Model-Driven Software Engineering

Code Generation

Dr. Jochen Küster (jku@zurich.ibm.com)

Contents

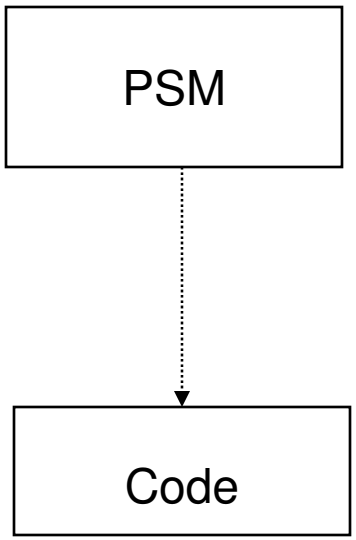
- Code Generation in Model-Driven Software Engineering
- Concept and Styles of Code Generators
- A Closer Look at Xpand
- Summary and References

Code Generation in Model-Driven Software Engineering

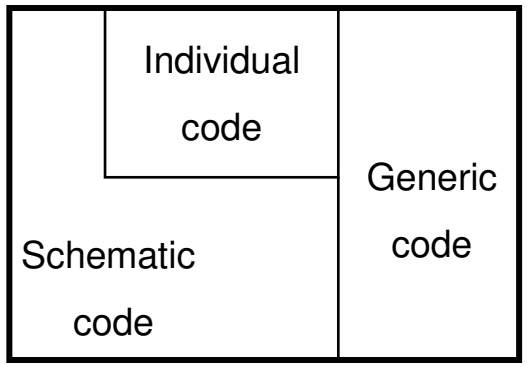
Motivation for Code Generators

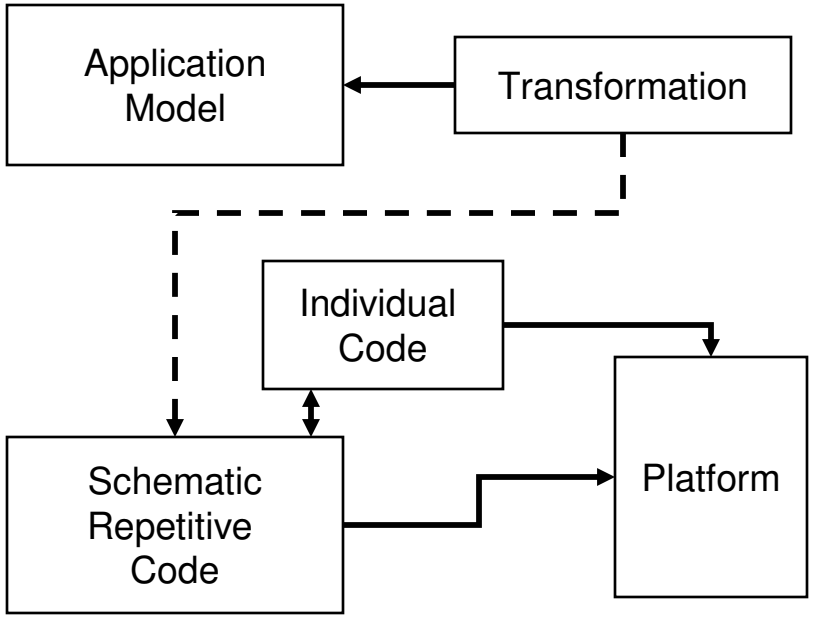
- Model-Driven Software Development makes models key artifacts in the software engineering process
- When working with models, automation of recurring tasks can often be achieved by **code generation**

- The MDA approach favors **code generation** from models
- Architecture-centric Model Driven Software Development **generates** code from models



Recap: Purpose of Code Generation in AC-MDSE





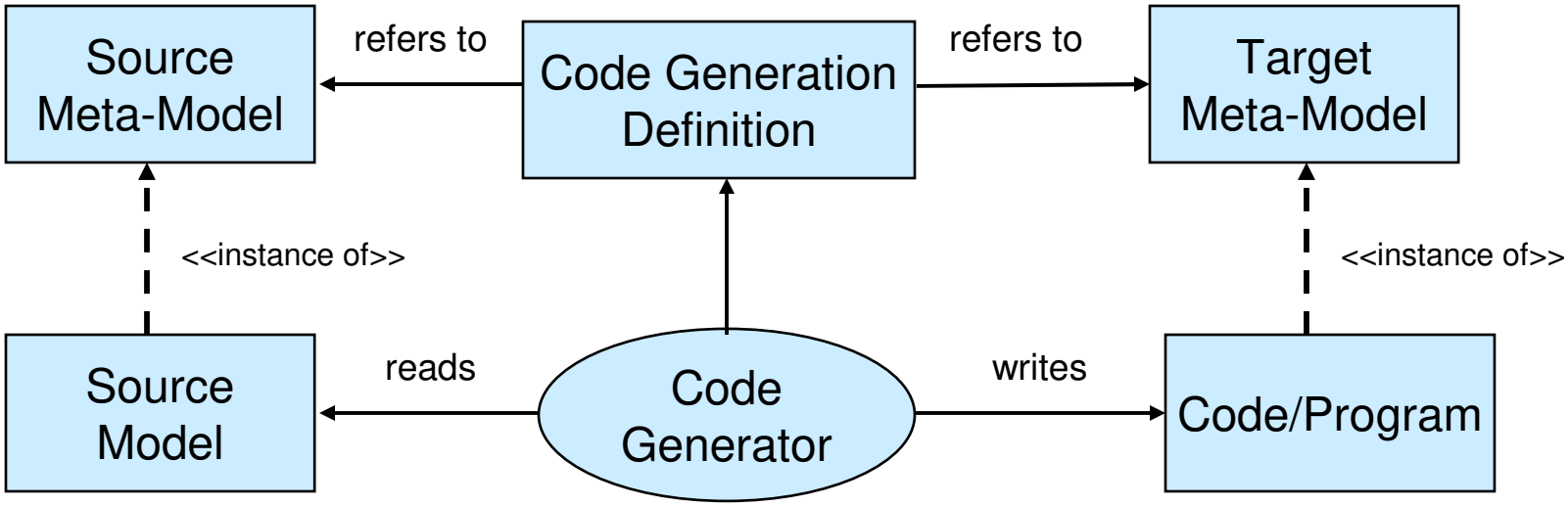
- Generate generic code for the platform instead of writing it
- Generate schematic code using transformations based on an application model
- Write individual code that is application specific

Purpose of Code Generation

- **Separation of application logic from implementation platform details** in order to ease transition to other platforms
  - Example: SOA solutions where a process model is implemented
- Improved **productivity** by using code generation
  - Example: EMF generation of code from domain models
- Improved **quality of the application** due to standardized implementation (patterns and best practices)
  - Example: EMF code generation
- Increased **performance of the application** using generators that produce efficient code
  - Example: Code generation from statecharts in embedded applications

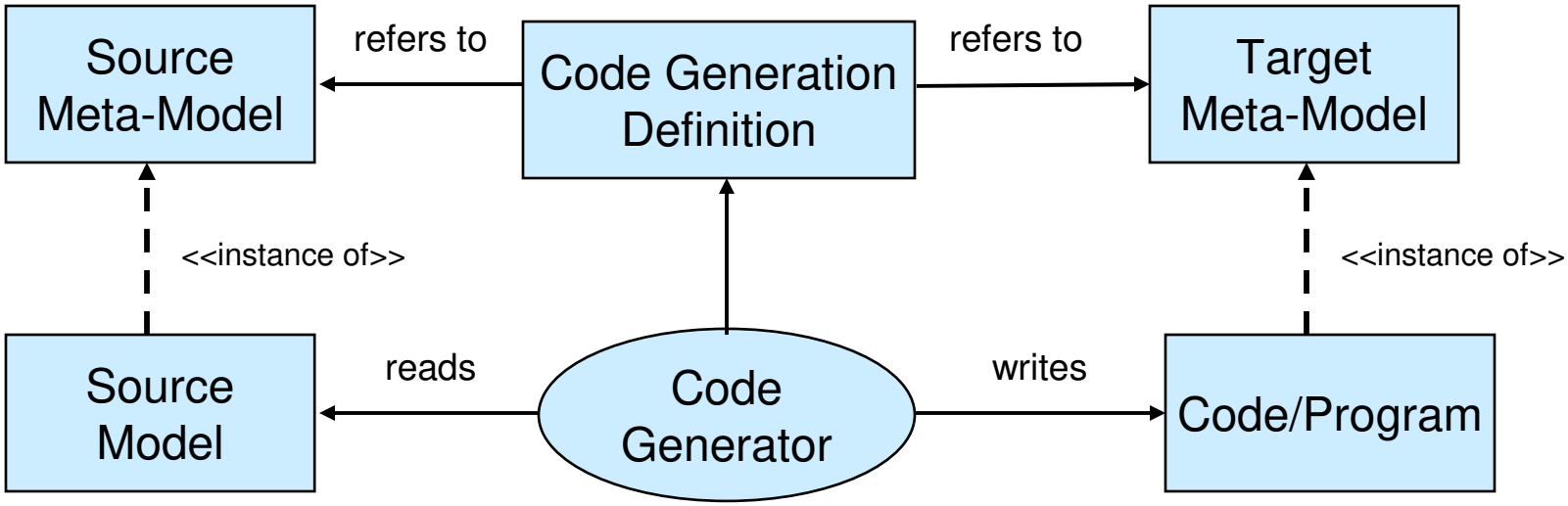
Concept and Styles of Code Generators

Code Generation as a form of Model Transformation



- **Source models** include models defined in various modeling languages and even program code
- The **target** is code which has to conform to the syntax of the target language
  - Java, C#, C++

Terminology of Code Generation



- Target is also referred to as **Program**
- Code Generator and Code Generation Definition also called **Meta Program**
- **Meta Program** and **Program** can be mixed or separated from each other

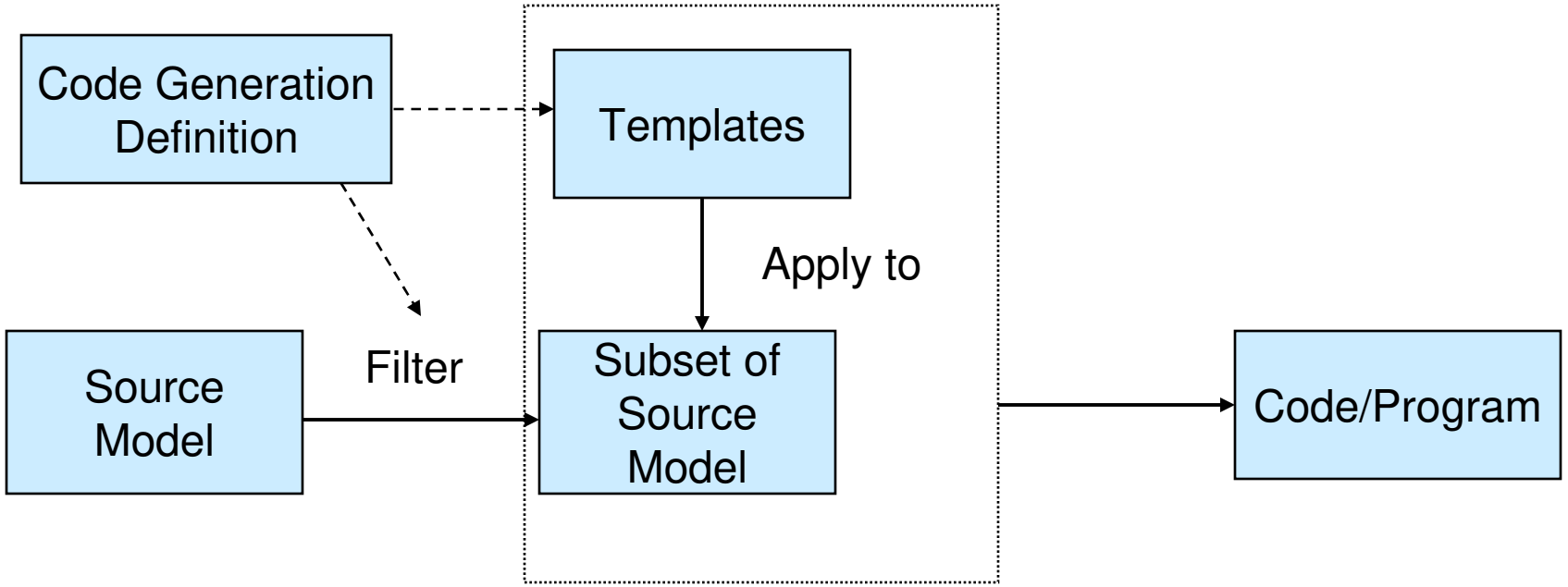
Generation Patterns

- There are different ways how to design and implement a code generator

- We discuss several well-known generation patterns:
  - Templates and filtering
  - Templates and metamodel
  - API-based generators
  - Inline code generation

- Different generation patterns have different advantages and disadvantages

Templates and Filtering



- Source model is in textual XMI/XML form
- Source model is filtered to obtain a subset of the source model
- Templates are instantiated using values of the filtered source model
- Result yields the code/program

XSLT Example (1)

- Code generation of a Java Bean class
- Input is an XML description of the required information

```
<?xml version="1.0" encoding="UTF-8"?>
<JavaBean name="Customer">
<Package>com.developer</Package> <Imports> <Import>java.util.Collection</Import> </Imports>
<Superclass name="Object"/>
<Properties>
<Property name="name" type="String"/> <Property name="addrln1" type="String"/>
<Property name="addrln2" type="String"/>
<Property name="city" type="String"/> <Property name="state" type="String"/>
<Property name="zip" type="String"/>
<Property name="contacts" type="Collection"/>
 </Properties>
</JavaBean>
```

Source: Jeff Ryan: Code Generation with XSL, developer.com

XSLT Example (2)

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/
    1999/XSL/Transform">
<xsl:output method="text"/>
<xsl:template match="/">
package <xsl:value-of select="//Package"/>;
<xsl:apply-templates select="//Import"/>
public class <xsl:value-of select="JavaBean/@name"/> extends <xsl:value-of
    select="//Superclass/@name"/> {
<xsl:apply-templates select="//Property" mode="instanceVariable"/>
public <xsl:value-of select="JavaBean/@name"/>()
{ }
<xsl:apply-templates select="//Property" mode="accessor"/>
<xsl:apply-templates select="//Property" mode="mutator"/> }
</xsl:template>

<xsl:template match="Import">
  import
  <xsl:value-of select="." />
  ;
  </xsl:template>

<xsl:template match="Property" mode="accessor">
  ...
```

generate import of Java class

generate properties

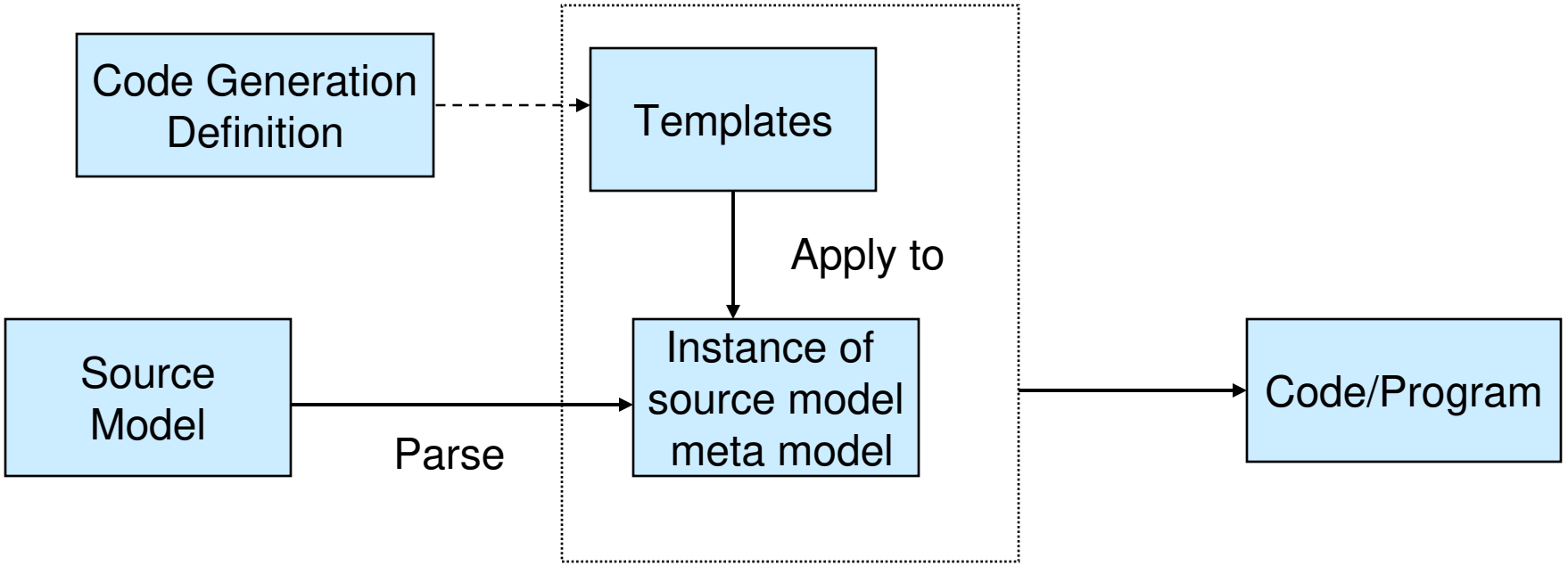
generate getters/setters

Source: Jeff Ryan: Code Generation with XSL, developer.com

Template and Filtering Drawbacks

- Templates become very complex for larger examples
- Approach tightly couples the generation definition (templates) to the concrete syntax of the model
- This yields low maintainability if source modeling language evolves

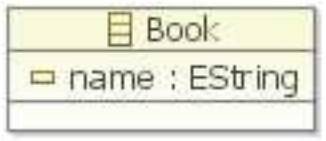
Templates and Meta Model



- **Source model is parsed** in order to create instance of source model meta model
- **Templates** are defined in terms of source model meta model terms
- **Templates are instantiated** using values of the instance of the source model meta model
- Result yields the code/program

Templates and Meta Model Example: Xpand

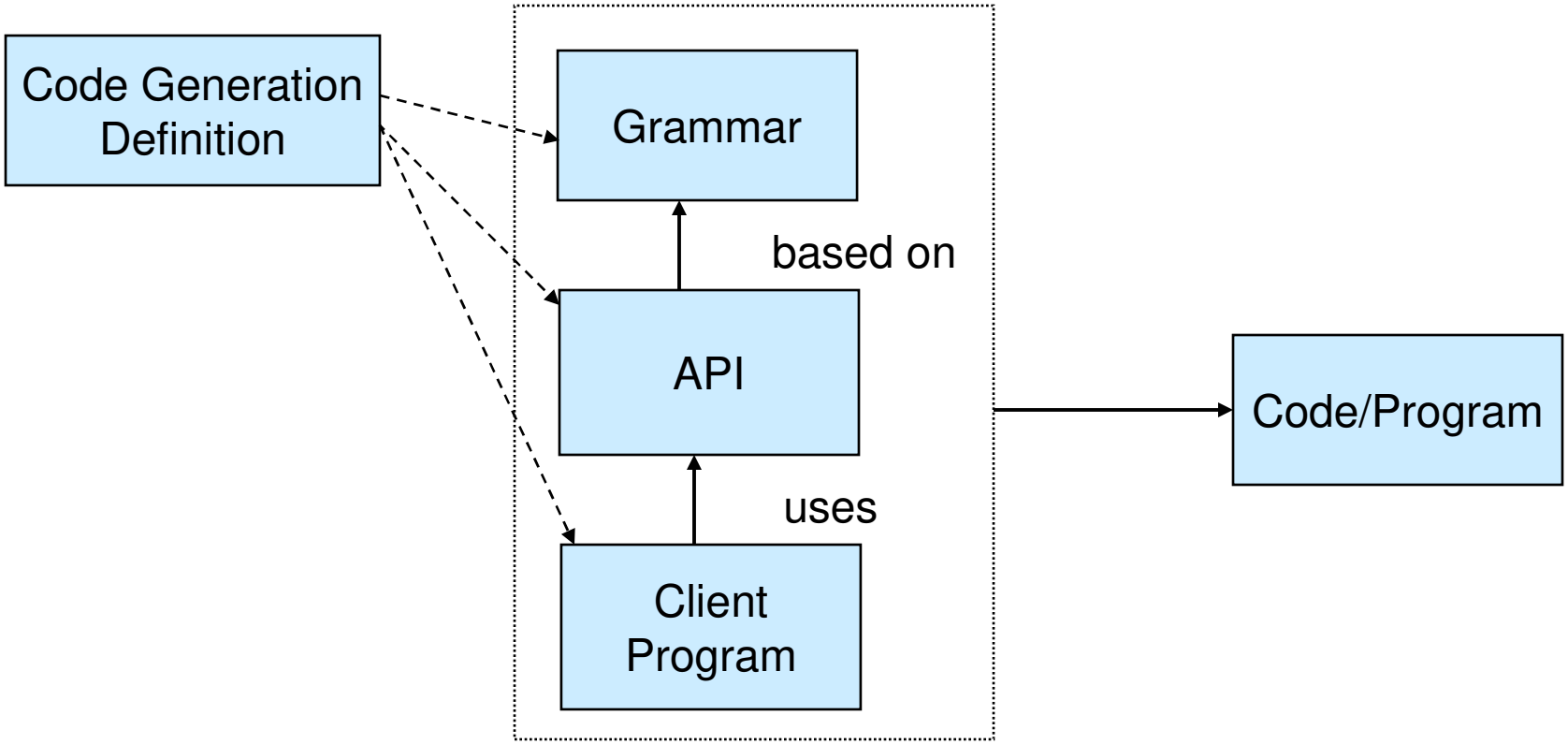
- Xpand is a model-to-text transformation language developed in the context of openArchitectureWare
- Recently added to be part of the Eclipse M2T project
- Suited for transformation of models to text for code generation
- Direct reference of metamodel elements in transformation rules




```
«IMPORT metamodel»

«DEFINE main FOR Book»
«FILE name+".java"»
public class «this.name» {
}
«ENDFILE»
«ENDDEFINE»
```

API –based Generators



- Client program uses API which is based on a Grammar

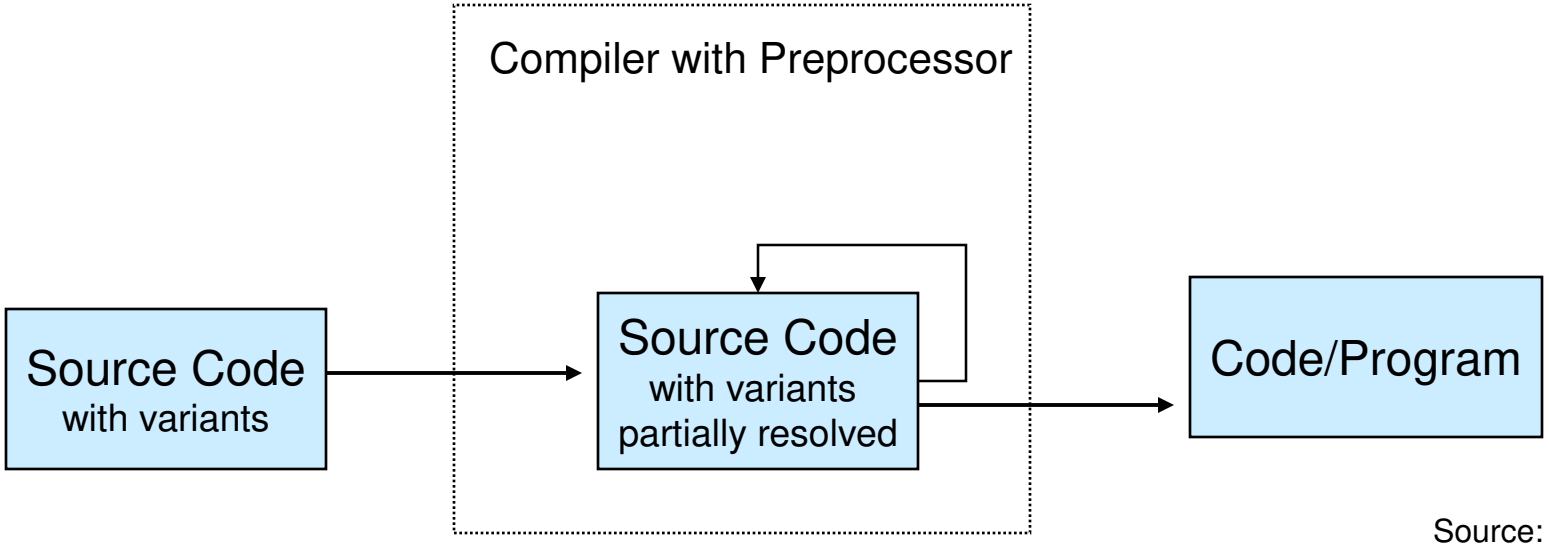
API-based Generators Example

- C# API for generating code

```
CodeNamespace n = ...
CodeTypeDeclaration c = new
CodeTypeDeclaration („Vehicle");
c.IsClass = true;
c.BaseTypes.Add (typeof (System.Object) );
c.TypeAttributes = TypeAttributes.Public;
n.Types.Add(c);
```

```
public class Vehicle : object {
}
```

Inline Code Generation



Source: [Voelter et al.]

- **Code generation is done implicitly** by means of a precompiler
- The precompiler modifies the program which is then compiled or interpreted
- Examples are common in the programming language domain (C++ precompiler)

Comparison of Approaches - Criteria

	Learning complexity	Suitability for complex uses	Suitability to model-to-code transformations
Template + Filtering	simple	not very good	good
Template + Metamodel	high	very good	very good
API-based	depends on API	depends on API	not very good
Inline code generation	simple	not very good	not very good

[Source: adapted from Voelter]

A Closer Look at XPand

Introduction to XPand

- XPand is a template language for model to text transformations
- Originally developed as part of openArchitectureWare
- Now part of the Eclipse M2T project
- XPand project contains Template files

Overview of XPand Templates

```
«IMPORT meta::model»
```
import meta models

```
«EXTENSION my::ExtensionFile»
```
specify meta model extensions

```
«DEFINE javaClass FOR Entity»
```
defines template for transformation

```
«FILE fileName()»
```
defines file for output

```
package «javaPackage()»;

public class «name» {
// implementation
}
«ENDFILE»
«ENDDEFINE»
```
generated text

Import and Extension of Meta Models

- Import meta models into templates

  «IMPORT meta::model»

- Describe extensions for meta models such as queries or target platform specific names
- Can be done using meta model extensions using the Xtend language

  «EXTENSION my::ExtensionFile»

Defining Templates for Transformation

«DEFINE templateName(formalParameterList) FOR MetaClass»

a sequence of statements

«ENDDEFINE»

- The template is executed for each instance of the MetaClass
- Support of polymorphism:
  - If there are two templates with the same name that are defined for two meta classes which inherit from the same super class, the corresponding subclass template is used in case the template is called for the super class. Vice versa the super class's template would be used in case a subclass template is not available.

Defining Subroutine Templates using EXPAND

«EXPAND definitionName [(parameterList)]

[FOR expression | FOREACH expression [SEPARATOR expression]]»

- EXPAND defines another template inside a given template
- «EXPAND myDef FOR this» evaluates myDef for the model element in the context of which EXPAND is situated
- «EXPAND myDef FOR entity» evaluates myDef for entity.
- «EXPAND myDef FOREACH entity.allAttributes»
- If FOREACH is specified, the target expression must evaluate to a collection type. In this case, the specified definition is executed for each element of that collection

Defining Files for Output

«FILE expression [outletName]»

a sequence of statements

«ENDFILE»

```
«FILE name+".java"»
public class «this.name» {
«EXPAND generateChapters FOREACH this.chapters»
}
«ENDFILE»
```

- FILE allows to redirect output into a file
- expression specifies the file name of the file
- The file is saved relative to the directory where the generator executes
- outletName allows to specify an identifier for reuse

Expressions in XPand

- Expressions provide access to instantiated meta model values
- Expressions are based on an expression language that has similarities to Java and OCL
- The expression language is based on a type system supporting built-in types such as Boolean, Integer, String and Real and Collection types

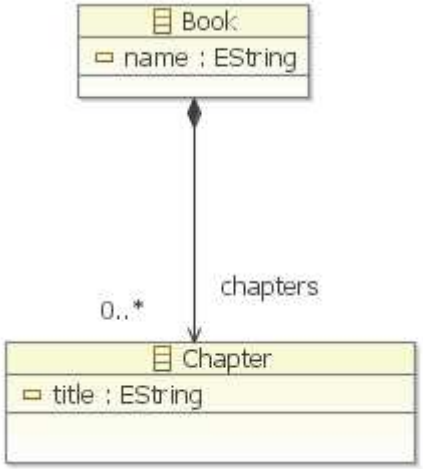
```
public class «this.name» {  .. }
```

Example

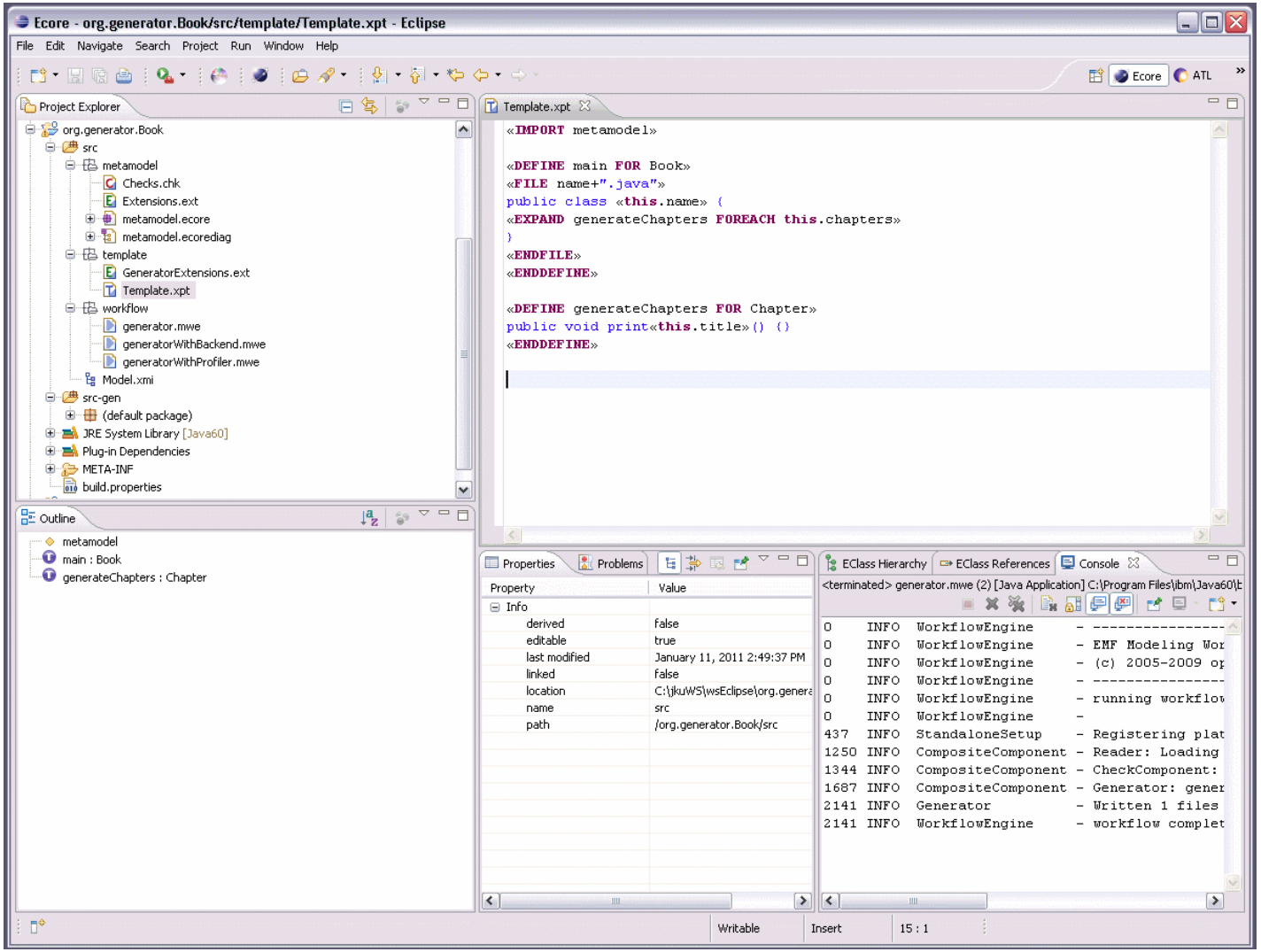
```
«IMPORT metamodel»

«DEFINE main FOR Book»
«FILE name+".java"»
public class «this.name» {
«EXPAND generateChapters FOREACH
this.chapters»
}
«ENDFILE»
«ENDDEFINE»

«DEFINE generateChapters FOR Chapter»
public void print«this.title»() {}
«ENDDEFINE»
```



Overview of XPand project in Eclipse



Workflow for Generation

```
<component class="oaw.xpand2.Generator2">
<fileEncoding value="ISO-8859-1"/>
<metaModel class="oaw.type.emf.EmfMetaModel">
<metaModelPackage value="org.eclipse.emf.ecore.EcorePackage"/>
</metaModel>
<expand value="example::Java::all FOR myModel"/>
<!-- aop configuration -->
<aspects value='example::Aspects1, example::Aspects2'/>
<!-- output configuration -->
<outlet path='main/src-gen'/>
<outlet name='TO_SRC' path='main/src' overwrite='false'/>
<beautifier class="oaw.xpand2.output.JavaBeautifier"/>
<beautifier class="oaw.xpand2.output.XmlBeautifier"/>
<!-- protected regions configuration -->
<prSrcPathes value="main/src"/>
<prDefaultExcludes value="false"/>
<prExcludes value="*.xml"/>
</component>
```

Defines generator

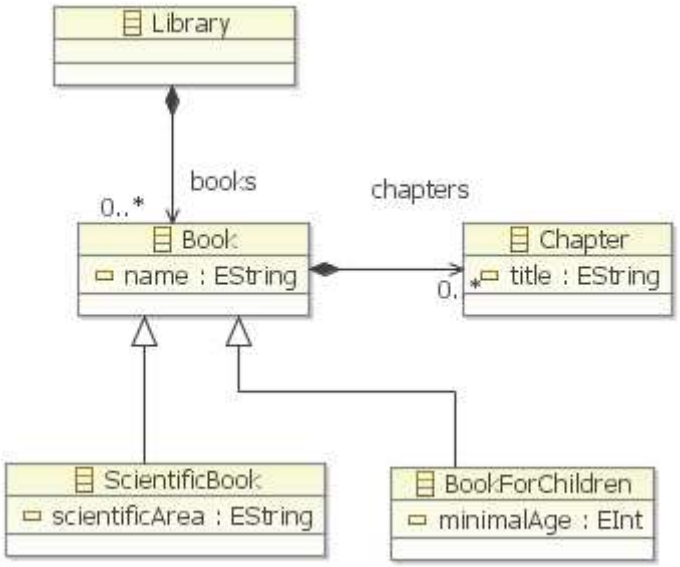
Defines meta model to be used for applying templates

Defines how to evaluate

Defines outlets for output

Specifies beautifier for output

Example with Polymorphism




```
«DEFINE generateBook FOR Book»
«FILE name+".java"»
public class «this.name» {
«EXPAND generateChapters FOREACH this.chapters»
}
«ENDFILE»
«ENDDEFINE»

«DEFINE generateBook FOR ScientificBook»
«FILE name+".java"»
public class «this.name» {
«EXPAND generateScientificArea»
«EXPAND generateChapters FOREACH this.chapters»

}
«ENDFILE»
«ENDDEFINE»
```

Summary of Lecture and References

- Code generation is an important aspect in model-driven software engineering
- Different forms of code generation from an architecture point of view
- XPand Eclipse code generation as an example for Template and Meta Model

References:

- Voelter et al. Model-Driven Software Development, Chapter on Code Generation.
- S. Efftinge and C. Kadura. OpenArchitectureWare 4.1 Xpand Language Reference.
- J. Oldevik. MOFScript User Guide, Version 0.8. Available online.