Eclipse Modeling Framework

Eclipse Modeling Framework (EMF)

- Modelling — more than just documentation
- Just about every program manipulates some data model
  - It might be defined using Java, UML, XML Schemas, or some other definition language
- EMF aims to extract this intrinsic “model” and generate some of the implementation code
  - Can be a tremendous productivity gain

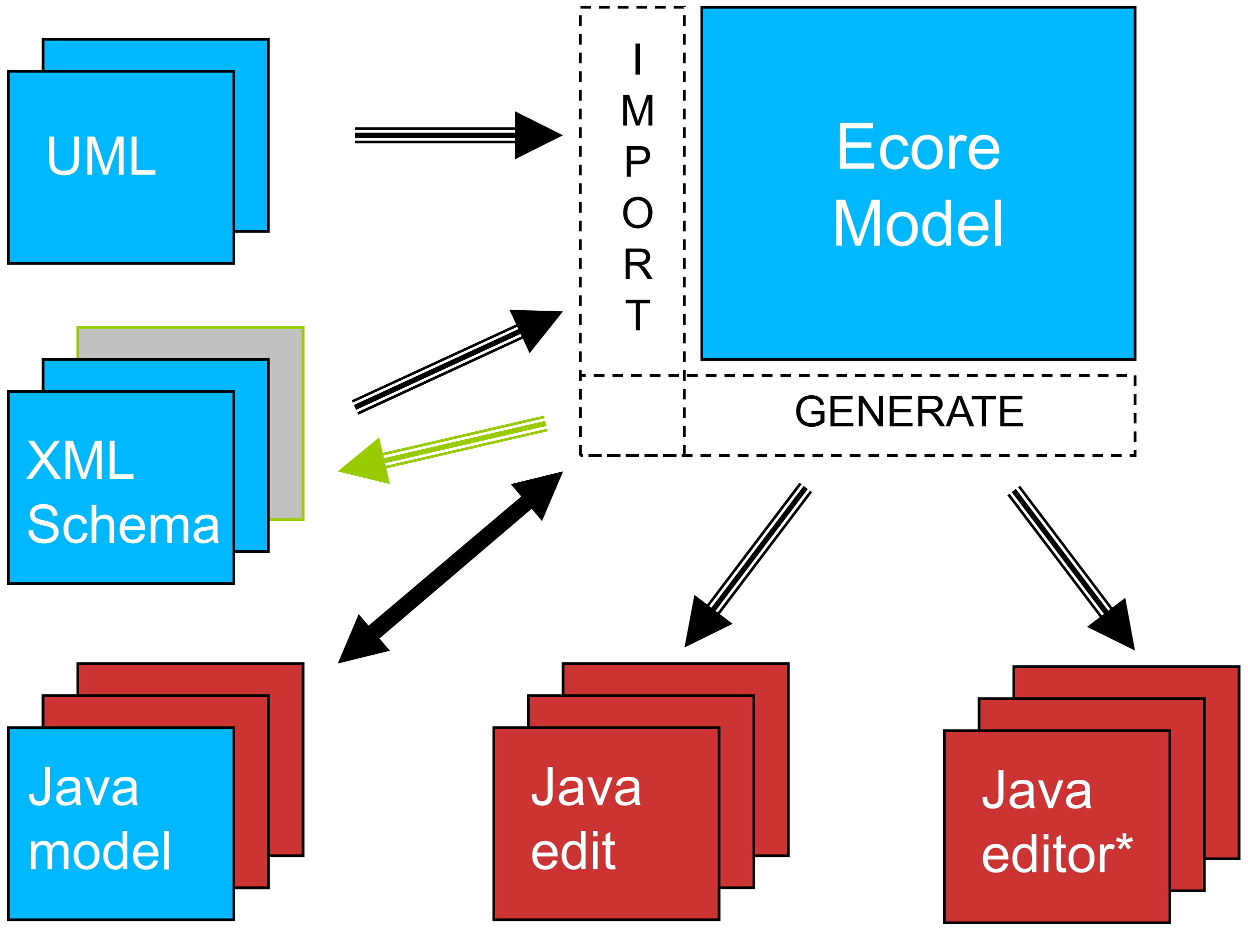
- EMF is one implementation of MOF (though it has differences)
  - EMF ≈ EMOF

http://www.eclipse.org/emf/

EMF

- EMF is a **modelling framework** and **code generation facility** for building tools and other applications based on a structured data model.
- From a model specification described in XMI, EMF provides
  - tools and runtime support **to produce a set of Java classes for the model**,
  - **adapter classes that enable viewing** and **command-based editing** of the model,
  - and a **basic editor**.
- Models can be specified using
  - Annotated Java
  - XML documents
  - Modelling tools like Rational Rose, MagicDraw, …
  - …
- **EMF provides the foundation for interoperability** with other EMF-based tools and applications.

EMF architecture: Model import and generation



Generator features:

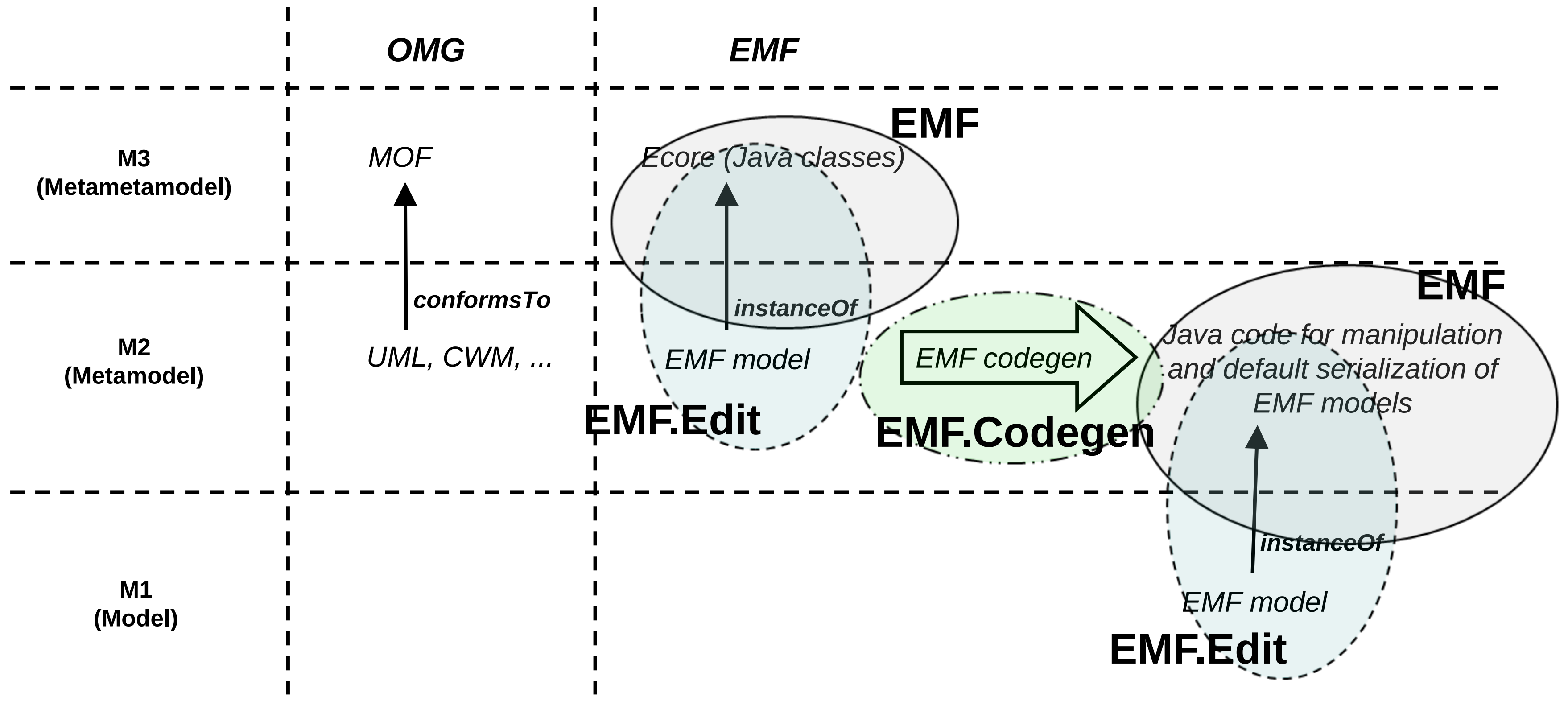
- Customizable JSP-like templates (JET)
- Command-line or integrated with Eclipse JDT
- Fully supports regeneration and merge

* requires Eclipse to run

EMF — Fundamental Pieces

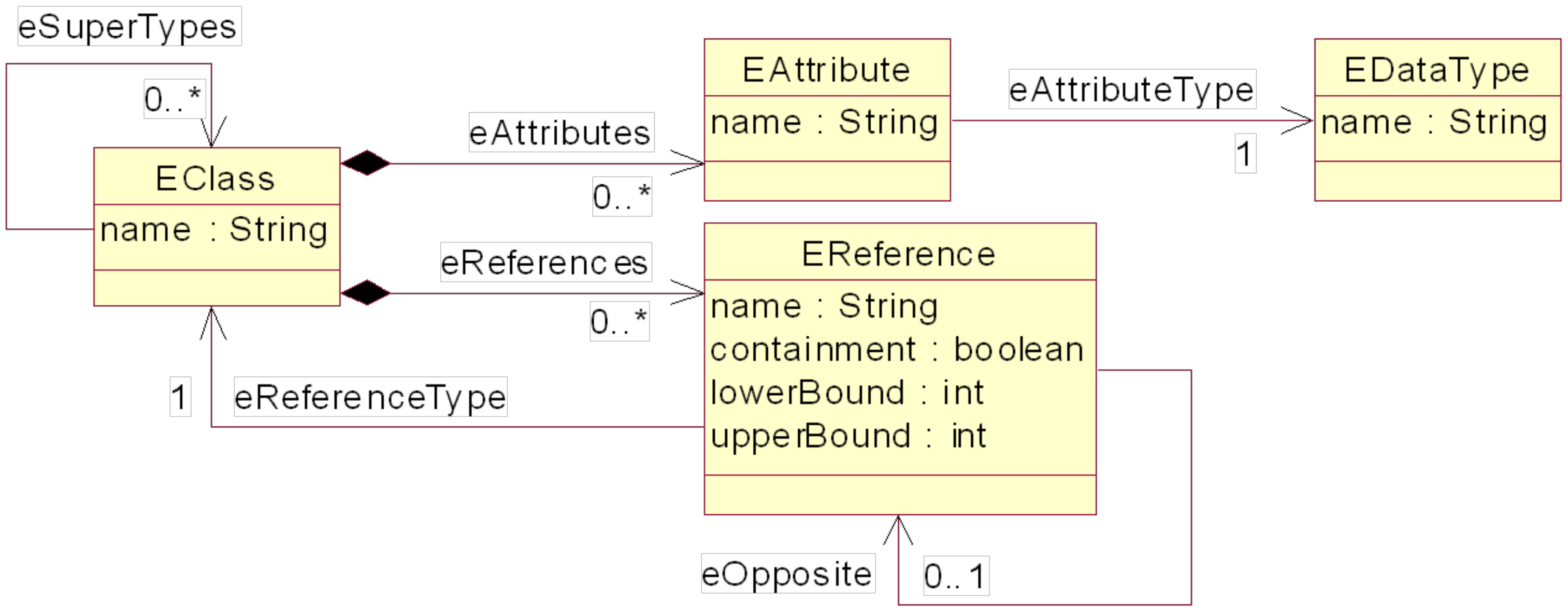
- **EMF**
  - The core EMF framework includes a **meta-model** (**Ecore**)
    - for describing models
    - **runtime support** for the models including change notification,
    - **persistence support** with default XMI serialization,
    - **reflective API** for manipulating EMF objects generically.
- **EMF.Edit**
  - Generic reusable **classes for building editors** for EMF models.
- **EMF.Codegen**
  - Capable of generating everything needed **to build a complete editor** for an EMF model.
  - **Includes a GUI** from which generation options can be specified, and generators can be invoked.

EMF in the meta-modelling architecture



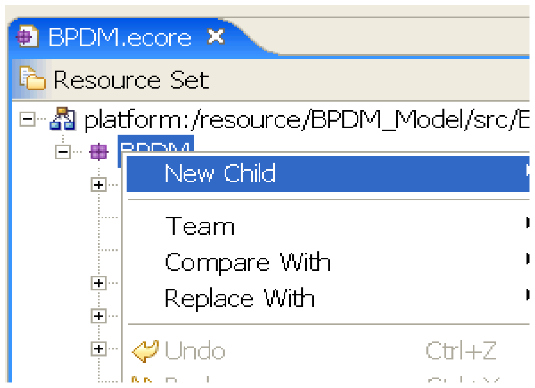
EMF architecture: Ecore

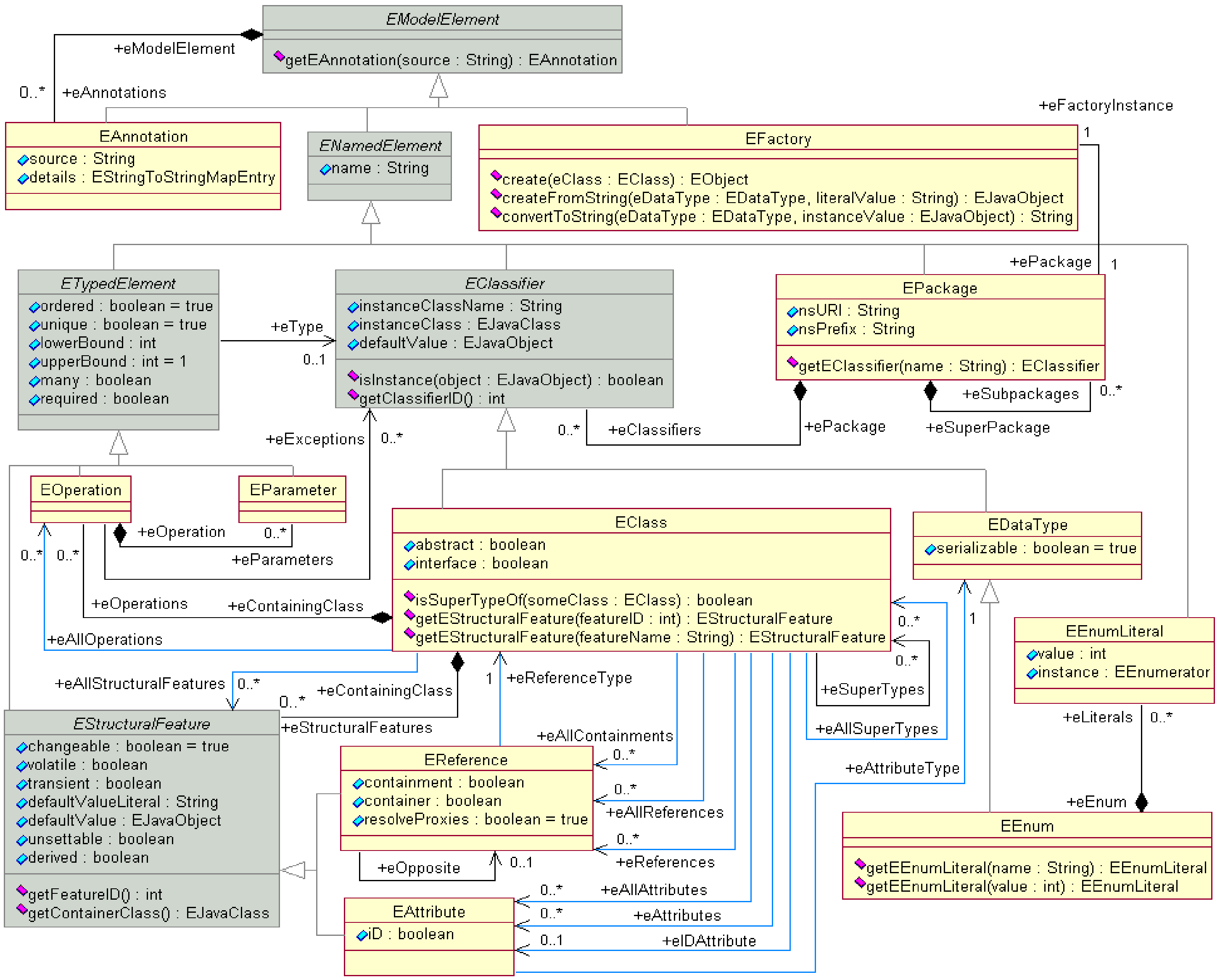
- Ecore is EMF’s model of models (meta-model)
  - Persistent representation is XMI
  - Can be seen as an implementation of UML Core::Basic



Ecore: Overview

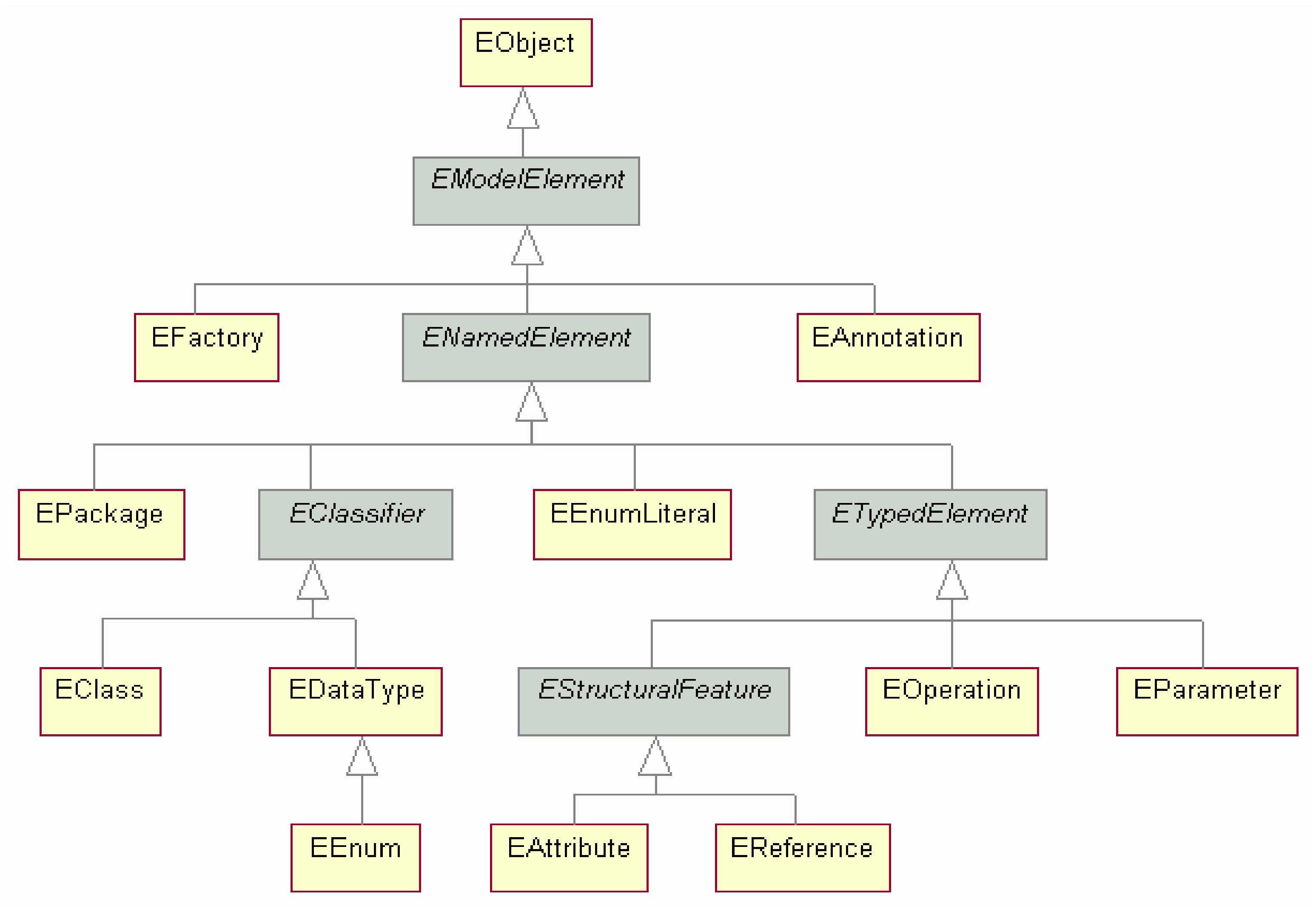
used for meta-modelling



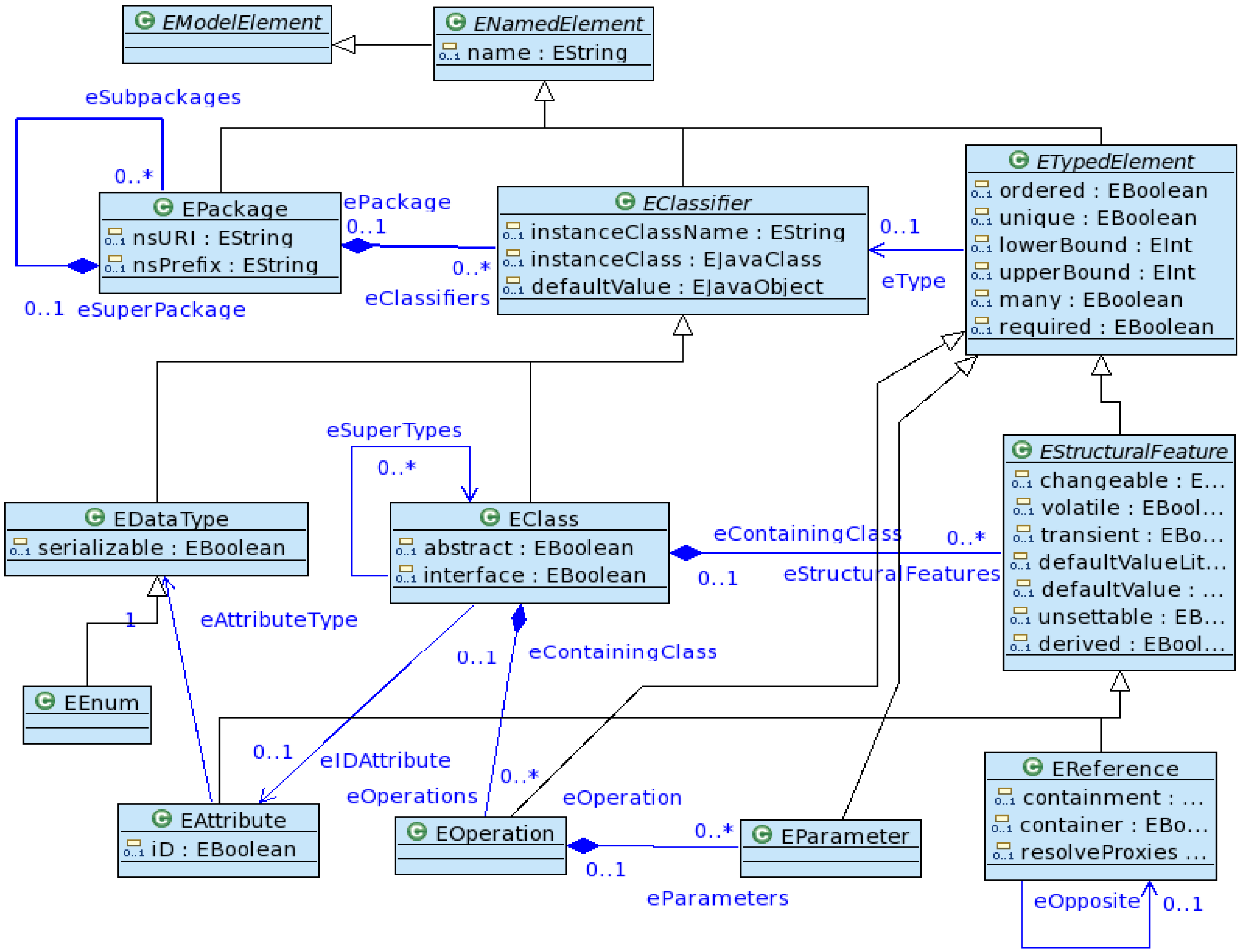


Modelling with UML, with semantics

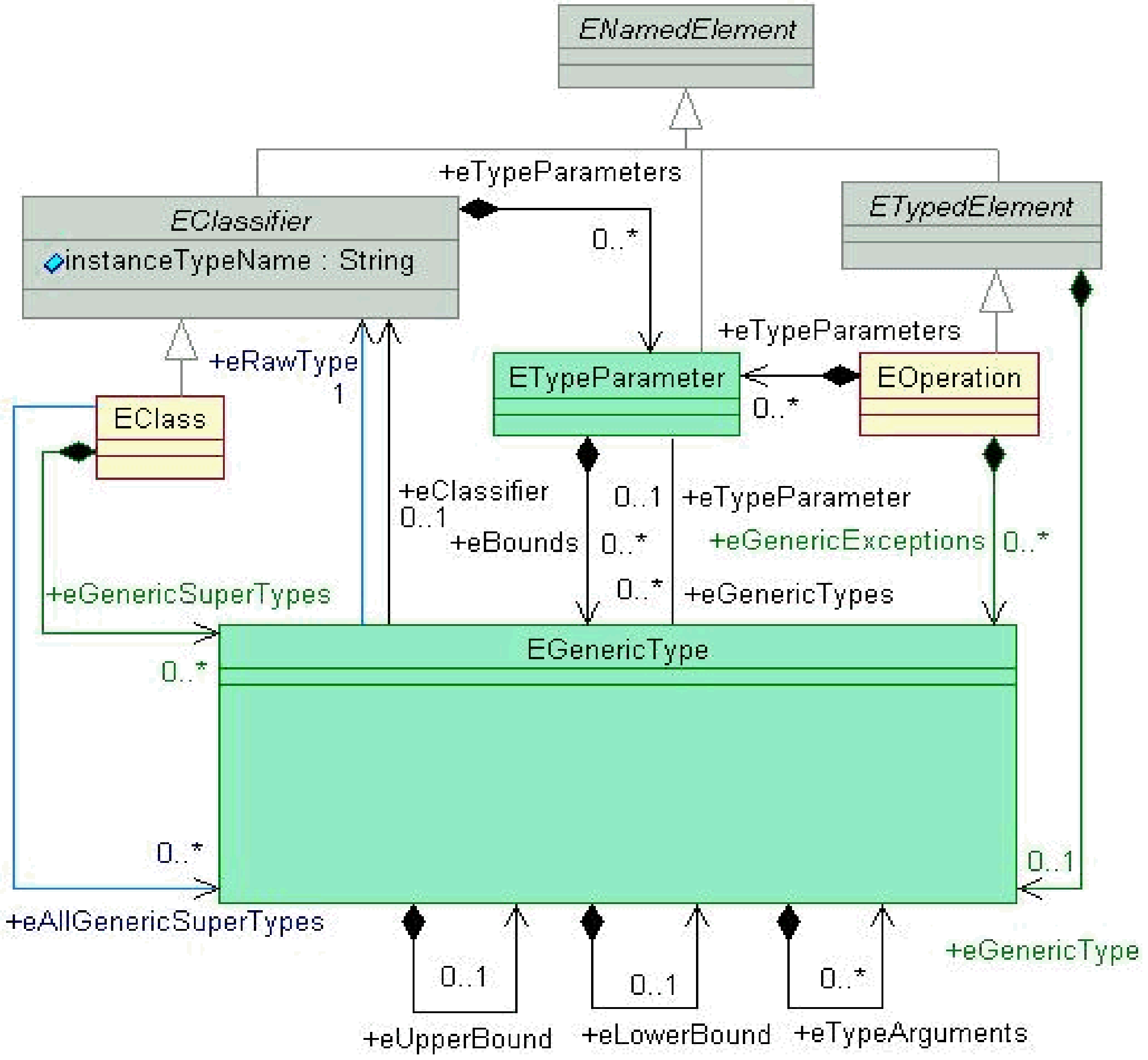
Ecore: Inheritance hierarchy



Ecore: Associations



Ecore: Generics



EMF model definition (1)

- Specification of an application's data
  - Object attributes
  - Relationships (associations) between objects
  - Operations available on each object
  - Simple constraints (e.g., multiplicity) on objects and relationships

EMF model definition: Programming (1)

```
import java.io.*;
import java.util.*;
import org.eclipse.emf.ecore.*;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.resource.*;
import org.eclipse.emf.ecore.resource.impl.*;
import org.eclipse.emf.ecore.xmi.impl.EcoreResourceFactoryImpl;

public class EMFTest {
  public static void main(String[] args) {
    EcoreFactory ecoreFactory = EcoreFactory.eINSTANCE;
```

factory for Ecore meta-models

EMF model definition: Programming (2)

```
EPackage aPackage = ecoreFactory.createEPackage();
aPackage.setName("somePackage");
aPackage.setNsPrefix("pkg");
aPackage.setNsURI("urn:www.pst.ifi.lmu.de/knapp/pkg");

EClass aClass = ecoreFactory.createEClass();
aClass.setName("SomeClass");
aPackage.getEClassifiers().add(aClass);

EAttribute anAttribute = ecoreFactory.createEAttribute();
anAttribute.setName("someAttribute");
anAttribute.setEType(ecoreFactory.getEcorePackage().
  getEString());
aClass.getEStructuralFeatures().add(anAttribute);

EReference aReference = ecoreFactory.createEReference();
aReference.setName("someReference");
aReference.setEType(aClass);
aClass.getEStructuralFeatures().add(aReference);
```

namespace settings

EMF model definition: Programming (3)

```
    try {
      Resource.Factory.Registry.INSTANCE.
        getExtensionToFactoryMap().put("ecore",
          new EcoreResourceFactoryImpl());
      ResourceSet resourceSet = new ResourceSetImpl();
      Resource resource = resourceSet.
        createResource(URI.createFileURI("test.ecore"));
      resource.getContents().add(aPackage);

      StringWriter stringWriter = new StringWriter();
      URIConverter.WriteableOutputStream outputStream =
        new URIConverter.WriteableOutputStream(stringWriter, "UTF-8");
      Map<String, String> options = new HashMap<String, String>();
      resource.save(outputStream, options);
      System.out.println(stringWriter.toString());
    } catch (IOException ioe) {
        ioe.printStackTrace();  }
  }
}
```

for saving as Ecore meta-model

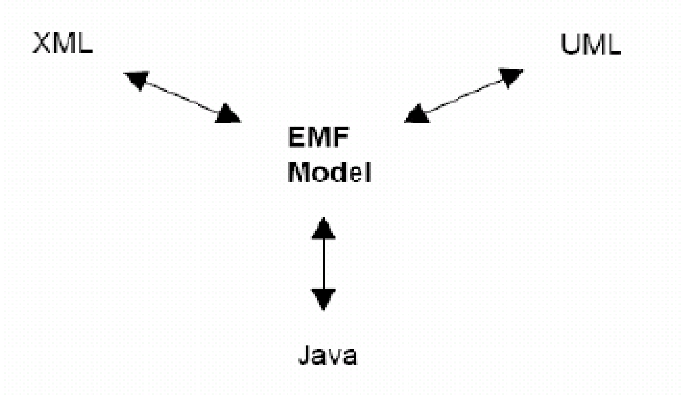
options for resources
(compress, encrypt, save only when modified, progress monitor, &c.)

EMF model definition (2)

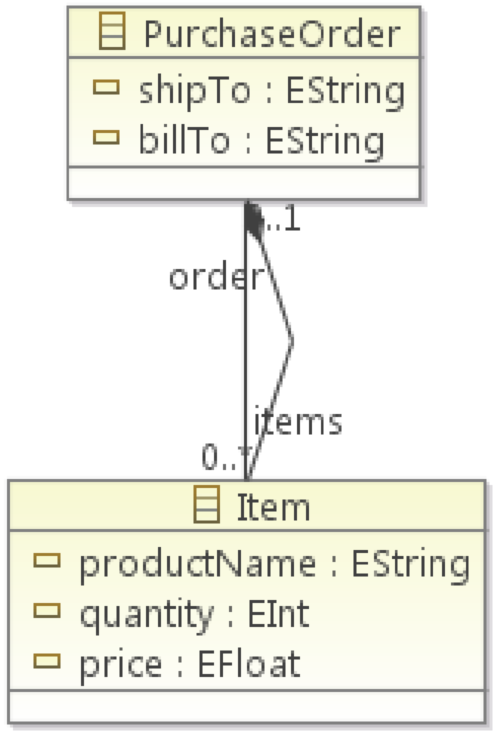
- Unifying Java, XML, and UML technologies

- All three forms provide the same information
  - Different visualization/representation
  - The application's "model" of the structure

- EMF models can be defined in (at least) four ways:
  1. ECore diagram
  2. Java interfaces
  3. UML Class Diagram
  4. XML Schema
- EMF can generate the others as well as the implementation code



EMF model definition: ECore diagrams



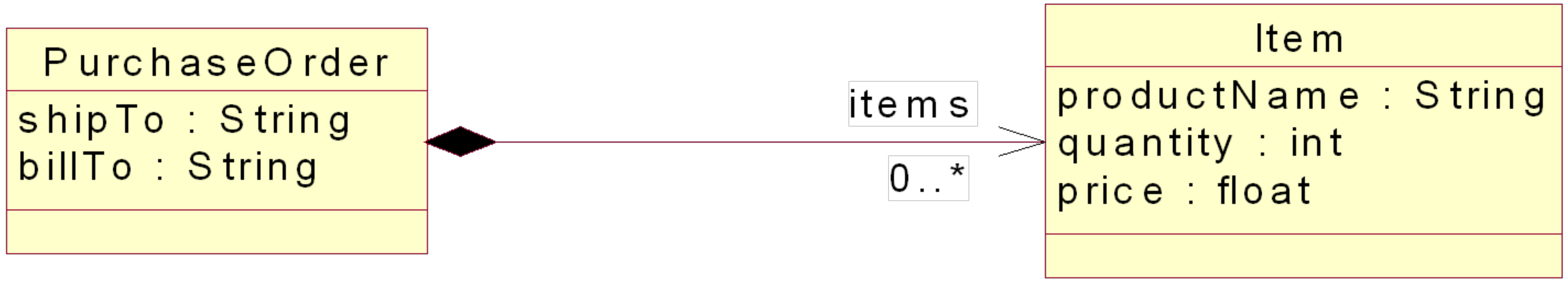
EMF model definition: Annotated Java interfaces

```
/** @model */
public interface PurchaseOrder {
  /** @model */ String getShipTo();
  /** @model */ String getBillTo();
  /** @model containment="true" opposite="order" */
  List<Item> getItems();
}

/** @model */
public interface Item {
  /** @model opposite="items" */
  PurchaseOrder getOrder();
  /** @model */ String getProductName();
  /** @model */ int getQuantity();
  /** @model */ float getPrice();
}
```

- Setter methods for attributes generated

EMF model definition: UML class diagrams



- From Rational Software Architect, Eclipse UML 2, &c.

EMF model definition: XML Schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/purchase"
        xmlns:tns="http://www.example.org/purchase">
  <complexType name="PurchaseOrder">
    <sequence>
      <element name="shipTo" type="string"/>
      <element name="billTo" type="string"/>
      <element name="items"  type="tns:Item"
               minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="id" type="ID"/>
  </complexType>
  <complexType name="Item">
    <sequence>
      <element name="order" type="IDREF" minOccurs="1" maxOccurs="1"/>
      <element name="productName" type="string"/>
      <element name="quantity" type="int"/>
      <element name="price" type="float"/>
    </sequence>
  </complexType>
</schema>
```

EMF architecture: Ecore/XMI (1)

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="purchase"
  nsURI="http://www.example.org/purchase" nsPrefix="purchase">
  <eClassifiers xsi:type="ecore:EClass" name="Item">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="order"
       lowerBound="1" eType="ecore:EDataType
         http://www.eclipse.org/emf/2003/XMLType#//IDREF"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
       name="productName" lowerBound="1"
       eType="ecore:EDataType
         http://www.eclipse.org/emf/2003/XMLType#//String"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="quantity"
        lowerBound="1" eType="ecore:EDataType
        http://www.eclipse.org/emf/2003/XMLType#//Int"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="price"
        lowerBound="1" eType="ecore:EDataType
          http://www.eclipse.org/emf/2003/XMLType#//Float"/>
  </eClassifiers>
```

EMF architecture: Ecore/XMI (2)

```
  <eClassifiers xsi:type="ecore:EClass" name="PurchaseOrder">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="shipTo"
       lowerBound="1" eType="ecore:EDataType
         http://www.eclipse.org/emf/2003/XMLType#//String"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="billTo"
       lowerBound="1" eType="ecore:EDataType
         http://www.eclipse.org/emf/2003/XMLType#//String"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="items"
        upperBound="-1" eType="#//Item" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="id"
        eType="ecore:EDataType
          http://www.eclipse.org/emf/2003/XMLType#//ID" iD="true"/>
</eClassifiers>
</ecore:EPackage>
```

- Alternative serialisation format is EMOF/XMI

EMF.Codegen: Interface completion

```
/** @model */
public interface Item extends EObject {
  /** @model opposite="items" */ PurchaseOrder getOrder();
  /** @generated */ void setOrder(PurchaseOrder value);
  /** @model */ String getProductName();
  /** @generated */ void setProductName(String value);
  /** @model */ int getQuantity();
  /** @generated */ void setQuantity(int value);
  /** @model */ float getPrice();
  /** @generated */ void setPrice(float value);
}
```

- No regeneration of implementations when changing `@generated` to `@generated NOT`

EMF.Codegen: Implementation of associations (1)

- Proper handling of binary associations
  - changes on either side of an association propagated to the other

- Special handling of composite associations
  - only a single container, stored in `eContainerFeatureID`

```
public PurchaseOrder getOrder() {
  if (eContainerFeatureID != PurchasePackage.ITEM__ORDER)
    return null;
  return (PurchaseOrder)eContainer();
}
```

EMF.Codegen: Implementation of associations (2)

```
public void setOrder(PurchaseOrder newOrder) {
  if (newOrder != eInternalContainer() ||
      (eContainerFeatureID != PurchasePackage.ITEM__ORDER &&
       newOrder != null)) {
    if (EcoreUtil.isAncestor(this, newOrder))
      throw new IllegalArgumentException("Recursive containment " +
        "not allowed for " + toString());
    NotificationChain msgs = null;
    if (eInternalContainer() != null)
      msgs = eBasicRemoveFromContainer(msgs);
    if (newOrder != null)
      msgs = ((InternalEObject)newOrder).eInverseAdd(this,
        PurchasePackage.PURCHASE_ORDER__ITEMS,
        PurchaseOrder.class, msgs);
    msgs = basicSetOrder(newOrder, msgs);
    if (msgs != null) msgs.dispatch();
  }
  else
    if (eNotificationRequired())
      eNotify(new ENotificationImpl(this, Notification.SET,
        PurchasePackage.ITEM__ORDER, newOrder, newOrder));
}
```

single container

consistent update for
binary associations

EMF: Creating models with generated code

```
PurchaseFactory purchaseFactory = PurchaseFactory.eINSTANCE;

PurchaseOrder order1 = purchaseFactory.createPurchaseOrder();
order1.setBillTo("X");
order1.setShipTo("Y");
Item item1 = purchaseFactory.createItem();
item1.setProductName("A");
item1.setQuantity(2);
item1.setPrice(10.0f);
item1.setOrder(order1);
Item item2 = purchaseFactory.createItem();
item2.setProductName("B");
item2.setQuantity(3);
item2.setPrice(100.0f);
item2.setOrder(order1);
```

factory for purchase models

EMF: Saving models

```
ResourceSet resourceSet = new ResourceSetImpl();
resourceSet.getResourceFactoryRegistry().
  getExtensionToFactoryMap().put("xmi", new XMIResourceFactoryImpl());

URI fileURI = URI.createFileURI(new File("orders.xmi").getAbsolutePath());
Resource resource = resourceSet.createResource(fileURI);

resource.getContents().add(order1);

try {
  resource.save(System.out, Collections.EMPTY_MAP);
  resource.save(Collections.EMPTY_MAP);
}
catch (IOException ioe) {
  ioe.printStackTrace();
}
```

mind containment
not `resource.getContents().add(item1);`

EMF: Ecore/XMI from generated code

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:purchaseJava="http:///purchaseJava.ecore">
  <purchaseJava:PurchaseOrder shipTo="Y" billTo="X">
    <items productName="A" quantity="2" price="10.0"/>
    <items productName="B" quantity="3" price="100.0"/>
  </purchaseJava:PurchaseOrder>
</xmi:XMI>
```

containment

EMF: Querying with OCL (1)

```
import org.eclipse.ocl.ecore.OCL;
import org.eclipse.ocl.ParserException;
import org.eclipse.ocl.OCLInput;
import org.eclipse.ocl.ecore.Constraint;
```

```
OCL purchaseOCL = OCL.newInstance();
try {
  purchaseOCL.parse(new OCLInput("package purchaseJava " +
                                   "context Item " +
                                   "inv: price < 100.0 " +
                                  "endpackage"));
  for (Constraint constraint : purchaseOCL.getConstraints()) {
    System.out.println(purchaseOCL.check(item2, constraint));
  }
}
catch (ParserException e) {
  e.printStackTrace();
}
```

EMF: Querying with OCL (2)

```
import org.eclipse.ocl.ecore.OCL;
import org.eclipse.ocl.ParserException;
import org.eclipse.ocl.expressions.OCLExpression;
import org.eclipse.ocl.helper.OCLHelper;
```

parametric in classifier, operation, property, and constraint representation of the meta-model

```
OCL purchaseOCL = OCL.newInstance();
OCLHelper<EClassifier, ?, ?, ?> purchaseOCLHelper =
  purchaseOCL.createOCLHelper();
purchaseOCLHelper.setContext(PurchaseJavaPackage.Literals.ITEM);
try {
  OCLExpression<EClassifier> priceExpression =
    purchaseOCLHelper.createQuery("price");
  System.out.println(purchaseOCL.evaluate(item1, priceExpression));
}
catch (ParserException e) {
  e.printStackTrace();
}
```

convenient for embedded OCL constraints

EMF: Querying with OCL (3)

```
import org.eclipse.emf.query.conditions.eobjects.EObjectCondition;
import org.eclipse.emf.query.ocl.conditions.BooleanOCLCondition;
import org.eclipse.emf.query.statements.FROM;
import org.eclipse.emf.query.statements.SELECT;
import org.eclipse.emf.query.statements.WHERE;
import org.eclipse.emf.query.statements.IQueryResult;
```

parametric in classifier, class, and element of the meta-model

context

```
try {
  EObjectCondition itemsOK =
    new BooleanOCLCondition<EClassifier, EClass, EObject>(
          purchaseOCL.getEnvironment(),
          "self.quantity < 2", PurchaseJavaPackage.Literals.ITEM);
  IQueryResult result = new SELECT(
                          new FROM(resource.getContents()),
                          new WHERE(itemsOK)).execute();
  for (Object next : result) {
    System.out.println("Quantity too little in " +
                       ((Item) next).getProductName());
  }
} catch (ParserException pe) {
  pe.printStackTrace();
}
```

EMF: Querying UML models with OCL

```
import org.eclipse.uml2.uml.UMLFactory;
```

```
UMLFactory umlFactory = UMLFactory.eINSTANCE;
org.eclipse.uml2.uml.Activity activity = umlFactory.createActivity();
activity.setName("test");
OCL umlOCL = OCL.newInstance();
try {
  umlOCL.parse(new OCLInput("package uml " +
                            "context Activity " +
                               "inv: name <> '' " +
                            "endpackage"));
  for (Constraint constraint : umlOCL.getConstraints()) {
    System.out.println(umlOCL.check(activity, constraint));
  }
} catch (ParserException e) {
  e.printStackTrace();
}
```

Xtext

Xtext

- Grammar language for describing domain-specific languages textually
  - Based on LL(*)-parser generator ANTLR
  - Generation of Eclipse-integrated editor (with validator, content assist, outline, formatting, …)
- Tightly integrated with EMF
  - Ecore meta-model inference from grammar
- Model querying (and transformation) with Xtend
- Model-to-text transformation with Xpand
- Integration into EMFT's Modeling Workflow Engine (MWE)
  - Dependency injection using Google's Guice

- Originally developed in the openArchitectureWare project (2006)
- Since 2008 integrated in the Textual Modeling Framework (TMF) of EMF
  - Current version (July 2011): Xtext 2.0

`http://www.eclipse.org/Xtext`

DSL example: Secret compartments (1)

“I have vague but persistent childhood memories of watching cheesy adventure films on TV. Often, these films would be set in some old castle and feature secret compartments or passages. In order to find them, heroes would need to pull the candle holder at the top of stairs and tap the wall twice.

Let’s imagine a company that decides to build security systems based on this idea. They come in, set up some kind of wireless network, and install little devices that send four-character messages when interesting things happen. For example, a sensor attached to a drawer would send the message D2OP when the drawer is opened. We also have little control devices that respond to four-character command messages—so a device can unlock a door when it hears the message D1UL.

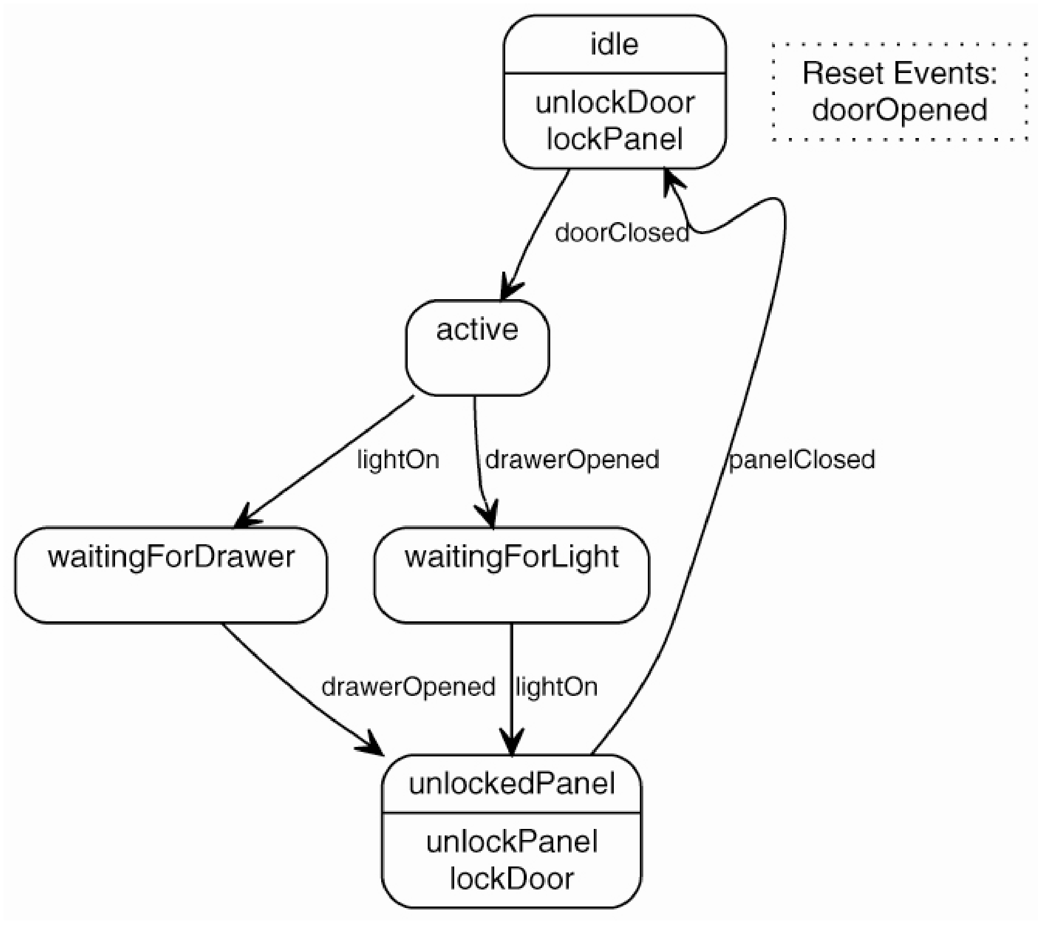
At the center of all this is some controller software that listens to event messages, figures out what to do, and sends command messages. The company bought a job lot of Java-enabled toasters during the dot-com crash and is using them as the controllers. So whenever a customer buys a gothic security system, they come in and fit the building with lots of devices and a toaster with a control program written in Java.

For this example, I’ll focus on this control program. Each customer has individual needs, but once you look at a good sampling, you will soon see common patterns.”

DSL example: Secret compartments (2)

“Miss Grant closes her bedroom door, opens a drawer, and turns on a light to access a secret compartment. Miss Shaw turns on a tap, then opens either of her two compartments by turning on correct light. Miss Smith has a secret compartment inside a locked closet inside her office. She has to close a door, take a picture off the wall, turn her desk light on three times, open the top drawer of her filing cabinet—and then the closet is unlocked. If she forgets to turn the desk light off before she opens the inner compartment, an alarm will sound.”

Martin Fowler. Domain-specific Languages, 2010.



DSL example: Secret compartments (3)

```
events
  doorClosed D1CL  drawOpened D2OP  lightOn L1ON  doorOpened D1OP  panelClosed PNCL end
resetEvents
  doorOpened end
commands
  unlockPanel PNUL  lockPanel PNLK  lockDoor D1LK  unlockDoor D1UL end

state idle
  actions { unlockDoor lockPanel }
  doorClosed => active end
state active
  drawOpened => waitingForLight
  lightOn => waitingForDraw end
state waitingForLight
  lightOn => unlockedPanel end
state waitingForDraw
  drawOpened => unlockedPanel end
state unlockedPanel
  actions { unlockPanel lockDoor }
  panelClosed => idle end
```

Secret compartments in Xtext: Grammar (1)

```
grammar org.eclipse.xtext.example.fowlerdsl.Statemachine
  with org.eclipse.xtext.common.Terminals

generate statemachine "http://www.eclipse.org/xtext/example/fowlerdsl/Statemachine"
                                              ← name and nsURI of EPackage
Statemachine :
  {Statemachine} ←——— action generating an Ecore object
  ('events'
     events+=Event+
   'end')?
  ('resetEvents'
     resetEvents+=[Event]+
   'end')?          ← cross-reference
  ('commands'
     commands+=Command+
   'end')?
  states+=State*
;
```

Secret compartments in Xtext: Grammar (2)

```
Event:
  name=ID code=ID
;

Command:
  name=ID code=ID
;

State:
  'state' name=ID
  ('actions' '{' actions+=[Command]+ '}')?
  transitions+=Transition*
  'end'
;

Transition:
  event=[Event] '=>' state=[State]
;
```

identifier token from `terminals`

Secret compartments: Code generation with Xtend/Xpand (1)

```
package org.eclipse.xtext.example.fowlerdsl.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.IGenerator
import org.eclipse.xtext.generator.IFileSystemAccess
import org.eclipse.xtext.example.fowlerdsl.statemachine.Statemachine
import org.eclipse.xtext.example.fowlerdsl.statemachine.Event
import org.eclipse.xtext.example.fowlerdsl.statemachine.Command
import org.eclipse.xtext.example.fowlerdsl.statemachine.State

class StatemachineGenerator implements IGenerator {
  override void doGenerate(Resource resource, IFileSystemAccess fsa) {
    fsa.generateFile(resource.className+".java",
                     toJavaCode(resource.contents.head as Statemachine))
  }

  def className(Resource res) {
    var name = res.URI.lastSegment
    return name.substring(0, name.indexOf('.'))
  }
```

Secret compartments: Code generation with Xtend/Xpand (2)

```
  def toJavaCode(Statemachine sm) '''
    import java.io.BufferedReader;
    import java.io.IOException;
    import java.io.InputStreamReader;

    public class «sm.eResource.className» {
      public static void main(String[] args) {
        new «sm.eResource.className»().run();
      }

      «FOR c : sm.commands»
        «c.declareCommand»
      «ENDFOR»

      protected void run() {
        boolean executeActions = true;
        String currentState = "«sm.states.head.name»";
        String lastEvent = null;
        while (true) {
          «FOR state : sm.states»
            «state.generateCode»
          «ENDFOR»
```

Secret compartments: Code generation with Xtend/Xpand (3)

```
          «FOR resetEvent : sm.resetEvents»
            if ("«resetEvent.name»".equals(lastEvent)) {
              System.out.println("Resetting state machine.");
              currentState = "«sm.states.head.name»";
              executeActions = true;
            }
          «ENDFOR»
        }
      }

      private String receiveEvent() {
        System.out.flush();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        try {
          return br.readLine();
        } catch (IOException ioe) {
            System.out.println("Problem reading input");
            return "";
        }
      }
    }
  '''
```

Secret compartments: Code generation with Xtend/Xpand (4)

```
  def declareCommand(Command command) '',
    protected void do«command.name.toFirstUpper»() {
      System.out.println("Executing command «command.name» («command.code»)");
    }
  '''

  def generateCode(State state) '',
    if (currentState.equals("«state.name»")) {
      if (executeActions) {
        «FOR c : state.actions» do«c.name.toFirstUpper»(); «ENDFOR»
        executeActions = false;
      }
      System.out.println("Your are now in state '«state.name»'. Possible events are
[«state.transitions.map(t | t.event.name).join(', ')»].");
      lastEvent = receiveEvent();
      «FOR t : state.transitions»
        if ("«t.event.name»".equals(lastEvent)) {
          currentState = "«t.state.name»";
          executeActions = true;
        }
      «ENDFOR»
    }
  '''
}
```

Secret compartments: Modelling workflow (1)

```
module org.eclipse.xtext.example.fowlerdsl.GenerateStatemachine

import org.eclipse.emf.mwe.utils.*
import org.eclipse.xtext.generator.*
import org.eclipse.xtext.ui.generator.*

var grammarURI = "classpath:/org/eclipse/xtext/example/fowlerdsl/Statemachine.xtext"
var file.extensions = "statemachine"
var projectName = "org.eclipse.xtext.example.fowlerdsl"
var runtimeProject = "../${projectName}"

Workflow {
  bean = StandaloneSetup {
    scanClassPath = true
    platformUri = "${runtimeProject}/.."
  }
  component = DirectoryCleaner {
    directory = "${runtimeProject}/src-gen"
  }
  component = DirectoryCleaner {
    directory = "${runtimeProject}.ui/src-gen"
  }
```

Secret compartments: Modelling workflow (2)

```
component = Generator {
  pathRtProject = runtimeProject
  pathUiProject = "${runtimeProject}.ui"
  pathTestProject = "${runtimeProject}.tests"
  projectNameRt = projectName
  projectNameUi = "${projectName}.ui"
  language = {
    uri = grammarURI
    fileExtensions = file.extensions
    fragment = grammarAccess.GrammarAccessFragment { }
    fragment = ecore.EcoreGeneratorFragment { }
    fragment = serializer.SerializerFragment { }
    fragment = resourceFactory.ResourceFactoryFragment {
      fileExtensions = file.extensions
    }
    fragment = parser.antlr.XtextAntlrGeneratorFragment { }
    fragment = validation.JavaValidatorFragment {
      composedCheck = "org.eclipse.xtext.validation.ImportUriValidator"
      composedCheck = "org.eclipse.xtext.validation.NamesAreUniqueValidator"
    }
    fragment = scoping.ImportNamespacesScopingFragment { }
    fragment = exporting.QualifiedNamesFragment { }
    fragment = builder.BuilderIntegrationFragment { }
```

Secret compartments: Modelling workflow (3)

```
      fragment = generator.GeneratorFragment {
        generateMwe = true
        generateJavaMain = true
      }
      fragment = formatting.FormatterFragment {}
      fragment = labeling.LabelProviderFragment {}
      fragment = outline.OutlineTreeProviderFragment {}
      fragment = outline.QuickOutlineFragment {}
      fragment = quickfix.QuickfixProviderFragment {}
      fragment = contentAssist.JavaBasedContentAssistFragment {}
      fragment = parser.antlr.XtextAntlrUiGeneratorFragment {}
      fragment = junit.Junit4Fragment {}
      fragment = types.TypesGeneratorFragment {}
      fragment = xbase.XbaseGeneratorFragment {}
      fragment = templates.CodetemplatesGeneratorFragment {}
      fragment = refactoring.RefactorElementNameFragment {}
      fragment = compare.CompareFragment {
        fileExtensions = file.extensions
      }
    }
  }
}
```

Model Transformations

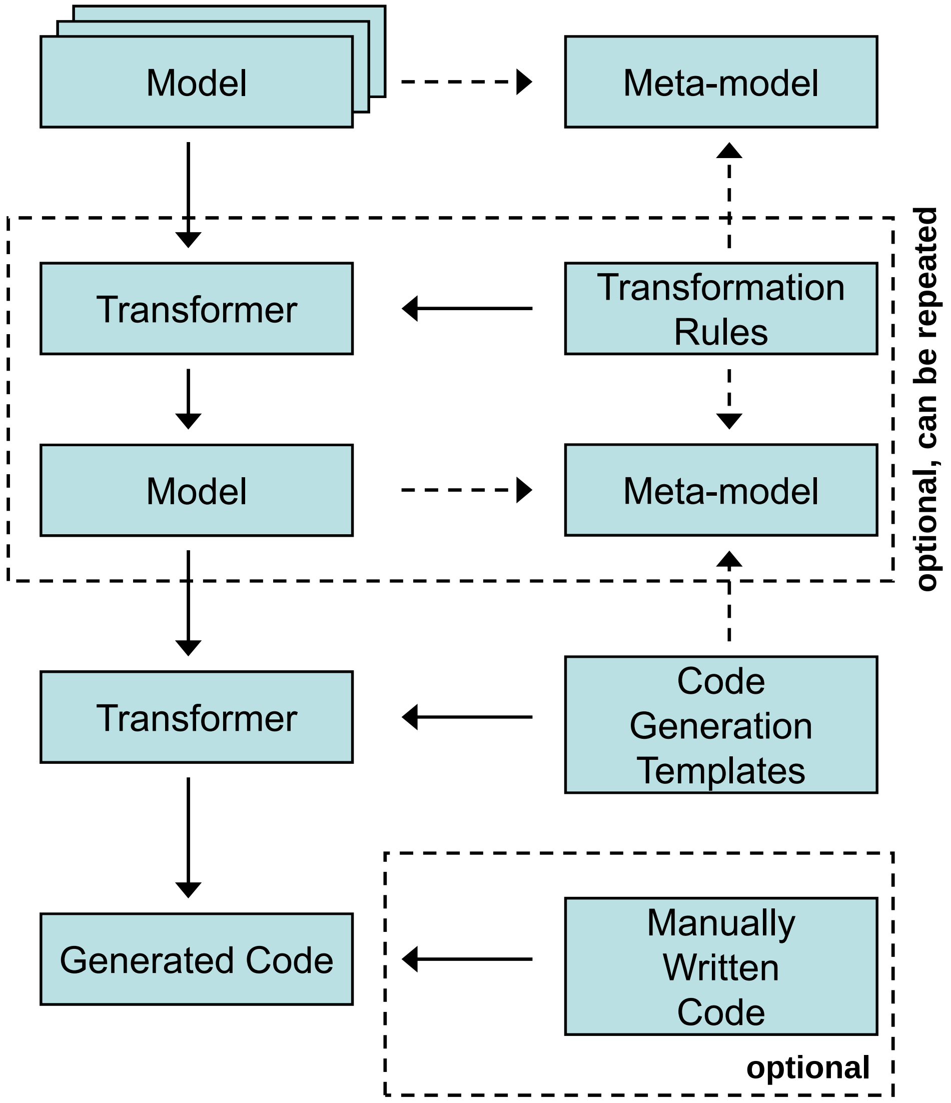
What is a transformation?

- A **transformation** is the automatic generation of a target model from a source model, according to a transformation definition.
- A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.
- A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.
  - Unambiguous specifications of the way that (part of) one model can be used to create (part of) another model

- Preferred characteristics of transformations
  - **semantics-preserving**

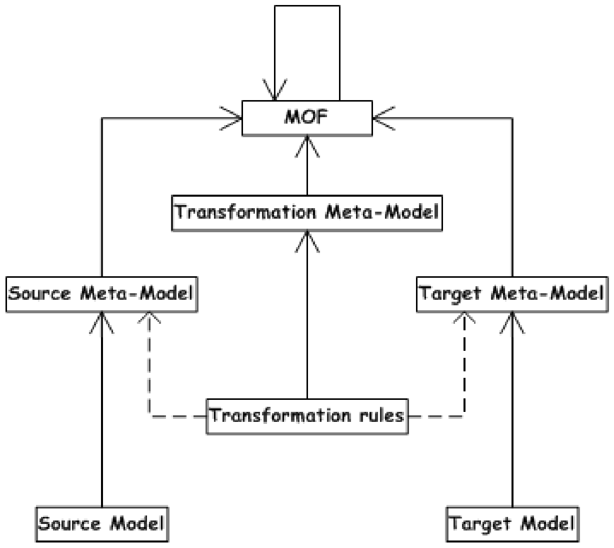
Model-to-model vs. Model-to-code

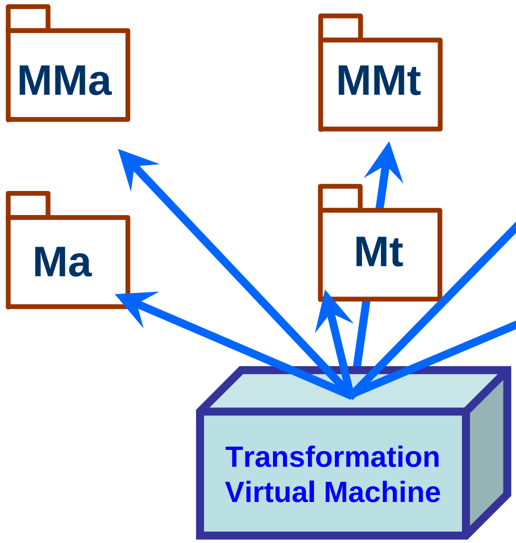
- **Model-to-model** transformations
  - Transformations may be between different languages. In particular, between different languages defined by MOF
- **Model-to-text** transformations
  - Special kind of model to model transformations
  - MDA TS to Grammar TS



Transformations as models

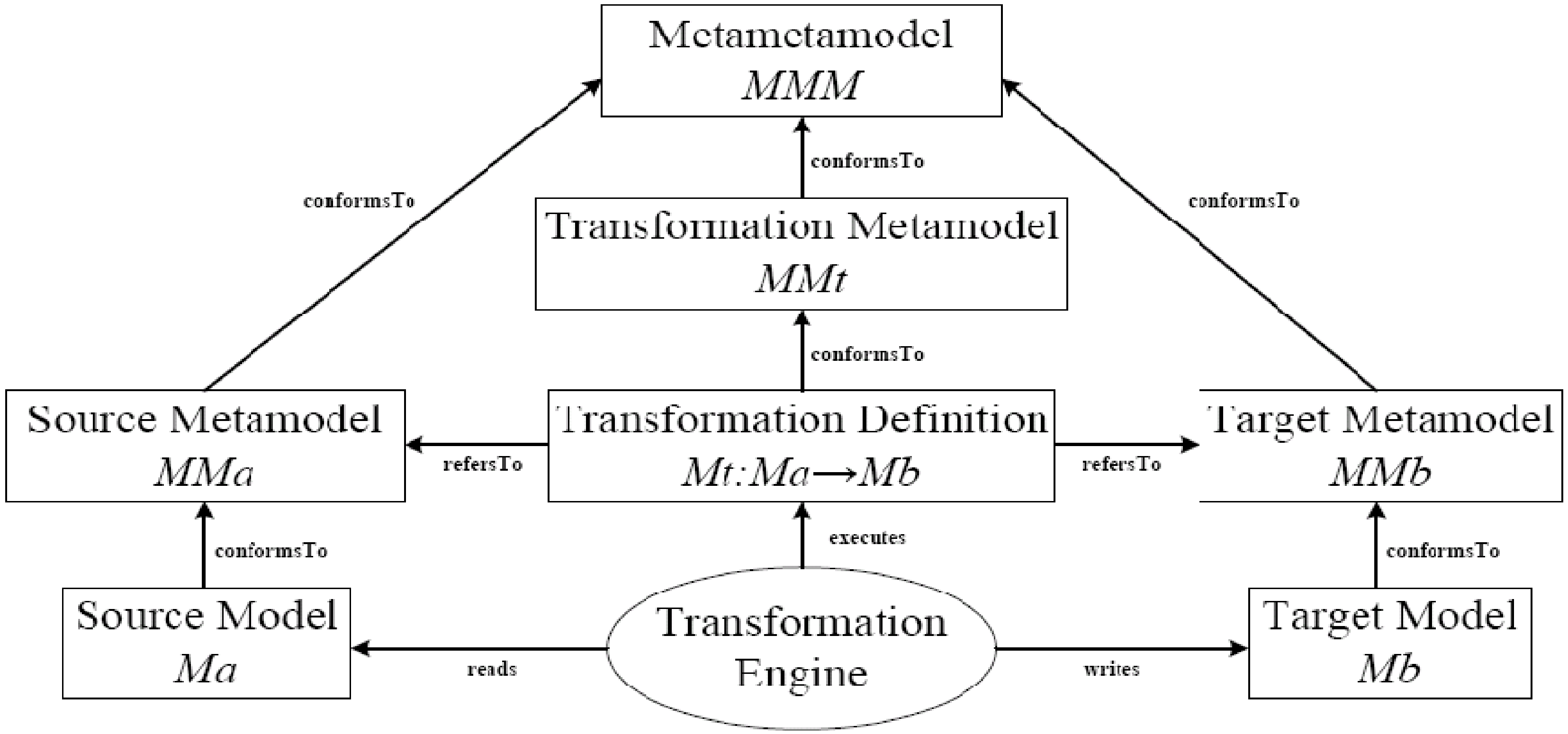
- Treating everything as a model leads not only to conceptual simplicity and regular architecture, but also to implementation efficiency.
- An implementation of a transformation language can be composed of a transformation virtual machine plus a metamodel-driven compiler.
- The transformation VM allows uniform access to model and metamodel elements.





Model transformation

- Each model conforms to a metamodel.
- A transformation builds a target model (Mb) from a source model (Ma).
- A transformation is a model (Mt, here) conforming to a metamodel (MMt).



Characterisation of model transformations (1)

- **Endogenous** vs. **exogenous**
  - **Endogenous** transformations are transformations between models expressed in the same metamodel. Endogenous transformations are also called **rephrasing**
    - Optimisation, refactoring, simplification, and normalization of models.
  - Transformations between models expressed using different meta-models are referred to as **exogenous** transformations or **translations**
    - Synthesis of a higher-level specification into a lower-level one, reverse engineering, and migration from a program written in one language to another

- **Horizontal** vs. **vertical**
  - **Horizontal** transformations are transformations where the source and target models reside at the same abstraction level
    - Refactoring (an endogenous transformation) and migration (an exogenous transformation)
  - **Vertical** transformations are transformation where the source and target models reside at different abstraction levels
    - Rrefinement, where a specification is gradually refined into a full-fledged implementation

Characterisation of model transformations (2)

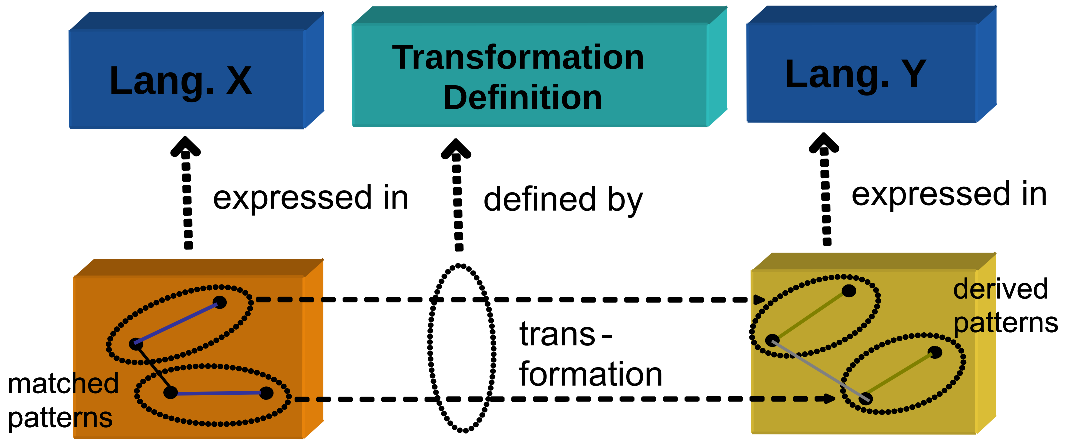
- **Level of automation**
  - The level of automation is the grade to which a model transformation can be automated.

- **Complexity**
  - Simple transformations
    - Mappings for identifying relations between source and target model elements
  - Complex transformations
    - Synthesis, where higher-level models are refined to lower-level models

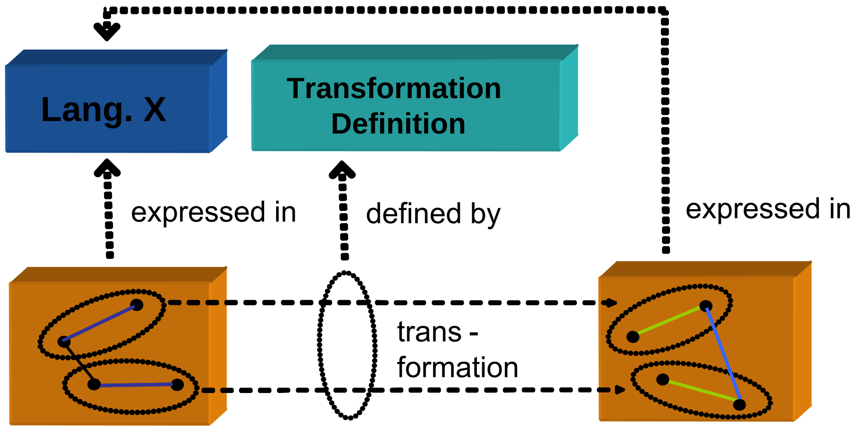
- **Preservation**
  - Each transformation preserves certain aspects of the source model in the transformed target model.
  - The properties that are preserved can differ significantly depending on the type of transformation.
    - With refactorings the (external) behaviour needs to be preserved, while the structure is modified.
    - With refinements, the program correctness needs to be preserved.

Characterisation of model transformations (3)

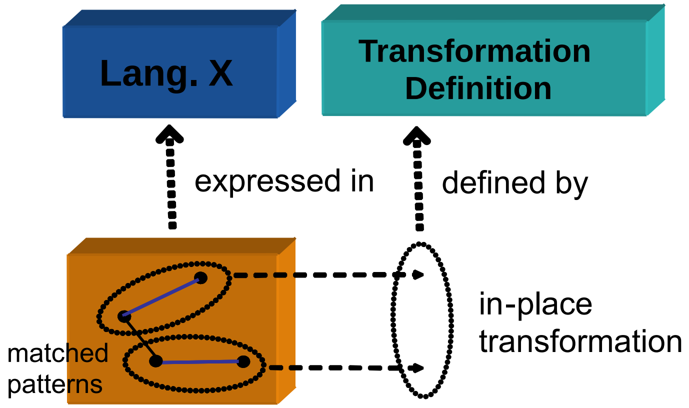
Transformation = Matching and deriving patterns



Transformation in the same meta-model

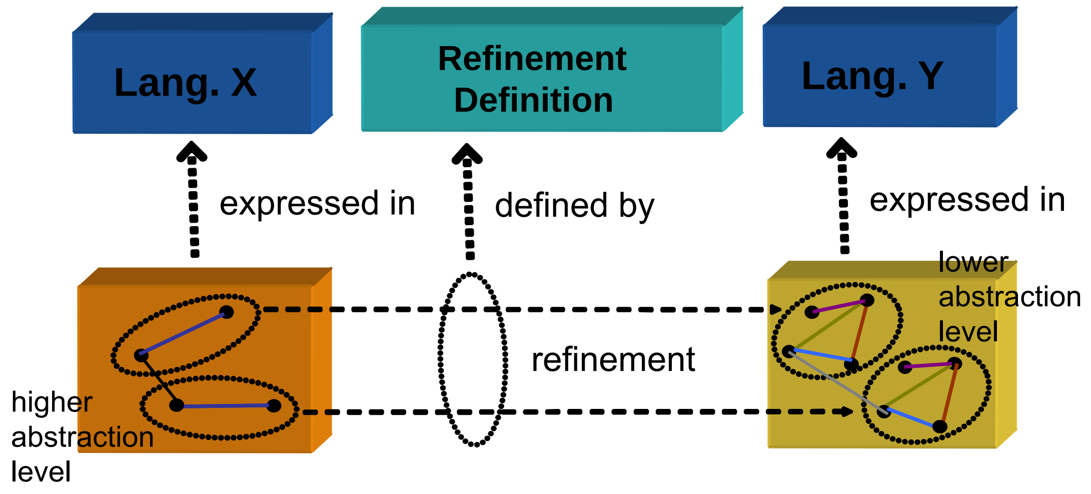


Transformation in the same model

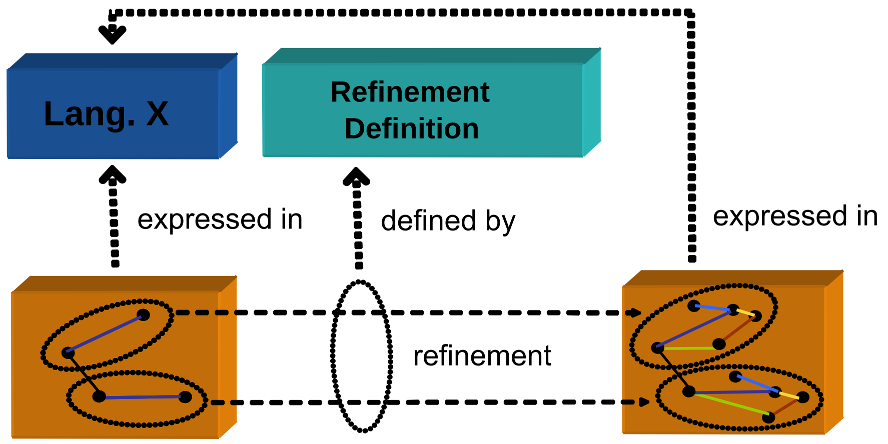


Characterisation of model transformations (4)

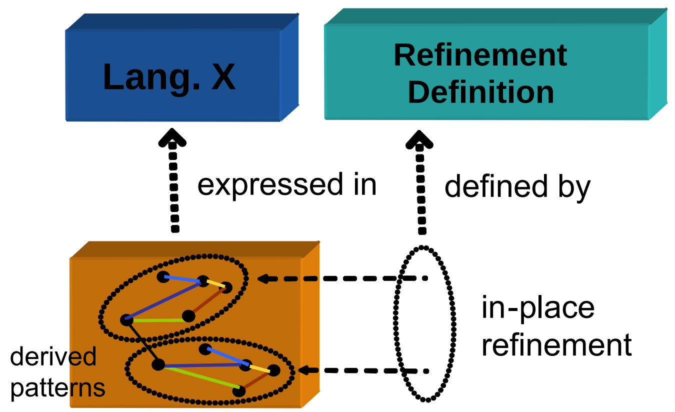
Refinement preserve meaning and derives complex patterns



Refinement in the same meta-model

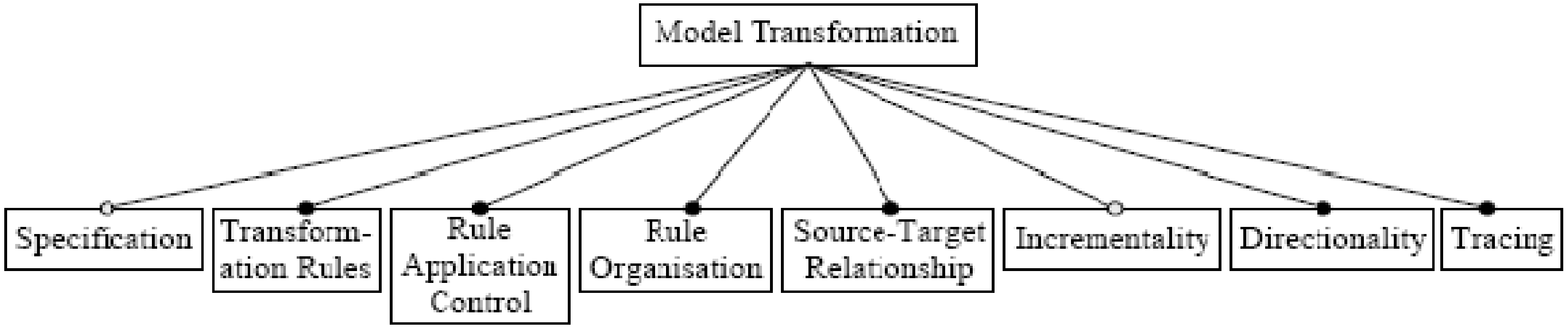


Refinement in the same model



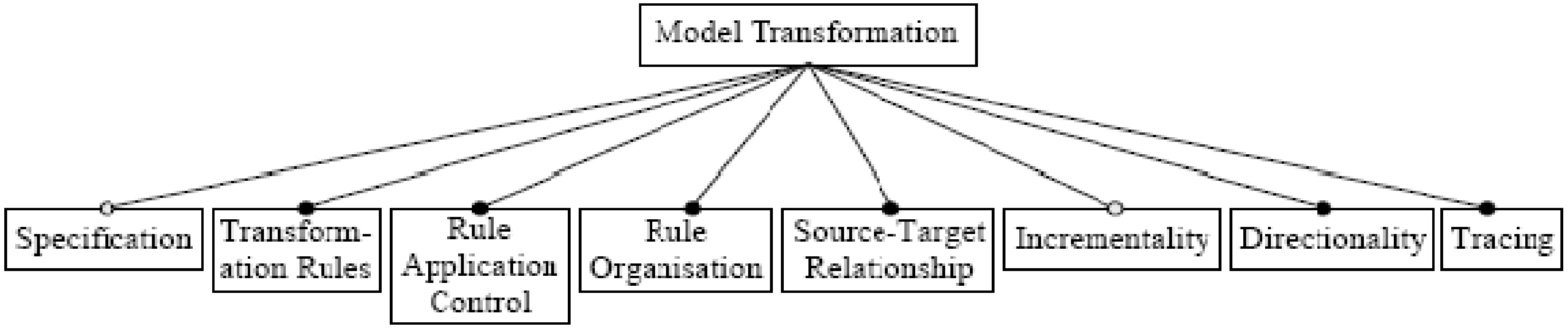
Features of model transformations

- **Specification**
  - Some approaches provide a dedicated specification mechanism, such as pre-/post-conditions expressed in OCL.
- **Transformation rules**
  - A transformation rule consists of two parts:
    - A left-hand side (LHS), which accesses the source model
    - A right-hand side right-hand side (RHS), which expands in the target model
  - A **domain** is the rule part used for accessing the models on which the rule operates
  - The **body** of a domain can be divided into three subcategories
    - Variables: Variables may hold elements from the source and/or target models
    - Patterns: Patterns are model fragments with zero or more variables
    - Logic:. Logic expresses computations and constraints on model elements
  - The transformations variables and patterns can be **typed**.



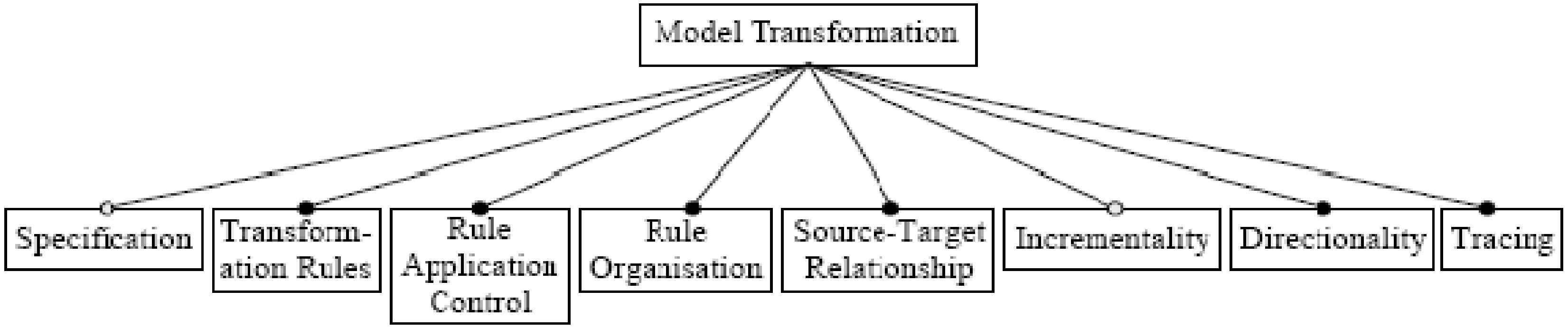
Features of model transformations

- **Rule application control**
  - **Location determination** is the strategy for determining the model locations to which transformation rules are applied.
  - **Scheduling** determines the order in which transformation rules are executed.
- **Rule organisation**
  - Rule organisation is concerned with composing and structuring multiple transformation rules by mechanisms such as modularisation and reuse.
- **Source-target relationship**
  - whether source and target are one and the same model or two different models
    - Create new models
    - Update existing models
    - In-place update



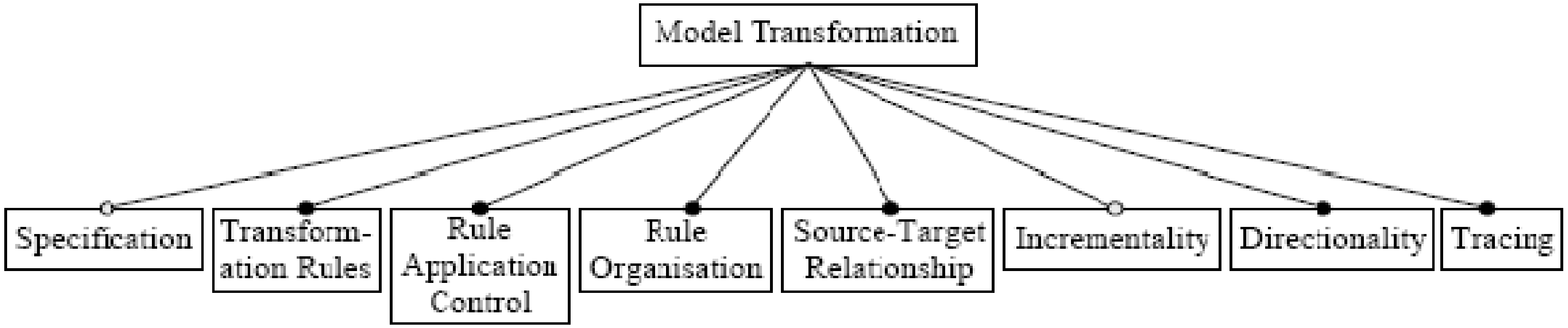
Features of model transformations

- **Incrementality**
  - Ability to update existing target models based on changes in the source models
- **Directionality**
  - Unidirectional transformations can be executed in one direction only, in which case a target model is computed (or updated) based on a source model
  - Multidirectional transformations can be executed in multiple directions, which is particularly useful in the context of model synchronisation.



Features of model transformations

- **Tracing**
  - Mechanisms for recording different aspects of transformation execution, such as creating and maintaining trace links between source and target model elements.
  - Trace information can be useful in
    - performing impact analysis (i.e. analyzing how changing one model would affect other related models),
    - determining the target of a transformation as in model synchronization
    - model-based debugging (i.e. mapping the stepwise execution of an implementation back to its high-level model)
    - debugging model transformations themselves



Model-to-model approaches (1)

- **Direct manipulation approaches**
  - Offers an internal model representation and some APIs to manipulate it
  - Usually implemented as an object-oriented framework
  - Users usually have to implement transformation rules, scheduling, tracing, etc.
  - Examples: Java Metadata Interface (JMI), EMF, …
- **Structure-driven approaches**
  - Two distinct phases:
    - The first phase is concerned with creating the hierarchical structure of the target model
    - The second phase sets the attributes and references in the target
  - The overall framework determines the scheduling and application strategy; users are only concerned with providing the transformation rules
  - Example: OptimalJ

Model-to-model approaches (2)

- **Template-based approaches**
  - Model templates are models with embedded meta-code that compute the variable parts of the resulting template instances.
  - Model templates are usually expressed in the concrete syntax of the target language, which helps the developer to predict the result of template instantiation
  - Typical annotations are conditions, iterations, and expressions, all being part of the meta-language. An expression language to be used in the meta-language is OCL.
  - Examples: Czarnecki, Antkiewicz (2005)

- **Operational approaches**
  - Similar to direct manipulation but offer more dedicated support for model transformation
  - Extend the utilized metamodeling formalism with facilities for expressing computations
    - Extend a query language such as OCL with imperative constructs.
    - The combination of MOF with such extended executable OCL becomes a fully-fledged object-oriented programming system.)
  - Examples: QVT Operational mappings, XMF-Mosaic’s executable MOF, MTL, C-SAW, Kermeta, etc.

Model-to-model approaches (3)

- **Relational approaches**
  - Declarative approaches in which the main concept is mathematical relations
  - The basic idea is to specify the relations among source and target element types using constraints
  - Since declarative constraints are non-executable, declarative approaches give them an executable semantics, such as in logic programming
  - Relational approaches are side-effect-free, support multidirectional rules, can provide backtracking …
  - Examples: QVT Relations, MTF, Kent Model Transformation Language, Tefkat, AMW, mappings in XMF-Mosaic, etc.

Model-to-model approaches (4)

- **Graph-transformation-based approaches**
  - Based on the theoretical work on graph transformations
  - Operates on typed, attributed, labelled graphs
  - Graph transformation rules have an LHS and an RHS graph pattern.
    - The LHS pattern is matched in the model being transformed and replaced by the RHS pattern in place
    - Additional logic, for example, in string and numeric domains, is needed to compute target attribute values such as element names
  - Examples: AGG, AToM3, VIATRA, GReAT, UMLX, BOTL, MOLA, Fujaba, etc.

Model-to-model approaches (5)

- **Hybrid approaches**
  - Hybrid approaches combine different techniques from the previous categories
    - as separate components
    - or/and , in a more fine-grained fashion, at the level of individual rules
  - In a hybrid rule, the source or target patterns are complemented with a block of imperative logic which is run after the application of the target pattern
  - Rules are unidirectional and support rule inheritance.
  - Examples:
    - Separate components: QVT (Relations, Operational mappings, and Core)
    - Fine-grained combination: ATL and YATL

Model-to-model approaches (6)

- **Other approaches**
  - Extensible Stylesheet Language Transformation (XSLT)
    - Models can be serialized as XML using the XMI
    - Model transformations can be implemented with Extensible Stylesheet Language Transformation (XSLT), which is a standard technology for transforming XML
    - The use of XMI and XSLT has scalability limitations
    - Manual implementation of model transformations in XSLT quickly leads to non-maintainable implementations
  - Application of meta-programming to model transformation
    - Domain-specific language for model transformations embedded in a meta-programming language.

Model-to-text approaches

- **Visitor-based** approaches
  - Use visitor mechanism to traverse the internal representation of a model and write text to a text stream
  - Example: Jamda

- **Template-based** approaches
  - The majority of currently available MDA tools support template-based model-to-text generation
    - structure of a template resembles more closely the code to be generated
    - Templates lend themselves to iterative development (they can be derived from examples)
  - A template consists of the target text containing slices of meta-code to access information from the source
  - Examples: oAW, JET, Codagen Architect, AndroMDA, ArcStyler, MetaEdit, OptimalJ, etc.

QVT Operational

MOF QVT: OMG's model-to-model transformation standard

- **QVT** stands for **Q**uery/**V**iews/**T**ransformations
  - OMG standard language for expressing *queries*, *views*, and *transformations* on MOF models

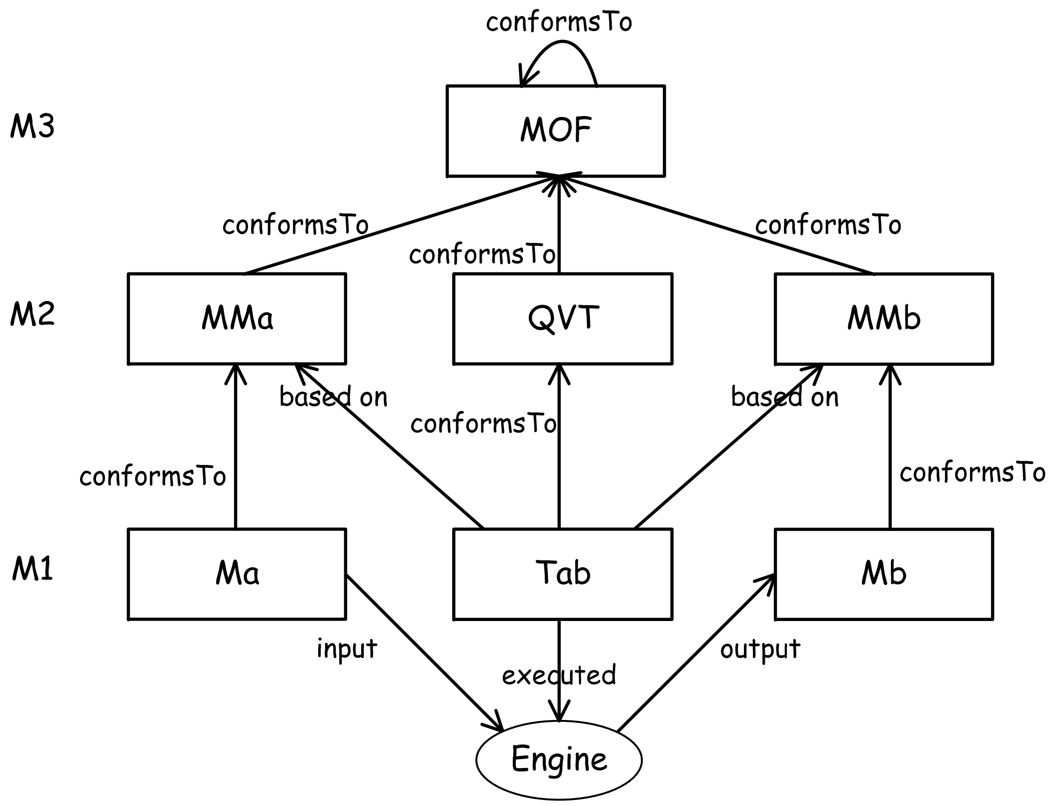
- OMG QVT Request for Proposals (QVT RFP, ad/02-04-10) issued in 2002
  - Seven initial submissions that converged to a common proposal
  - Current status (June, 2011): version 1.1, formal/11-01-01

http://www.omg.org/spec/QVT/1.0/

http://www.omg.org/spec/QVT/1.1/

MOF QVT context

- Abstract syntax of the language is defined as MOF 2.0 metamodel
  - Transformations (*Tab*) are defined on the base of MOF 2.0 metamodels (*MMa*, *MMb*)
  - Transformations are executed on instances of MOF 2.0 metamodels (*Ma*)



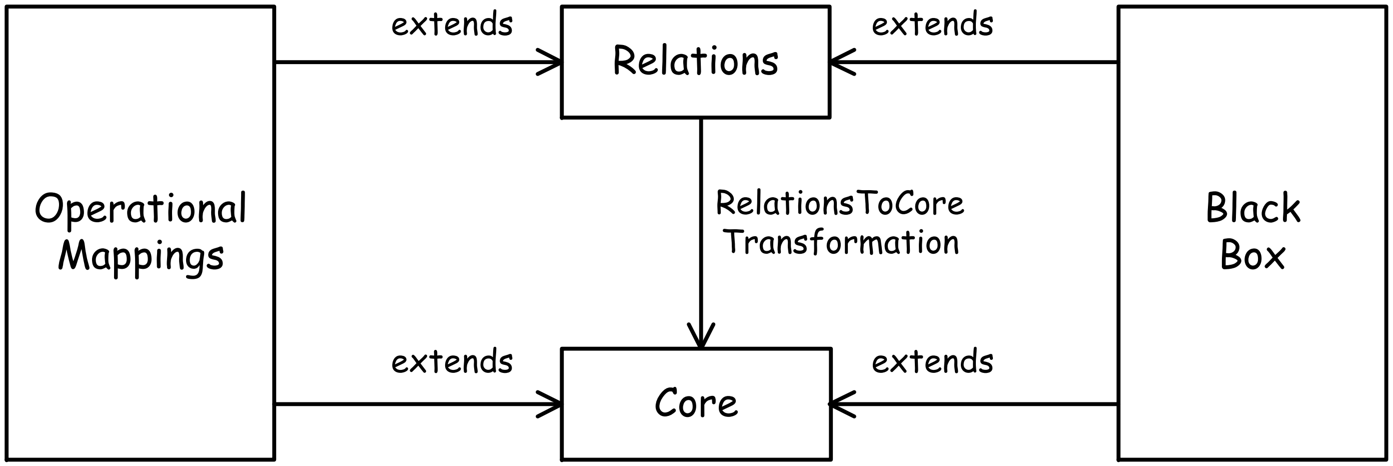
Requirements for MOF QVT language

- Some requirements formulated in the QVT RFP

Mandatory requirements	
Query language	Proposals shall define a language for querying models
Transformation language	Proposals shall define a language for transformation definitions
Abstract syntax	The abstract syntax of the QVT languages shall be described as MOF 2.0 metamodel
Paradigm	The transformation definition language shall be declarative
Input and output	All the mechanisms defined by proposals shall operate on models instances of MOF 2.0 metamodels
Optional requirements	
Directionality	Proposals may support transformation definitions that can be executed in two directions
Traceability	Proposals may support traceability between source and target model elements
Reusability	Proposals may support mechanisms for reuse of transformation definitions
Model update	Proposals may support execution of transformations that update an existing model

MOF QVT architecture

- Layered architecture with three transformation languages:
  - **Relations** (declarative)
  - Core (declarative, simpler than Relations)
  - **Operational Mappings** (imperative)
- Black Box is a mechanism for calling external programs during transformation execution
- QVT is a set of three languages that collectively provide a hybrid "language".



Overview of Operational Mappings (OM)

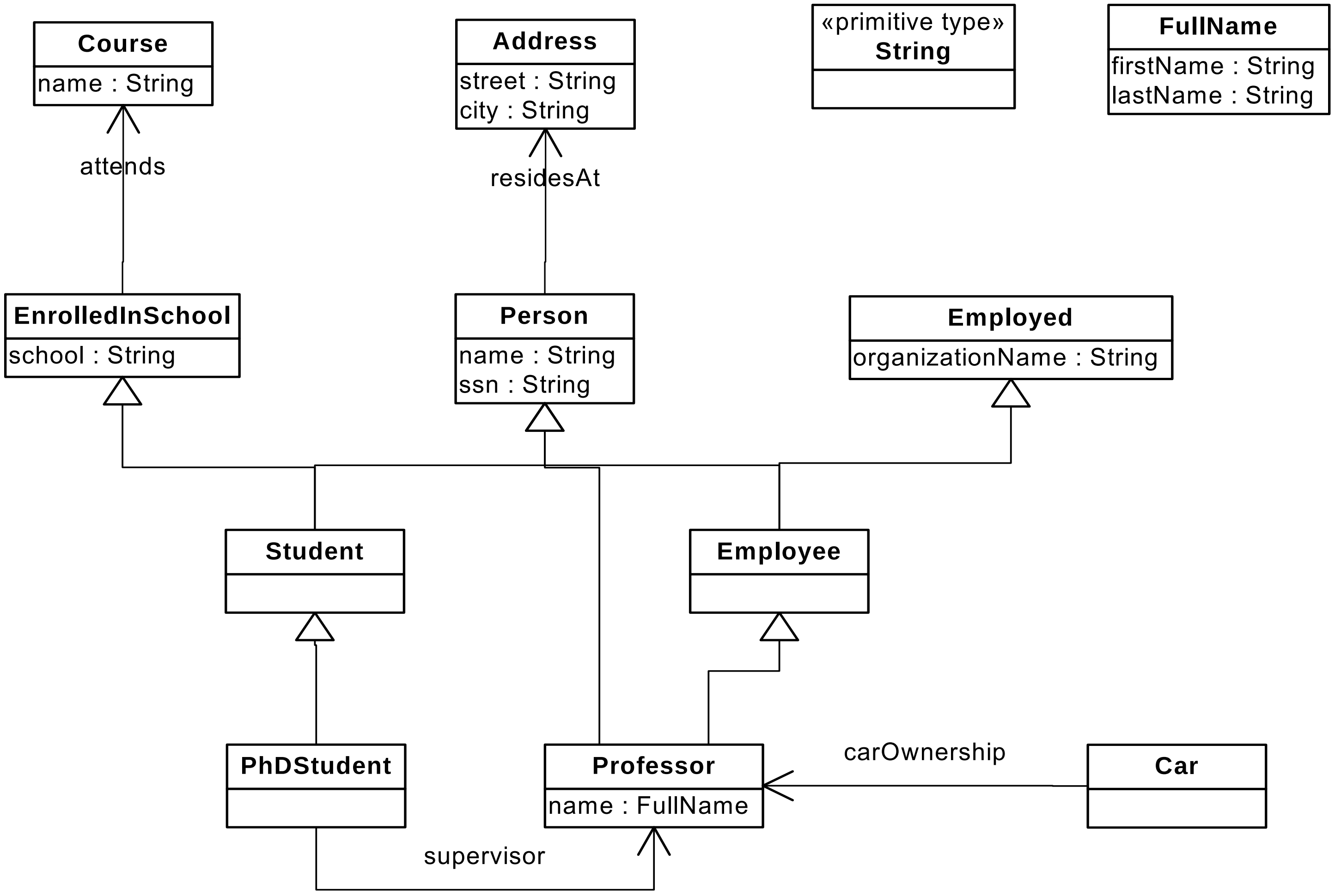
- Imperative transformation language that extends relations
- OM execution overview:
  - **Init**: code to be executed before the instantiation of the declared outputs.
  - **Instantiation** (internal): creates all output parameters that have a null value at the end of the initialization section
  - **Population**: code to populate the result parameters and the
  - **End**: code to be executed before exiting the operation. Automatic handling of traceability links
- Transformations are unidirectional
- Supported execution scenarios:
  - Model transformations
  - In-place update
- OM uses explicit internal scheduling, where the sequence of applying the transformation rules is specified within the transformation rules
- Updates have to be implemented in the model transformations

Flattening class hierarchies example

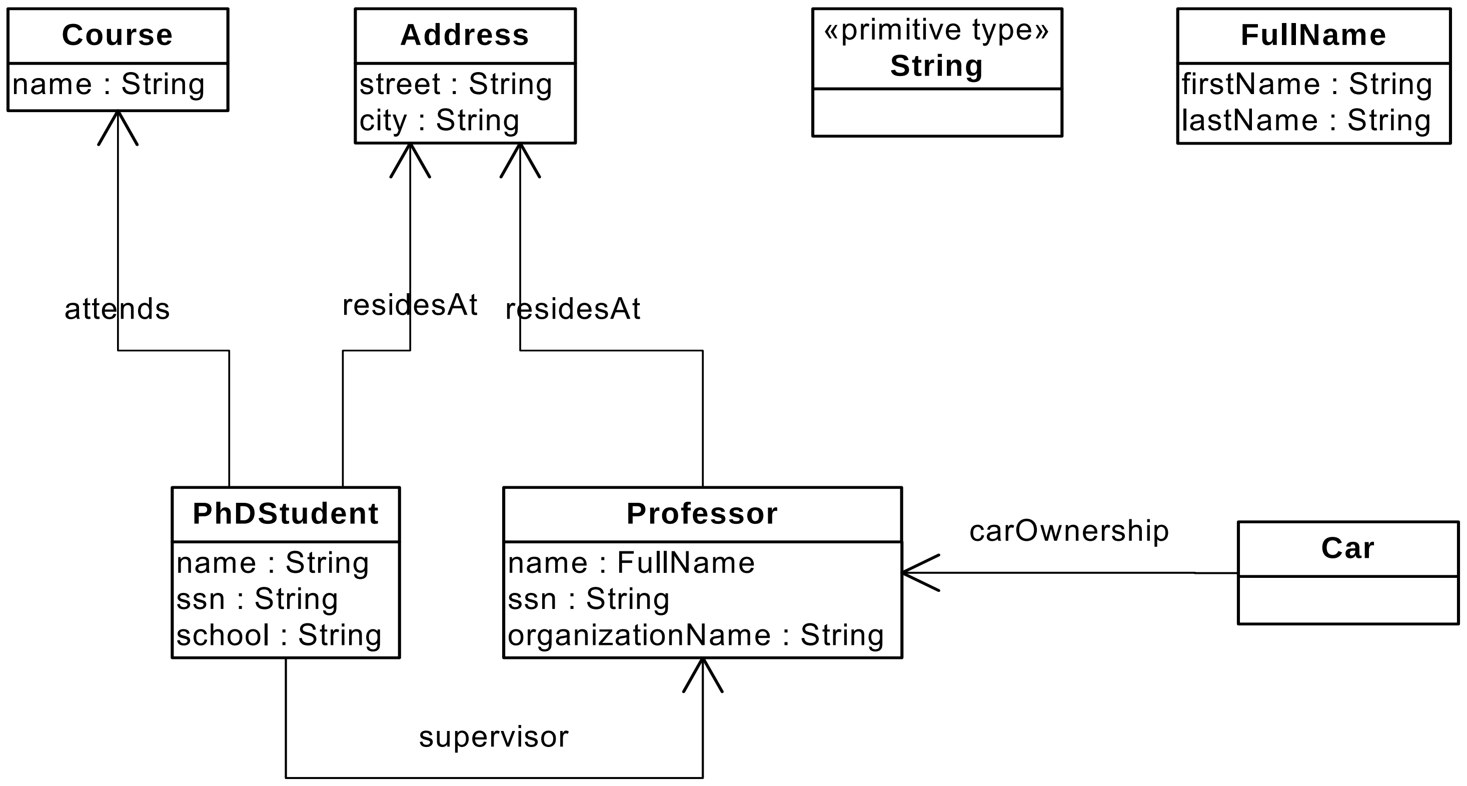
- Flattening UML class hierarchies: given a source UML model transform it to another UML model in which only the leaf classes (classes not extended by other classes) in inheritance hierarchies are kept.

- Rules:
  - Transform only the leaf classes in the source model
  - Include the inherited attributes and associations
  - Attributes with the same name override the inherited attributes
  - Copy the primitive types

Sample input model



Sample output model



OM language: Transformation program structure

```
transformation
flatten
(in hierarchical : UML,
 out flat : UML);
```

Signature: declares the transformation name and the source and target metamodels.
in and **out** keywords indicate source and target model variables.

```
main() {
  …
}
…
```

Entry point: execution of the transformation starts here by executing the operations in the body of `main`

helper declarations
…

mapping operations declarations

Transformation elements:
Transformation consists of mapping operations and helpers forming the transformation logic.

Mapping operations

- A mapping operation maps one or more source elements into one or more target elements
- Always unidirectional
- Selects source elements on the base of a type and a Boolean condition (guard)
- Executes operations in its body to create target elements
- May invoke other mapping operations and may be invoked
- Mapping operations may be related by inheritance, merging, and disjunction

General structure of mapping operations

```
mapping Type::operationName(((in|out|inout) pName : pType)*)
   : (rName : rType)+
   when {guardExpression}            pre-condition
   where {guardExpression} {         post-condition
   init {
      …        init section contains code executed before the instantiation of the declared result
   }           elements
```

There exists an implicit instantiation section that creates all the output parameters not created in the init section. The trace links are created in the instantiation section.

```
   population {
      …
   }
```

population section contains code that sets the values or the result and the parameters declared as `out` or `inout`. The `population` keyword may be skipped. The population section is the default section in the operation body.

```
   end {
   …
   }
}
```

end section contains code executed before exiting the operation

Mapping operations: Example

- Rule for transforming leaf classes
  - selects only classes without subclasses, collects all the inherited properties and associations, creates new class in the target model

target type: instance created on call

object on which mapping is called

```
mapping Class::copyLeafClass() : Class
  when {
    not hierarchical.allInstances(Generalization)->exists(g | g.general = self)
  } {
  name := self.name;
  ownedAttribute += self.ownedAttribute.
                      map copyOwnedProperty();
  ownedAttribute += (self.allFeatures()[Property] –
                       self.ownedAttribute).copyProperty(self);
  self.allFeatures()[Property]->select(p |
    not p.association.oclIsUndefined()).association.copyAssociation(self);
}
```

call of another mapping

guard: mapping operation only executed for elements for which the guard expression evaluates to true

call of a helper

- Mappings only executed once
- Call of mappings with OCL-syntax (*collection*`->map` vs. *object*`.map`)

Helpers: Example

meta-model extension

```
intermediate property Property::mappedTo : Set(Tuple(c : Class, p : Property));

helper Property::copyProperty(in c : Class) : Property {
  log('[Property] name = ' + self.name);
  var copy := object Property {
    name := self.name;
    type := self.type.map transformType();
  };
  self.mappedTo += Tuple{ c = c, p = copy };
  return copy;
}
```

object creation and population

Resolving object references

- The transformation engine maintains links among source and target model elements. These links are used for resolving object references from source to target model elements and back.
    - `resolveIn` is an operation that looks for model elements of a given type (`Class`) in the target model derived from a source element by applying a given rule (`copyLeafClass`).

```
helper Association::copyAssociation(in c : Class) : Association {
  var theOwnedEnd : Property := self.ownedEnd->any(true); …
  return object Association {
    name := self.name;
    package := self.package.resolveoneIn(Package::transformPackage, Package);
    ownedEnd += new Property(theOwnedEnd.name,
                             c.resolveoneIn(Class::copyLeafClass, Class)); …
  }
}
```

call to constructor

- Variants: `resolve(`*i* `|` *exp*`)`, `resolveone(`*i* `|` *exp*`)`
- `late resolve` for resolving **after** the transformation (in order of calls)

Mapping operations: Disjunction, inheritance, merging

```
mapping DataType::copyDataType() : DataType {
  name := self.name;
  ownedAttribute += self.ownedAttribute.map copyOwnedProperty();
}

mapping PrimitiveType::copyPrimitiveType() : PrimitiveType {
  init {
    result := self.deepclone().oclAsType(PrimitiveType);
  }
}

mapping Type::transformType() : Type
  disjuncts DataType::copyDataType,
            Class::copyLeafClass,
            PrimitiveType::copyPrimitiveType;
```

- Inherited rules executed after `init`
- Merged rules executed after `end`

Imperative OCL constructs

- More sophisticated control flow
  - `compute (`*v* `:` *T* `:=` *exp*`)` *body*
    - like `let … in`
  - `while (`*cond*`)` *body*
  - *coll*`->forEach (`*i* `|` *exp*`)` *body*
  - `break`, `continue`
  - `switch`-statement, exceptions

MOFM2T

MOFM2T: OMG’s model-to-text transformation standard

- **M2T** stands for **M**odel-**to**-**T**ext
  - OMG standard language for *transforming* MOF models into text

- Current status (June, 2011): version 1.0, formal/08-01-16

http://www.omg.org/spec/MOFM2T/1.0/

M2T Transformations: Example (1)

```
[comment encoding = UTF-8 /]
[** Java Beans-style code from UML static structure */]
[module generate('http://www.eclipse.org/uml2/3.0.0/UML')]

[**
 * Generate a Java file from a UML class
 * @param aClass
 */]
[template public generateClass(aClass : Class)]
[comment @main/]
[file (aClass.name.concat('.java'), false, 'UTF-8')]
public class [aClass.name/] {
[for (p : Property | aClass.attribute) separator('\n')]
[generateClassAttribute(p)/]
[/for]
}
[/file]
[/template]
```

metamodel type

top-level rule (several possible)

output in file, not appending

call of another template

verbatim text

M2T Transformations: Example (2)

```
[template public generateClassAttribute(aProperty : Property)]
private [getTypeName(aProperty.type)/] [aProperty.name/];

public [getTypeName(aProperty.type)/] [aProperty.name.toUpperFirst()/]() {
    // [protected(aProperty.name)]
    // TODO implement
    // [/protected]
    return this.[aProperty.name/];
  }
[/template]

[template public generateDataType(aDataType : DataType)]
[comment @main/]
[file (aDataType.name.concat('.java'), false, 'UTF-8')]
public class [aDataType.name/] [for (p : Property | aDataType.attribute)
                                  before(' {\n') separator('\n') after('\n}')]
  public [getTypeName(aProperty.type)/] [aProperty.name/]; [/for]
[/file]
[/template]

[query public getTypeName(aType : Type) : String = aType.name /]
```

protected code message

output in file, not appending

top-level rule (several possible)

before first, in-between each, and after last item

MOFM2T features

- Tracing
  - `[trace(`*id*`)]` … `[/trace]`
- Change of escape direction
  - `@text-explicit` (default, shown above)
  - `@code-explicit`
- Macros
- Module structure
  - public module elements visible outside a module
  - guards on templates for selecting a template when overriding (overridden template callable with `[super/]`)

- No type checking of output