Domain Specific Language Design (2IMP20)

Modelware. Introduction to Eclipse Modeling Framework

Loek Cleophas, Ivan Kurtev

Agenda

Topics in applied generic language technologies

- Overview of the Modelware part of the course

EMF - Eclipse Modeling Framework

- Motivation and applications

EMF metamodels

- Ecore language

EMF Code Generation

EMF Dynamic Features

Topics in Applied Generic Language Technologies

Modelware approach to DSLs

Eclipse Modeling Framework (EMF)

Metamodeling

- Concepts
- Model constraints and validation, Object Constraints Language (OCL)

Textual Concrete Syntax

- Xtext

Model Transformations

- Classifications; languages: ETL, QVTo
- Design of model transformations

Code Generators

Modelware

A development approach based on:

- concepts
- principles
- tools
- standards

for creating, processing and using **models** in systems/software development

Modelware

The term emerged in the context of the transition from using objects and components to using models in software development

- Model Driven Engineering (MDE)

Concepts

- Model, metamodel, modeling language, domain-specific modeling language, model transformations, …

Principles

- Everything is a model !

Modelware

Standards and Tools

- UML, MOF, SysML, EMF, OCL, QVTo, …

Modelware applied to DSL development:

- All language aspects are defined in models, possibly themselves expressed in a DSL
  - Tools are as much as possible automatically generated from these models
  - Metamodel: the corner stone of the DSL definition
- Main operations: text-to-model, model-to-model, model-to-text transformations

Modelware vs Grammarware

Grammarware:

- Classical, grammar-based approach to DSL development
- Language
  - A set of sentences over an alphabet. A set of sentences that can be derived from a grammar
  - Fundamental relation: sentence-grammar
- Structures
  - Abstract and concrete syntax trees

Modelware:

- DSL development influenced by conceptual and OO modeling techniques
- Language
  - A set of models conforming to a metamodel
  - Fundamental relation: model-metamodel
- Structures
  - Models are typically represented as graphs

Modelware vs Grammarware

Modelware includes many Grammarware concepts

- Abstract and concrete syntax, static and dynamic semantics, …

Modelware aims at rapid and generative development of DSL tools

- Often, a further abstraction on top of Grammarware tools
  - Example: Xtext relies on and under the hood uses ANTLR, a parser generator

The Modelware part of this course generally requires to first grasp the Grammarware concepts

Relevancy of Modelware

Why study the Modelware approach?

- Growing number of applications and tools based on MDE in both research and industry

- Several tools and ecosystems with industrial application for DSL construction
  - Language Workbenches like Eclipse EMF, Xtext, JetBrains MPS, MetaEdit+

Agenda

Topics in applied generic language technologies

- Overview of the Modelware part of the course

EMF - Eclipse Modeling Framework

- Motivation and applications

EMF metamodels

- Ecore language

EMF Code Generation

EMF Dynamic Features

Eclipse Modeling Framework - EMF

Part of a larger context: Eclipse modeling project

- https://www.eclipse.org/modeling/emf/

Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) supports the definition of *domain models*, creation and manipulation of *instances* of the domain models

- Key technology for DSL development
- Provides a language for defining domain models called *Ecore*
- Provides default persistence mechanism based on XML for storing models
- Provides generation of Java code for manipulation of models

Eclipse Modeling Framework

In EMF, a domain model is called *metamodel*

- contains classes that can be instantiated

Instances of a metamodel are *models*

<u>Attention</u>

In the next lecture we will look in details on the terms *model*, *metamodel, metametamodel* and their relation to *modeling language*, *metalanguage*, and others

Eclipse Modeling Framework

EMF is widely used in industry for developing the core infrastructure for Domain-Specific Modeling Languages (DSML)

Some technologies that use EMF (most are covered in the course):

- Object Constraints Language (OCL)
- Xtext: a language workbench for textual DSLs
- GMF and Sirius: frameworks for developing graphical editors for DSMLs
- Model transformation languages such as QVTo and Epsilon Transformation Language (ETL)

Running Example

This course will use a running example of a language to be developed

SLCO – Simple Language for Communicating Objects

In the end of the course, a full implementation of SLCO will be achieved

- Textual editor, concrete and abstract syntax definitions, type checker, simulator,…

EMF is explained using SLCO as an example

SLCO Overview

SLCO is a modeling language for communicating objects

- *Objects* are instances of *classes*; classes define the structure and the behavior of the objects

Structure:

- *Ports* on which named *signals* can be sent and received. Signals may carry data
- Typed instance *variables*

Behavior:

- One or more *state machines*; *transitions* with *statements*

SLCO Overview

Objects communicate by sending *signals* over *channels*

Channels connect two ports belonging to different objects

Channel kinds:

- *Bidirectional* vs *Unidirectional*
- *Synchronous*, Asynchronous *Lossy*, Asynchronous *Lossless*

Example Model in SLCO

```
model SimpleCalculator {
    classes

    Calculator {
        ports in out

        state machines
        Main {
            variables
            Integer a
            Integer b

            initial S

            transitions
            read_calculate from S to S {
                receive input(a, b) from in;
                send result(a + b) to out
            }
        }
    }
```

Class Calculator

- Two ports
- One state machine

Receives two integers as input (signal *input* on port *in*) and produces the sum as an output (signal *result* on port *out*)

Example Model in SLCO

```
User {
    ports in out

    state machines
    Loop {
        variables
        Integer r

        initial Start
        state Wait

        transitions

        send_receive from Start to Wait {
            send input(1, 2) to out;
            receive result(r) from in
        }

        delay from Wait to Start {
            after 1500 ms //wait 1.5 secs
        }
    }
}
```

```
objects
c : Calculator
u : User

channels
c0(Integer, Integer) sync from u.out to c.in
c1(Integer) sync from c.out to u.in
```

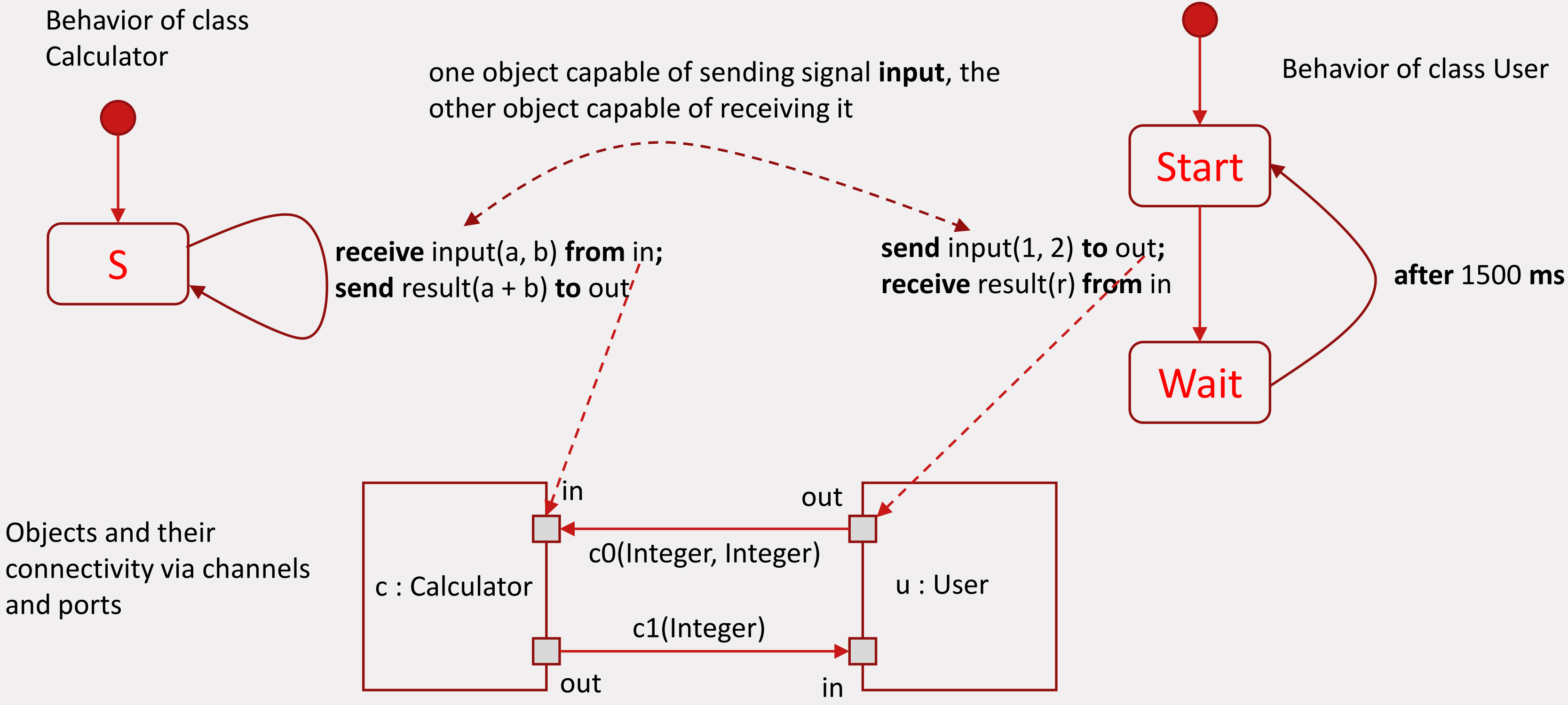
Class User:

- Supplies two integers by sending a signal; expects a result (some calculation on the integers)

Two objects:

- Communicate via two channels
- Channels indicate the types of data that are transmitted

Example Model in Graphical Notation



Domain Modeling

In this example, our domain consists of the communicating objects, their structure and behavior defined in the classes, the communication channels

We aim at creating a model of this domain using EMF

- Such a domain model is called *metamodel*
- SLCO models are instances of the domain model
- In effect, the metamodel will define the concepts in the SLCO without concidering how they are rendered to the user (via a concrete syntax)

The Metamodel of SLCO

A metamodel defines the *concepts* in a domain (as classes), their *attributes*, and the *relations* among them

In EMF, metamodels are expressed in a language that is a subset of the UML class diagrams language

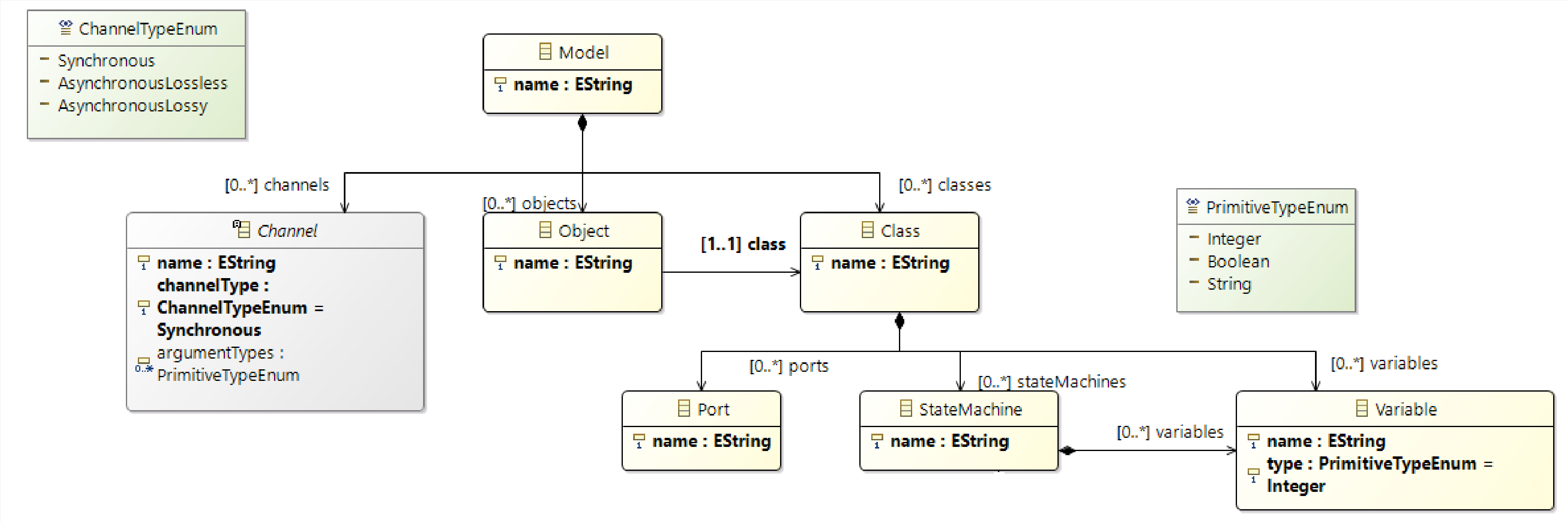
- Visual syntax very close to UML class diagrams

Ecore:

- language for defining metamodels in EMF

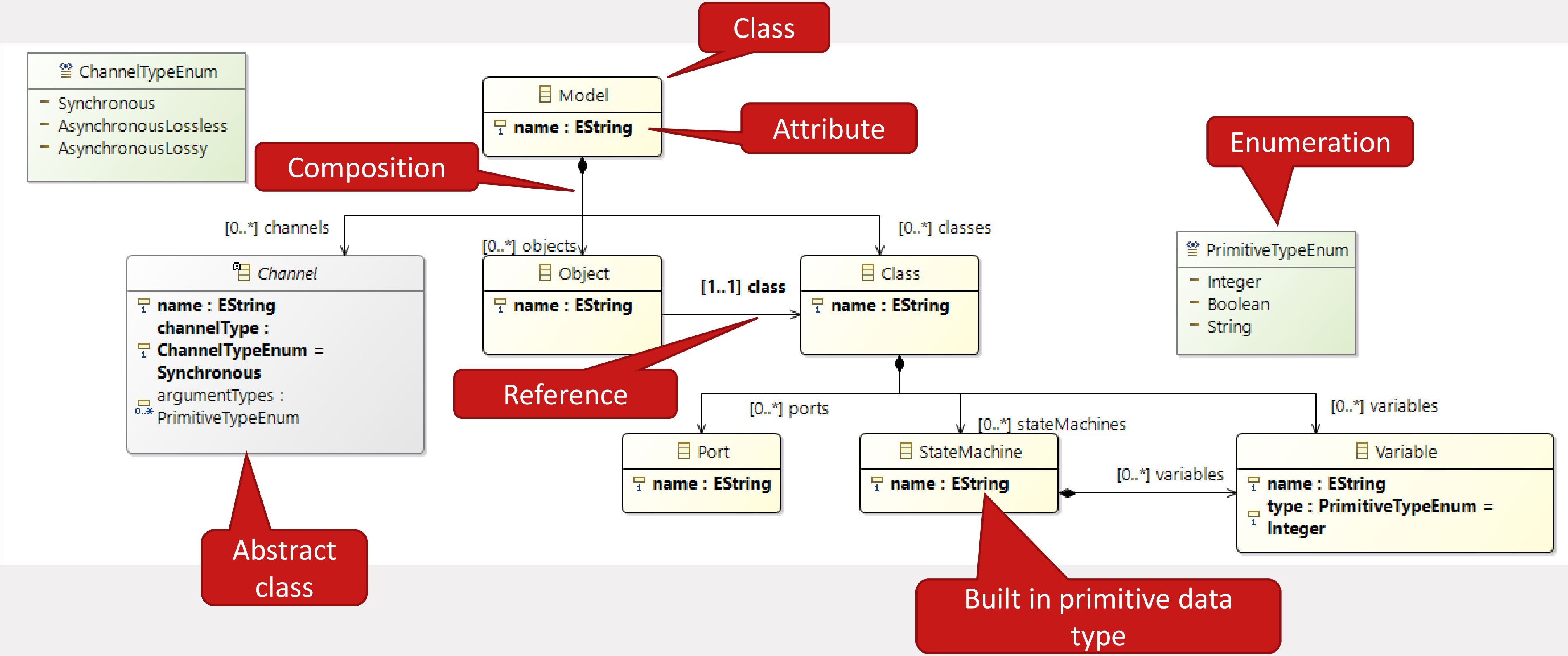
The Metamodel of SLCO

The main SLCO concepts and their relations



The Metamodel of SLCO

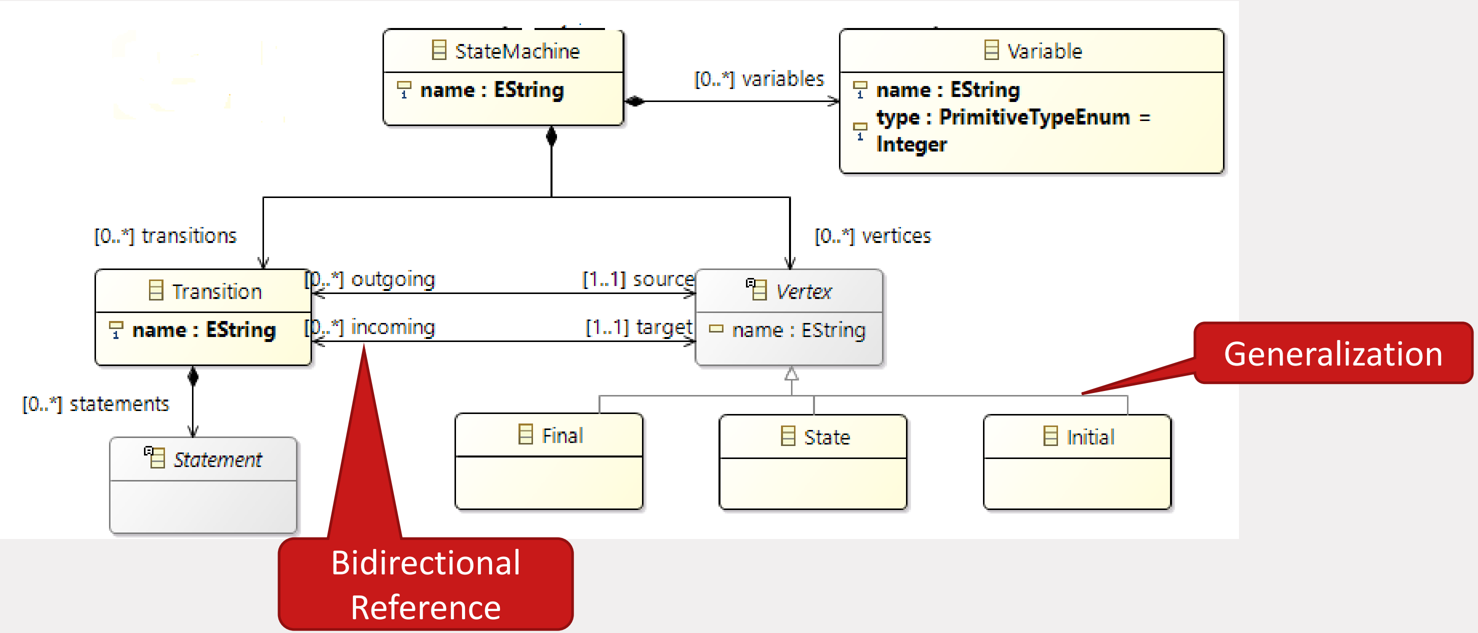
The main Ecore constructs



TU/e

The Metamodel of SLCO

The main Ecore constructs



Main Ecore Language Constructs

- Class
  - has named and typed structural features: attributes and outgoing references
- Attribute
- Enumeration
- Data type
- Generalization
  - multiple inheritance allowed
  - overriding of inherited features not allowed (except for operations)
  - name duplications of features not allowed

Semantics similar to the one of the constructs in UML

Main Ecore Language Constructs

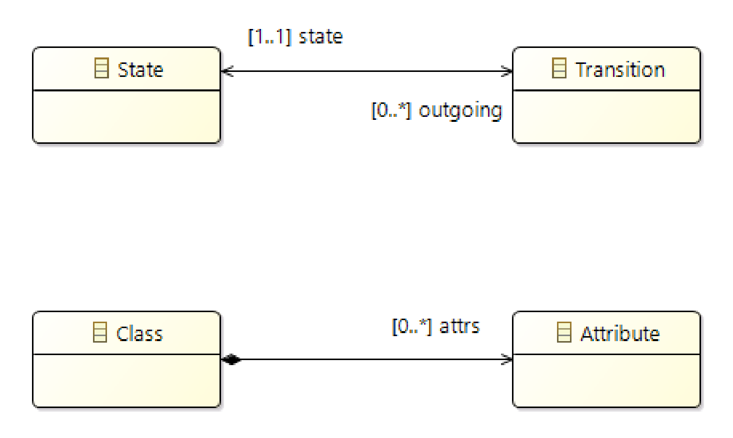
Only binary unidirectional associations between classes are allowed

- these are called *reference*
- association end only on the target side (no multiplicity and role name on the source side)

Reference may:

- be defined as a *composition*
- be an *opposite to* another reference thus achieving bidirectional associations

Composition and Bi-directionality



Pair of references opposite to each other: if a state *S* has a transition *T* as value of feature *outgoing* then *T* has *S* as value of feature *state*

Composition: for a given attribute *A*, there is at most one class *C* that has *A* among the values of feature *attrs*.

Note: composition does not imply that all attributes must have a container

Ecore Packages

Package:

- The main grouping construct
- An Ecore metamodel contains at least one package
- Packages may be nested

Package properties:

- name (e.g. slco)
- Unique (namespace) identifier, also called Ns URI (e.g. http://glt.tue.nl/slco)
- Ns prefix, a shorthand for the Ns URI used typically when Ecore models are stored as XML

Other Ecore Constructs

Constructs inspired by OO programming and modeling

- Interface
- Type parameters used in generic classes

Class operations

- operations can be defined but Ecore has no constructs to implement the body of the operations

Comparison to UML Class Diagrams

In contrast to UML class diagrams, Ecore does not support (list not exhaustive):

- N-ary associations, N > 2
- Association classes
- Bidirectional associations
- Specialization of associations
- Generalization sets and higher-order types

Demo: Metamodel Creation with EMF

Tasks:

- Inspect the complete SLCO metamodel
- Create an Ecore Modeling Project
- Create a simple Ecore metamodel in the graphical editor

These tasks are also explained step by step in the supplemental *Tool Guide*

Visualizing and Editing Ecore Metamodels

Different editors and views for Ecore models

Tree editor

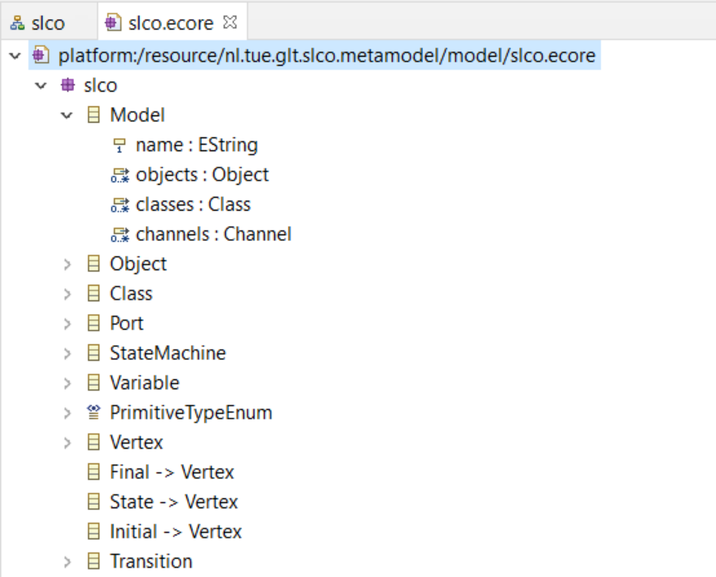
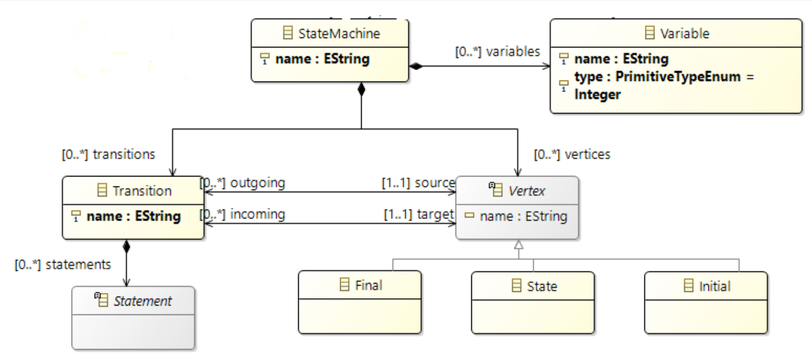


Diagram editor



Persisting Ecore Models

By default all Ecore (meta)models are persisted as XML in a dialect called XMI

- *XMI* – XML Metadata Interchange, standard serialization format for EMF

slco   slco.ecore

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xm
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="slco" nsU
  <eClassifiers xsi:type="ecore:EClass" name="Model">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerB
    <eStructuralFeatures xsi:type="ecore:EReference" name="objects" upp
        eType="#//Object" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="classes" upp
        eType="#//Class" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="channels" up
        eType="#//Channel" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Object">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerB
    <eStructuralFeatures xsi:type="ecore:EReference" name="class" lower
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Class">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerB
    <eStructuralFeatures xsi:type="ecore:EReference" name="ports" upper
        eType="#//Port" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="stateMachine
        eType="#//StateMachine" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="variables" u
        eType="#//Variable" containment="true"/>
  </eClassifiers>
```

Notes:

- users of EMF do not need to use any generic XML tools to parse Ecore files. The EMF does this transparently
- XMI files are for storage purposes only. It is not recommended to work with the XMI files directly

Using Ecore Metamodels and Models

Ecore editors can be used to create metamodels

Using the metamodel:

- How do we create instance models?
- How do we process instance models, for example, in a program?

A widely used approach is to access metamodels and models via an API in a programming language

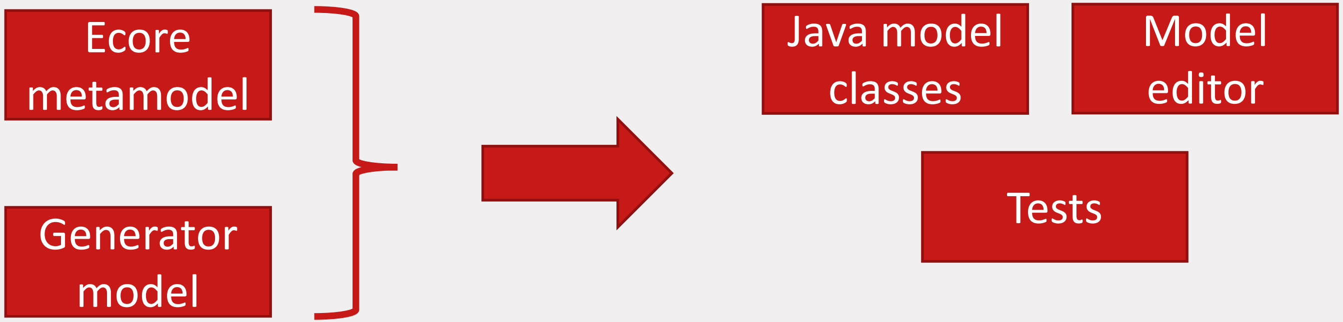
- Ecore provides facility for generating Java code from metamodels

Generating Code from Ecore Metamodels

Ecore metamodels can be transformed to a set of Java classes

- users can create models by instantiating these Java classes
- model instances thus contain Java objects

Java code generation from a metamodel:



Generator model (.genmodel) contains parameters for controlling the generation

Generation of Editors

EMF provides generic tools for generating and customizing simple editors for creating models that conform to a given metamodel

In practice, dedicated editors are usually created:

- based on textual syntax, provide the typical IDE features
- graphical editors that support diagrammatical (visual) syntax

Technologies that support development of dedicated model editors (e.g. Xtext, Sirius) are covered further in the course

Demo: Java Code Generation

Tasks:

- Generate Java model code from an Ecore metamodel and its generator model
- Generate editor
- Create instance model with the generated editor
- Inspect the generated model code

Anatomy of the Generated Model Code

- One Java interface for each Ecore class
  - getters and setters for attributes and references
- One Java class that implements the interface
- Package, Factory and Utility classes

```java
public interface Class extends EObject {
    /**
     * Returns the value of the '<em><b>Name</b></em>' attribute.
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @return the value of the '<em>Name</em>' attribute.
     * @see #setName(String)
     * @see nl.tue.glt.slco.metamodel.slco.SlcoPackage#getClass_Name()
     * @model required="true"
     * @generated
     */
    String getName();

    /**
     * Sets the value of the '{@link nl.tue.glt.slco.metamodel.slco.Class#getName
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @param value the new value of the '<em>Name</em>' attribute.
     * @see #getName()
     * @generated
     */
    void setName(String value);
```

```java
public class ClassImpl extends MinimalEObjectImpl.Container implements nl.tue.glt
    /**
     * The default value of the '{@link #getName() <em>Name</em>}' attribute.
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @see #getName()
     * @generated
     * @ordered
     */
    protected static final String NAME_EDEFAULT = null;

    /**
     * The cached value of the '{@link #getName() <em>Name</em>}' attribute.
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @see #getName()
     * @generated
     * @ordered
     */
    protected String name = NAME_EDEFAULT;
```

Sample Code for Model Creation

```
public class UsingEMFModel {

        public static void main(String[] args) {

                // Initialize the package before using the metamodel classes
                SlcoPackage pack = SlcoPackage.eINSTANCE;

                // Creation of instances of Ecore classes follows the Factory design pattern
                // Obtain the factory to create model elements
                SlcoFactory factory = SlcoFactory.eINSTANCE;

                // Create the root model element that contains all other elements
                Model slcoModel = factory.createModel();
                slcoModel.setName("My first SLCO Model"); //setter for the 'name' attribute

                // Create a class and set its name
                nl.tue.glt.slco.metamodel.slco.Class c = factory.createClass();
                c.setName("Calculator");

                // Create an object and set its class
                nl.tue.glt.slco.metamodel.slco.Object o = factory.createObject();
                o.setName("calc");
                o.setClass(c);

                // Add the object and the class to the model
                slcoModel.getClasses().add(c); //features with multiplicity > 1 are implemented as lists
                slcoModel.getObjects().add(o);
```

For every class: 1 factory method

Sample Code for Saving Models

```
public static void saveModel(Model model) {
	// We will be using the default Ecore serialization in XML (XMI)
	// provided by the EMF API
	Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
	Map<String, Object> m = reg.getExtensionToFactoryMap();

	// Register 'slco' file extension to be treated as XMI format
	// Alternatively, the default generic 'xmi' extension can always be used
	m.put("slco", new XMIResourceFactoryImpl());

	// In EMF, files are treated as resources. A resource belongs to a resource set
	// Obtain a new resource set
	ResourceSet resSet = new ResourceSetImpl();

	// First we need to create a resource
	// Every resource has an unique identifier (URI)
	Resource resource = resSet.createResource(URI.createURI("models/MyFirstSLCOModel.slco"));

	// Add the root object to the resource. All contained objects will be added transitively
	resource.getContents().add(model);
	// Save the content
	try {
		resource.save(Collections.EMPTY_MAP);
	} catch (IOException e) {
		e.printStackTrace();
	}
```

Persisting Models via EMF API

The example code uses the EMF API for managing content (files on disk, content on internet). Main points are:

- Models are contained in *resources*
- A collection of available resources forms *resource set*
- A resource has an *identifier* (URI – uniform resource identifier)
- EMF maintains a registry that maps extensions (or resource kinds) to processors that are capable of reading a resource and instantiating an Ecore model

EMF can be extended wrt managing resources and their formats, for example

- Resources can be in binary format
- A parser can be used to process textual resources and instantiate Ecore objects

Users can implement their own resource factories

Everything is an EObject

All generated model classes from Ecore metamodels implement *EObject* interface

EObject:

- The EMF analog of *java.lang.Object*
- Provides reflective access to object's features
  - E.g. get the object's class, container, containing resource, features, feature values, etc.

EObject Interface

These are some of the methods of *EObject* interface

```
public interface EObject extends Notifier {
        EClass eClass();
        Resource eResource();
        EObject eContainer();
        EList<EObject> eContents();
        Object eGet(EStructuralFeature feature);
        void eSet(EStructuralFeature feature, Object newValue);
        Object eInvoke(EOperation operation, EList<?> arguments) throws InvocationTargetException;
        ………
}
```

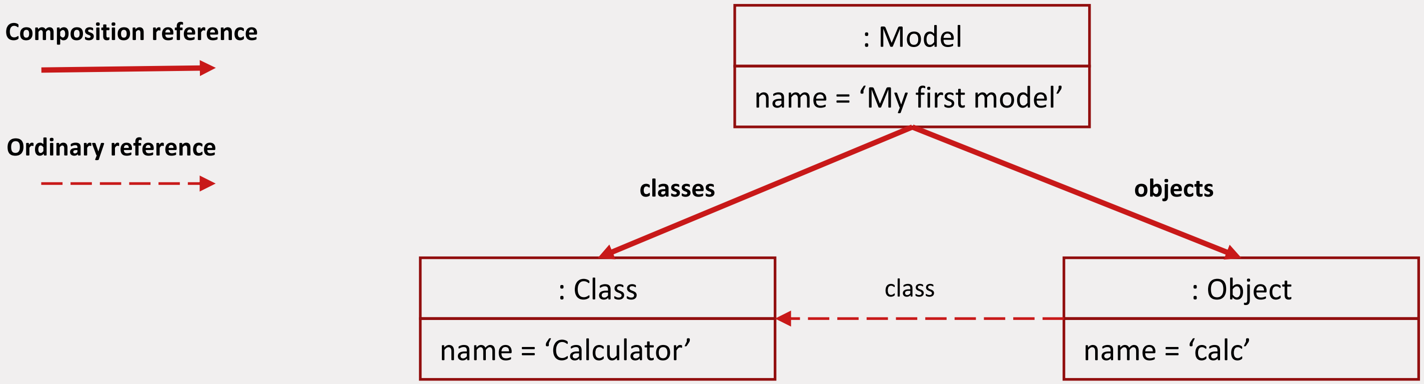
In this generic view of objects:

- An Ecore model is a *forest* of EObject *trees*, possibly contained in a resource
- A tree is defined by the *part-whole* relation induced by the composition references (recall that an object may be a part of at most one object);
- Normal object *references* do not follow *part-whole* semantics thus turning the tree into a *graph*

EObject Interface

In the generic view of EObject:

- Every EObject is an instance of an Ecore class
- Every object has structural features with values; structural features are derived from class attributes and outgoing references



A tree of Eobjects over composition references

Dynamic EMF Features

EMF metamodels can be transformed to Java code and models can be created by instantiating generated Java classes

- Classes are known *statically*, at compile time

EMF also provides the possibility to create model objects *without generating code* from metamodels

- metamodels are still needed, however

It is even possible to *create a metamodel dynamically* in a Java program and instantiate it without any code being generated

Other EMF Features

- Elements from one metamodel can refer to and use elements in another metamodel (e.g. a class extends another class)
  - Real life language definitions often use a dozen of metamodels
- Providing implementation body of operations
- Change listeners: listeners can be registered with EObjects and notifications can be sent when objects change
- Constraints on metamodels and model validation (follow up lectures)
- Model element annotations

Some Tips for Metamodeling

- Determine the concepts that *exist* in the domain; their sub-class/super-class hierarchy and relations
- Denote one *model container* class that transitively contains all other elements (even if such a container is artificially introduced)
- Many OO *design patterns* are applicable to metamodels:
  - Composite pattern
  - Class-Object
  - Visitor, Interpreter

- Sometimes, next to domain concepts, additional constructs are required: e.g. generic *modularity constructs*, *import* clauses, *references*

Summary

EMF:

- Bridge between modeling and programming world
- Transforms metamodels to customizable Java code for creating and manipulating models
- Foundation for many industrial technologies for developing domain-specific modeling languages
  - Graphical and textual editors
  - Model transformations
  - Code generators

Summary

EMF:

- With the code generation facility, turns *models* into *data*
- Models and metamodels can be read and created programmatically thus enabling *metaprogramming*
- No difference between a *general-purpose programming language* and a *domain-specific language*
    - Both can be defined in a metamodel
    - *Programs* and *models* treated *uniformly* (confirms ‘everything is a model’ principle)

Resources and Further Reading

Book *Software Languages*, Chapter 1 about Modelware and MDE

Supplemental *Tool Guide* (available on Canvas)

- Eclipse installation, basic tasks for creating Ecore projects, metamodels and models
- Small self-study exercise
- Reference to other tutorials on EMF/Ecore

Projects with code used in the demos (available on Canvas)

- The SLCO metamodel