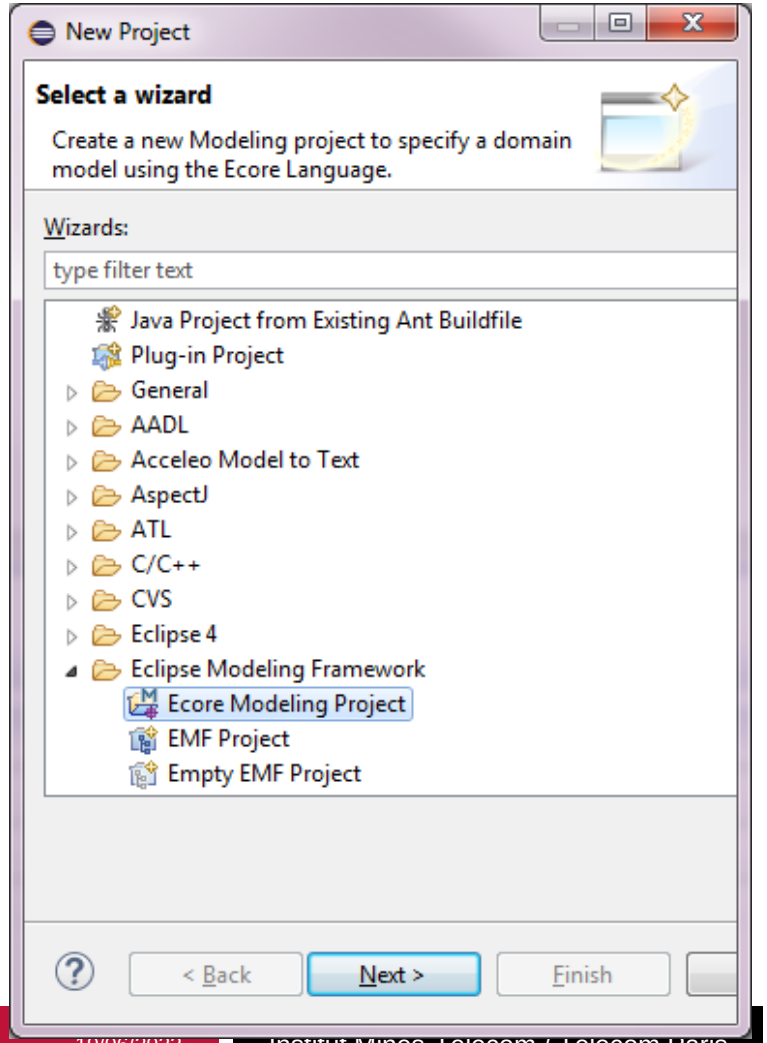
Instructions pour la réalisation du TP

- **Exécuter Eclipse tel que vous l'avez installé sur votre machine**
  - Voir instructions ici: https://se206.wp.imt.fr/tp-installation-des-outils-pour-travailler-de-chez-soi/

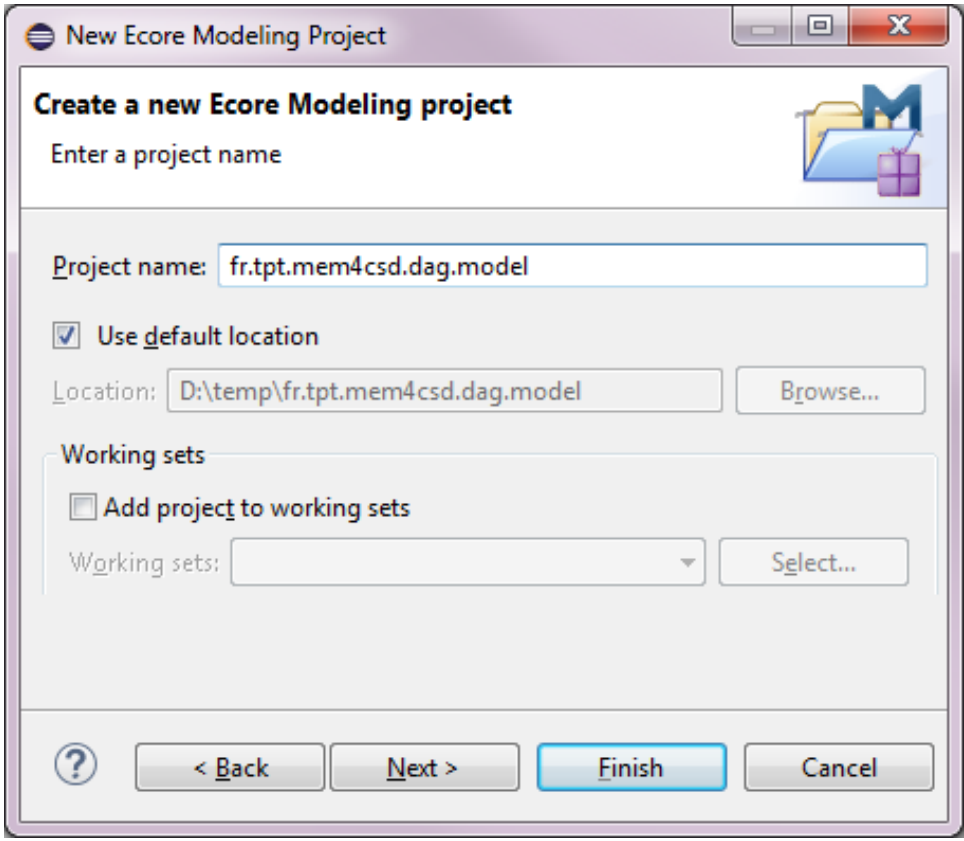
- **Ou connectez-vous sous votre compte Telecom Paris**
  - Dans une fenêtre de commande Linux exécutez les commandes suivantes:
    - Exécuter Eclipse:
      - /comelec/softs/opt/aadl-tools/eclipse-modeling-2022-06-R/eclipse &

Creating an Ecore Class Diagram

- Create a new “Ecore Modeling Project”

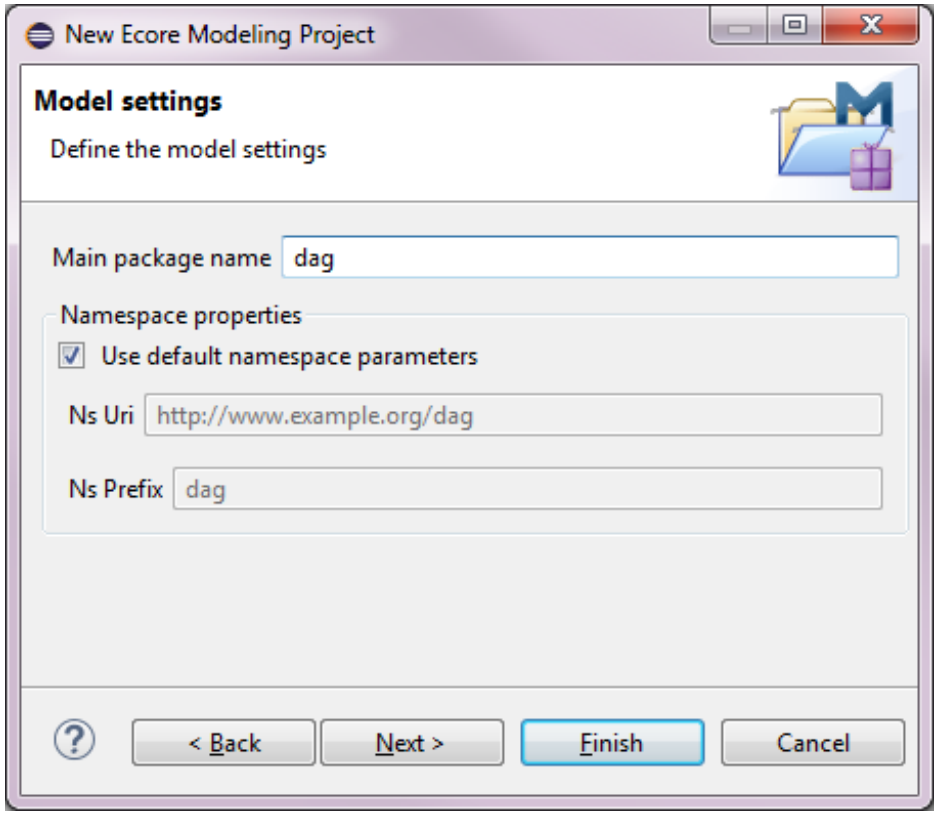


- Create a new “Ecore Modeling Project”

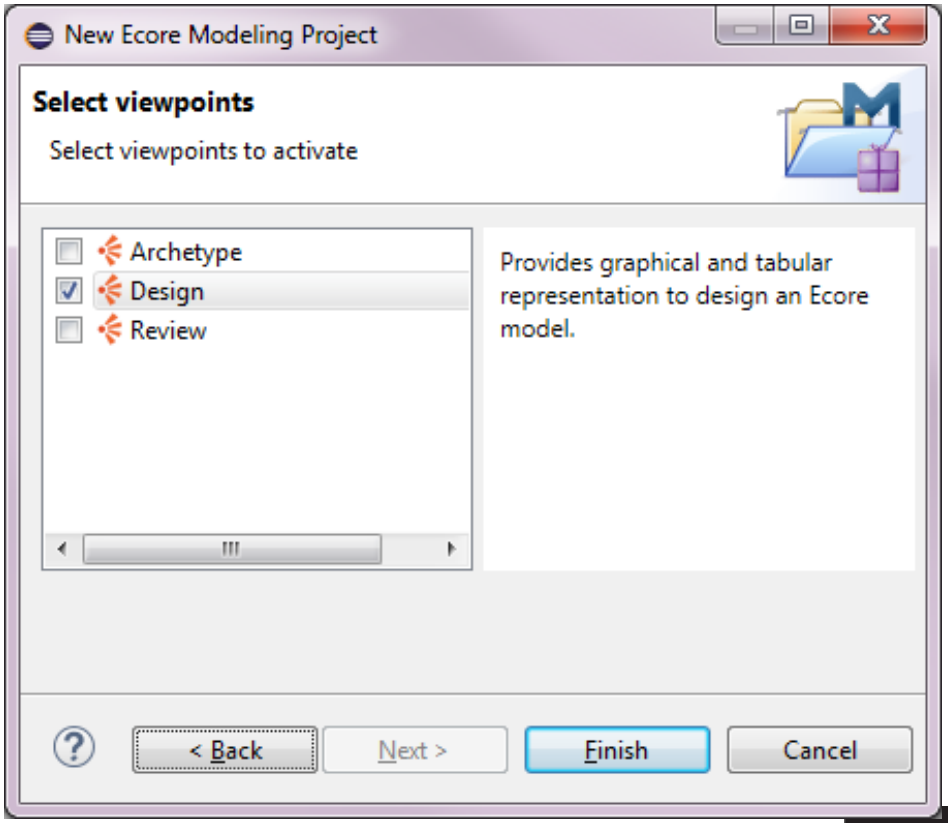


Creating an Ecore Class Diagram

- **Specify package name, Ns Uri, Ns prefix**

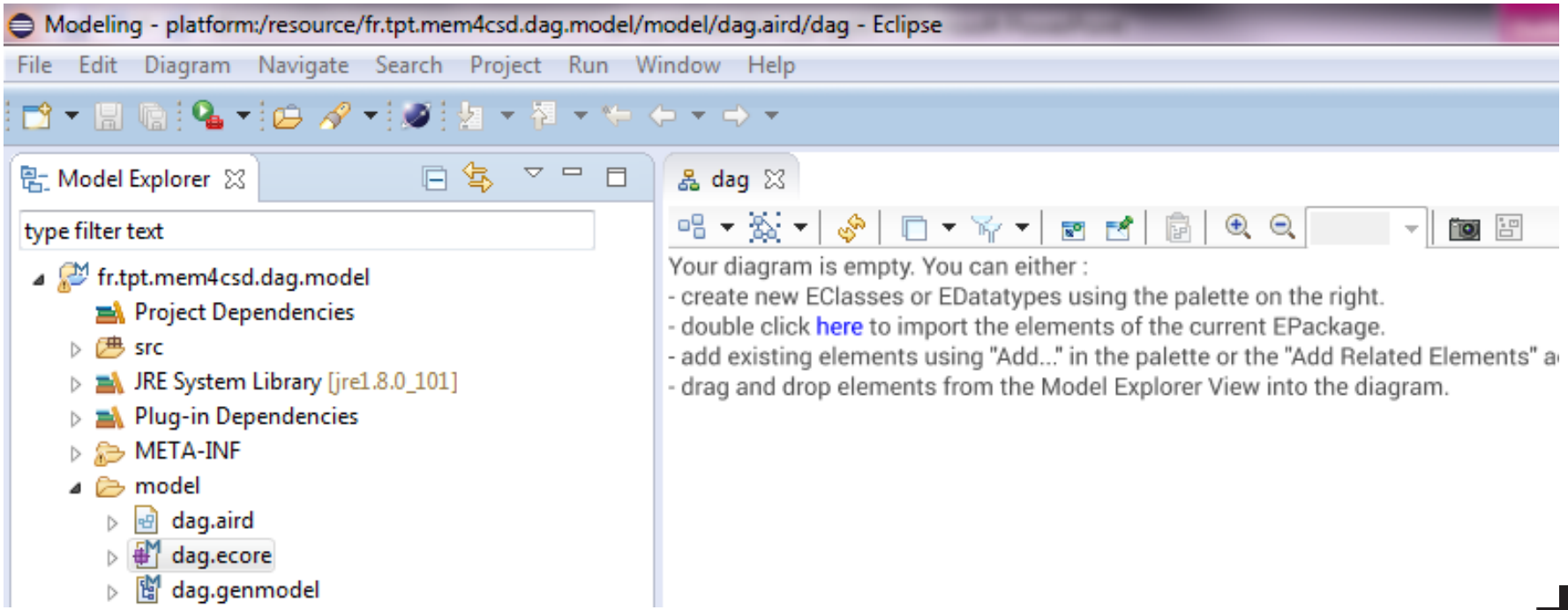


- **Select viewpoint “Design”**

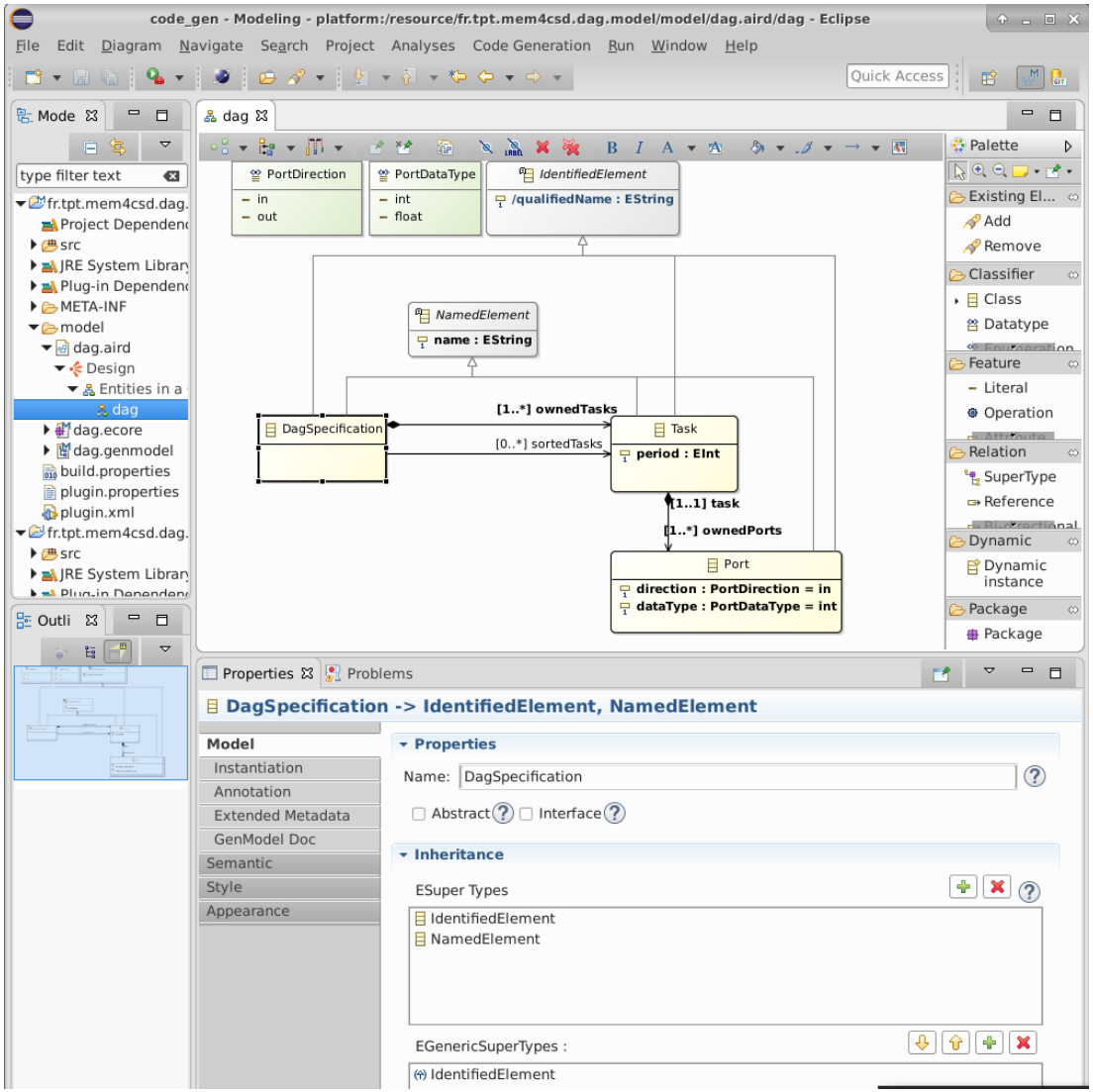


Created Files

- *.ecore = metamodel
- *.aird = viewpoint / class diagram representation
- *.genmodel = generator model for generating model code and tree-based editor

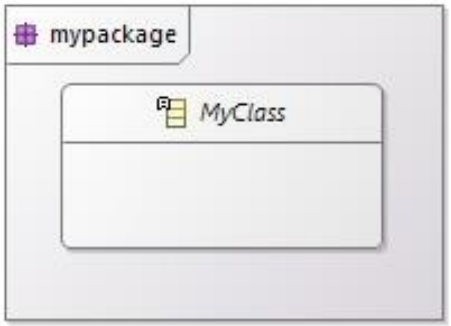


Ecore Diagram Editor



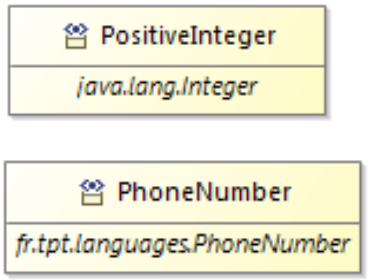
Ecore Concepts: Packages

- **Used to organize classes into groups**
- **Characteristics of well-defined packages:**
  - **High cohesion**: classes strongly dependent on each other that are to be used together (e.g., for a domain) should be grouped into a common package
  - **Low coupling**: packages should be loosely coupled and as independent as possible form each other to be reusable in an independent manner
- **Graphical representation:**
  - Main package as canvas of graphical editor
  - Sub-packages as square nodes containing classes



Ecore Concepts: Datatypes

- **Datatypes: represent primitive data types such as integer, real, string, enumerations, etc.:**
  - Cannot declare properties
  - Can create your own data types
  - Specify the equivalent class in the generated code

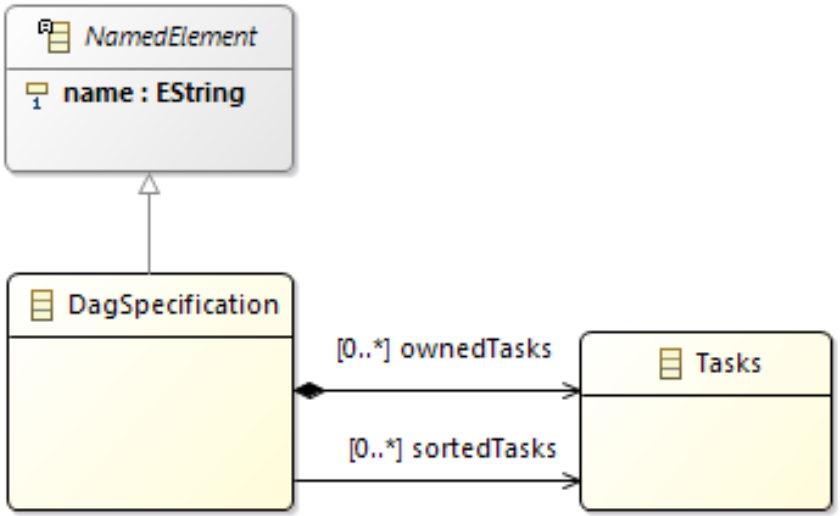


Ecore Concepts: Classes and Properties

- **Classes: represent a set of instance objects sharing the same characteristics (properties)**

- **2 different types of properties:**
  - *Attribute*: property of primitive type or data type (e.g. *name*)
  - *Reference*: property of a class type (e.g.: *ownedTasks*)
  - Cardinalities: max. and min. number of occurrences [min..max]

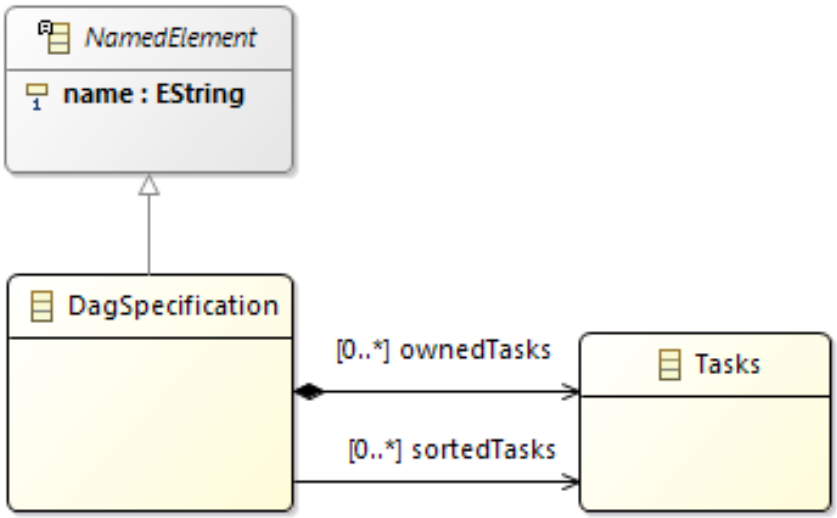
- **Inheritance**
  - A class can subclass many classes
    - Different from Java
  - Inherits properties of extended classes
    - (e.g.: Dag inherits the *name* attribute)



- **Abstract**
  - Cannot be instantiated
  - E.g.: *Named Element*

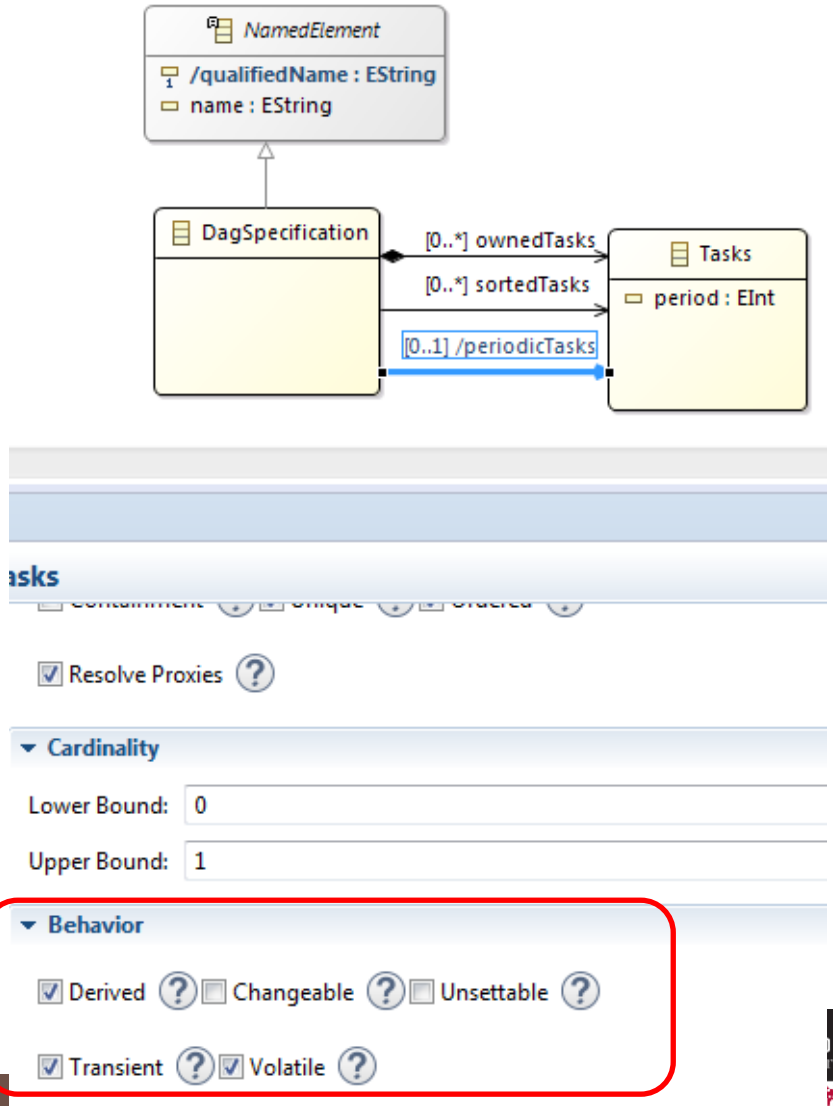
Containment / Composition

- **Reference can be declared as *containment* or not:**
  - Containment: the referred objects cannot exist if the object holding the references does not exist (e.g.: *Task*)
  - Indicated by diamond symbol at beginning of arrow
- **All model objects must be contained by a single object of the model**
- **Define a root container class that will contain model elements (directly or not)**
  - E.g.: *DagSpecification*



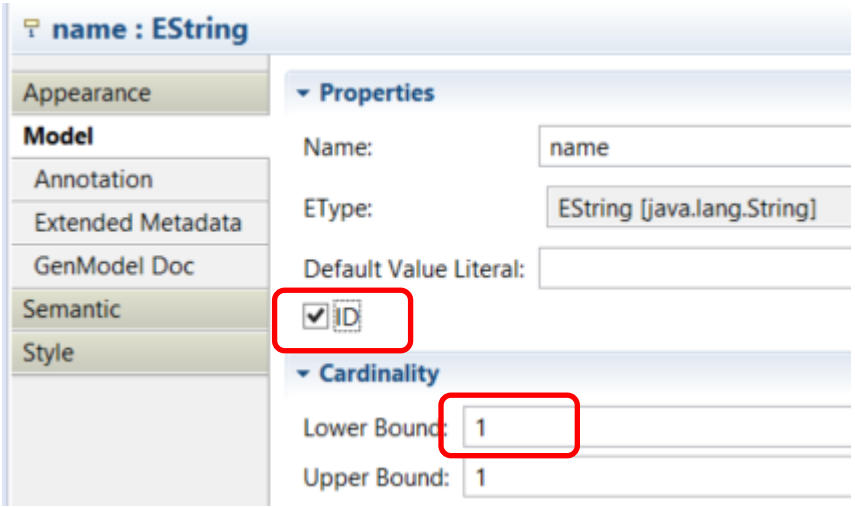
Derived Properties

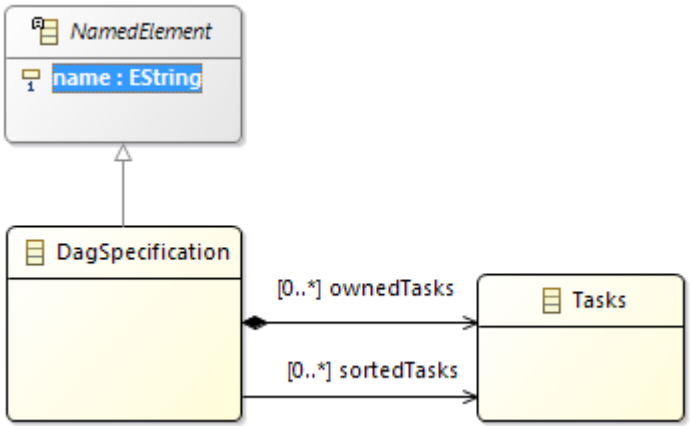
- **Computed like an operation**
- **Code must be provided as:**
  - OCL annotation within the class
  - Or coded manually in the generated code
- **Check “Transient” and “Volatile”**
  - Property will not be serialized
  - No storage (no attribute variable will be declared)



Identification Property

- **IDs: Used by model elements references to refer to other elements within the model**
  - Default behavior: position of the elements are used
  - One attribute can be declared as *ID* and used

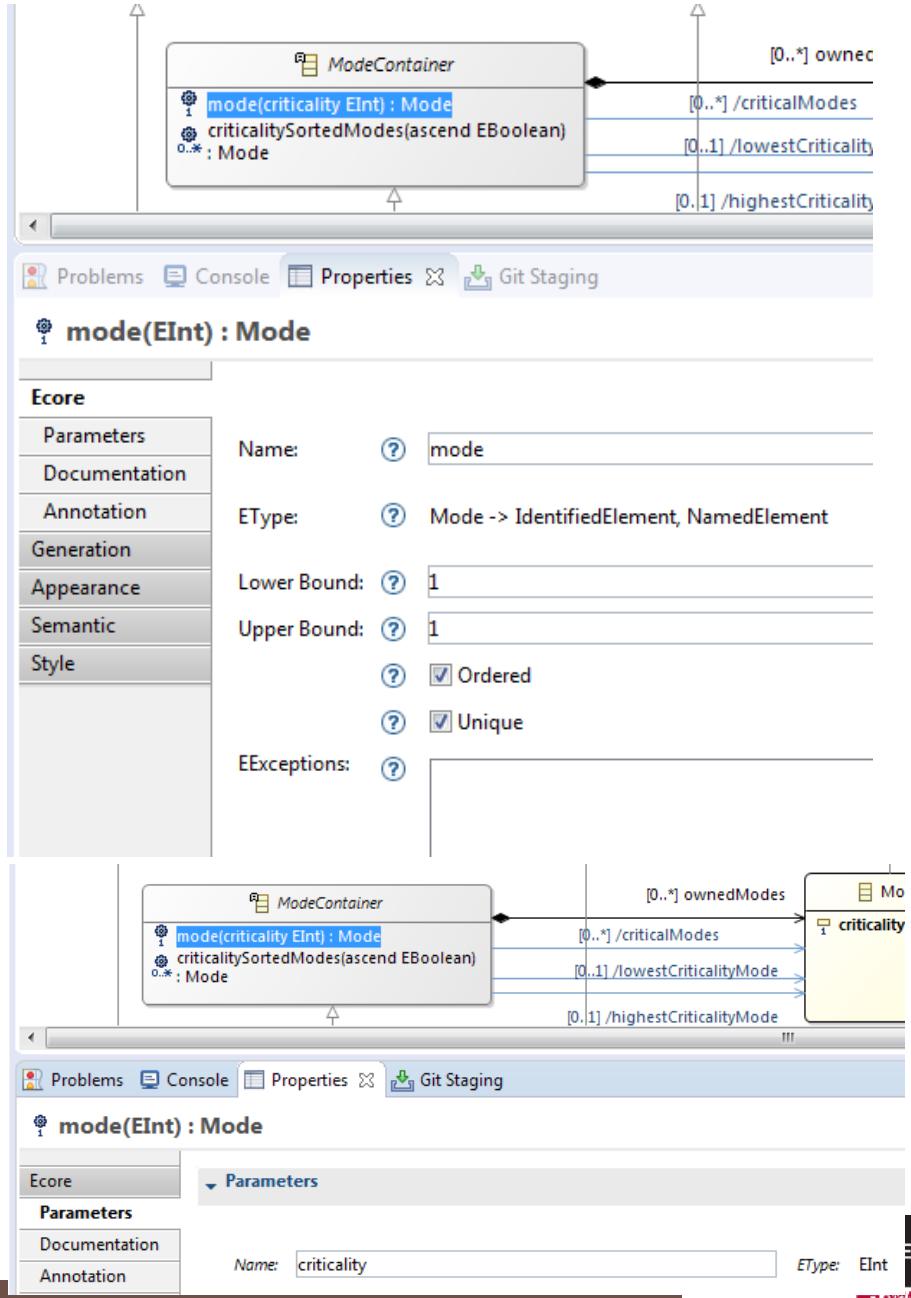




- **Qualified names or UUIDs? How to choose?**
  - If names are used, renaming implies changing all references (e.g. Java)

Ecore Concepts: Operations

- **Provides a return type and parameters with their cardinalities**
- **Implementation must be provided like for derived properties**
  - OCL annotation within the class
  - Or coded manually in the generated code
- **Difference between derived attributes / references:**
  - Operations have parameters



Metamodel Naming Conventions

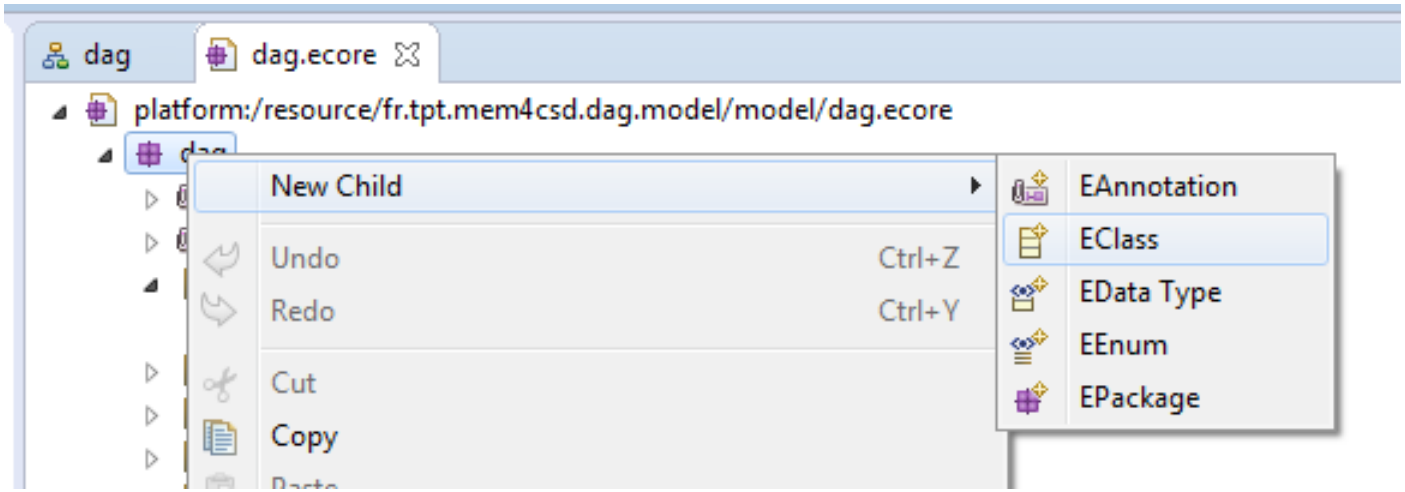
- **Similar to Java**
- **Packages:**
  - Only lower-case characters
- **Classes:**
  - Starts with capital letter and capitalized
    - E.g.: *DagSpecification*
- **References and attributes:**
  - Starts with lower case and capitalized
  - When many ends with *s*
  - Containment starts with *owned*
    - E.g.: *ownedTasks*
- **Operations:**
  - Start with lower case and capitalized

Ecore Metamodel Tree Editor

- **Containment arborescence**
- **Right-click classes to add properties**
- **Edit with bottom property view**

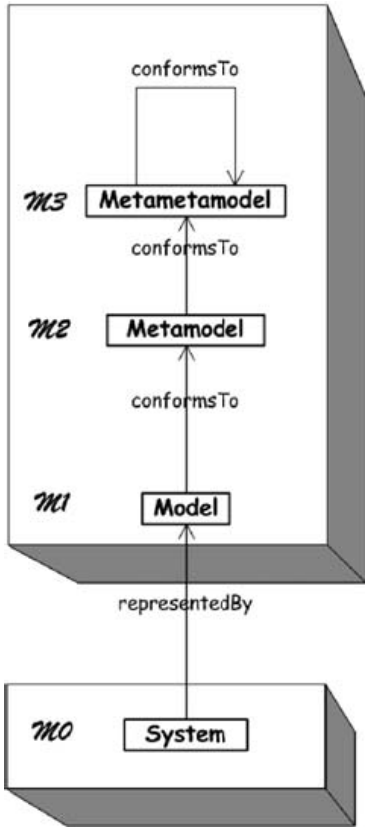
- **Some elements are easier to edit in tree editor**

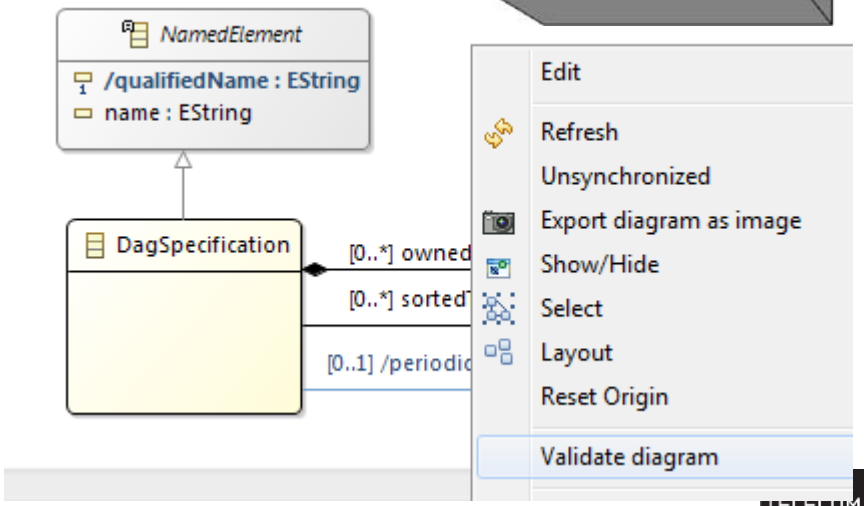
- **Tree editor shows everything of meta-model:**
  - Class diagrams are a view and do not necessarily show everything



Metamodel Validation

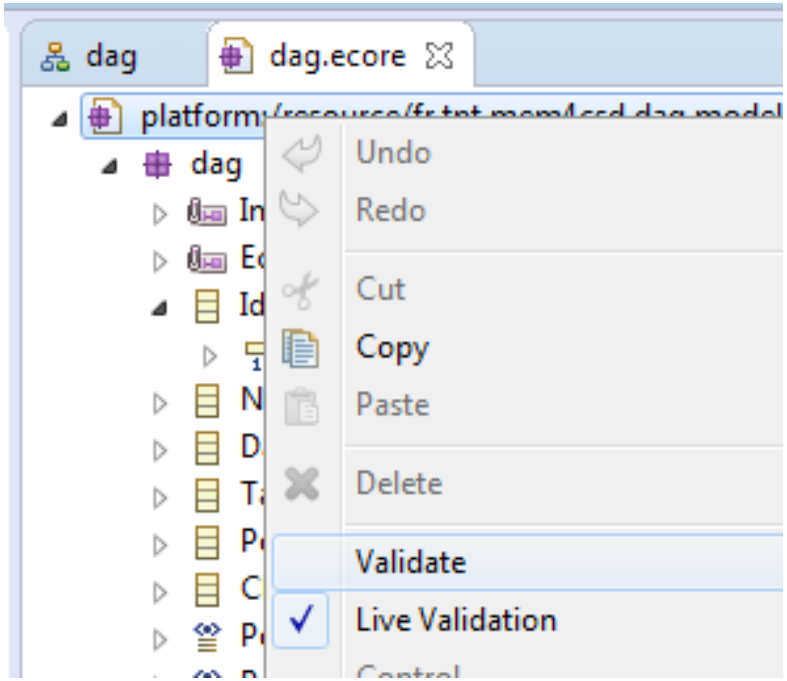
- **A metamodel is also a model conforming to the Ecore meta-meta-model**
- **Therefore, it must be validated to check conformance**
  - E.g.: all classes must have a unique name
- **For diagram editor:**
  - Errors are indicated as the model is changed
  - Validation can also be triggered by user by contextual menu





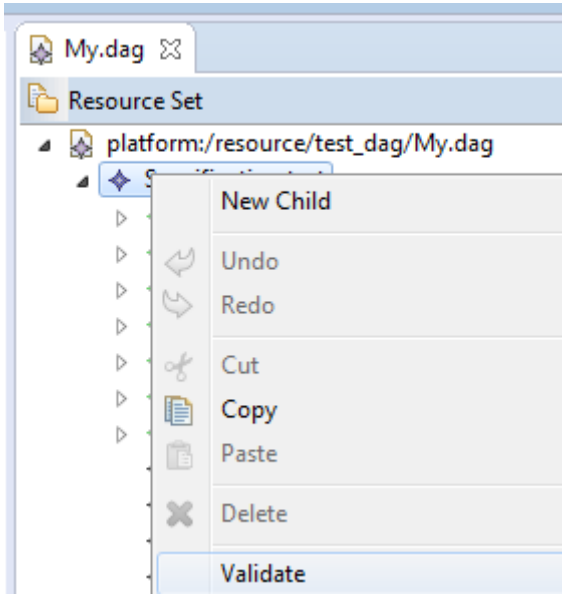
Metamodel Validation from Tree Editor

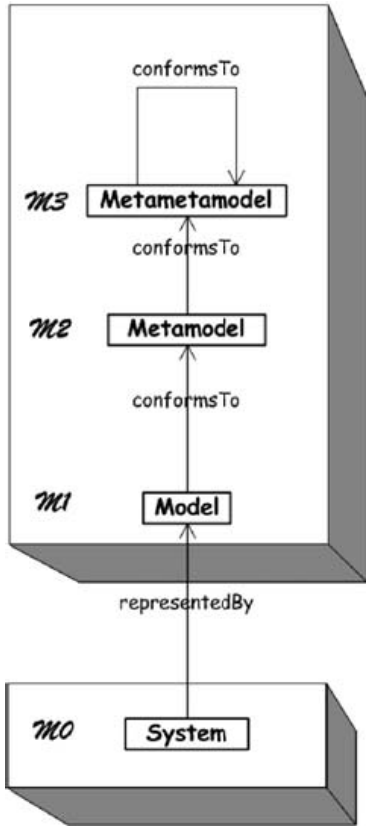
- **Upon request or live validation**



Model Validation

- **Models can also be validated against their metamodels to check conformance:**
  - Check cardinalities of properties
  - Check types of objects referred through properties
  - Custom structural constraints



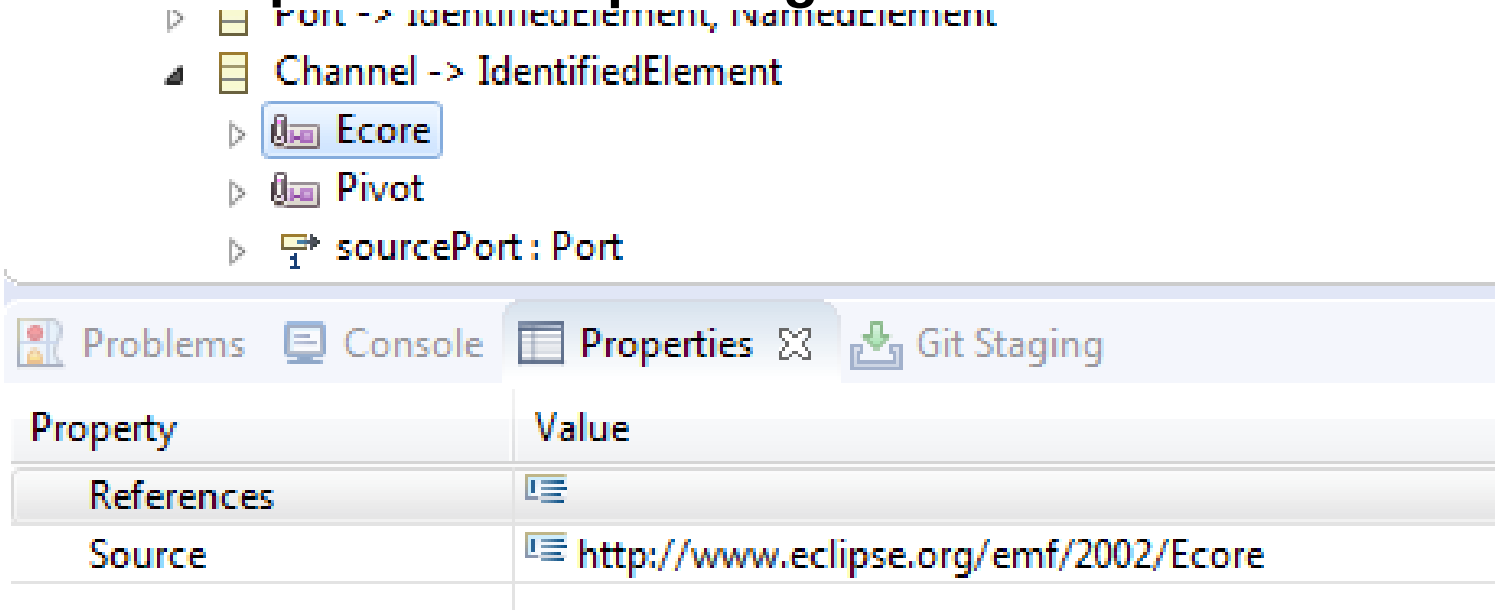


Custom Structural Constraints

- **Additional custom constraints can be added to a metamodel using the predefine constraints annotation**
  - E.g.: Adult class with constraint on age >= 18
- **Constraints can be specified as:**
  - Java code: the content of a specific operation provided in a validator class generated with the model classes must be hand coded
  - Alternatively, the constraint annotation can be extended to specify constraints in OCL
- **These constraints are automatically evaluated when model validation is executed**

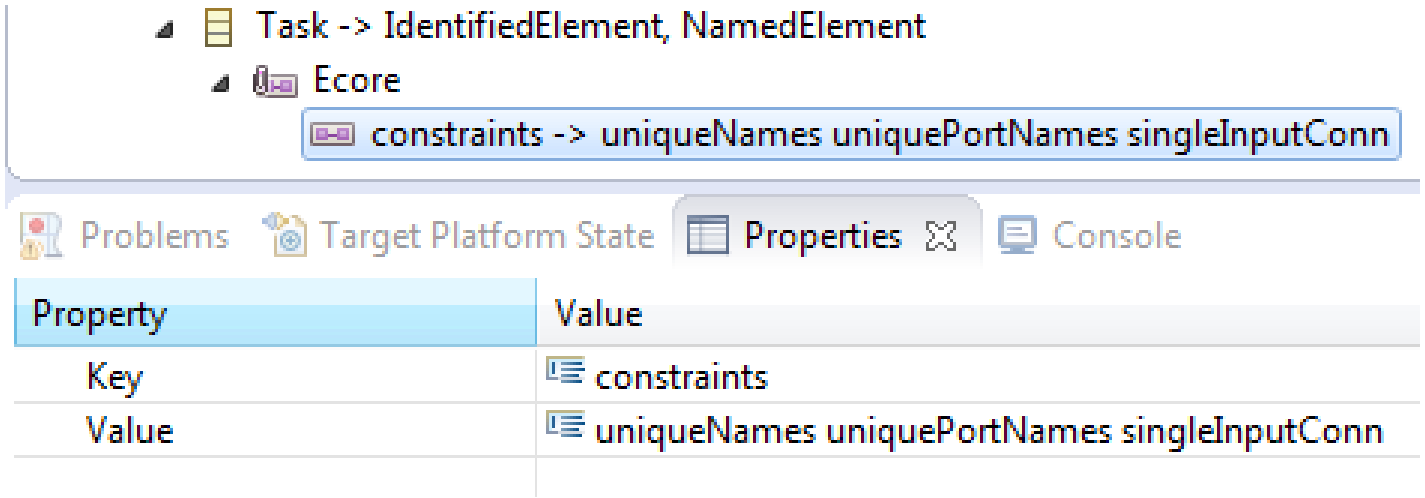
Adding a Structural Constraint Coded in Java (continued)

- Add an Ecore annotation to the class (Ecore tree editor)
- Value the source as "http://www.eclipse.org/emf/2002/Ecore"



Adding a Structural Constraint Coded in Java (continued)

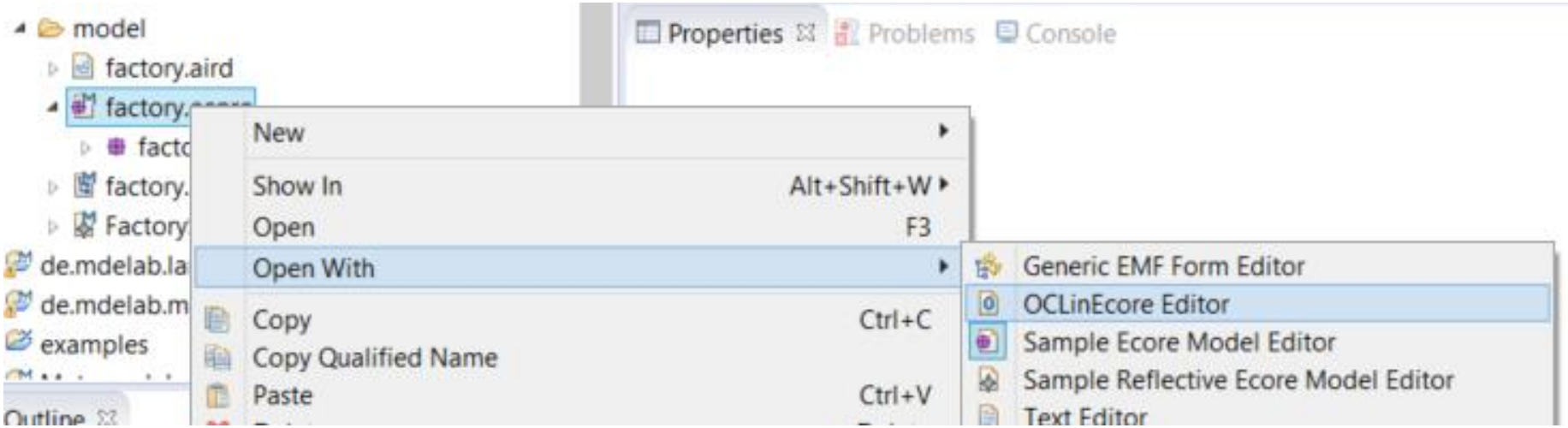
- **Key: “constraints”**
- **Value: lists of constraint names separated by space character**



OCLInEcore Editor

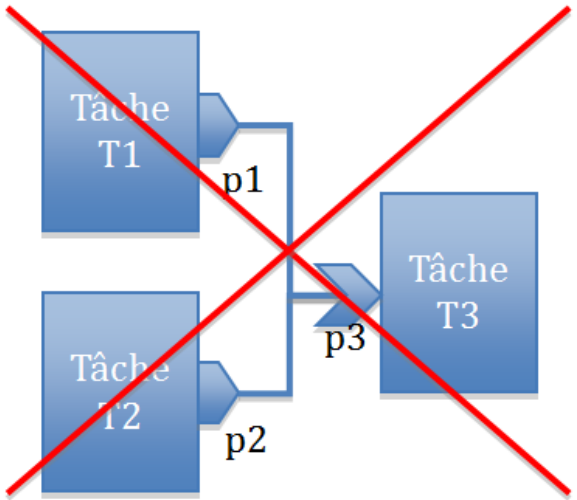
- **The OCL in Ecore Editor can be used to specify a metamodel and OCL constraints for it**

- **Opening the editor:**



Example: Task Class and Uniqueness of Task Names

```
class Task extends IdentifiedElement,NamedElement
{
    attribute period : ecore::EInt[1];
    property ownedPorts#task : Port[+] { ordered composes };
    invariant
    uniqueNames: not NamedElement.allInstances()->exists( element | element <> self and element.name = self.name );
    invariant
    uniquePortNames: self.ownedPorts->forAll( port : Port | not self.ownedPorts->exists( portIt | port <> portIt and portIt.name = port.name ) );
    invariant
    singleInputConn: self.ownedPorts->select( port | port.direction = PortDirection::_in )->
    forAll( inPort | Channel.allInstances()->select( conn | conn.destPort = inPort )->size() < 2 );
}
```



Content

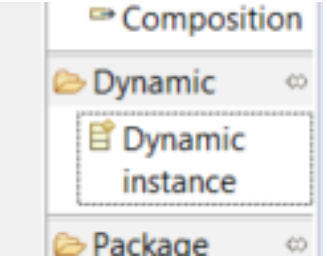
- Model-Based Engineering
- Domain Specific Languages in a Nutshell
- Overview of Eclipse Modeling Framework (EMF)
- Creating Ecore Metamodels
- **Model Code Generation**
- Exercise: Code Generation for Directed Acyclic Graph DSL

Model Code Generation

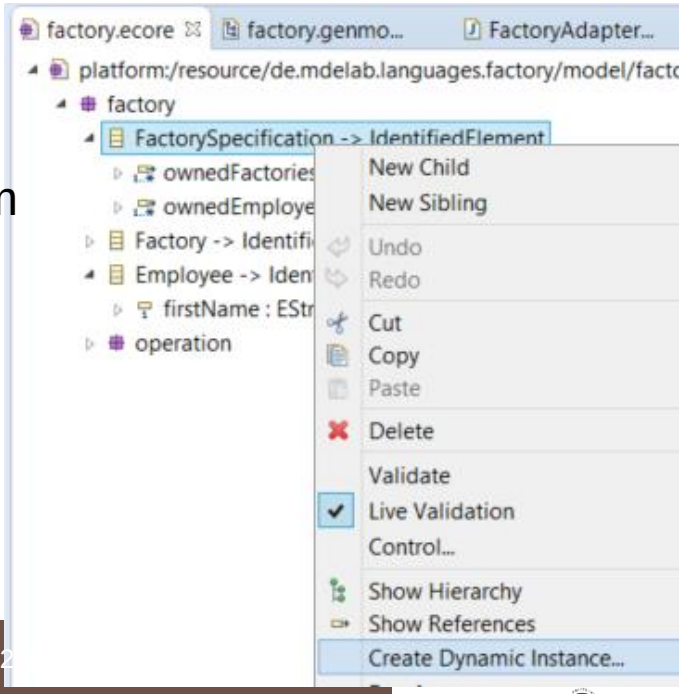
- **Ecore models can be interpreted directly by tools**
- **But code can also be generated (typically):**
  - Provides API usable by other application code
  - Better performances
  - Generation can be customized through configuration of generation model (.genmodel file)
  - Methods and validators can be coded manually and will **not be lost** when code is regenerated
  - Generators available for different programming languages (Java, Python and C++)

Instantiation

- **Models of the metamodel can be quickly instantiated for testing:**
- **From the diagram editor:**
  - Select “Dynamic instance” from the palette and click on the root container class



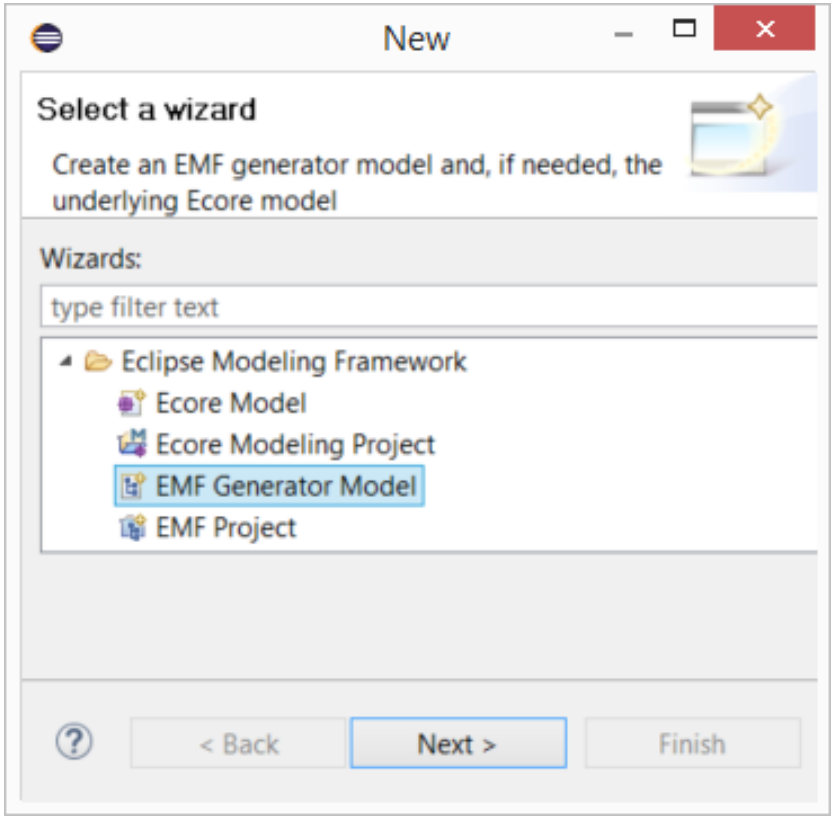
- **From the tree editor:**
  - Select the root instance class, right-click and select the “Create dynamic instance” menu item



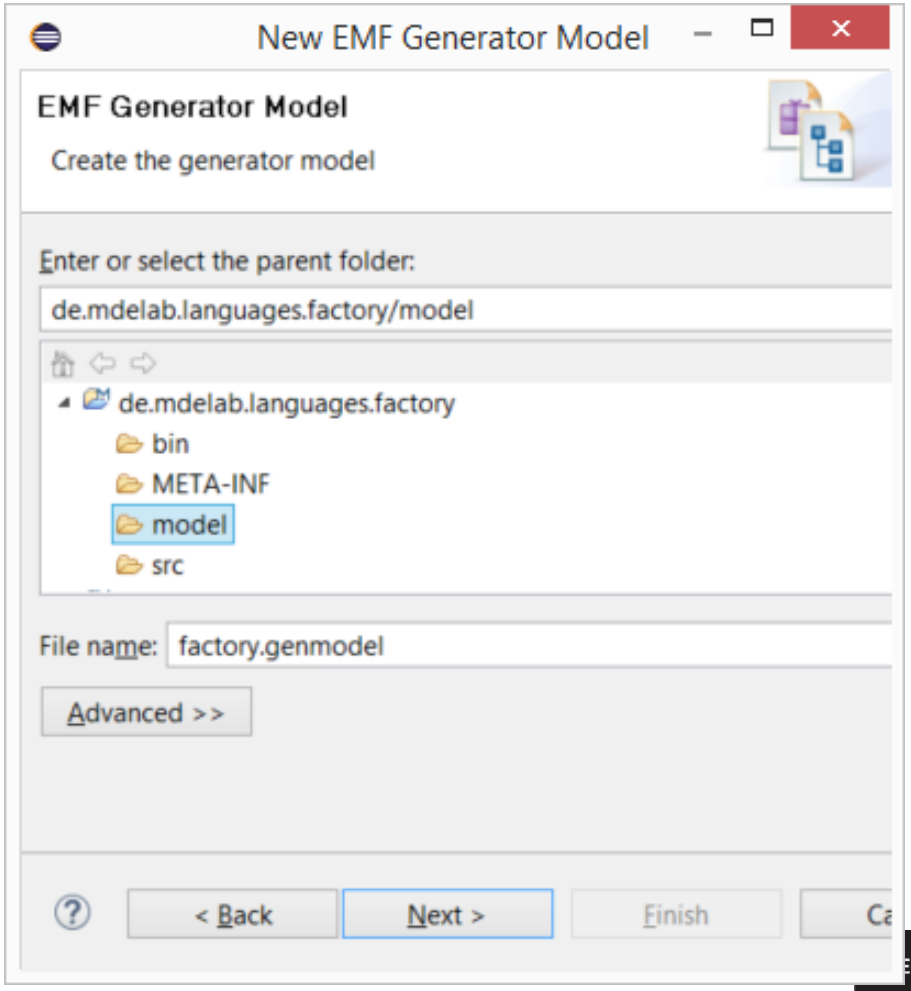
- **Does not make use of generated code**

Creating a Code Generation Model

- **Create EMF generator model**

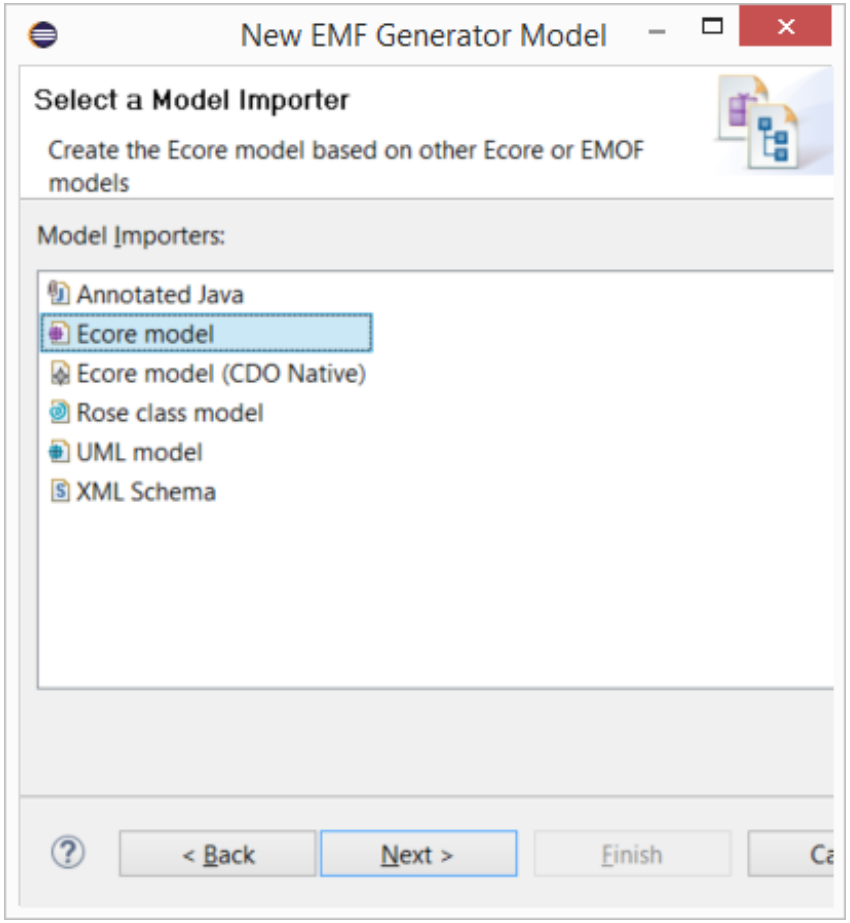


- **Select directory**

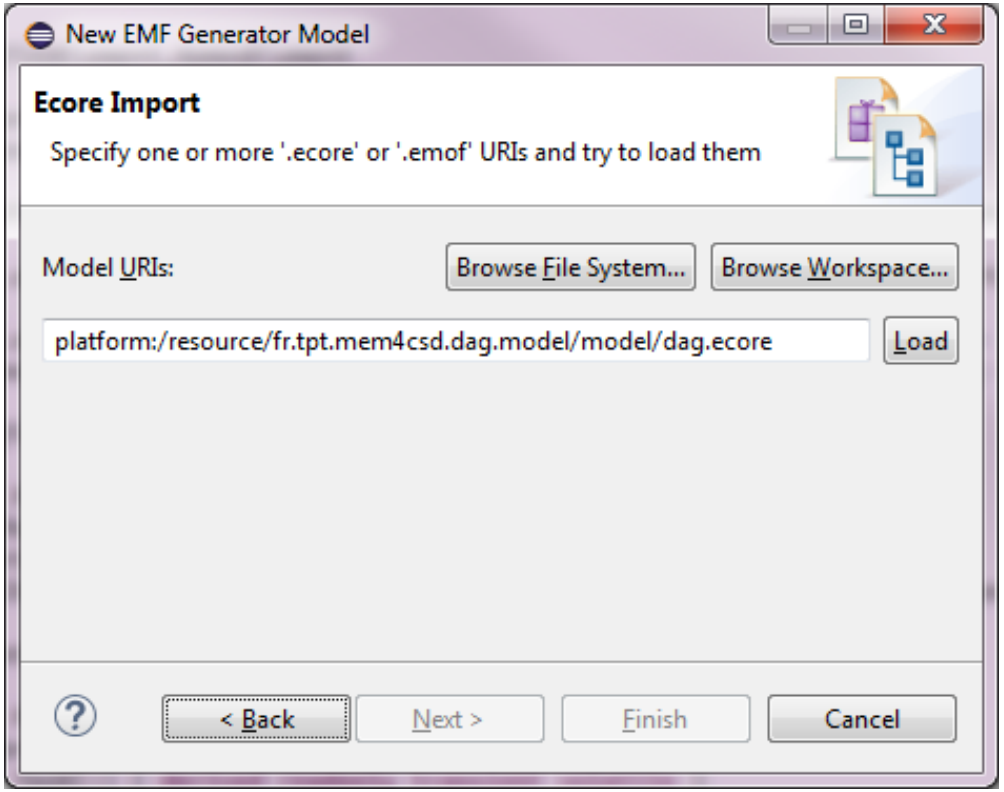


Creating a Code Generation Model (continued)

- **Create from Ecore model**



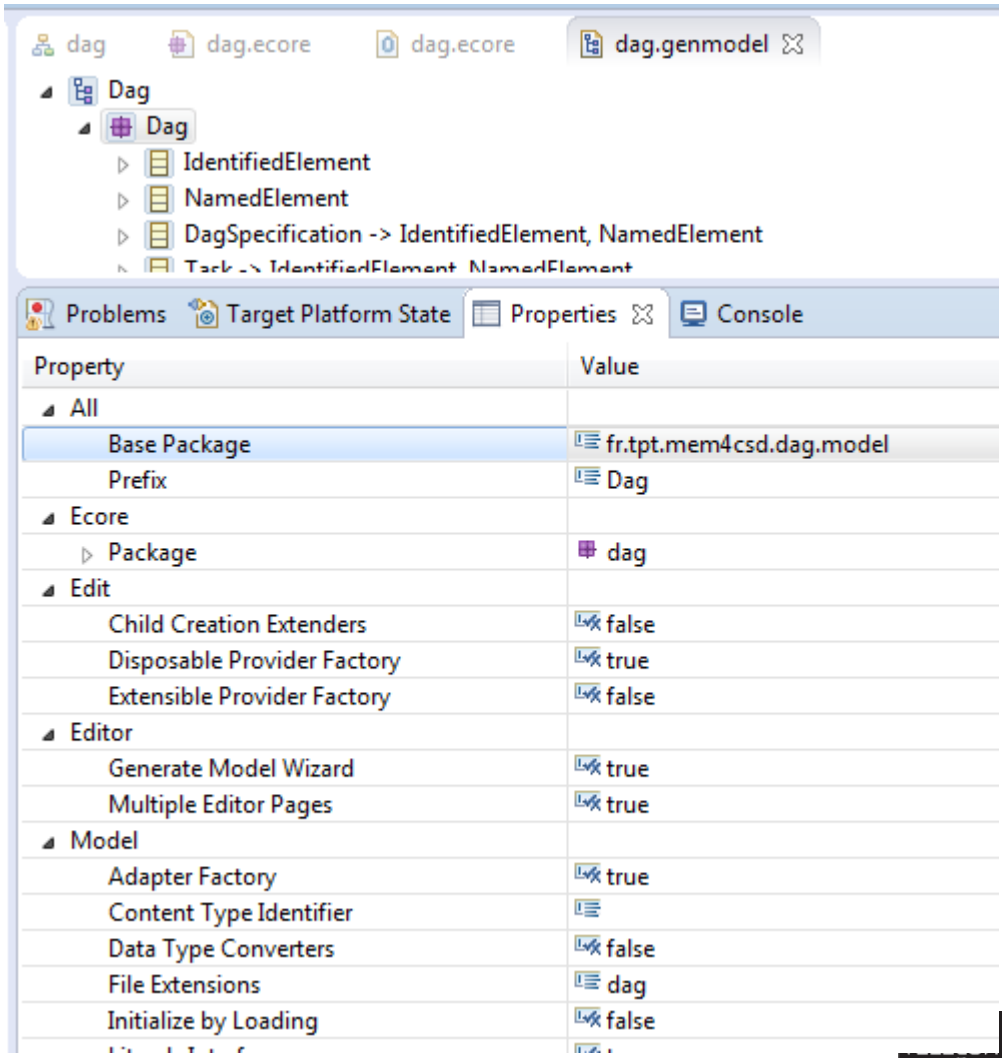
- **Select Ecore model, “Load”, “Next” and “Finish”**



Customization of Generation Model

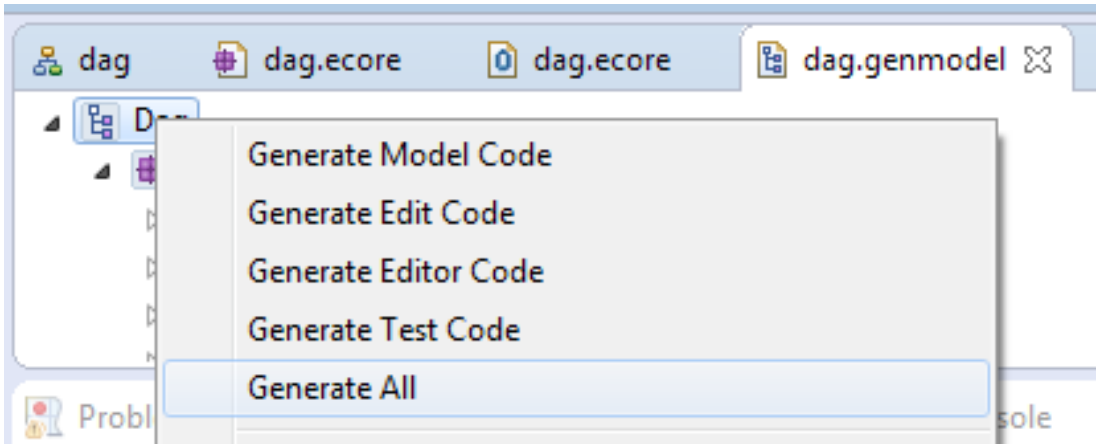
- **Set base package for Java classes**

- **Many other customizations available:**
  - Proxies
  - Pure Java beans
  - Customizable code generation templates
  - Etc.



Code Generation Execution

- **Launch:**



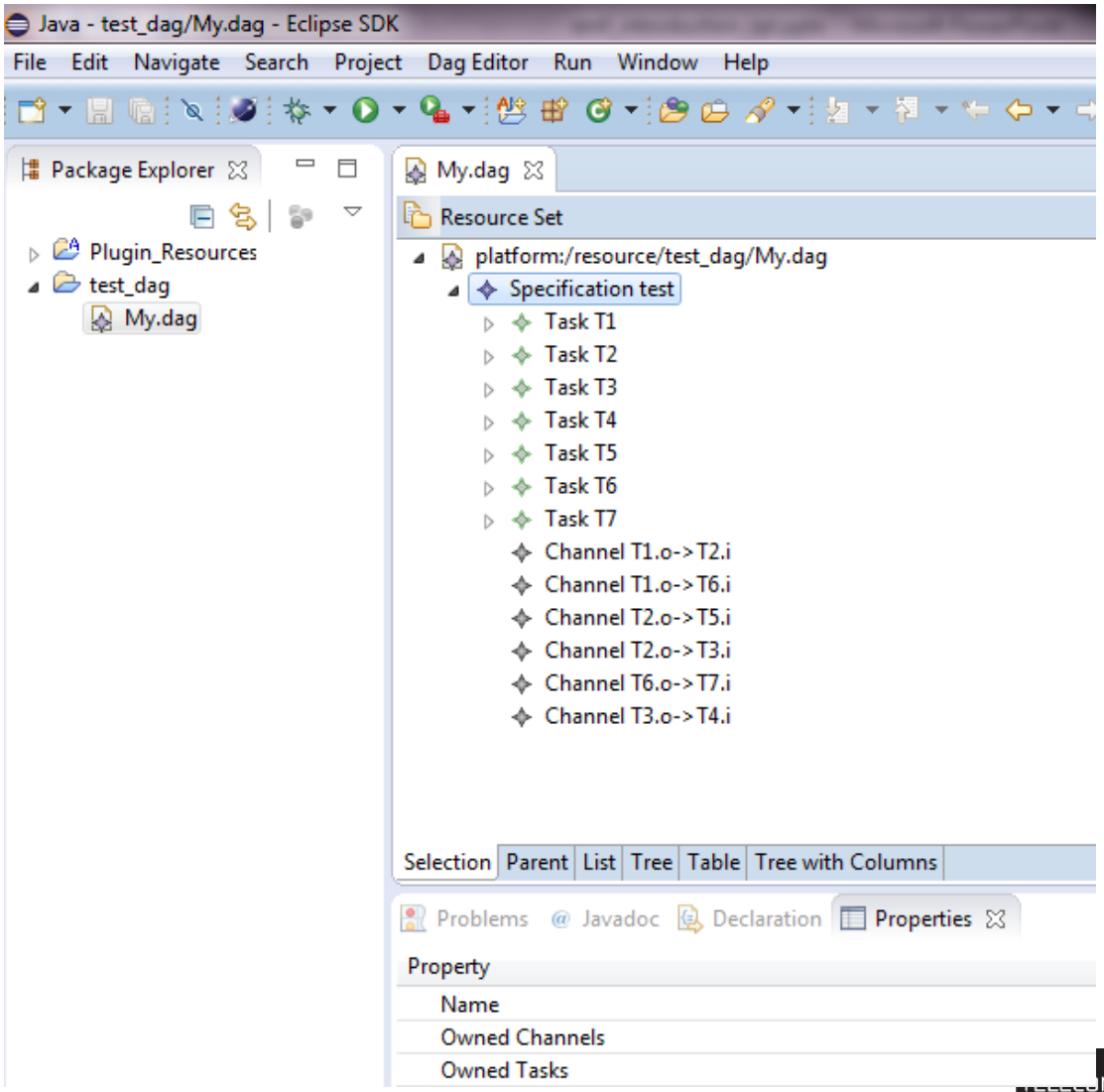
- **Several plugins are generated:**
  - x.model: provides code for meta-model classes
  - x.edit: provides classes responsible for providing labels and icons for the various model elements
  - x.editor: provides a basic tree editor that can be customized by changing the generated code

Generated Java Code

- **One interface and one implementation per class:**
  - E.g.: Task.java and TaskImpl.java
  - Handle automatically:
    - Management of bi-directional references
    - Proxy management
    - Object deletion: when an instance is deleted, all references targeting it are automatically removed
- **One interface and one implementation per package:**
  - Provide constants and API to retrieve information on the metamodels
  - E.g.: classes, references, attributes and identifiers for them
- **One factory interface and class per package:**
  - Provide API to instantiate metamodel classes

Generated Model Editor

- **Basic tree editor for models automatically generated**
- **Code and icons can be customized**



Coding a Derived Property in the Generated Code

- **Open the generated Java class implementation**
  - E.g., ChannelImpl.java
  - Code methods « basicGetSourceTask » and « basicGetDestTask »
  - Add « @generated NOT » annotation

```
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public Task basicGetSourceTask() {
    // TODO: implement this method to return the 'Source Task' reference
    // -> do not perform proxy resolution
    // Ensure that you remove @generated or mark it @generated NOT
    throw new UnsupportedOperationException();
}

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public Task basicGetDestTask() {
    // TODO: implement this method to return the 'Dest Task' reference
    // -> do not perform proxy resolution
    // Ensure that you remove @generated or mark it @generated NOT
    throw new UnsupportedOperationException();
}
```

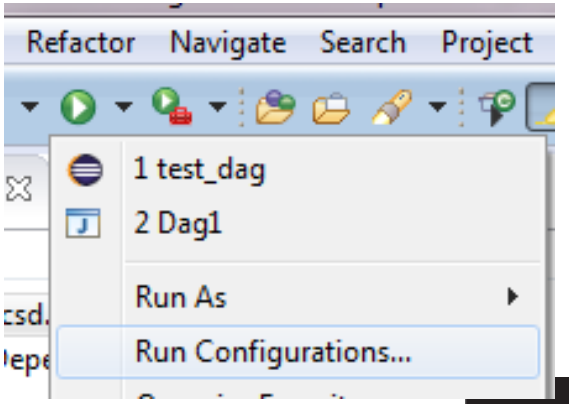
Coding the Validation of a Structural Constraint in the Generated Code

- **Open the generated Java class validator**
  - E.g., DagValidator.java
  - The skeleton of a method named «validateChannel_*constraintname*» will have been generated that you must implement
  - Add “@generated NOT” annotation

```
/**
 * Validates the consistentDirections constraint of '<em>Channel</em>'.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public boolean validateChannel_consistentDirections(Channel channel, DiagnosticChain diagnostics, Map<Object, Object> context) {
    // TODO implement the constraint
    // -> specify the condition that violates the constraint
    // -> verify the diagnostic details, including severity, code, and message
    // Ensure that you remove @generated or mark it @generated NOT
    if (false) {
        if (diagnostics != null) {
            diagnostics.add
                (createDiagnostic
                    (Diagnostic.ERROR,
                    DIAGNOSTIC_SOURCE,
                    0,
                    "_UI_GenericConstraint_diagnostic",
                    new Object[] { "consistentDirections", getObjectLabel(channel, context) },
                    new Object[] { channel },
                    context));
        }
        return false;
    }
    return true;
}
```

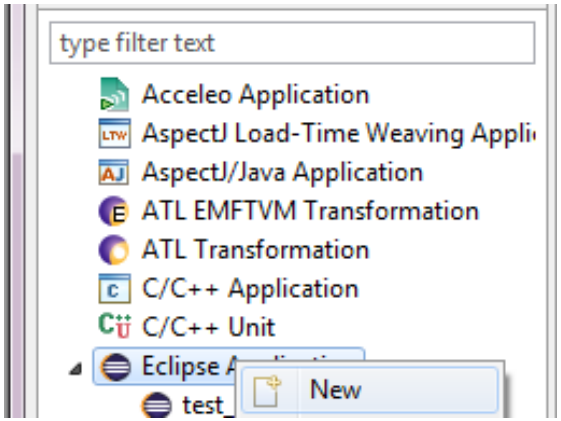
Creating a Model of a Metamodel

- An Eclipse instance can be launched from the Eclipse used for development.
- This Eclipse will have the generated meta-model released in its plugin registry
- For this, create a new Eclipse launch configuration by opening the configuration dialog box:



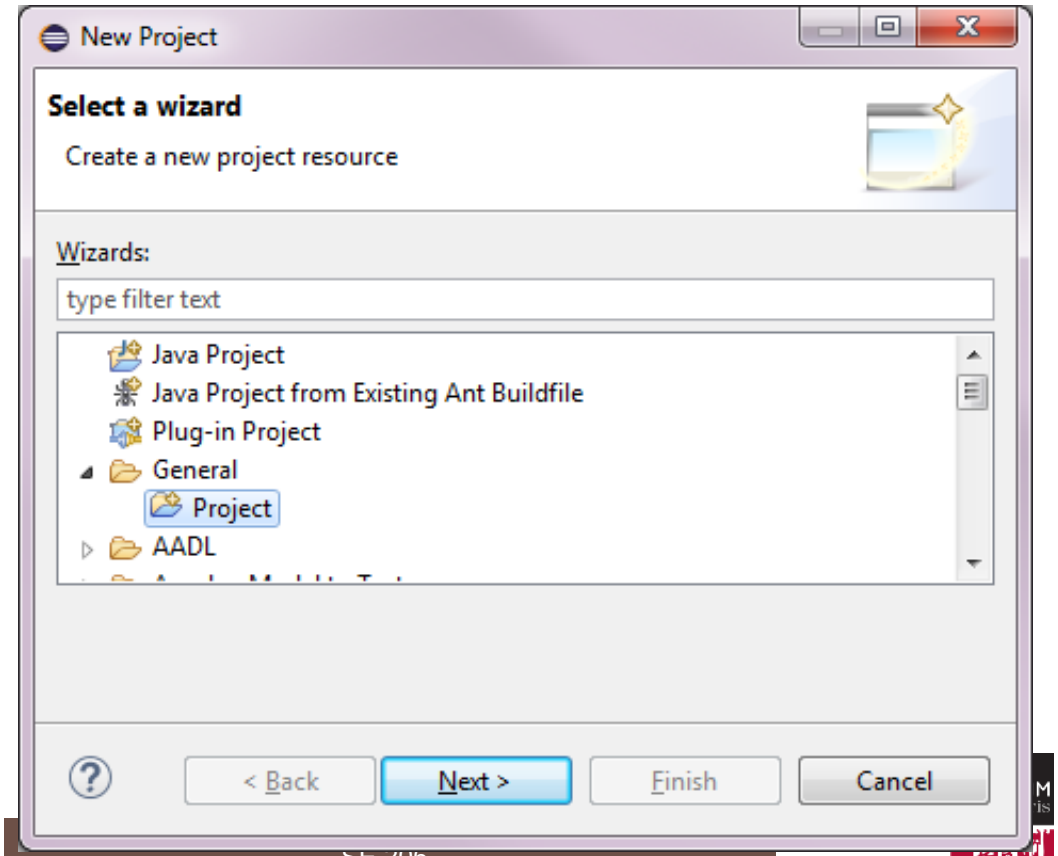
Creating a Model of a Metamodel (continued)

- Create a new Eclipse launch configuration
- Click “Run” to execute this new Eclipse for testing the metamodel



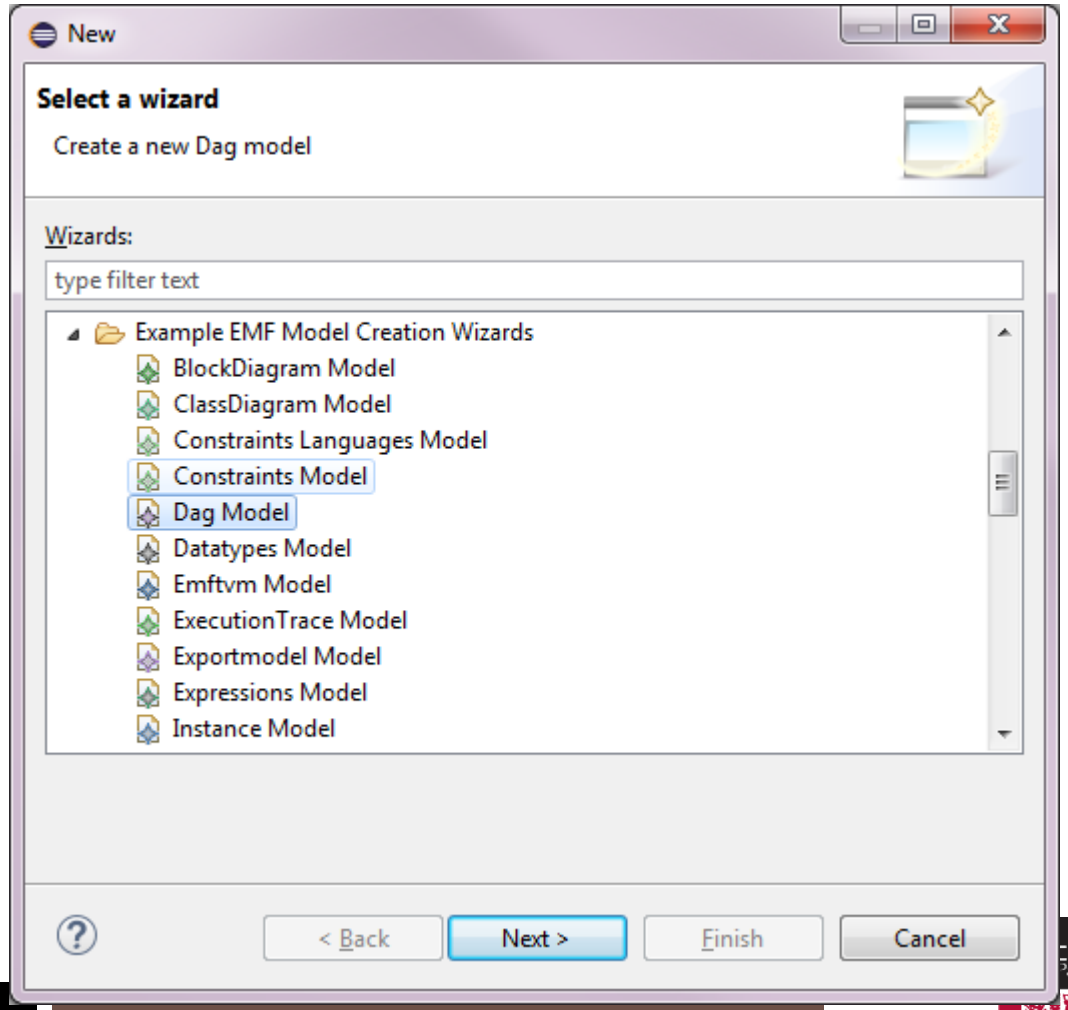
Creating a Model of a Metamodel (continued)

- From the test Eclipse instance, create a resource project to contain the test model: Menu File>>New>>Project
- Select “Project” under “General”



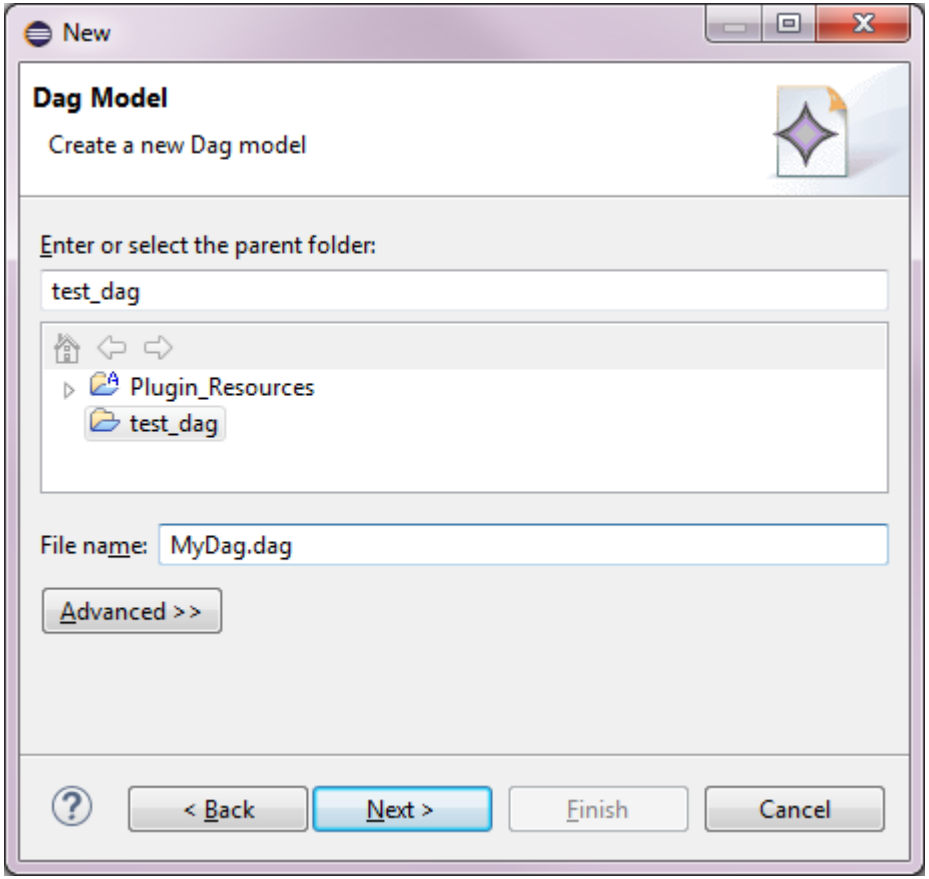
Creating a Model of a Metamodel (continued)

- **Create a new model: Menu File>>New>>Other**
- **Select “Dag model”:**



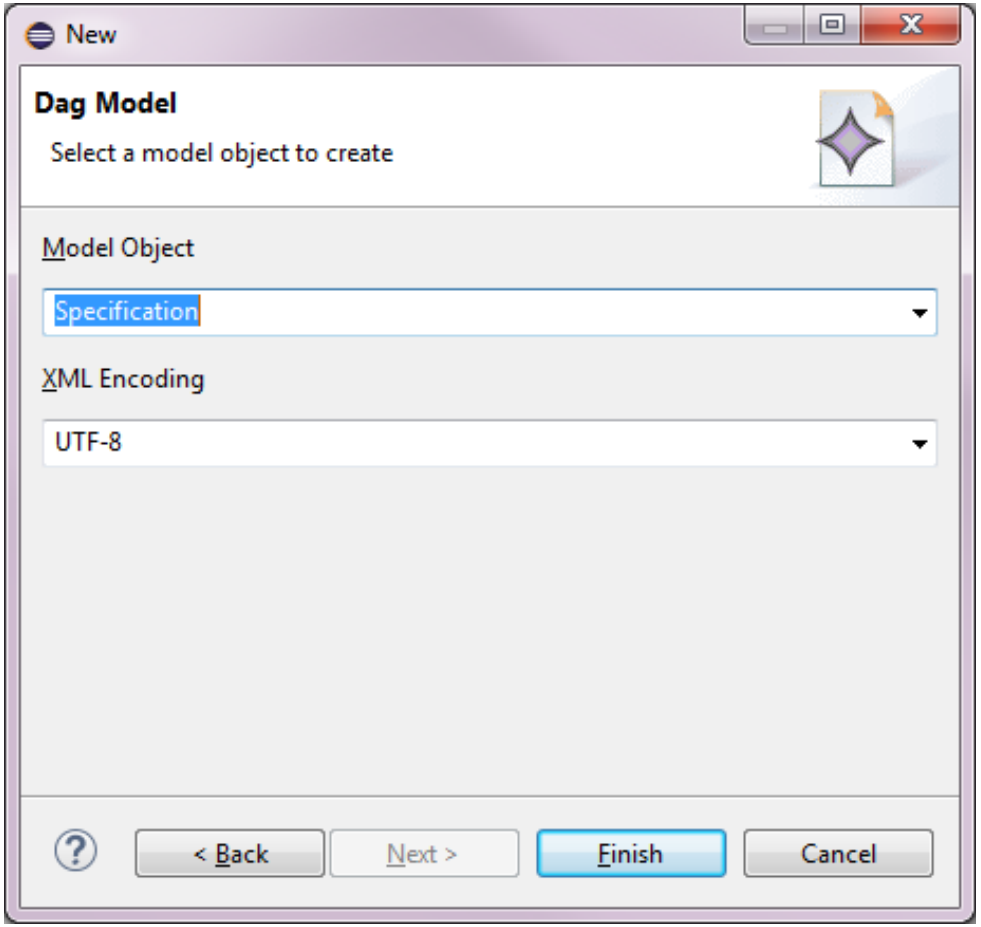
Creating a Model of a Metamodel (continued)

- **Select the project to contain the model:**

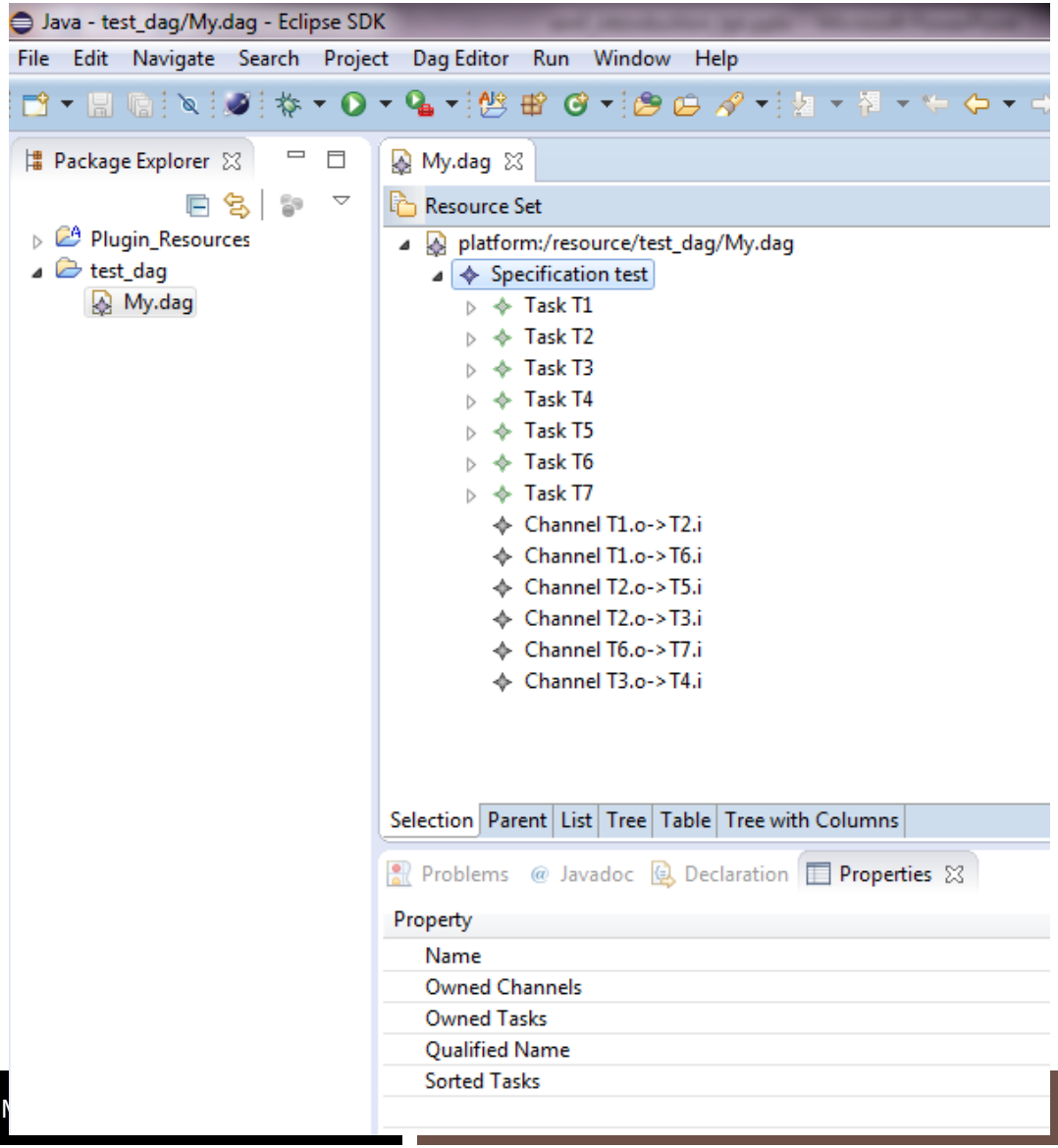


Creating a Model of a Metamodel (continued)

- **Select the root class for the model:**



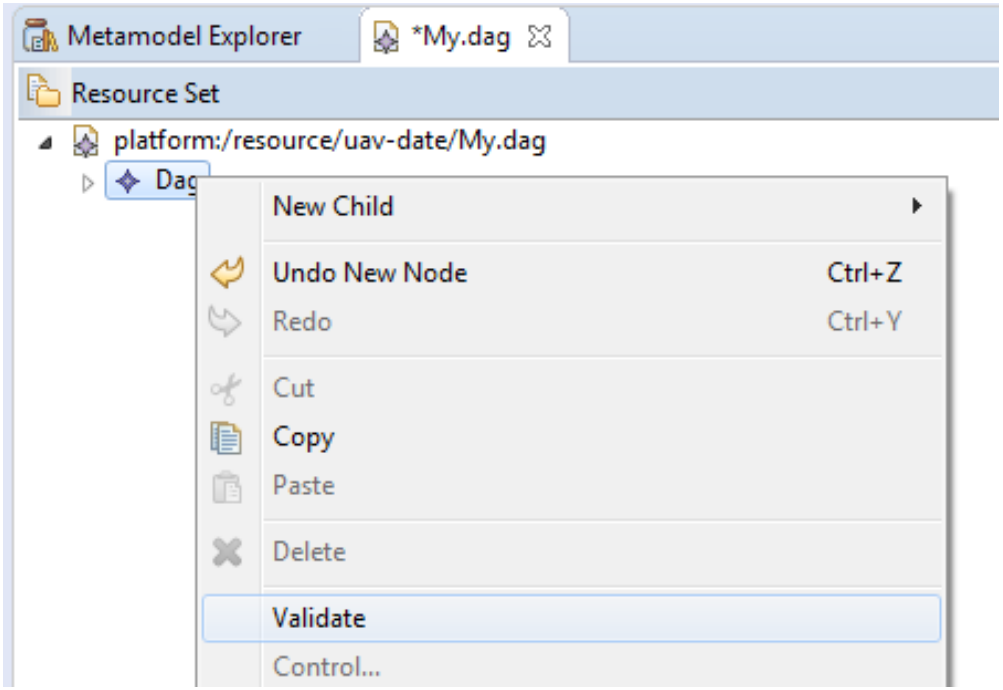
Edit the Model



Validate the Model

- Like for metamodels, models can be validated

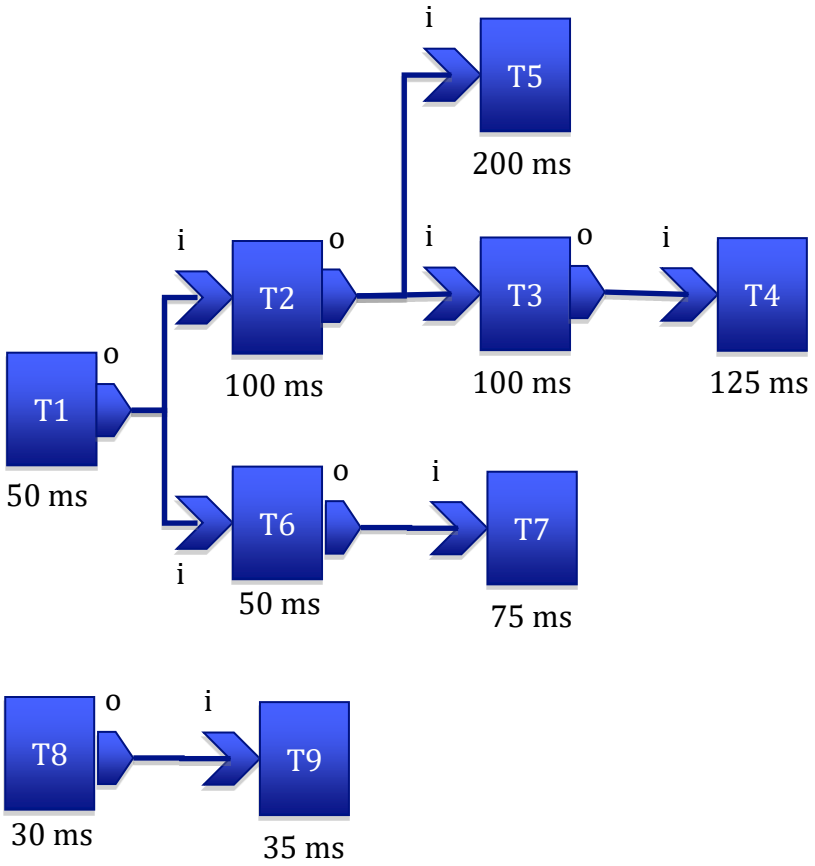
- Validation triggers the evaluation of structural constraints (from metamodel and custom)



Content

- Model-Based Engineering
- Domain-Specific Languages in a Nutshell
- Overview of Eclipse Modeling Framework
- Creating Ecore Metamodels
- Model Code Generation
- **Exercise: Code Generation for Directed Acyclic Graph DSML**

Exercise: Complete the Directed Acyclic Graph Tasks DSL (DAG)



- See exercise description of course website

Travaux Pratiques

- **Sur le site web du cours (https://se206.wp.imt.fr/sujets-de-tp/):**
  - Télécharger cette présentation
  - Télécharger l'archive de projets à compléter
  - Télécharger le document d'instructions pour le TP

- **Suivre les instructions du document pour réaliser le TP**

- **Le TP sera évalué:**
  - M'envoyer tous vos projets Eclipse dans une archive pour que je puisse les évaluer
  - **Vous devrez compléter par vous-même si temps insuffisant en classe**

Instructions pour la réalisation du TP

- **Exécuter Eclipse tel que vous l'avez installé sur votre machine**
  - Voir instructions ici: https://se206.wp.imt.fr/tp-installation-des-outils-pour-travailler-de-chez-soi/

- **Ou connectez-vous sous votre compte Telecom Paris**
  - Dans une fenêtre de commande Linux exécutez les commandes suivantes:
    - Exécuter Eclipse:
      - /comelec/softs/opt/aadl-tools/eclipse-modeling-2022-06-R/eclipse &

Rendu du TP

- **A rendre pour le TBD**
- **Archiver tous les projets du workspace dans un seul fichier .zip**
  - Ne pas inclure le répertoire **.metadata**
- **Déposer l'archive sur eCampus**