

# 5. Bridging the Technical Spaces Grammarware and EMOFware using EMFText

Florian Heidenreich, Jendrik  
Johannes, Sven Karol, Mirko  
Seifert, Christian Wende, Uwe  
Aßmann

Version 15-0.3, 10/22/12

- 1) What is a DSL?
- 2) How to build a DSL
  - 1) Defining/Using a meta model
  - 2) Syntax Definition
    - 1) Generating an initial syntax (HUTN)
  - 3) Refining the syntax
- 3) Advanced features of EMFText
  - 1) Mapping text to data types
  - 2) Reference resolving
  - 3) Syntax modules (Import and Reuse)
  - 4) Interpretation vs. Compilatio
- 4) Integrating DSLs and GPLs
- 5) Other DSL examples in the Zoo
- 6) Conclusion

# Obligatory Literature

2

Model-Driven Software Development in Technical Spaces (MOST)

- ▶ Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Model-based language engineering with EMFText. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, GTTSE, volume 7680 of Lecture Notes in Computer Science, pages 322-345. Springer, 2011.

# Recommended Literature

EMFText is used by our start-up, DevBoost  
[www.devboost.de](http://www.devboost.de)

3

Model-Driven Software Development in Technical Spaces (MOST)

- ▶ <http://www.emftext.org>
- ▶ [http://www.emftext.org/index.php/EMFText\\_Publications](http://www.emftext.org/index.php/EMFText_Publications)
- ▶ Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In Proc. of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009).
- ▶ Mirko Seifert and Christian Werner. Specification of Triple Graph Grammar Rules using Textual Concrete Syntax. 7th International Fujaba Days, 2009
- ▶ Florian Heidenreich, Jendrik Johannes, Mirko Seifert and Christian Wende. Construct to Reconstruct - Reverse Engineering Java Code with JaMoPP. In Proc. of the International Workshop on Reverse Engineering Models from Software Artifacts (R.E.M.'09).
- ▶ Florian Heidenreich, Jendrik Johannes, Mirko Seifert and Christian Wende. Closing the Gap between Modelling and Java Tool demonstration at the 2nd International Conference on Software Language Engineering (SLE'09).
- ▶ Florian Heidenreich, Jendrik Johannes, Mirko Seifert, Christian Wende and Marcel Böhme. Generating Safe Template Languages. In Proc. of the 8th International Conference on Generative Programming and Component Engineering (GPCE 2009).
- ▶ Christian Wende and Florian Heidenreich. A Model-based Product-Line for Scalable Ontology Languages. In Proc. of the 1st International Workshop on Model-Driven Product-Line Engineering (MDPLE 2009) collocated with ECMDA-FA 2009. Enschede, The Netherlands, June 2009.
- ▶ Mirko Seifert and Roland Samlaus. Static Source Code Analysis using OCL. In Proc. of OCL Workshop 2008 at MODELS 2008
- ▶ Jakob Henriksson, Florian Heidenreich, Jendrik Johannes, Steffen Zschaler and Uwe Aßmann. Extending Grammars and Metamodels for Reuse -- The Reuseware Approach. IET Software Journal 2008.

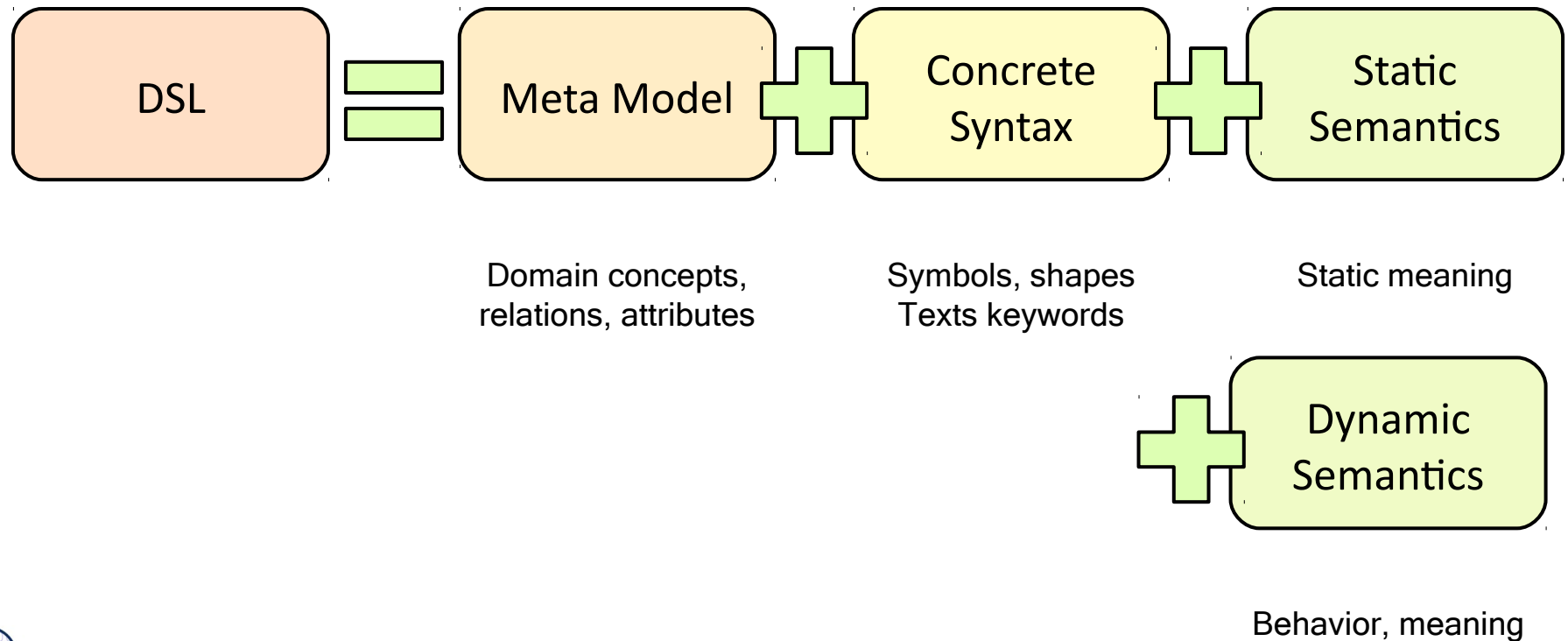
## 5.1 What is a Domain-Specific Language (DSL)?



# What's in a Domain-Specific Language (DSL)?

5

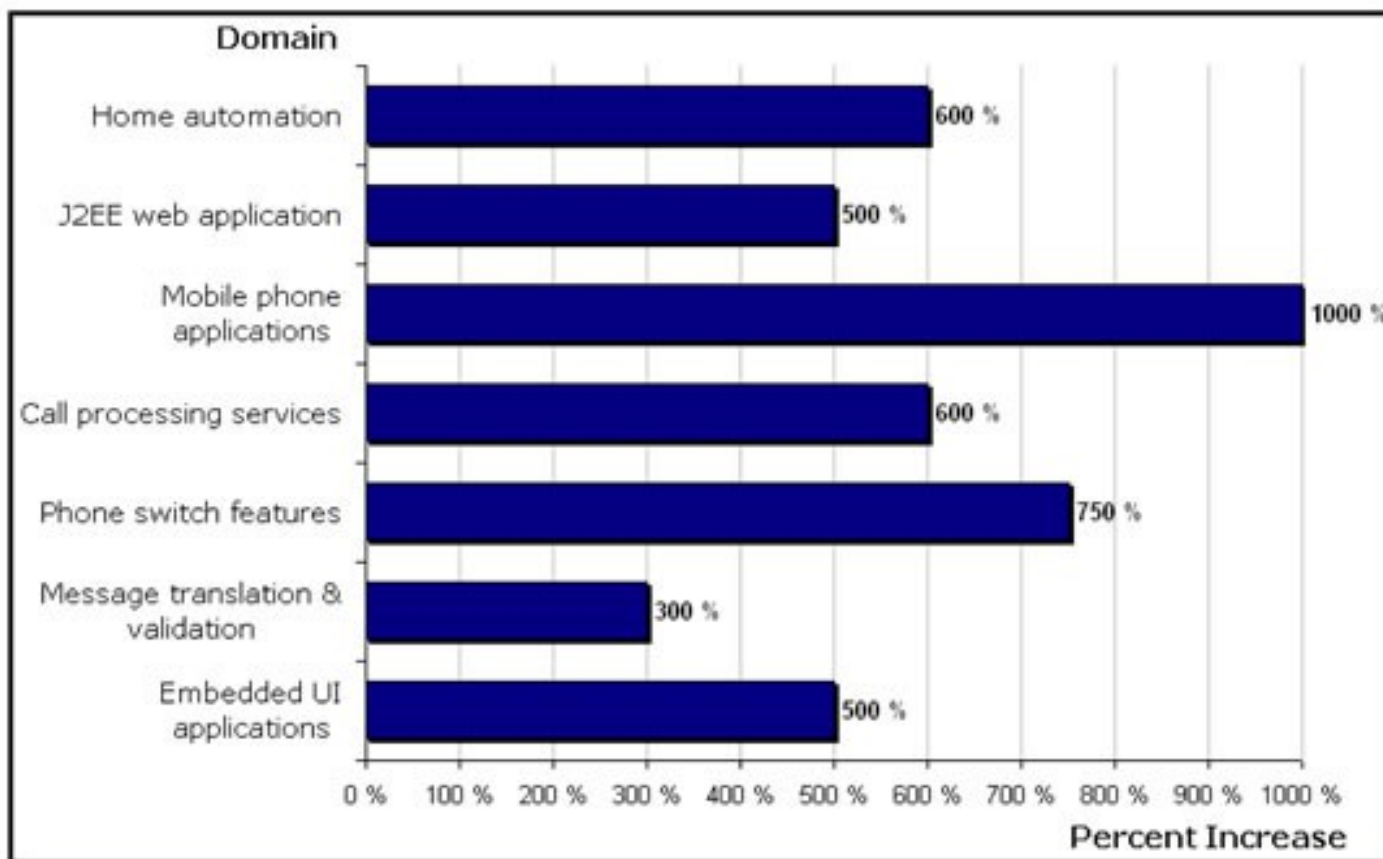
Model-Driven Software Development in Technical Spaces (MOST)



# Productivity Gains with DSL

6

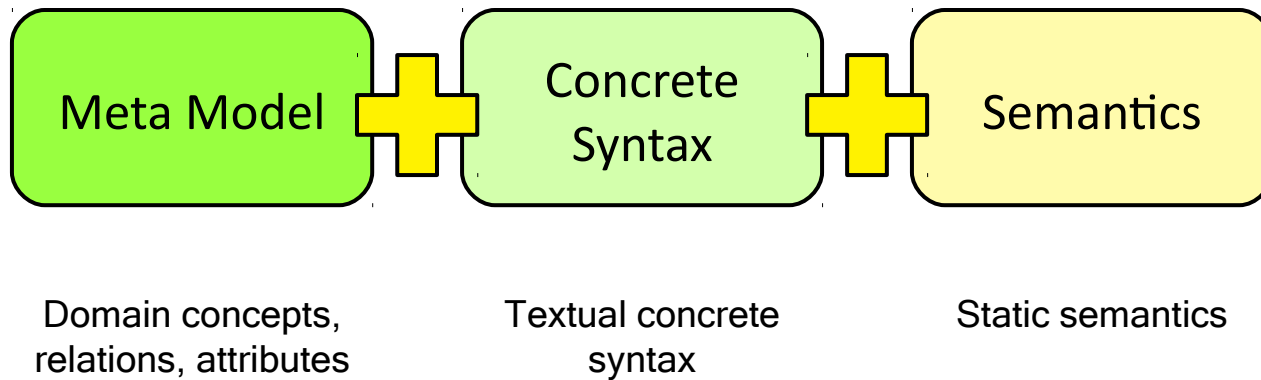
Model-Driven Software Development in Technical Spaces (MOST)



Juha-Pekka Tolvanen. Domain-Specific Modeling for Full Code Generation. January 2010. Vol. 12, Number 4. <http://journal.thedacs.com/issue/52/144>

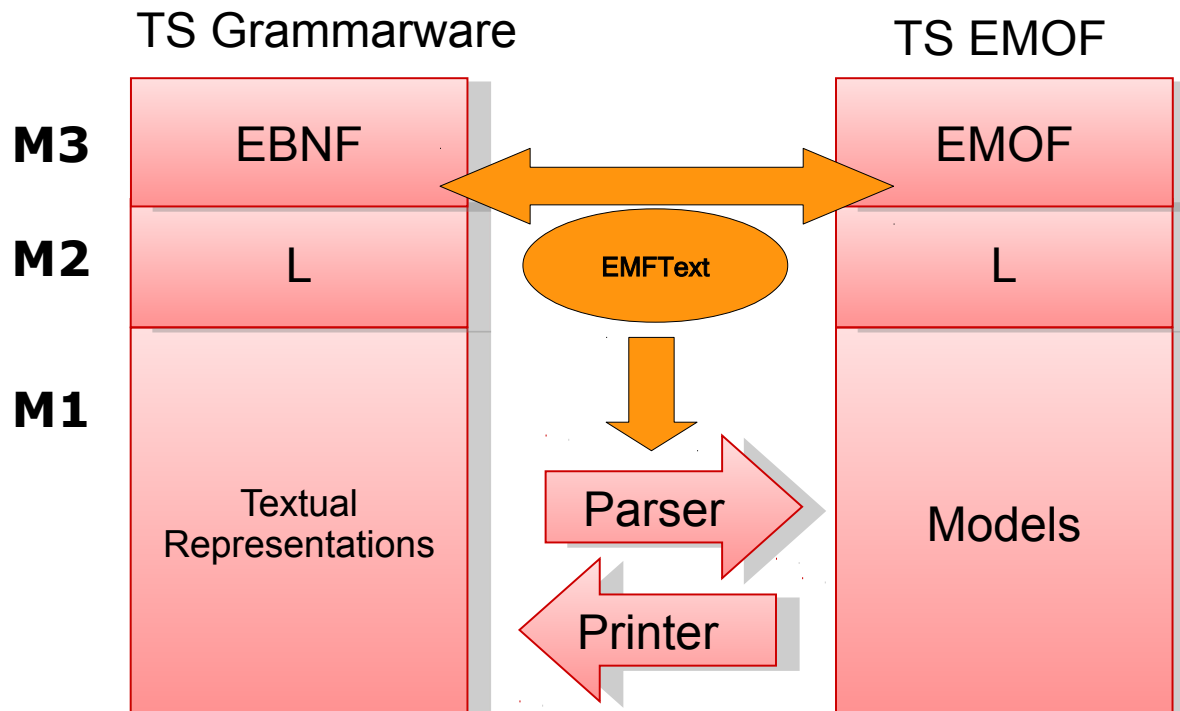
# What is a Textual Domain-Specific Language (DSL)?

- ▶ EMFText relates a concrete syntax specification (grammar in EBNF) to a EMOF/Ecore-based metamodel.
- ▶ From this language mapping, printers, parsers and editors for a DSL can be generated



# Textual DSL rely on a Transformation Bridge between the Technical Spaces EMOF and Grammarware

- ▶ EMFText relates a concrete syntax specification (grammar in EBNF) to a EMOF/Ecore-based metamodel.
- ▶ From this language mapping, printers (unparsers), parsers and editors are generated
- ▶ EMFText can be used to produce normative concrete syntax for exchange formats





# Motivation – Why DSLs?

- + Use the concepts and idioms of a domain
- + Domain experts can understand, validate and modify DSL programs
- + Concise and self-documenting
- + Higher level of abstraction
- + Can enhance productivity, reliability, maintainability and portability
- + Embody domain knowledge, enabling the conservation and reuse of this knowledge

## But:

- Costs of design, implementation and maintenance
- Costs of education for users
- Limited availability of DSLs

From: <http://homepages.cwi.nl/~arie/papers/dslbib/>

# EMFText Motivation – Why Textual syntax?

## Why use textual syntax for models?

- Readability
- Diff/Merge/VCS
- Evolution
- Tool autonomy
- Quick model instantiation

## Why create models from text?

- Tool reuse (e.g., to perform transformations (ATL) or analysis (OCL))
- Know-how reuse
- Explicit representation of text document structure
- Tracing software artifacts
- Graphs instead of strings

Be aware: exchange syntax is like a textual DSL

# EMFText Philosophy and Goals

Design principles:

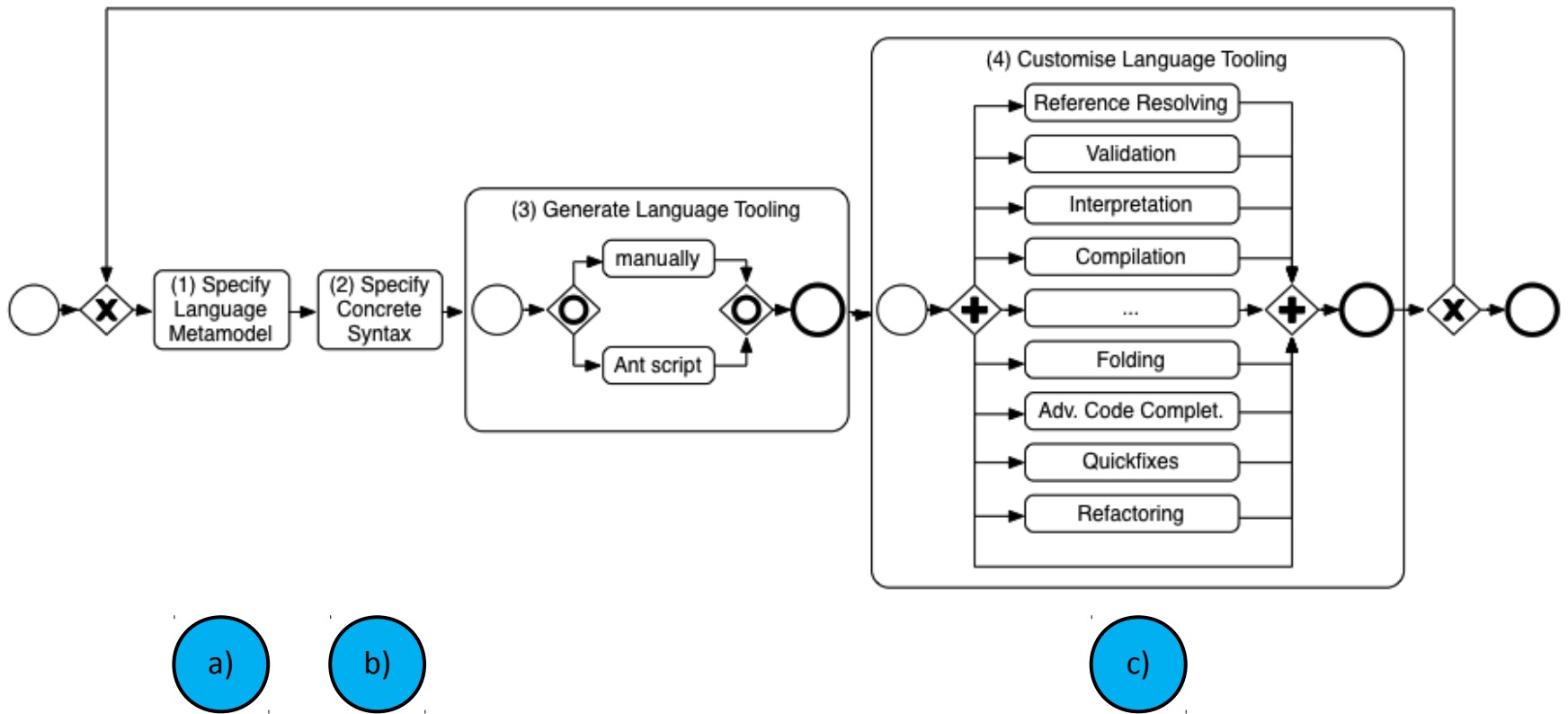
- Convention over Configuration
- Provide defaults wherever possible
- Allow customization for all parts of a syntax

Syntax definition should be

- Simple and easy for small DSLs
- Yet powerful for complex languages

- ▶ **Generation Features**
  - Generation of independent code
  - Generation of Default Syntax
  - Customizable Code Generation
  
- ▶ **Specification Features**
  - Modular Specification
  - Default Reference Resolving
  - Comprehensive Syntax Analysis
  
- ▶ **Editor Features**
  - Code Completion, Customizable Syntax and Occurrence Highlighting, Code Folding, Error Marking, Hyperlinks, Text Hovers, Outline View, ...
  
- ▶ **Other Highlights**
  - ANT Support, Post Processors, Builder, Interpreter and Debugger Stubs, Quick Fixes

# EMFText Language Development Process



a)

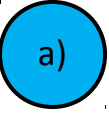
b)

c)

## 5.2 How to Build a DSL with EMFText

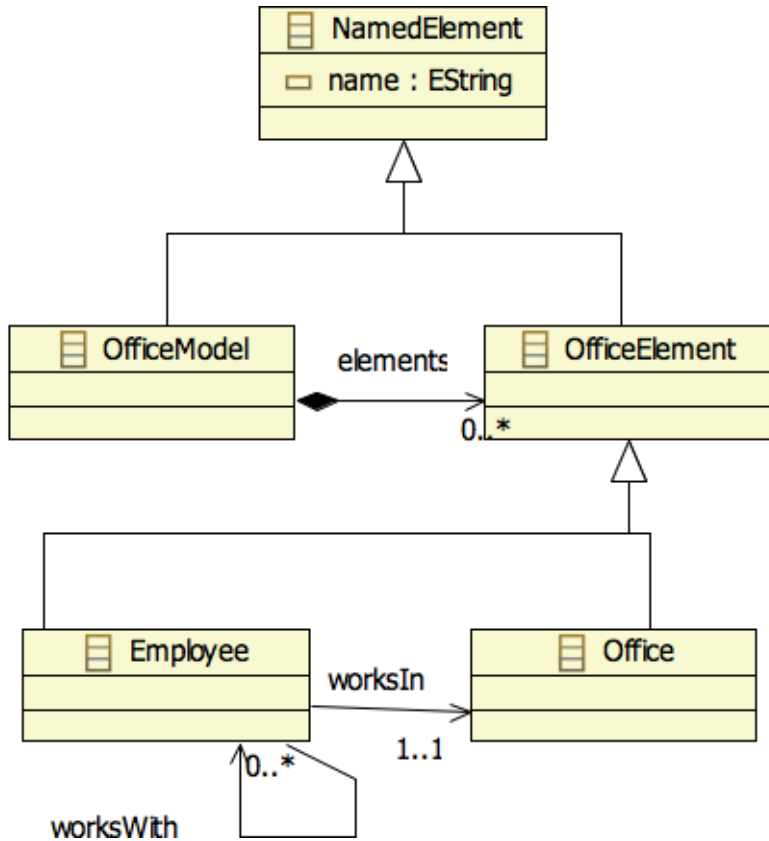


# How to build a DSL – Metamodel



Creating a new meta model:

- ▶ Define concepts, relations and properties in an Ecore model
- ▶ Existing meta models can be imported (e.g., UML, Ecore, ...)

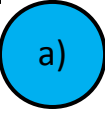


# How to build a DSL – Metamodel

16

Model-Driven Software Development in Technical Spaces (MOST)

## Metamodel elements (EMF concepts):



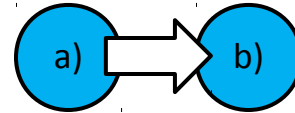
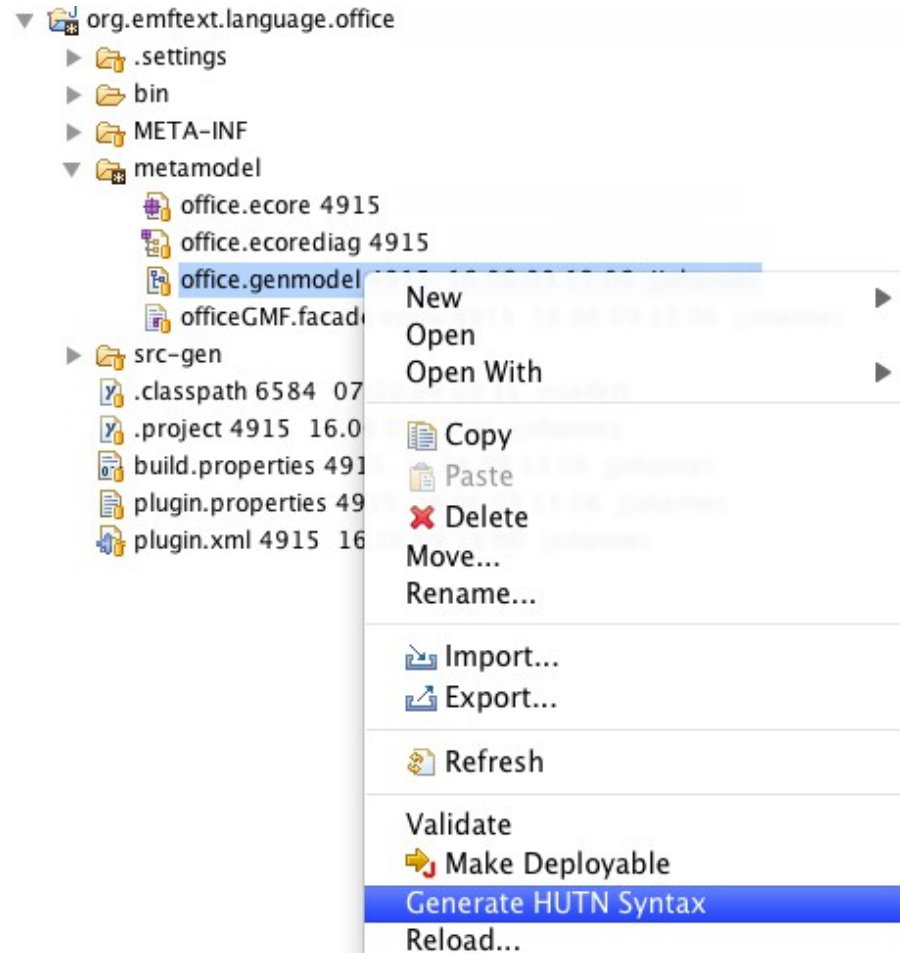
- Classes
- Data Types
- Enumerations
  
- Attributes
- References (Containment, Non-containment)
- Cardinalities
  
- Inheritance
  
- The Office DSL is a Material metamodel, because offices are treated as data



# Generate initial syntax (Human Usable Text Notation)

17

Model-Driven Software Development in Technical Spaces (MOST)



# Initial HUTN Syntax - Grammar

18

Model-Driven Software Development in Technical Spaces (MOST)

b)

```
office.cs
| SYNTAXDEF office
FOR <http://emftext.org/office>
START OfficeModel

TOKENS{
  DEFINE COMMENT$'//'(~/(\n|\r|\uffff))*$;
  DEFINE INTEGER$('-')?('1'..'9')('0'..'9')*|'0'$;
  DEFINE FLOAT$('-')?('1'..'9') ('0'..'9')* | '0' '.' ('0'..'9')+ $;
}

TOKENSTYLES{
  "OfficeModel" COLOR #7F0055, BOLD;
  "name" COLOR #7F0055, BOLD;
  "elements" COLOR #7F0055, BOLD;
  "Employee" COLOR #7F0055, BOLD;
  "worksIn" COLOR #7F0055, BOLD;
  "worksWith" COLOR #7F0055, BOLD;
  "Office" COLOR #7F0055, BOLD;
}

RULES{
  OfficeModel ::= "OfficeModel" "{" ( "name" ":" name['"', '"] | "elements" ":" elements )* "}" ;
  Employee ::= "Employee" "{" ( "name" ":" name['"', '"] | "worksIn" ":" worksIn[] | "worksWith" ":" worksWith[] )* "}" ;
  Office ::= "Office" "{" ( "name" ":" name['"', '"] )* "}" ;
}
```

# Full Grammar

```
//*****  
// Copyright (c) 2006-2010  
// Software Technology Group, Dresden University of Technology  
//  
// All rights reserved. This program and the accompanying materials  
// are made available under the terms of the Eclipse Public License v1.0  
// which accompanies this distribution, and is available at  
// http://www.eclipse.org/legal/epl-v10.html  
//  
// Contributors:  
//   Software Technology Group - TU Dresden, Germany  
//   - initial API and implementation  
// *****/  
SYNTAXDEF office  
FOR <http://emftext.org/office>  
START OfficeModel  
OPTIONS {  
    licenceHeader = "../..org.dropsbox/licence.txt";  
    generateCodeFromGeneratorModel = "true";  
    disableLaunchSupport = "true";  
    disableDebugSupport = "true";  
}  
RULES {  
    OfficeModel ::= "officemodel" name[]  
                  "{" elements* "}" ;  
  
    Office ::= "office" name[];  
  
    Employee ::= "employee" name[]  
                "works" "in" worksIn[]  
                "works" "with"  
                worksWith[] ("," worksWith[])* ;  
}
```

# Initial, Generated HUTN Syntax – Example Document

b)

```
st.office X
OfficeModel {
  name : "SoftwareTechnology"

  elements :
    Office {
      name : "INF2080"
    }

  elements :
    Office {
      name : "INF2084"
    }

  elements :
    Employee {
      name : "Florian"
      worksIn : INF2080
      worksWith : Jendrik
    }

  elements :
    Employee {
      name : "Jendrik"
      worksIn : INF2084
      worksWith : Florian
    }
}
```

# Syntax Refinement – The Concrete Syntax Language CS

21

Model-Driven Software Development in Technical Spaces (MOST)

b)

Structure of a .cs file:

- Header
  - File extension
  - Meta model namespace URI, *location*
  - Start element(s)
  - *Imports (meta models, other syntax definitions)*
- *Options*
- *Token Definitions*
- Syntax Rules

The screenshot displays two windows from an IDE. The left window, titled 'office.cs', shows the following code:

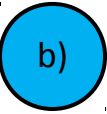
```
SYNTAXDEF office
FOR <http://emftext.org/office>
START OfficeModel

RULES{
  OfficeModel ::= "officemodel" ;
}
```

The right window, titled 'Outline', shows a tree view of the file's structure:

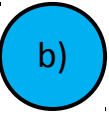
- office : http://emftext.org/office
  - [ab] TEXT
  - [ab] WHITESPACE
  - [ab] LINEBREAK
  - abc officemodel
  - OfficeModel
    - Choice

# Syntax Refinement – Syntax Rules in EBNF

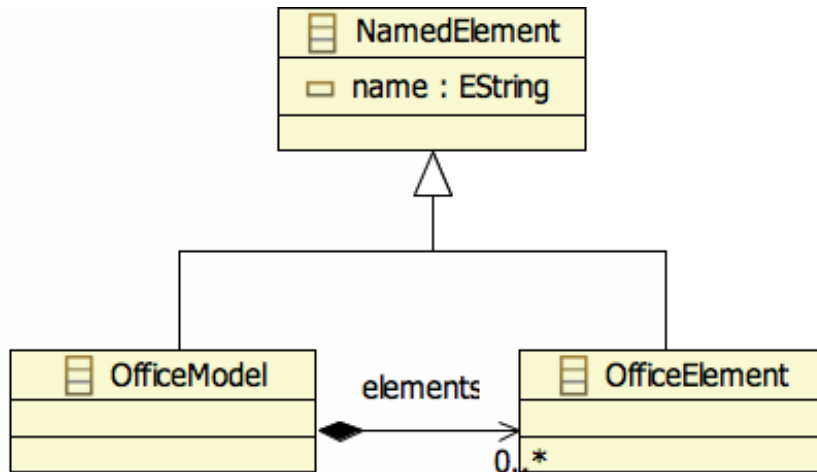


- ▶ EBNF (Extended Backus-Naur Form) is the dominating metalanguage of the Grammarware technical space
  
- ▶ **Definition elements in EBNF rules:**
  - Static strings (keywords)      "public"
  - Choices                              a|b
  - Multiplicities                        +, \*
  - Compounds                            (ab)
  - Terminals                             a[]                    (Non-containment references, attributes)
  - Non-terminals                        a                      (Containment references)
  
- ▶ **Language mapping:** One syntax rule per metaclass defines the *language mapping* between EBNF and EMF metaclasses
  - Syntax: MetaClassName ::= *Syntax Definition* ;
  
- ▶ All concept mappings define a *language mapping*

# Customized Syntax Rules - Examples



// EMOF Metamodel



// Grammar Rule

```
OfficeModel ::= "officemodel" name[]
               "{" elements* "}" ;
```

// Textual model

```
officemodel
SoftwareTechnology {
    ...
}
```



# Customized Syntax Rules - Examples

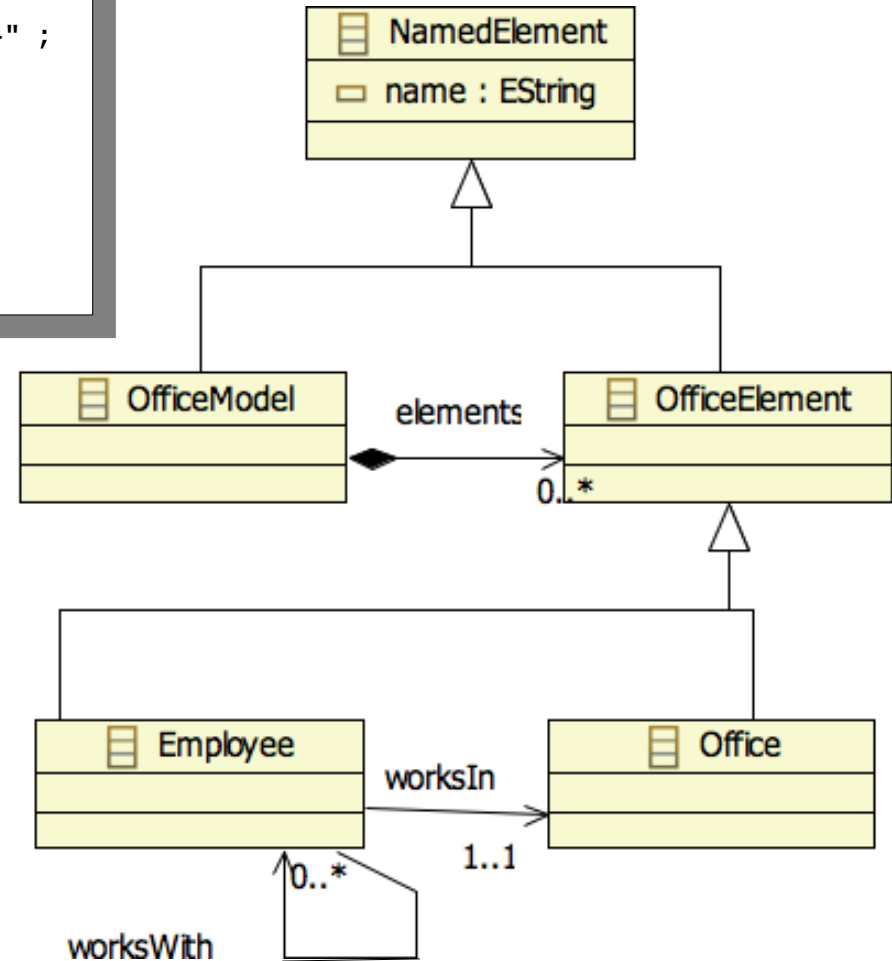
24

Model-Driven Software Development in Technical Spaces (MOST)

```
RULES {  
  OfficeModel ::= "officemodel" name[] "{" elements* "}" ;  
  
  Office ::= "office" name[] ;  
  
  Employee ::= "employee" name[]  
             "works" "in" worksIn[]  
             "works" "with"  
             worksWith[] ("," worksWith[])* ;  
}
```

```
// Text  
officemodel SoftwareTechnology {  
  office INF2080  
  employee Florian  
  works in INF2018  
}
```

// EMOF Metamodel

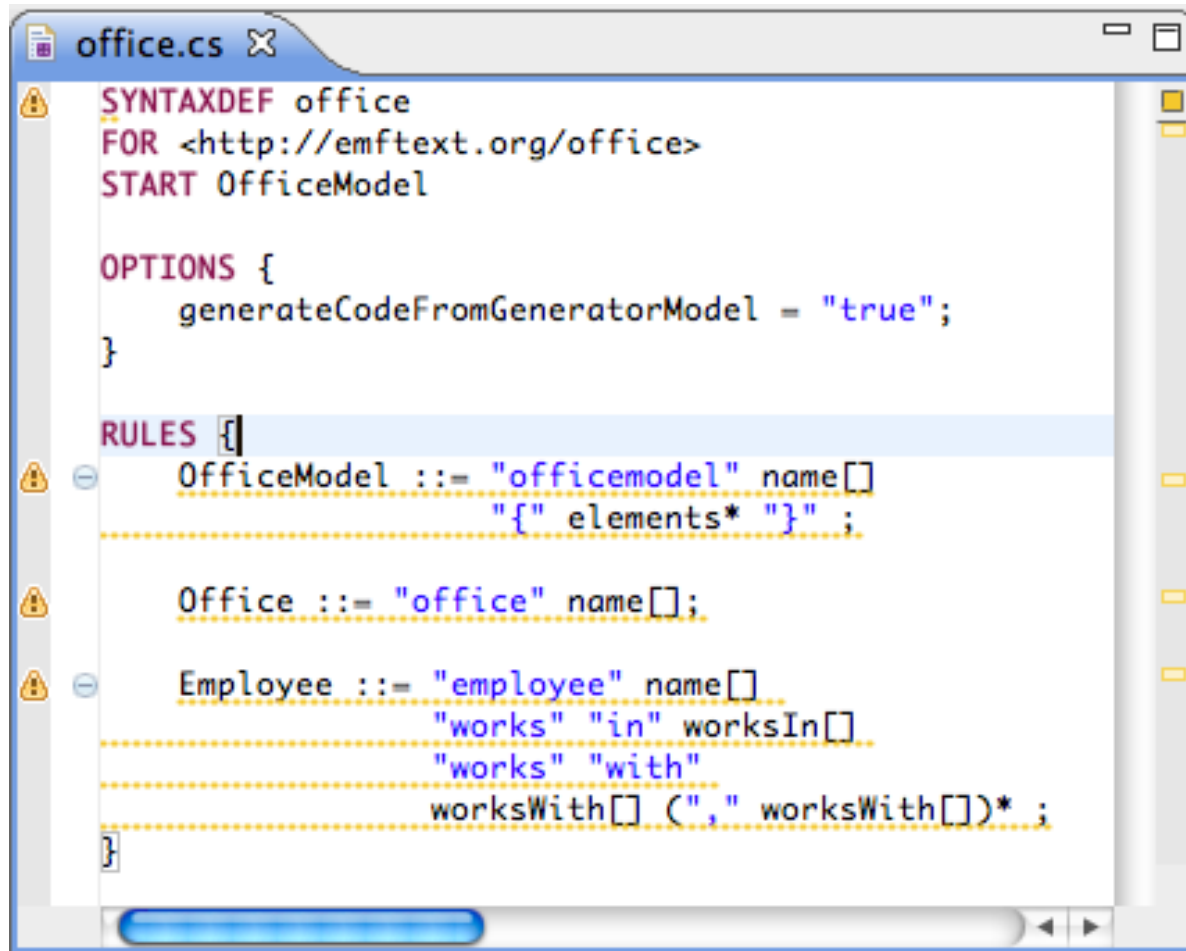


b)



# Grammar of Complete Customized Syntax

b)



```
SYNTAXDEF office
FOR <http://emftext.org/office>
START OfficeModel

OPTIONS {
    generateCodeFromGeneratorModel = "true";
}

RULES {
    OfficeModel ::= "officemodel" name[]
                "{" elements* "}";

    Office ::= "office" name[];

    Employee ::= "employee" name[]
               "works" "in" worksIn[]
               "works" "with"
               worksWith[] ("," worksWith[])* ;
}
```

# DSM: Generic Syntax vs. Custom Syntax

b)

```
st.office x
OfficeModel {
  name : "SoftwareTechnology"

  elements :
    Office {
      name : "INF2080"
    }

  elements :
    Office {
      name : "INF2084"
    }

  elements :
    Employee {
      name : "Florian"
      worksIn : INF2080
      worksWith : Jendrik
    }

  elements :
    Employee {
      name : "Jendrik"
      worksIn : INF2084
      worksWith : Florian
    }
}
```

```
st.office x
officemodel SoftwareTechnology {

  office INF2080

  office INF2084

  employee Florian
    works in INF2080
    works with Jendrik

  employee Jendrik
    works in INF2084
    works with Florian
}
```



## 5.3. Advanced Features of EMFText

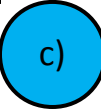


# Advanced Features – Attribute Mapping

28

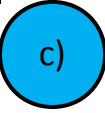
Model-Driven Software Development in Technical Spaces (MOST)

- ▶ Putting strings into EString attributes is easy
- ▶ How about EInt, EBoolean, EFloat, ..., custom data types?
- ▶ Solution A: Default mapping: The generated classes use the conversion methods provided by Java (java.lang.Integer, Float etc.)
- ▶ Solution B: Customize the mapping using a token resolver



```
// resolver interface looks up a lexem in the context
// example implementation for TRUE == yes
public void resolve(String lexem,
    EStructuralFeature feature,
    ITokenResolveResult result) {
    if ("yes".equals(lexem)) result.setResolvedToken(Boolean.TRUE);
    else result.setResolvedToken(Boolean.FALSE);
}
public String deResolve(Object value,
    EStructuralFeature feature,
    EObject container) {
    if (value == Boolean.TRUE) return "yes"; else return "no";
}
```

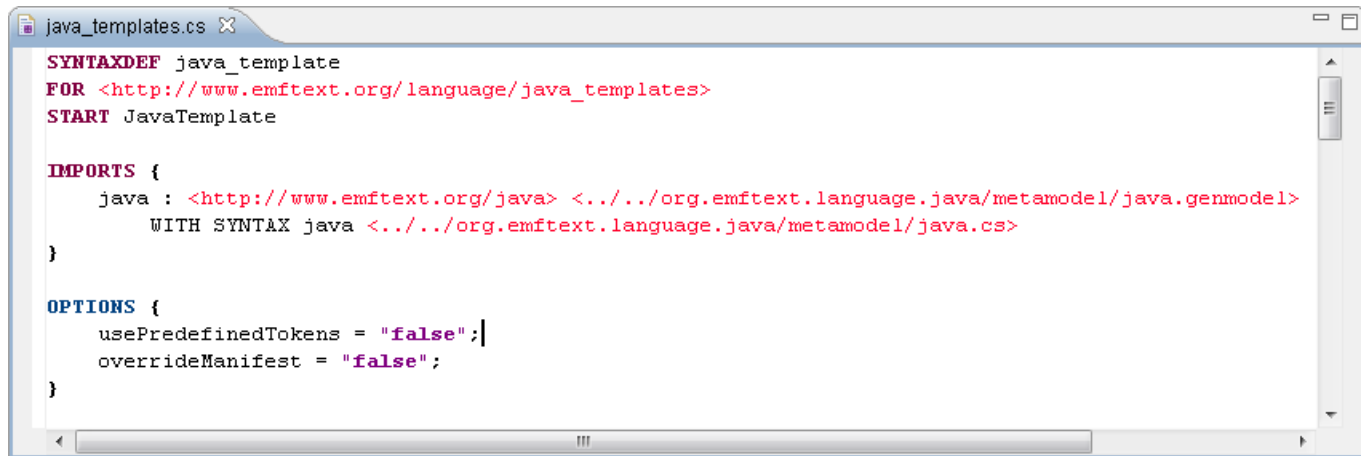
# Advanced Features – Resolving Cross References



Well, quite similar to attribute mappings:

- ▶ **Solution A:** Default resolvingSearches for matching elements that have an ID attribute, a name attribute or a single attribute of type EString and picks the first(Works well for simple DSLs without scoping rules)
- ▶ **Solution B:** Custom resolvingChange the generated resolver class (implements IReferenceResolver<ContainerType, ReferenceType>)For examples see the resolvers for the Java language

# Advanced Features – Syntax Modules for Language Composition



```
java_templates.cs X
SYNTAXDEF java_template
FOR <http://www.emftext.org/language/java_templates>
START JavaTemplate

IMPORTS {
  java : <http://www.emftext.org/java> <../../org.emftext.language.java/metamodel/java.genmodel>
        WITH SYNTAX java <../../org.emftext.language.java/metamodel/java.cs>
}

OPTIONS {
  usePredefinedTokens = "false";|
  overrideManifest = "false";
}
```

c)

- ▶ Import meta models optionally with syntax
- ▶ Extend, Combine, Compose existing DSLs
- ▶ Create embedded DSLs (e.g., for Java)
- ▶ Create a template language from your DSL
- ▶ ...

# Disclaimer: Pitfalls of Language Composition (with EMFText)

## Syntax and model extensions can be non-trivial

In EMFText problems may be caused by

- Unexpected inclusions between token definitions
- Intersections between token definitions (partial overlaps)
- Problems with the underlying parser generator, e.g. left-recursion or extensive backtracking
- Ambiguous grammars (may require non monotonic changes to the grammar and the metamodels)
- Interference between reference resolvers
- Different language semantics

Detected by EMFText

Avoided by extensive use of keywords

**Alternative parsing technologies:** Scannerless Parsing, Context-Aware Scanning, SDF/SGLR, MPS, Packrat Parsing, Parsing Expression Grammars ....

# The EMFText Syntax Zoo (>90 residents)

- ▶ Ecore, KM3 (Kernel Meta Meta Model)
- ▶ Quick UML, UML Statemachines
- ▶ Java 5 (complete), C# (in progress)
  
- ▶ Feature Models
- ▶ Regular Expressions
- ▶ OWL2 Manchester Syntax
  
- ▶ Java Behavior4UML
- ▶ DOT (Graphviz language)

...and lots of example DSLs

<http://emftext.org/zoo>



# Conclusion

- Few concepts to learn before using EMFText
- Creating textual syntax for new languages is easy, for existing ones it is harder, but possible (we did Java)
- Rich tooling can be generated from a syntax definition
- Textual and graphical syntax can complement each other (e.g., to support version control)
- Semantics (Interpretation/Compilation) must be defined manually – At most it can be reused

*Language is the blood of the soul into which thoughts run  
and out of which they grow.*

*(Oliver Wendell Holmes)*

## 5.4 Use EMFText for Embedded DSL



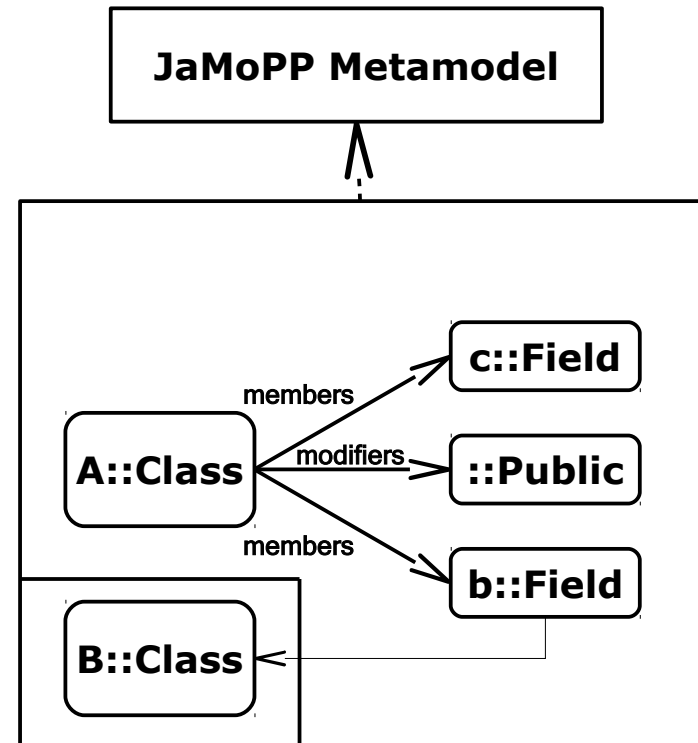
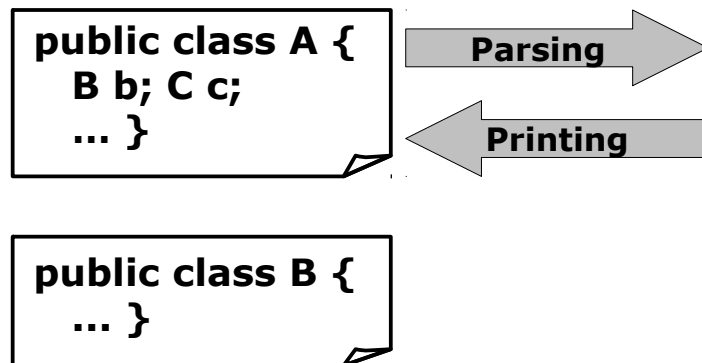
# JaMoPP: Lifting Java to DSLs

35

Model-Driven Software Development in Technical Spaces (MOST)

## Ingredients:

- Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)
- EMFText .cs definition for each concrete class



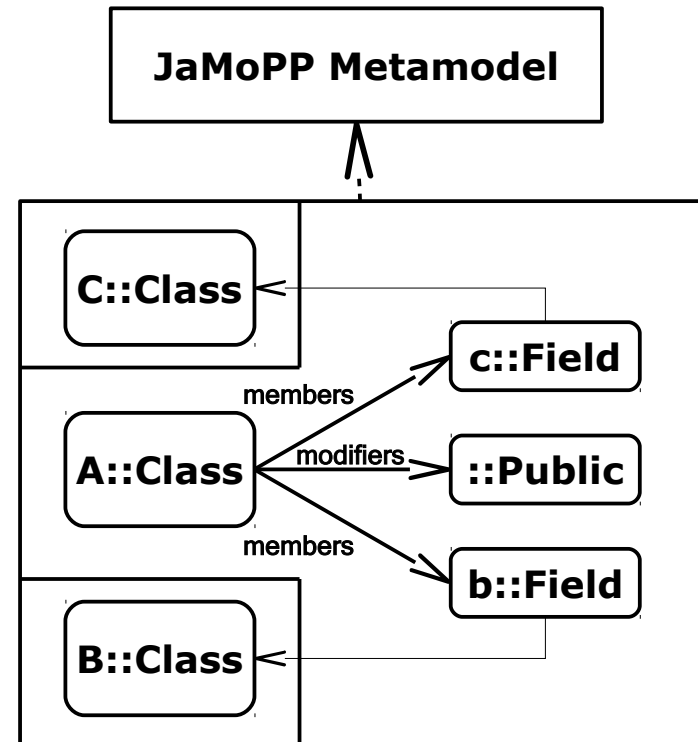
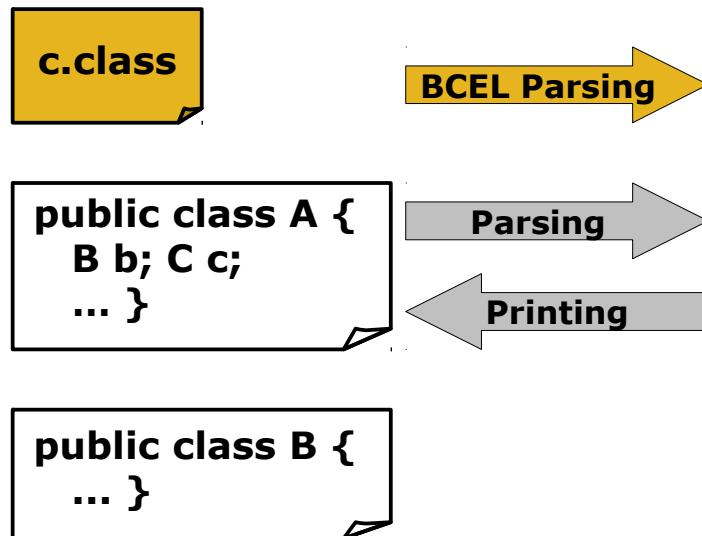
# JaMoPP: Lifting Java to DSLs

36

Model-Driven Software Development in Technical Spaces (MOST)

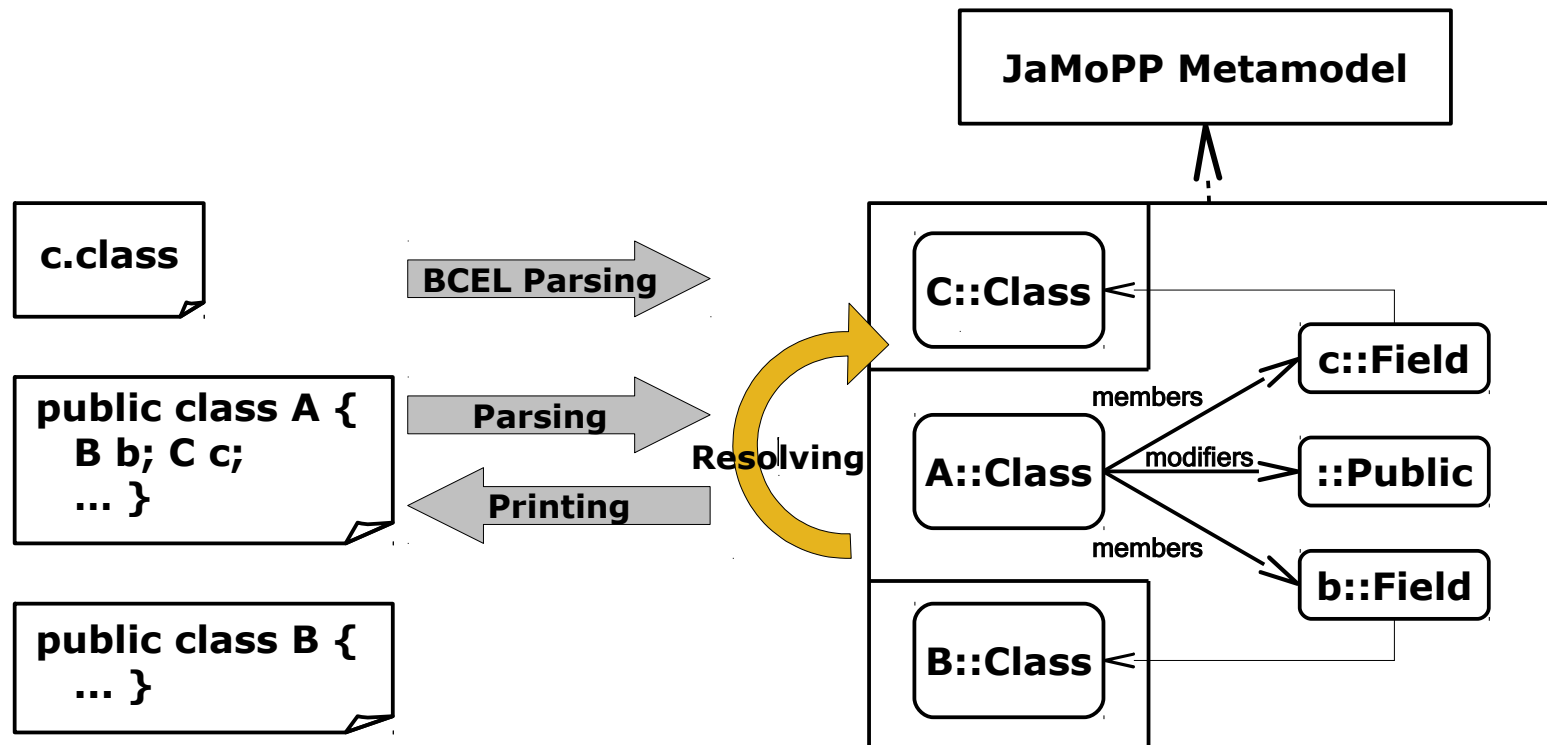
## Ingredients:

- Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)
- EMFText .cs definition for each concrete class
- BCEL Bytecode-Parser – to handle third-party libraries

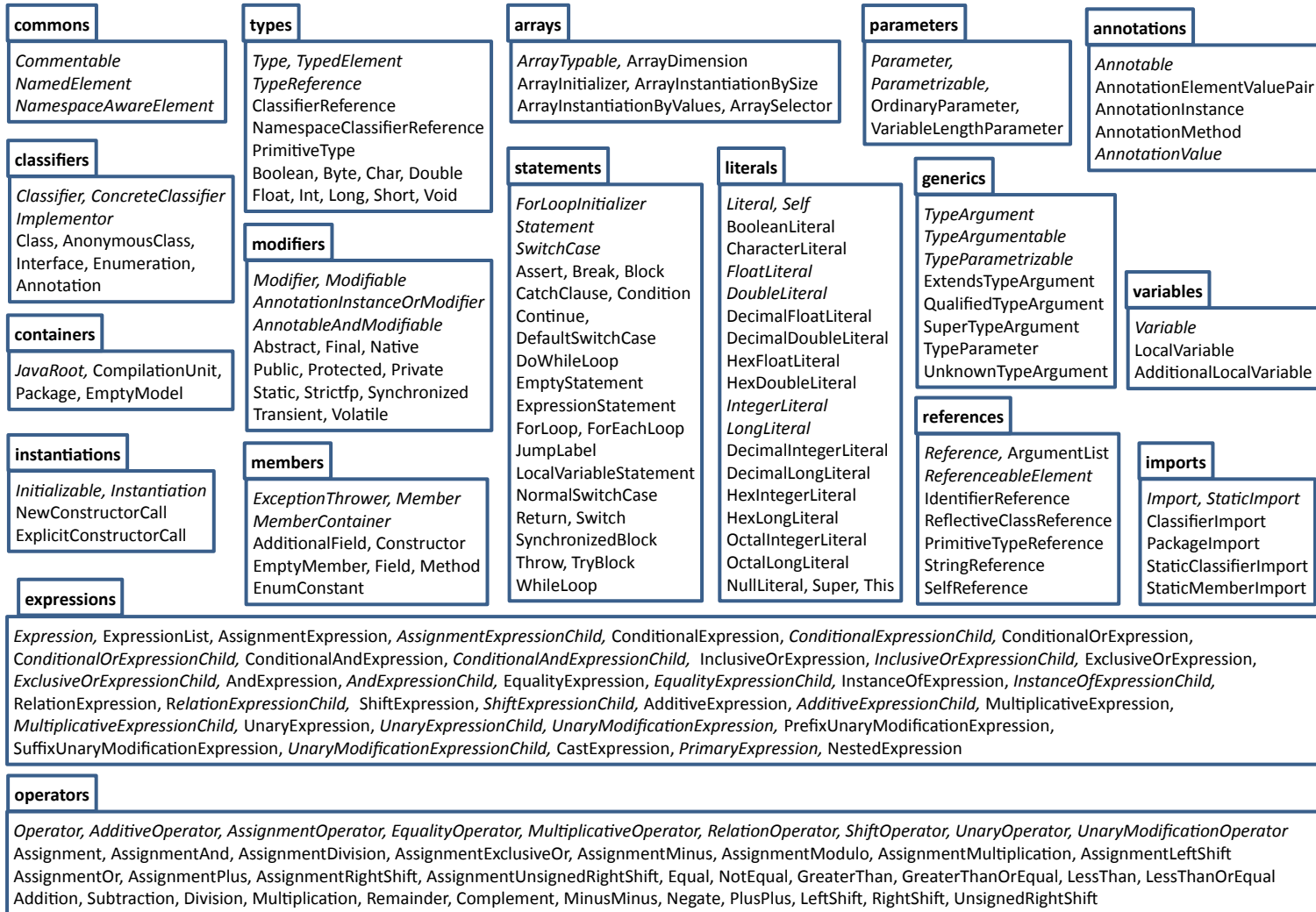


# JaMoPP: Lifting Java to DSLs

- Reference Resolvers that implement Java-specific static semantics (e.g., typing rules, scoping rules, referencing rules)



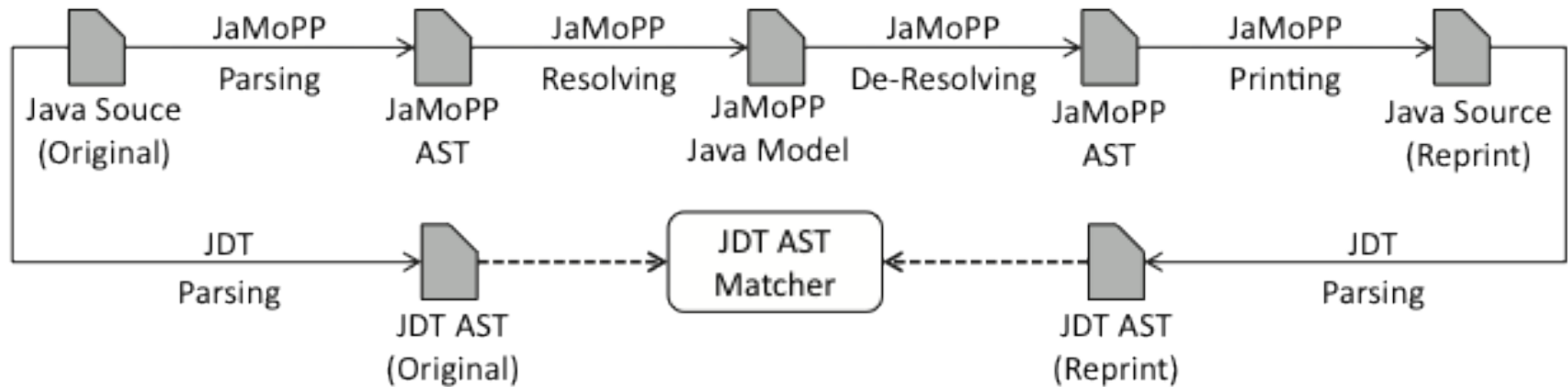
# JaMoPP Metamodel



# JaMoPP Testing

39

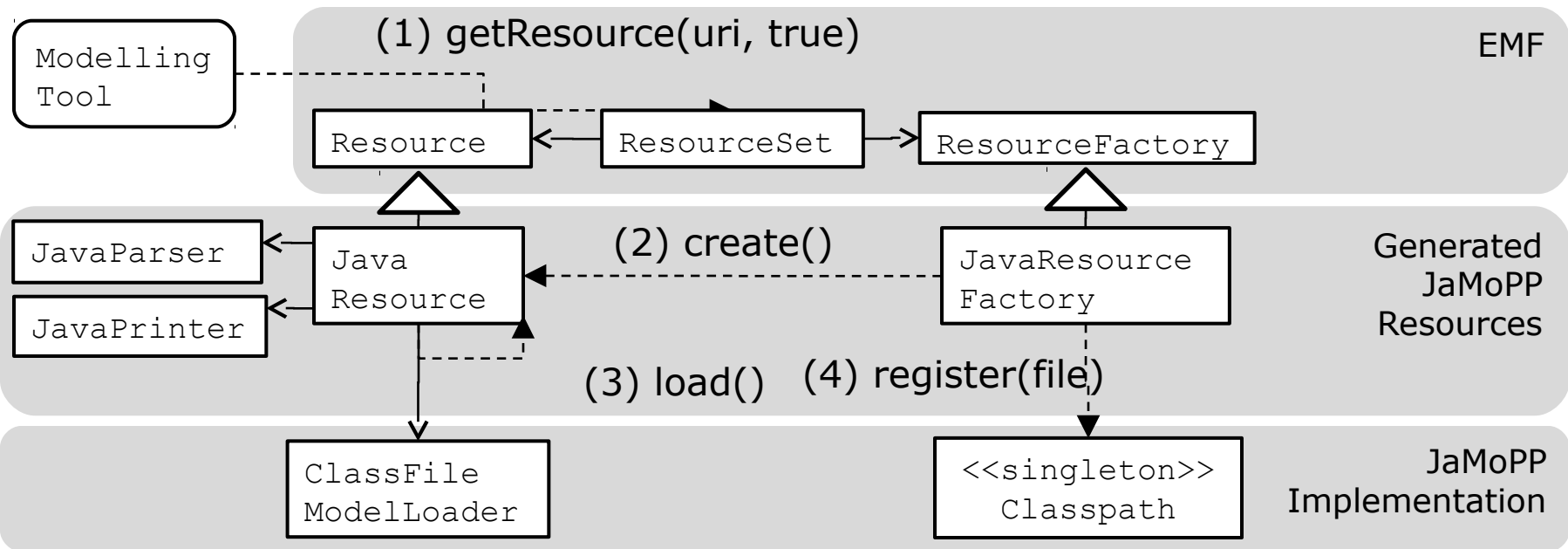
Model-Driven Software Development in Technical Spaces (MOST)



- ▶ Parsing public class A is easy, but parsing Java 5 is not (Unicode, Generics, Annotations and lots of weird things allowed by the JLS)
- ▶ JaMoPP Test suite:
  - 88.595 Java files (14.7 million non-empty lines including comments)
- ▶ Open Source projects:
  - AndroMDA 3.3, Apache Commons Math 1.2, Apache Struts 2.1.6, Apache Tomcat 6.0.18, Eclipse 3.4.1, Google Web Toolkit 1.5.3, JBoss 5.0.0 GA, Mantissa 7.2, Netbeans 6.5, Spring 3.0.0M1, Sun JDK 1.6.0 Update 7, XercesJ 2.9.1

# JaMoPP Tool Integration

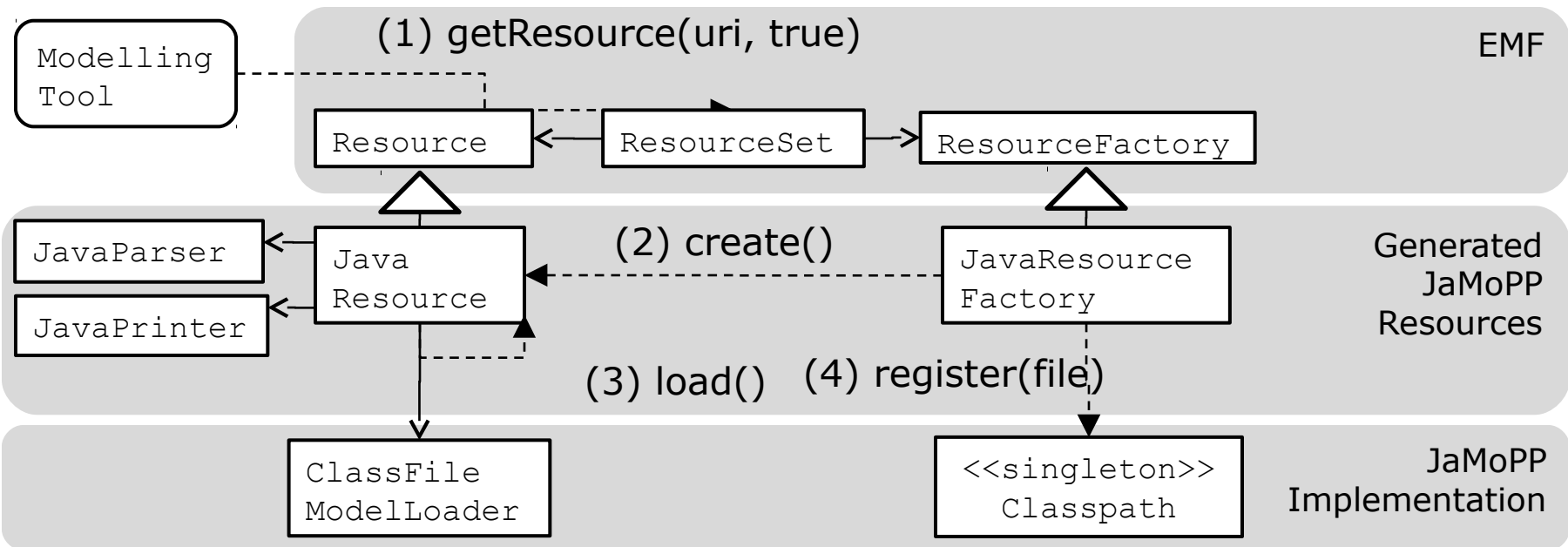
- ▶ JaMoPP seamlessly and transparently integrates with arbitrary EMF-based Tools
- ▶ Parsing Java files to models and Printing Java Files is simple





# JaMoPP Tool Integration

- ▶ JaMoPP seamlessly and transparently integrates with arbitrary EMF-based Tools
- ▶ Parsing Java files to models and Printing Java Files is simple



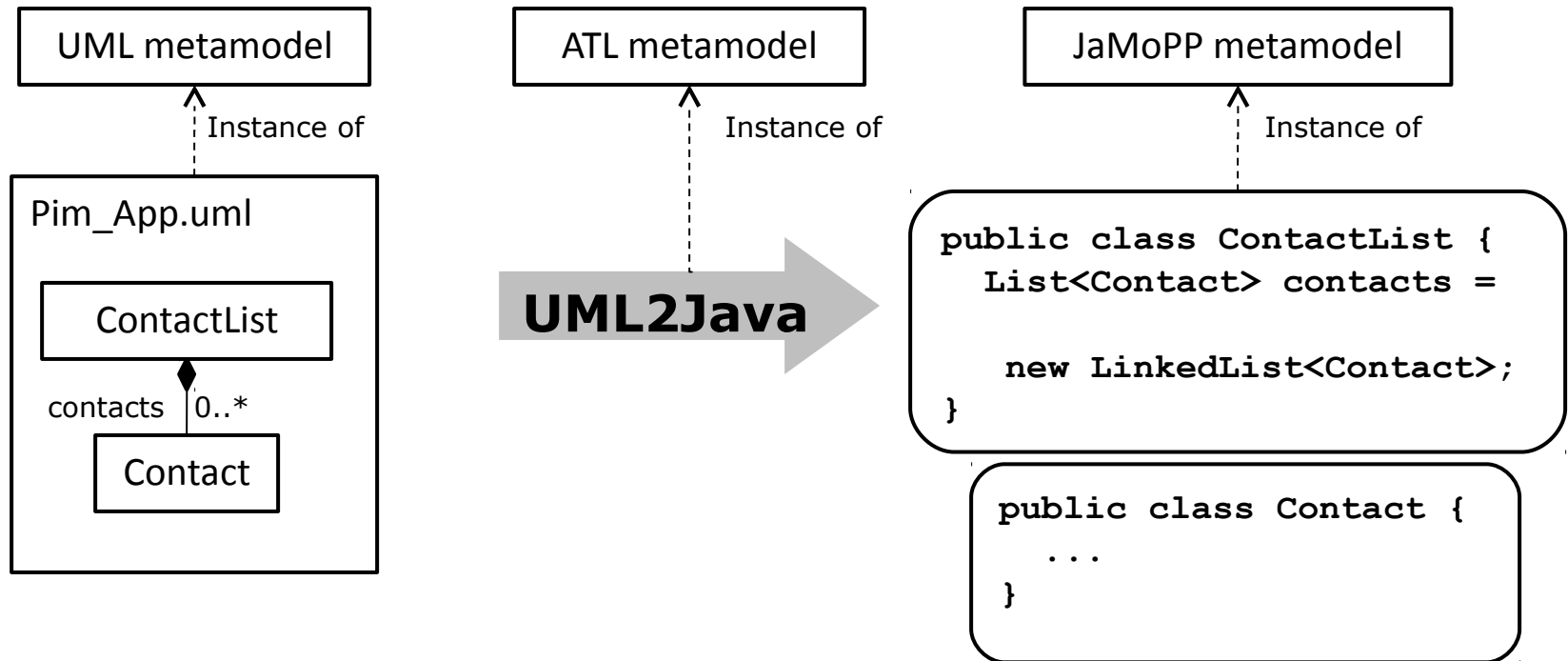
```
ResourceSet rs = new ResourceSetImpl();  
Resource javaResource = rs.getResource(URI.createFileURI("A.java"), true); //parsing  
javaResource.save(); // printing
```

# JaMoPP Application: Code Generation with the Code-To-Text Transformation Tool ATL

42

Model-Driven Software Development in Technical Spaces (MOST)

- Design UML model, apply M2M transformation, print JaMoPP model
- Syntactic and semantic correctness



# JaMoPP Application: Code Generation with the Code-To-Text Transformation Tool ATL

- Design UML model, apply M2M transformation, print JaMoPP model

```
rule Property {
  from umlProperty : uml!Property
  to javaField : java!Field (
    name <- umlProperty.name,
    type <- typeReference
  ),

  typeReference : java!TypeReference (
    target <- if (umlProperty.upper = 1) then umlProperty.type
      else
        java!Package.allInstances()->any(p | p.name = 'java.lang').compilationUnits->collect(
          cu | cu.classifiers)->flatten()->any(c | c.name = 'LinkedList')
      endif,
    typeArguments <- if (umlProperty.upper = 1) then
      Sequence{} -- empty type argument list
    else
      Sequence{typeArgument}
    endif
  ),

  typeArgument : java!QualifiedTypeArgument (
    target <- umlProperty.type
  )
}
```

# JaMoPP Application: Code Analysis with the Query Language OCL

44

Model-Driven Software Development in Technical Spaces (MOST)

- ▶ Parse Java source files to model instances
- ▶ Run OCL queries to find undesired patterns

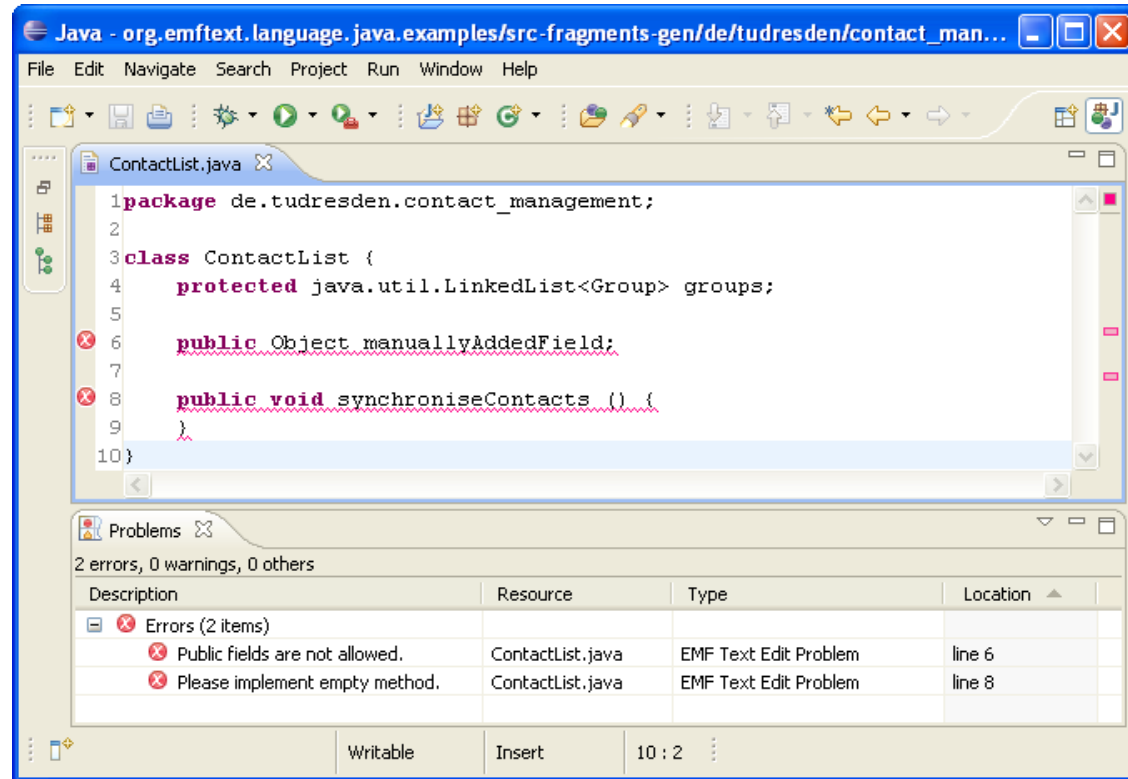
```
context members::Field inv:  
  self->modifiers->select(m|m.ocIsKindOf(modifiers::Public))->size() = 0
```

# JaMoPP Application: Code Analysis (OCL)

45

Model-Driven Software Development in Technical Spaces (MOST)

- ▶ Parse Java source files to model instances
- ▶ Run OCL queries to find undesired patterns



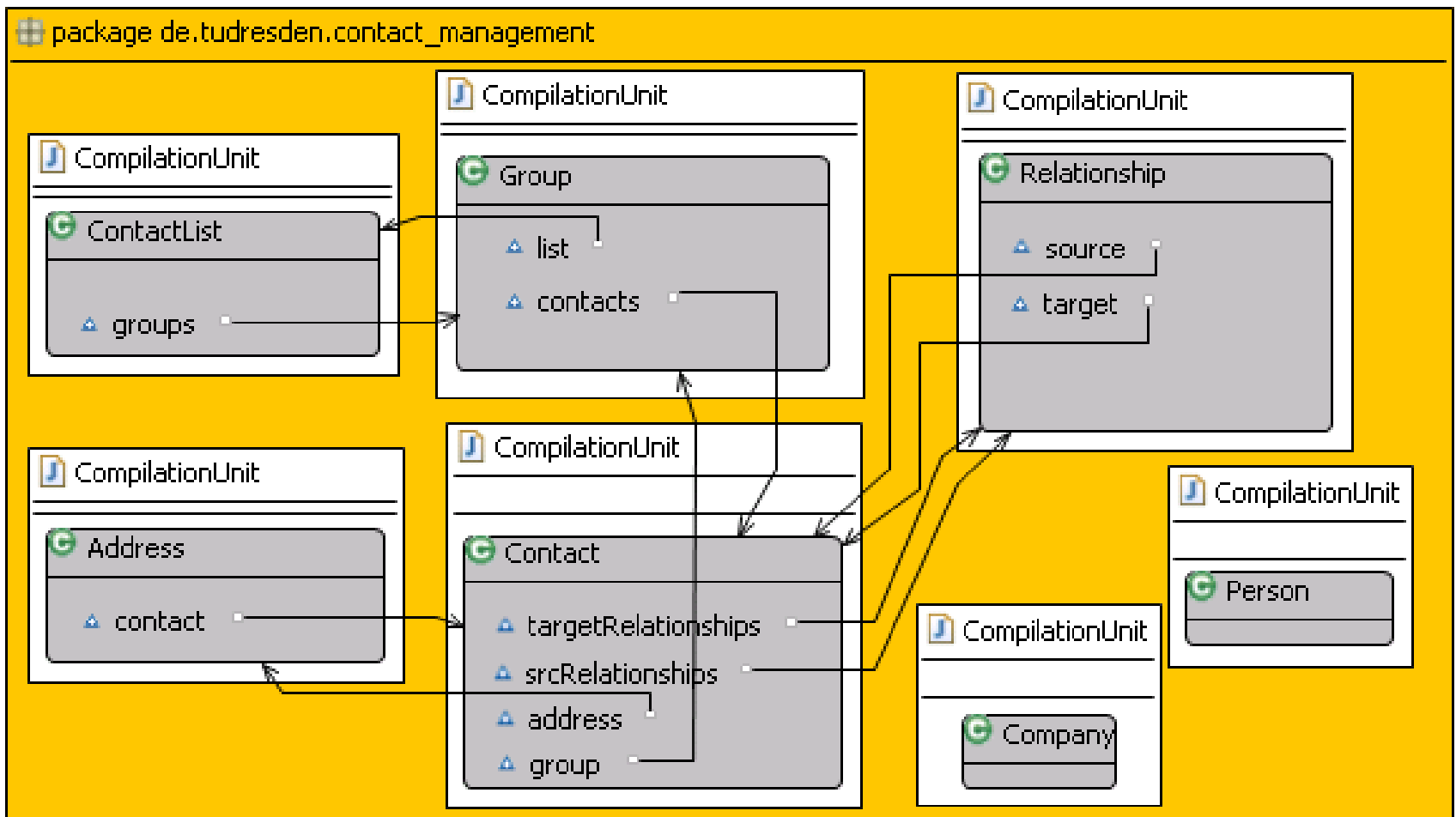
**context members::Field inv:**

**self->modifiers->select(m|m.ocIsKindOf(modifiers::Public))->size() = 0**



# JaMoPP Application: Code Visualization (via Eclipse GMF)

- 46 Model-Driven Software Development in Technical Spaces (MOST)
- ▶ Create .gmfgraph, gmftool, and gmfmap model
  - ▶ Generate Graphical Editor for Java



# JaMoPP Application: Software Product Line Engineering (FeatureMapper)

47

Model-Driven Software Development in Technical Spaces (MOST)

The screenshot displays two windows from the JaMoPP application. The left window, titled 'MappingView', shows a hierarchical feature model for 'Feature Contact Management'. The model includes a 'Group 0' with sub-features: 'Feature Addresses', 'Feature Relationships', 'Feature Contact Opportunities', 'Feature Notes', 'Feature Groups' (highlighted with a blue box and a circled '1'), and 'Feature Synchronisation'. Below the model, there are sections for 'Current Expression' (containing 'Not') and 'Assigned Feature Expression'. The right window, titled 'ContactList.java', shows the Java code for the 'ContactList' class. The code is as follows:

```
public class ContactList {  
    protected LinkedList < Group > groups ;  
    protected LinkedList < Contact > contactsList ;  
  
    public ContactList() {  
        groups = new LinkedList<Group>();  
        contactsList = new LinkedList<Contact>();  
    }  
  
    public void addContact(Contact toBeAdded, Group group) {  
        if (groups.indexOf(group) == -1) groups.add(group);  
        group.getContacts().add(toBeAdded);  
        contactsList.add(toBeAdded);  
    }  
}
```

The code is annotated with a circled '2' on the right side, indicating that the code for the feature 'Feature Groups' is highlighted in blue.

1 - Feature Model

2 - EMFText Editor for Java (code for feature highlighted)

## 5.4. Integrating DSLs and GPLs (Embedded DSL)

- ▶ <http://www.jamopp.org/index.php/JaMoPP>





# JaMoPP Applications: What else?

- ▶ Round-trip Support for template-based code generators
- ▶ Refactoring, Optimization using model transformations
- ▶ Traceability-related activities
  - Certification (Map code to the model elements)
  - Impact analysis (How much of the code will change if I do this?)
- ▶ Embedded DSL
- ▶ Model-based compilation to byte code
  
- ▶ ...

# Using the DSL – Interpretation vs. Compilation

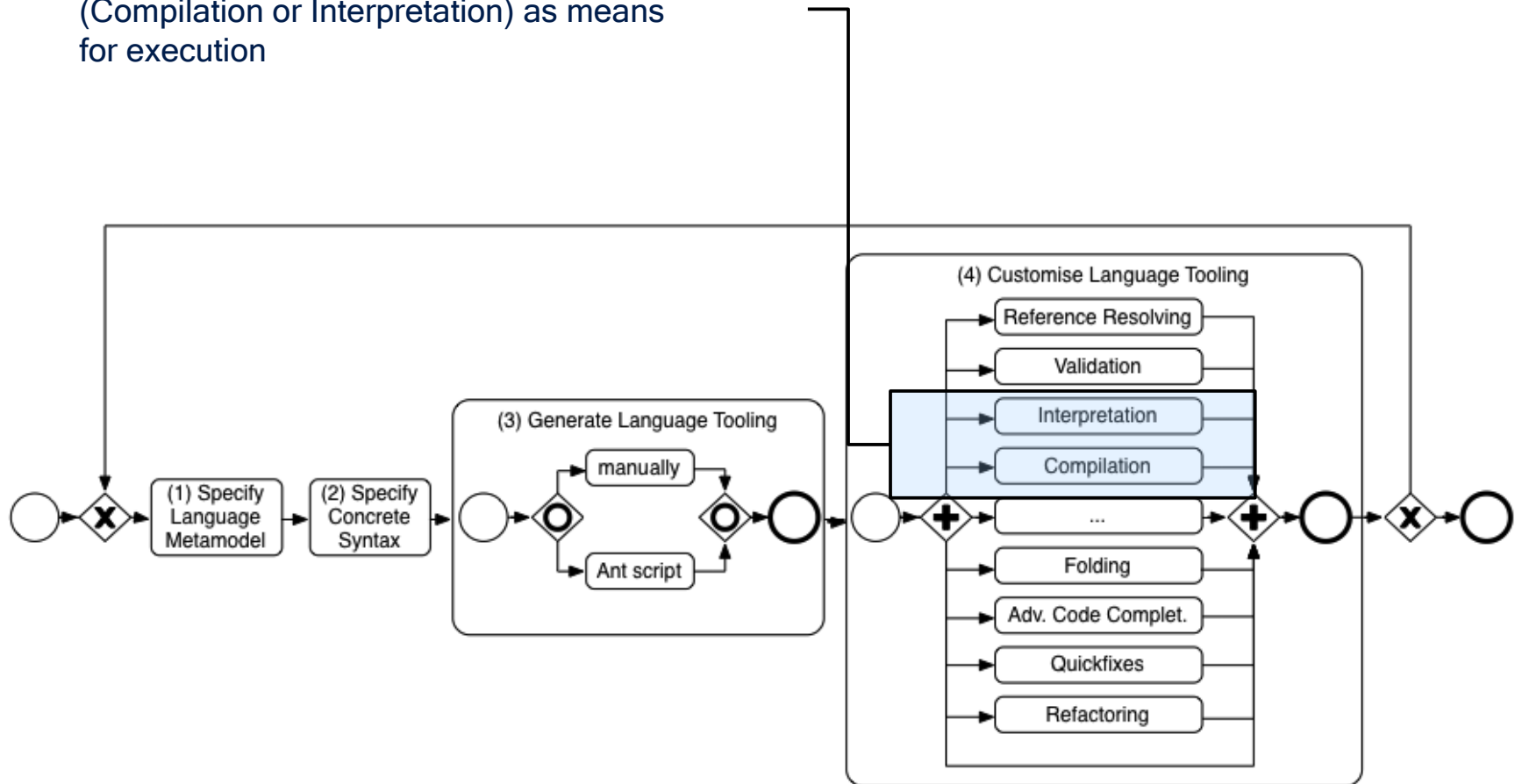
EMFText provides an extension point to perform interpretation (or compilation) whenever DSL documents change

To use the DSL we need to assign meaning by

- ▶ **Interpretation:** Traverse the DSL model and perform appropriate actions
- ▶ **Compilation:** Translate the DSL constructs to another (possibly executable) language
  - In principle compilation is an interpretation where the appropriate action is to emit code of the target language

# Integrating DSLs and GPLs

- **Technical Gap:** Mapping DSLs to GPLs (Compilation or Interpretation) as means for execution



# Embedded DSL “Embedded DieSeL”

55

Model-Driven Software Development in Technical Spaces (M

- ▶ An embedded domain-specific language (embedded DSL) is a language extension of a general-purpose language (GPL), the base language
- ▶ An embedded SDL can be *interpreted* or *reduced* (compiled to the core language)
- ▶ Typesafe Template Languages
  - Same syntax as string-based templates
  - [http://emftext.org/index.php/EMFText\\_Concrete\\_Syntax\\_Zoo\\_Java\\_Templates](http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Java_Templates)

```
1<%TEMPLATE INPUT=
2  "http://www.emftext.org/language/sandwich::Recipe" %>
3
4public class SandwichMaker {
5  private class Instr {};
6  public Instr <%= "instructions" %> = new Instr();
7
8  private boolean cleanIngredients() {}
9  private void addSingleIngredient() {}
10
11  public void addIngredients() {
12    <%FOR "engredients" %> // typo 'ingredients'
13      addSingleIngredient();
14    <%ENDFOR%>
15  }
16
17  public void makeSandwich() {
18    <%IF "instructions->exists(i|i.oclIsKindOf(Clean))" %>
19      boolean success = cleanIngredients();
20    <%ENDIF%>
21
22    if (!success) return; // May not be available
23    addIngredients();
24  }
25}
```

Description	Loc...
Errors (3 items)	
The expression in the placeholder has wrong type (was org.eclipse.ocl.ecore.impl.Order	line 6
The expression "engredients" is invalid (Unrecognized variable: (engredients)).	line 12
ReferenceableElement 'success' not declared	line 22

# Extended Code Generation for a Embedded DSL

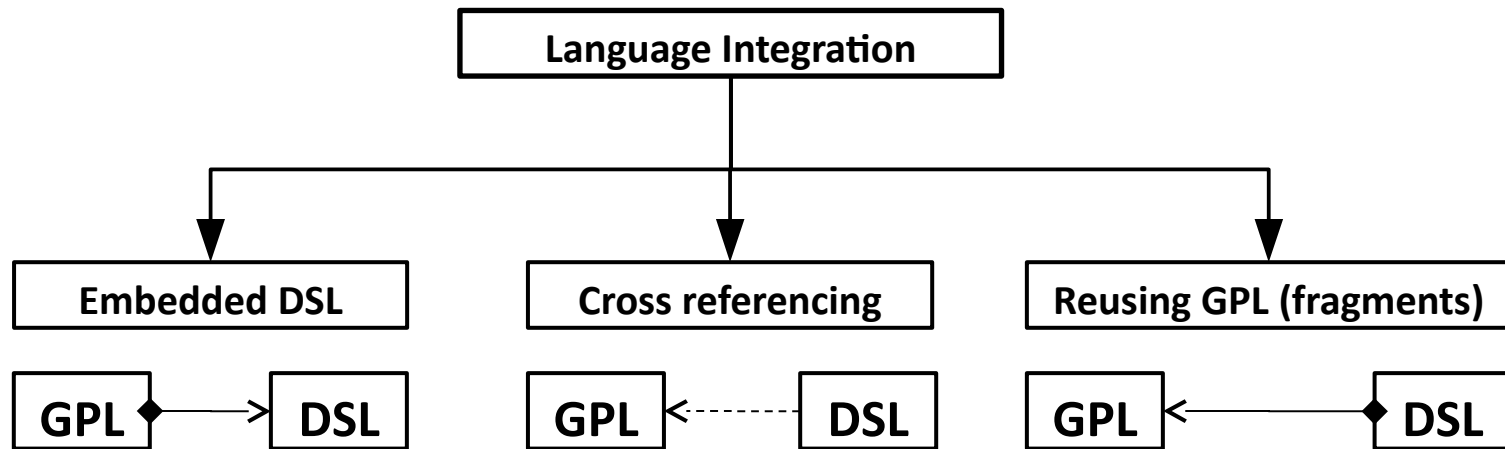
- ▶ An **Reducer** for an **embedded DSL** is a set of additional code generation rules
  - extending the rule set of the code generation of the base language
- ▶ Reducers should be composed modularly with the modules of the base language

## 5.4.2 Language Integration by Metamodel and Grammar Inheritance



# Integrating DSLs and GPLs

- ▶ Different integration scenarios

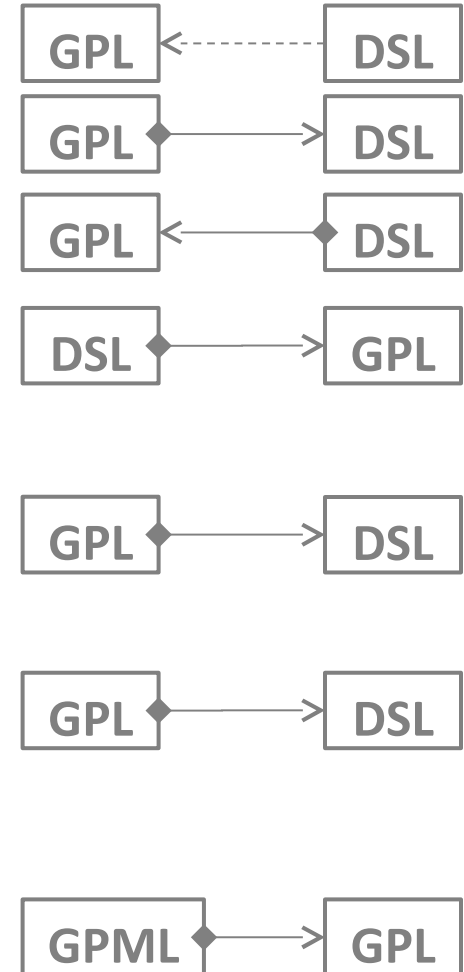


# Language Integration Examples

59

Model-Driven Software Development in Technical Spaces (MOST)

- ▶ FormsExtension
- ▶ FormsEmbedded
- ▶ JavaForms
- ▶ eJava
  - Provides metamodels with Eoperations
  - implementations without touching the generated java files
- ▶ JavaTemplate
  - Syntax safe templates with JaMoPP
- ▶ PropertiesJava
  - Experimental extension for Java to define C# like properties
- ▶ JavaBehaviour4UML
  - An integration of JaMoPP and the UML
  - Methods can be directly added to Classes in class diagrams





Thank you!

Questions?

<http://www.emftext.org>

emftext