40. Exchange Syntax and Textual DSLs using EMFText

1

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

Version 12-0.3, 10/22/12

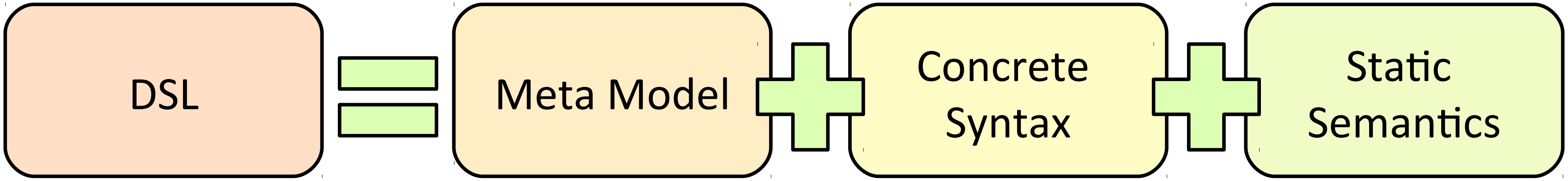
Literature

- http://www.emftext.org
- 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.

EMFText is used by our new start-up, DevBoost
www.devboost.de

40.1 What is a DSL?

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



Domain concepts, relations, attributes

Symbols, shapes
Texts keywords

Behavior, meaning

Productivity Gains with DSL

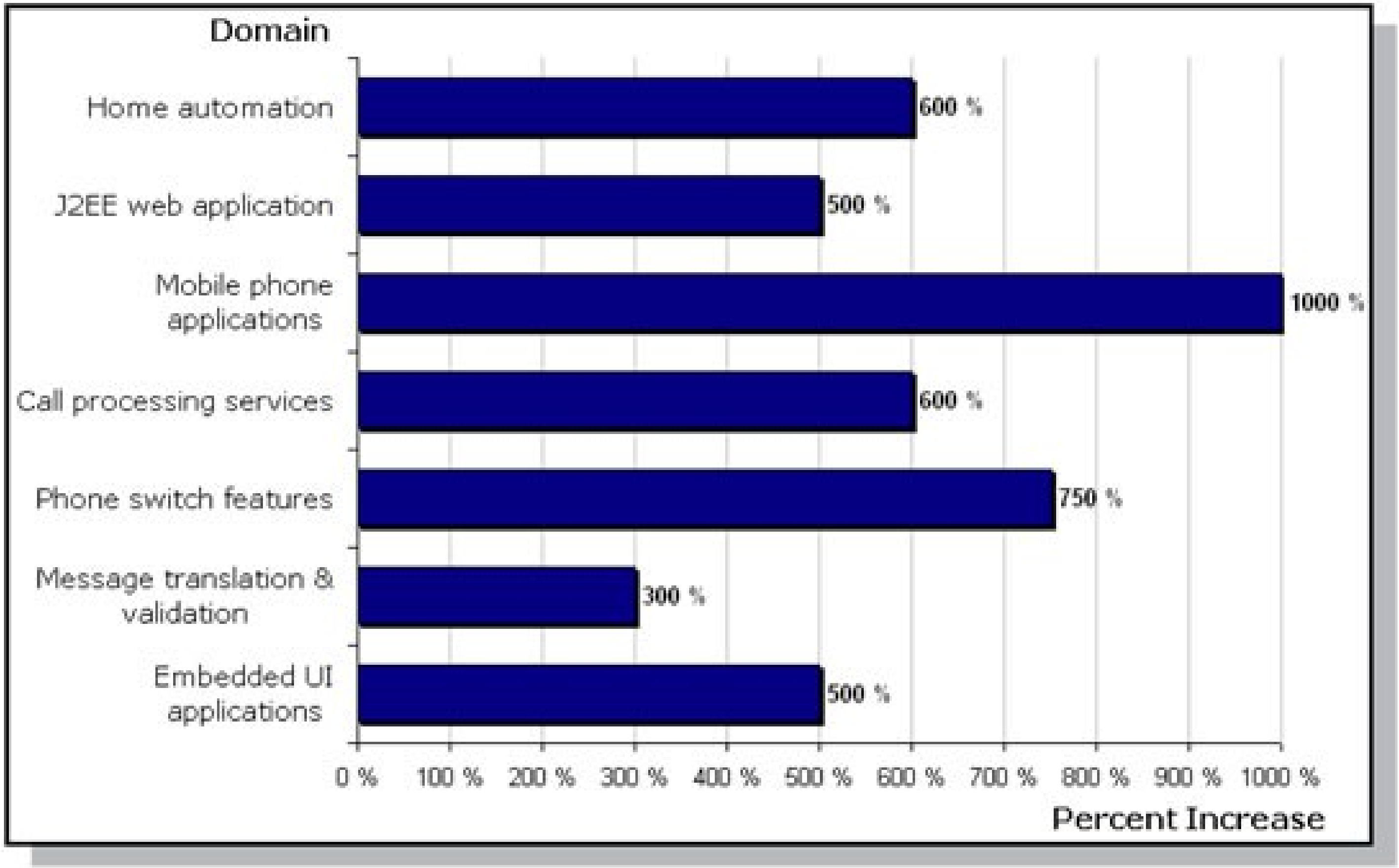
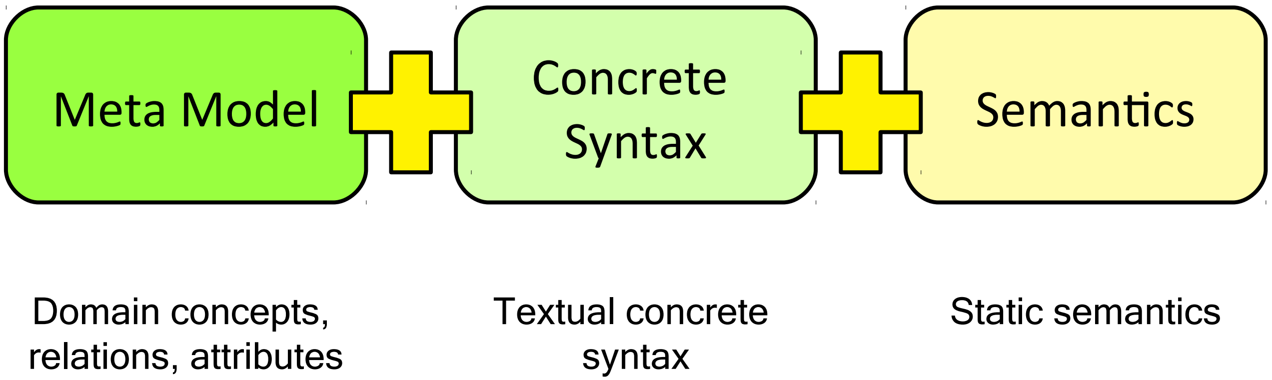


Figure 3: Measured productivity improvements in various domains

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

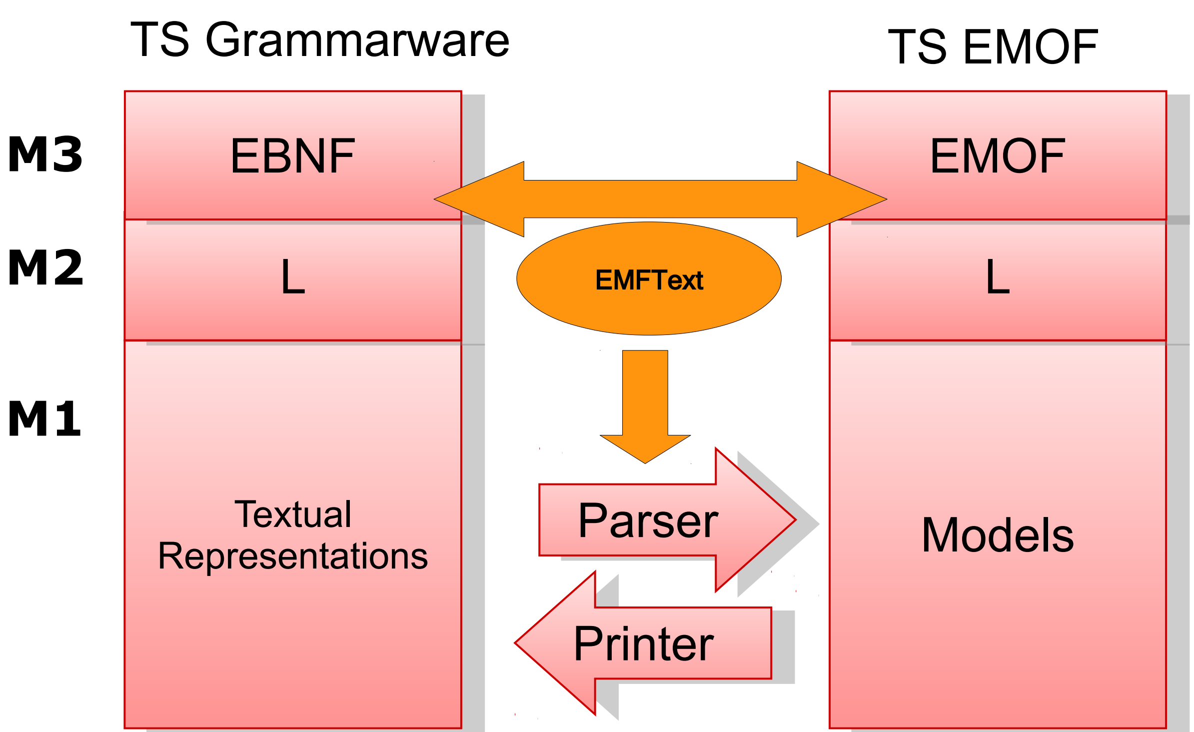
What is a Textual Domain-Specific Language (DSL)?

- 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 from EMOF to 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/

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

Philosophy and Goals

Design principles:

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

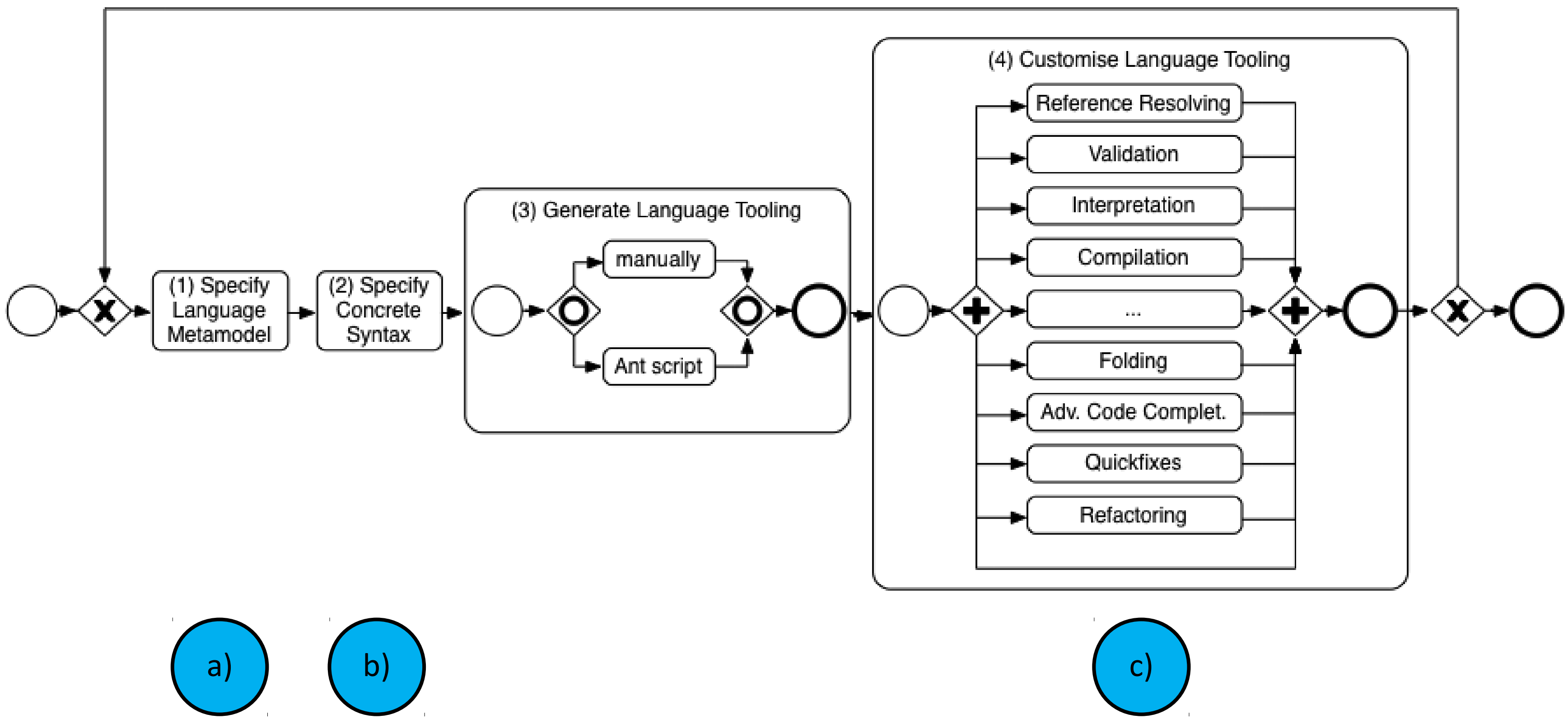
Syntax definition should be

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

EMFText Features

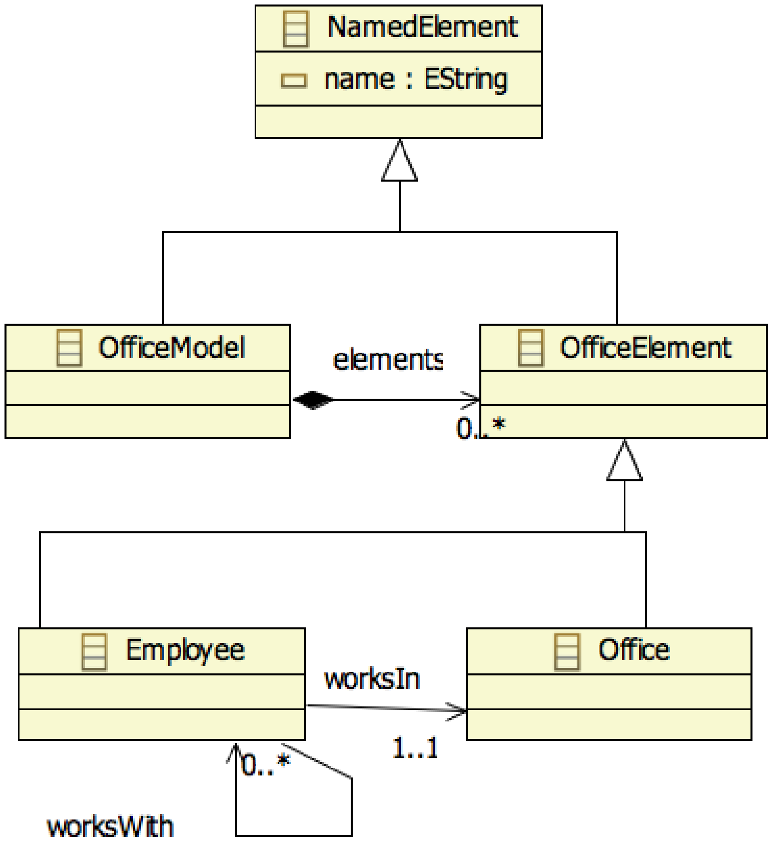
- **Generation Features**
  - Generation of independent code
  - Generation of Default Syntax
  - Customizable Code Generation
- **Specification Features**
  - Modular Specification
  - Default Reference Resolving
  - Comprehensive Syntax Analysis
- **Editor Features**
  - Code Completion, Customizable Syntax and Occurence 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



40.2 How to Build a DSL

How to build a DSL – Meta model



Creating a new meta model:

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

a)

How to build a DSL – Meta model

a)

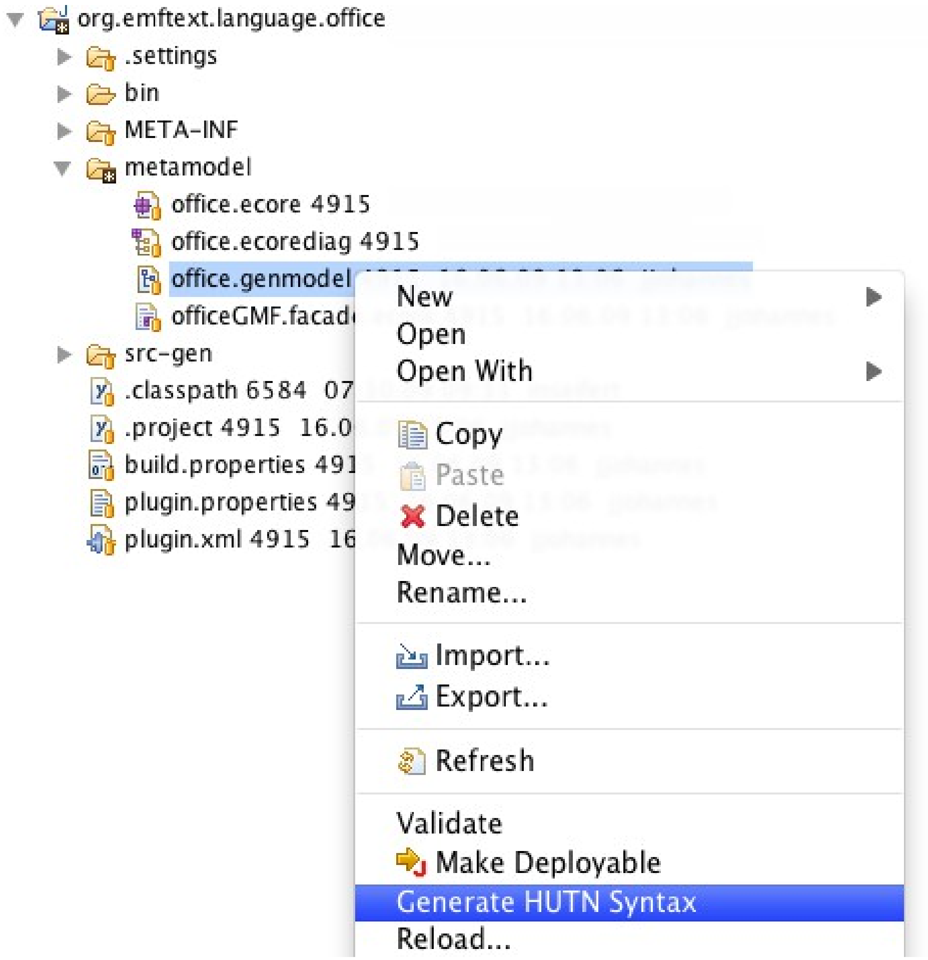
Meta model elements:

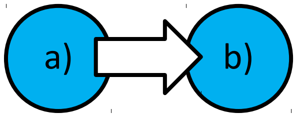
- Classes
- Data Types
- Enumerations

- Attributes
- References (Containment, Non-containment)
- Cardinalities

- Inheritance

Generate initial syntax (Human Usable Text Notation)





Initial HUTN syntax

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['"','"']  )* "}"  ;

}
```

Initial HUTN syntax – Example Document

b)

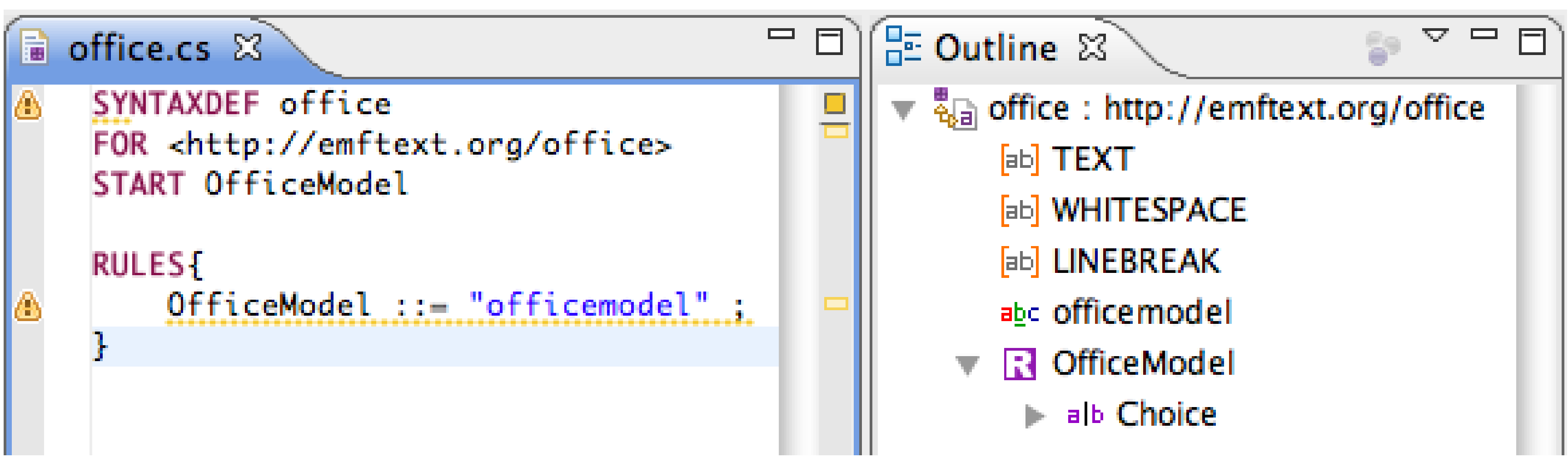
```
st.office

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

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



Syntax refinement – Syntax rules in EBNF

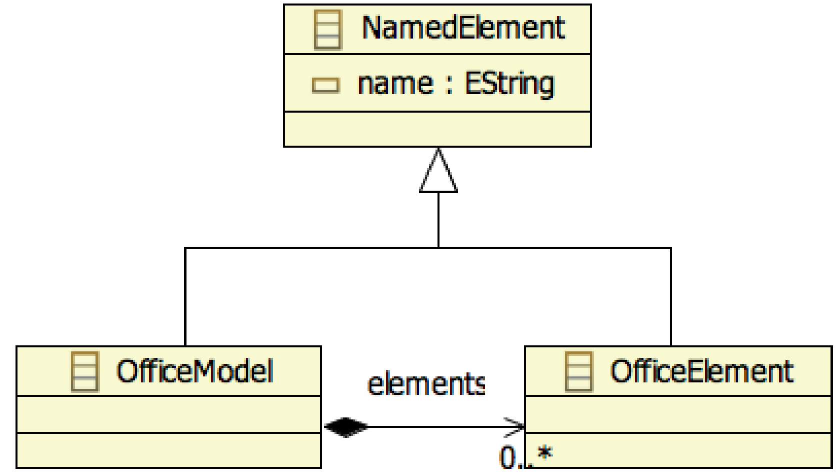
- One syntax rule per meta class defines the *language mapping* between EBNF and EMF metaclasses
  - Syntax: MetaClassName ::= *Syntax Definition* ;
- All concept mappings define a *language mapping*
- Definition elements in EBNF rules:
  - Static strings (keywords) “public”
  - Choices a|b
  - Multiplicities +,*
  - Compounds (ab)
  - Terminals a[] (Non-containment references, attributes)
  - Non-terminals a (Containment references)

b)

Customized Syntax Rules - Examples

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

b)



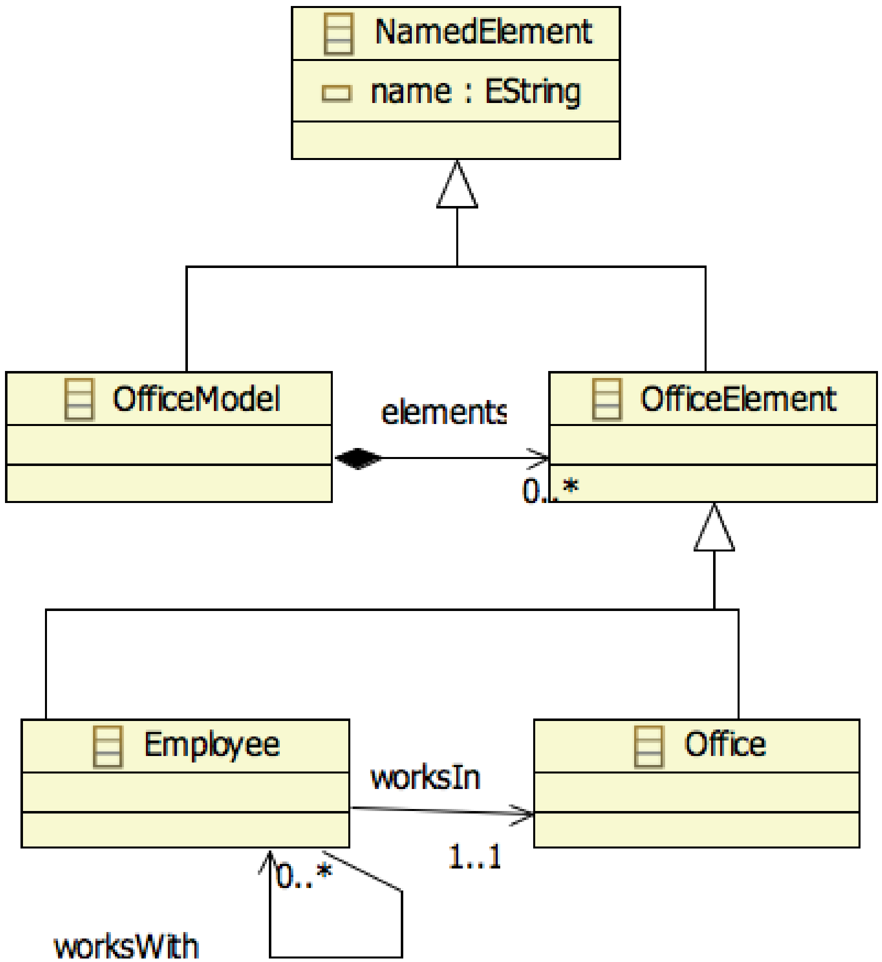
officemodel SoftwareTechnology {
...
}

Customized Syntax Rules - Examples

b)

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

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

Office ::= "office" name[];
```

```
officemodel SoftwareTechnology {
  office INF2080
  employee Florian
    works in INF2080
}
```



Complete Customized Syntax

b)

office.cs

```
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[])* ;
}
```

Generic Syntax vs. Custom Syntax

b)

st.office

```
OfficeModel {
    name : "SoftwareTechnology"

    elements :
        Office {
            name : "INF2080"
        }

    elements :
        Office {
            name :
        }

    elements :
        Employee {
            name :
            worksIr
            worksWi
        }

    elements :
        Employee {
            name :
            worksIr
            worksWi
        }
}
```

st.office

```
officemodel SoftwareTechnology {

    office INF2080

    office INF2084

    employee Florian
        works in INF2080
        works with Jendrik

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

40.3. Advanced Features of EMFText

Advanced Features – Attribute Mapping

c)

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

```
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

c)

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

c)

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

java_templates.cs

```
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";
}
```

Using the DSL – Interpretation vs. Compilation

So far we achieved to

- map input documents (text) to models
- do the inverse
-

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

To use the DSL we need to assign meaning by

- InterpretationTraverse the DSL document and perform appropriate actions
- CompilationTranslate 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)

Challenges for MDSD

- Developers are required to use different tool machinery for DSLs and GPLs.
- Explicit references between DSL and GPL code are not supported. Their relations are, thus, hard to track and may become inconsistent

- DSLs can not reuse (parts of) the expressiveness of GPLs
- Naive embeddings of DSL code (e.g., in Strings) do not provide means for syntactic and semantic checking
- Interpreted DSL code is hard to debug

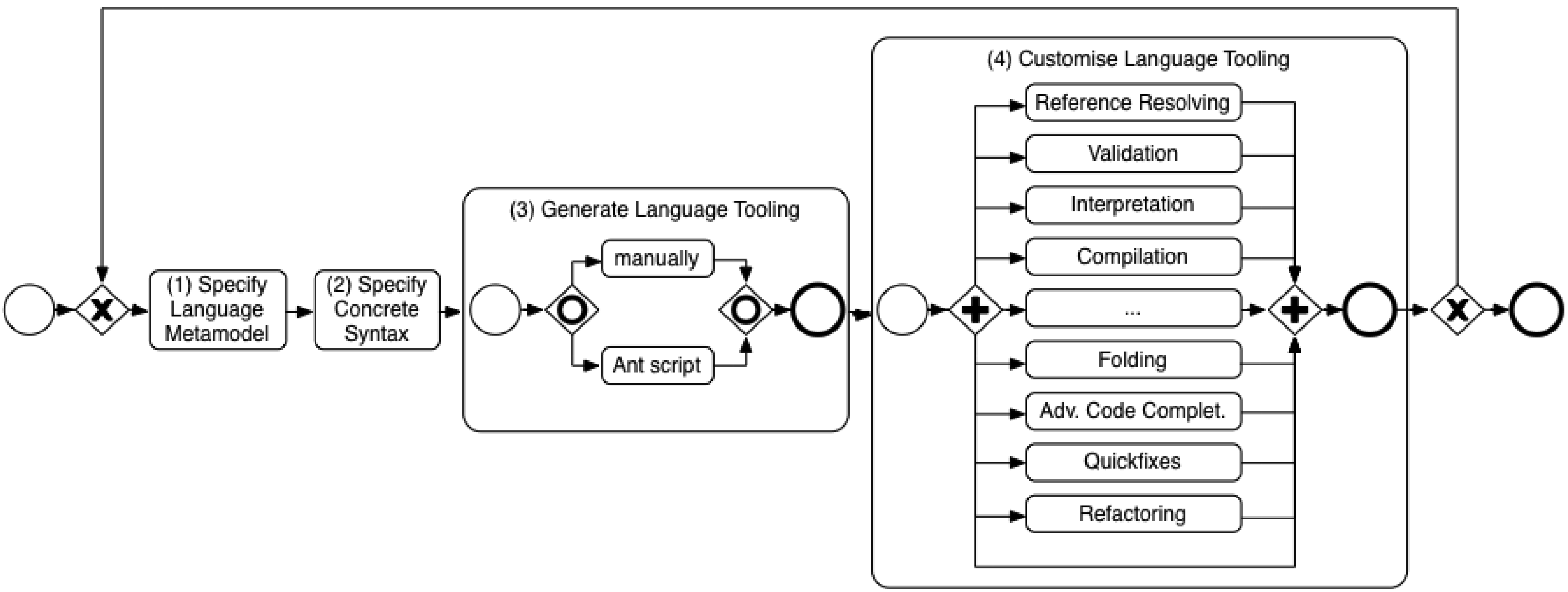
- Generated GPL code is hard to read, debug and maintain

Using the DSL – Interpretation vs. Compilation

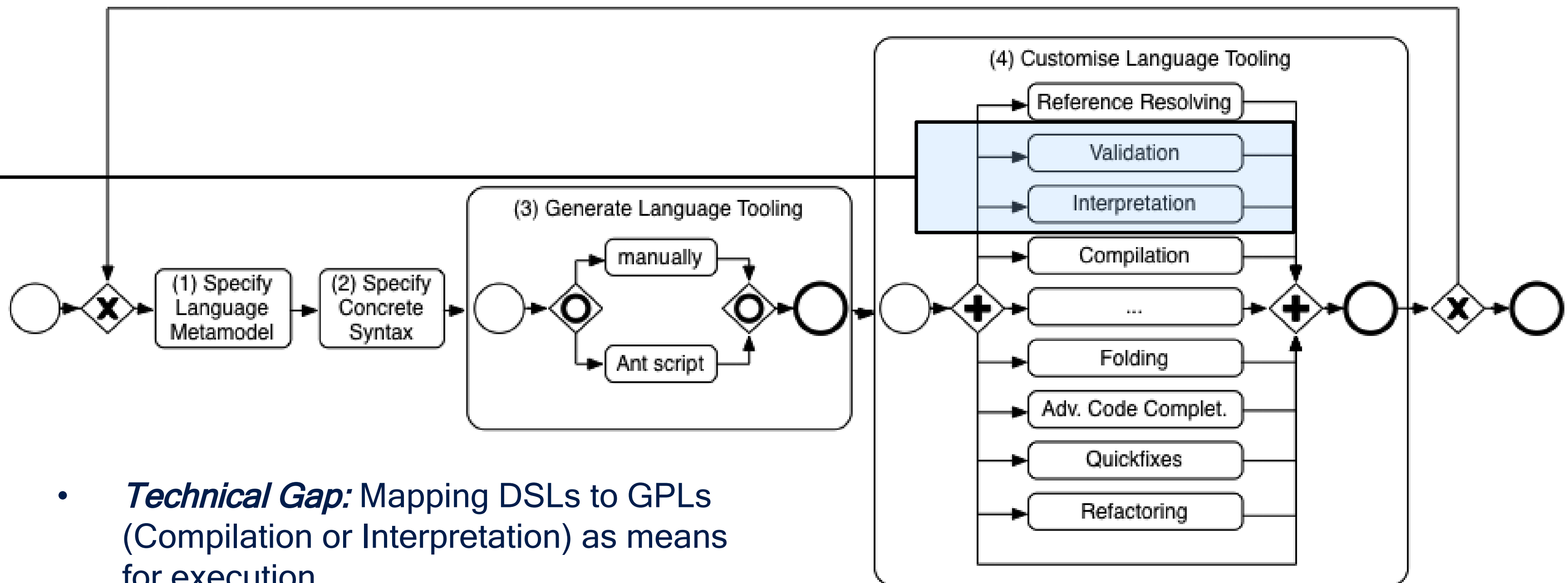
- Create an interpreter/compiler in Java
  - Initially easy, but hard to maintain
  -
- Use a model transformation
  - ATL, Epsilon, ...
-
- Use a template engine
  - DSL documents are the parameter (models)

40.4. Integrating DSLs and GPLs

Integrating DSLs and GPLs



Integrating DSLs and GPLs



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

40.4.1 Integrating DSLs and GPLs - Approach

(1) Use EMFText to *lift* GPLs to the technical space of DSLs

(2) Language integration by metamodel and grammar inheritance

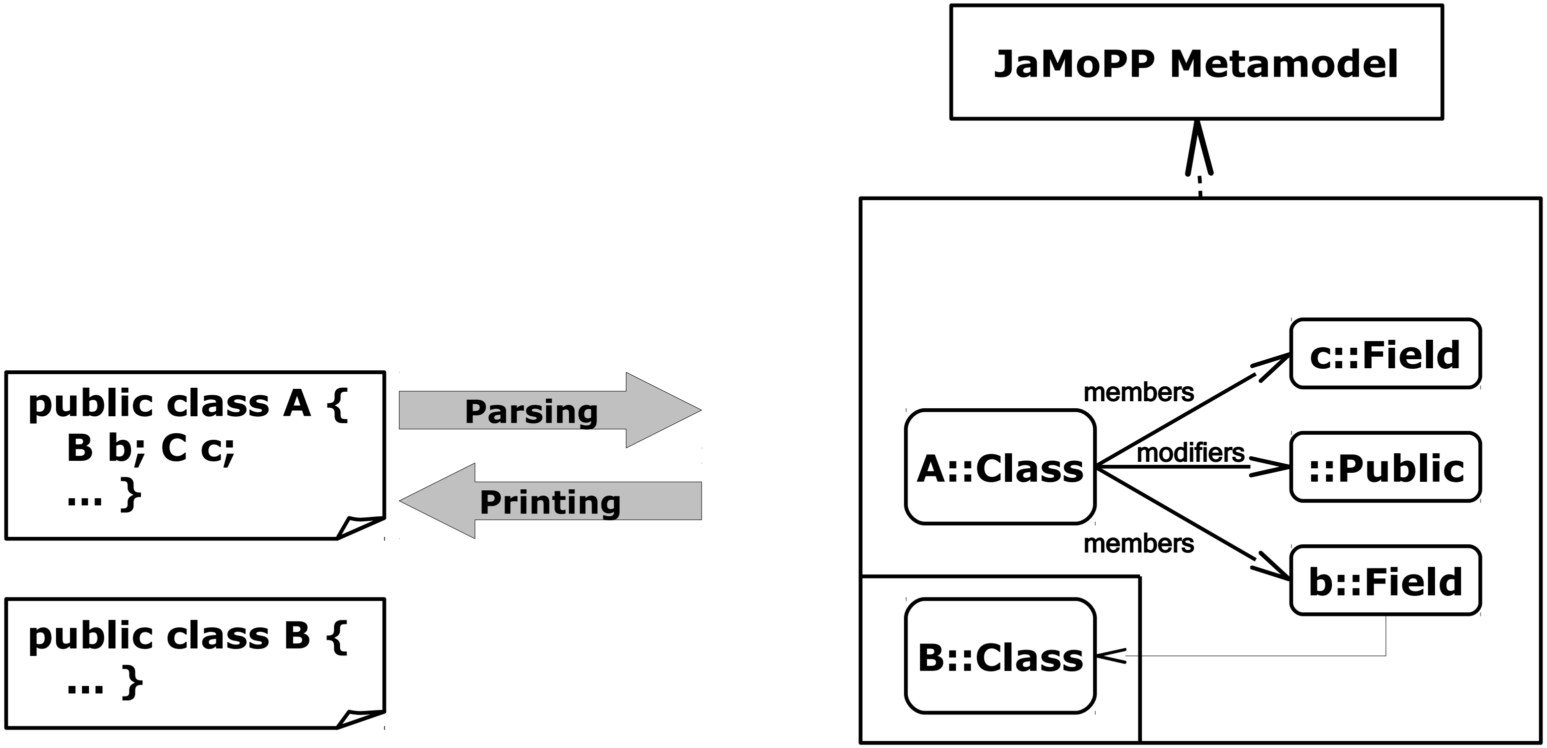
JaMoPP: Lifting Java to TS of DSLs

- Ingredients:
- Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)

JaMoPP Metamodel

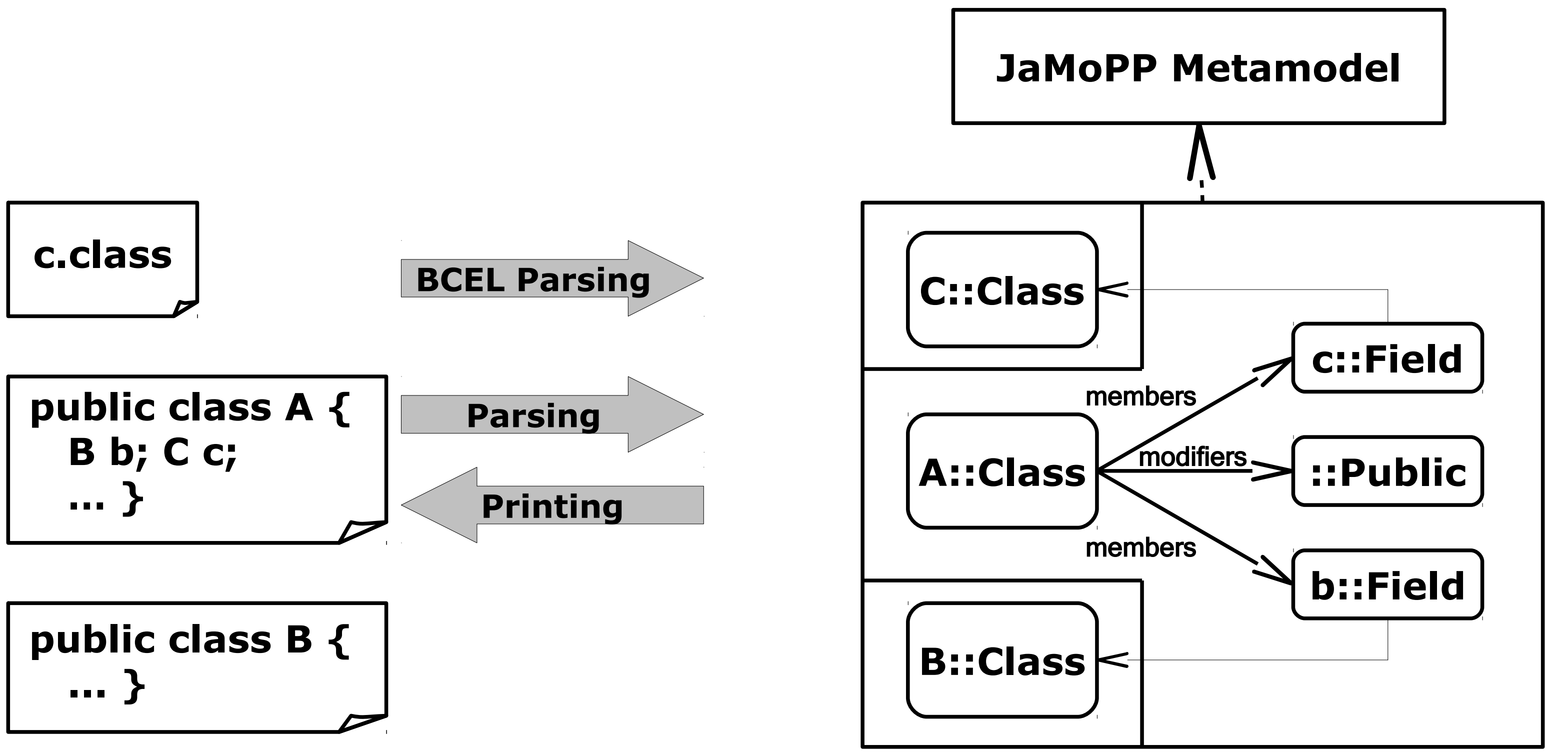
JaMoPP: Lifting Java to TS of DSLs

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



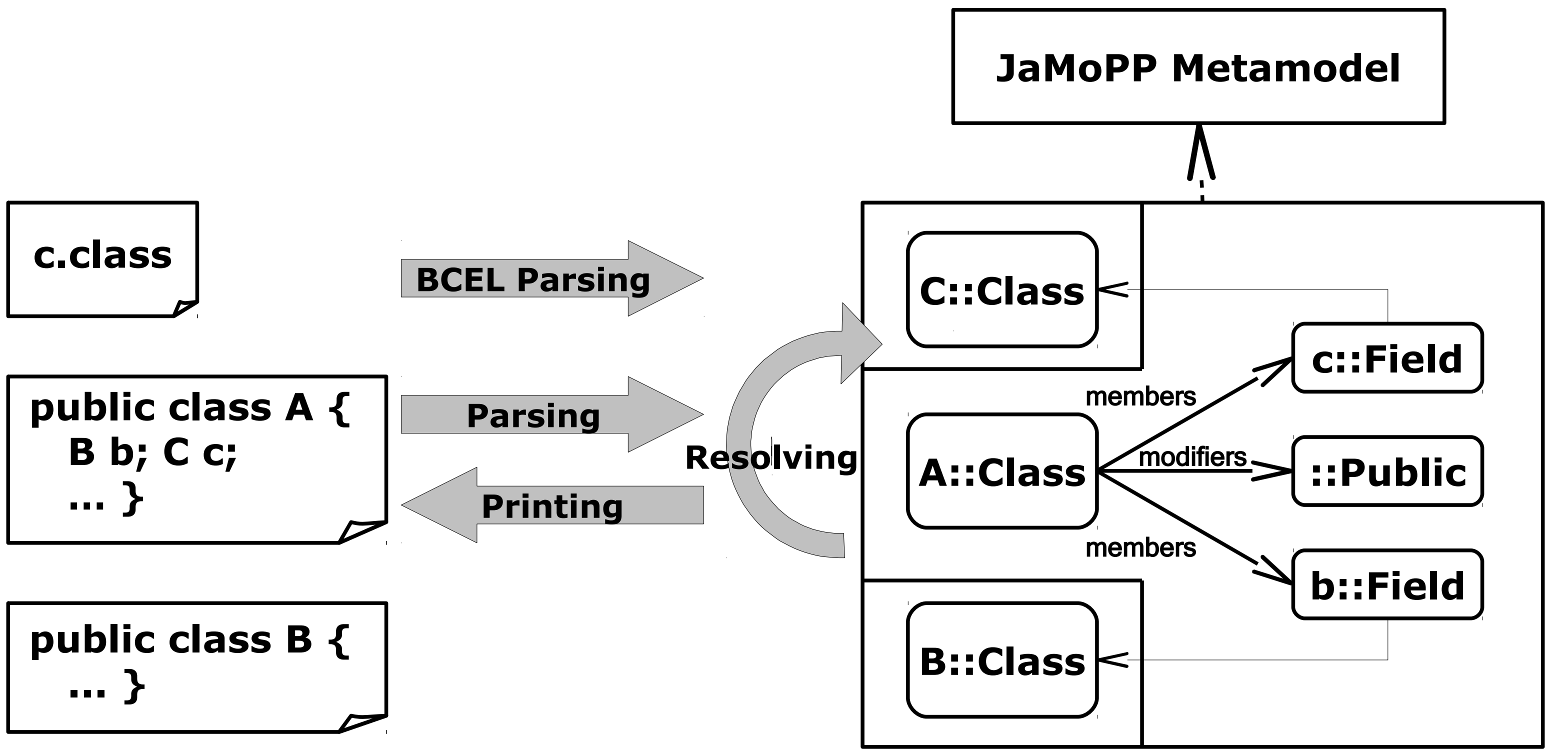
JaMoPP: Lifting Java to TS of DSLs

- 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 TS of DSLs

- 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
- Reference Resolvers that implement java-specific scoping (static semantics)



JaMoPP Metamodel

commons
Commentable
NamedElement
NamespaceAwareElement

classifiers
Classifier, ConcreteClassifier
Implementor
Class, AnonymousClass,
Interface, Enumeration,
Annotation

containers
JavaRoot, CompilationUnit,
Package, EmptyModel

instantiations
Initializable, Instantiation
NewConstructorCall
ExplicitConstructorCall

types
Type, TypedElement
TypeReference
ClassifierReference
NamespaceClassifierReference
PrimitiveType
Boolean, Byte, Char, Double
Float, Int, Long, Short, Void

modifiers
Modifier, Modifiable
AnnotationInstanceOrModifier
AnnotableAndModifiable
Abstract, Final, Native
Public, Protected, Private
Static, Strictfp, Synchronized
Transient, Volatile

members
ExceptionThrower, Member
MemberContainer
AdditionalField, Constructor
EmptyMember, Field, Method
EnumConstant

arrays
ArrayTypable, ArrayDimension
ArrayInitializer, ArrayInstantiationBySize
ArrayInstantiationByValues, ArraySelector

statements
ForLoopInitializer
Statement
SwitchCase
Assert, Break, Block
CatchClause, Condition
Continue,
DefaultSwitchCase
DoWhileLoop
EmptyStatement
ExpressionStatement
ForLoop, ForEachLoop
JumpLabel
LocalVariableStatement
NormalSwitchCase
Return, Switch
SynchronizedBlock
Throw, TryBlock
WhileLoop

literals
Literal, Self
BooleanLiteral
CharacterLiteral
FloatLiteral
DoubleLiteral
DecimalFloatLiteral
DecimalDoubleLiteral
HexFloatLiteral
HexDoubleLiteral
IntegerLiteral
LongLiteral
DecimalIntegerLiteral
DecimalLongLiteral
HexIntegerLiteral
HexLongLiteral
OctalIntegerLiteral
OctalLongLiteral
NullLiteral, Super, This

parameters
Parameter,
Parametrizable,
OrdinaryParameter,
VariableLengthParameter

generics
TypeArgument
TypeArgumentable
TypeParametrizable
ExtendsTypeArgument
QualifiedTypeArgument
SuperTypeArgument
TypeParameter
UnknownTypeArgument

references
Reference, ArgumentList
ReferenceableElement
IdentifierReference
ReflectiveClassReference
PrimitiveTypeReference
StringReference
SelfReference

annotations
Annotable
AnnotationElementValuePair
AnnotationInstance
AnnotationMethod
AnnotationValue

variables
Variable
LocalVariable
AdditionalLocalVariable

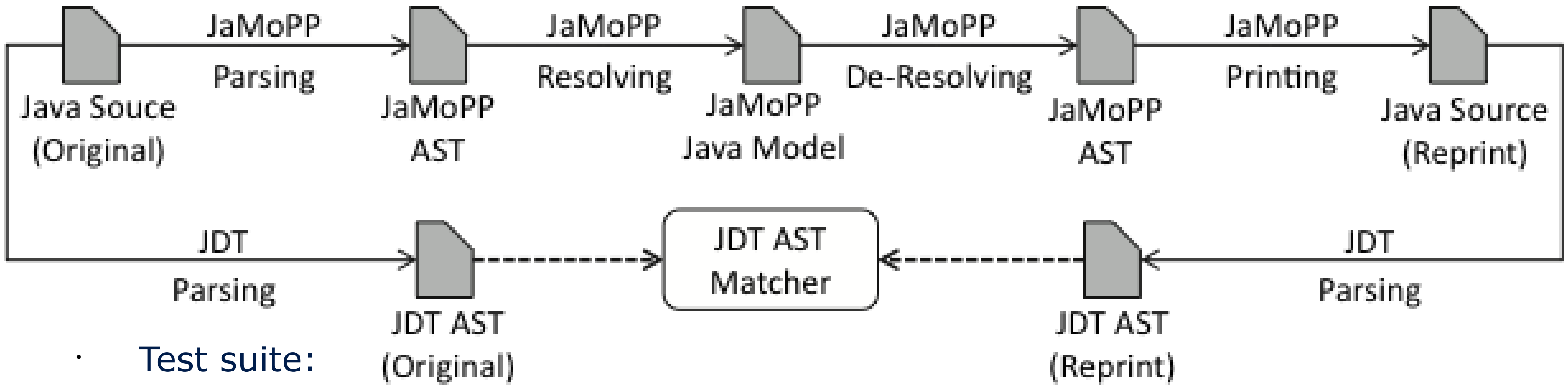
imports
Import, StaticImport
ClassifierImport
PackageImport
StaticClassifierImport
StaticMemberImport

expressions
Expression, ExpressionList, AssignmentExpression, *AssignmentExpressionChild,* ConditionalExpression, *ConditionalExpressionChild,* ConditionalOrExpression, *ConditionalOrExpressionChild,* ConditionalAndExpression, *ConditionalAndExpressionChild,* InclusiveOrExpression, *InclusiveOrExpressionChild,* ExclusiveOrExpression, *ExclusiveOrExpressionChild,* AndExpression, *AndExpressionChild,* EqualityExpression, *EqualityExpressionChild,* InstanceOfExpression, *InstanceOfExpressionChild,* RelationExpression, *RelationExpressionChild,* ShiftExpression, *ShiftExpressionChild,* AdditiveExpression, *AdditiveExpressionChild,* MultiplicativeExpression, *MultiplicativeExpressionChild,* UnaryExpression, *UnaryExpressionChild, UnaryModificationExpression,* PrefixUnaryModificationExpression, SuffixUnaryModificationExpression, *UnaryModificationExpressionChild,* CastExpression, *PrimaryExpression,* NestedExpression

operators
Operator, AdditiveOperator, AssignmentOperator, EqualityOperator, MultiplicativeOperator, RelationOperator, ShiftOperator, UnaryOperator, UnaryModificationOperator
Assignment, AssignmentAnd, AssignmentDivision, AssignmentExclusiveOr, AssignmentMinus, AssignmentModulo, AssignmentMultiplication, AssignmentLeftShift
AssignmentOr, AssignmentPlus, AssignmentRightShift, AssignmentUnsignedRightShift, Equal, NotEqual, GreaterThan, GreaterThanOrEqual, LessThan, LessThanOrEqual
Addition, Subtraction, Division, Multiplication, Remainder, Complement, MinusMinus, Negate, PlusPlus, LeftShift, RightShift, UnsignedRightShift

JaMoPP Testing

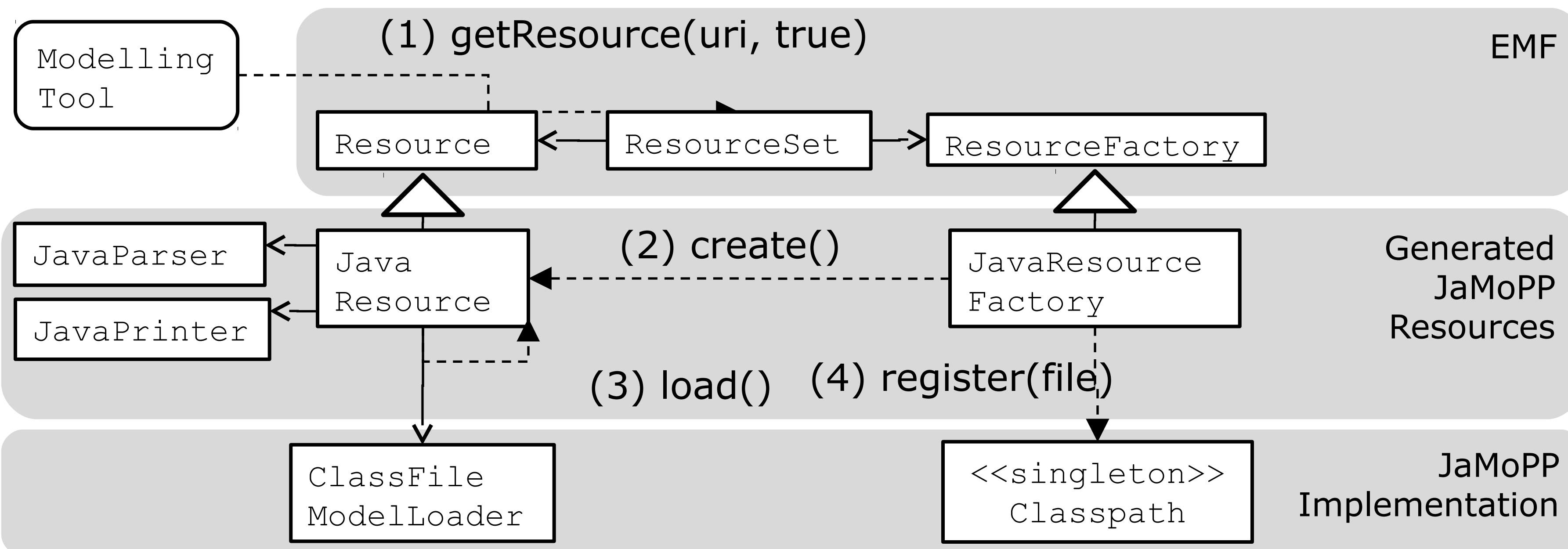
- Parsing public class A is easy, but parsing Java 5 is not (Unicode, Generics, Annotations and lots of weird things allowed by the JLS)
- We wanted JaMoPP to be complete



- 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

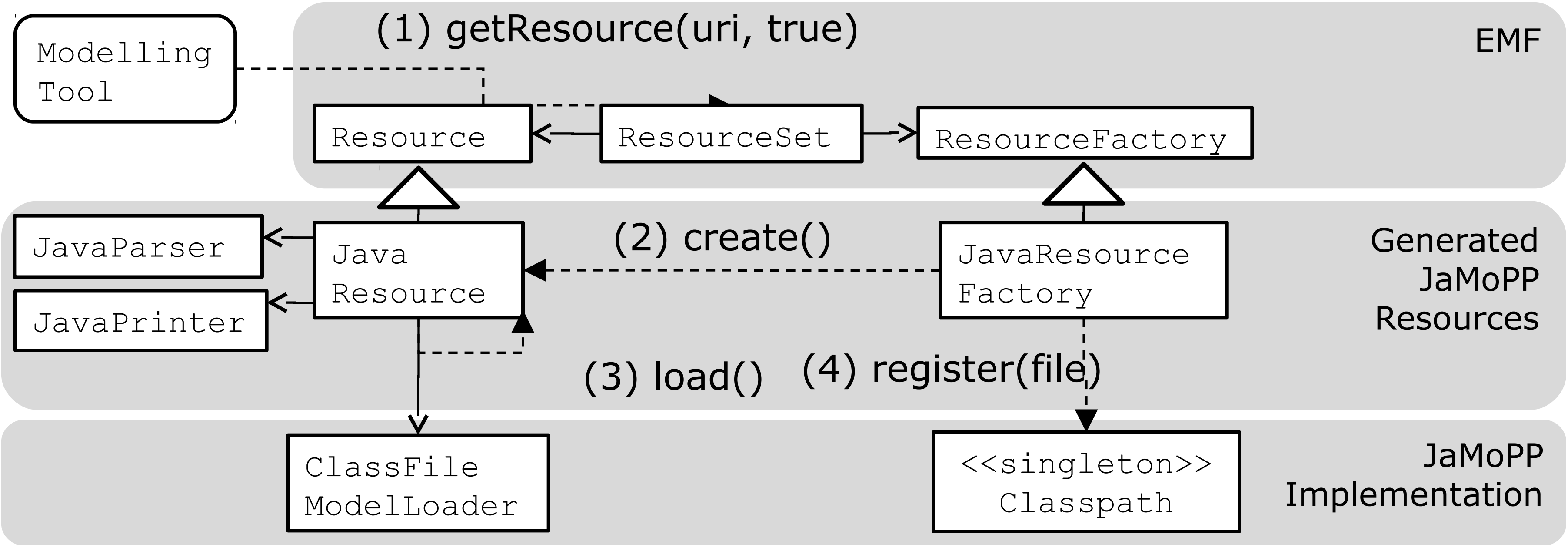
JaMoPP Tool Integration

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



JaMoPP Tool Integration

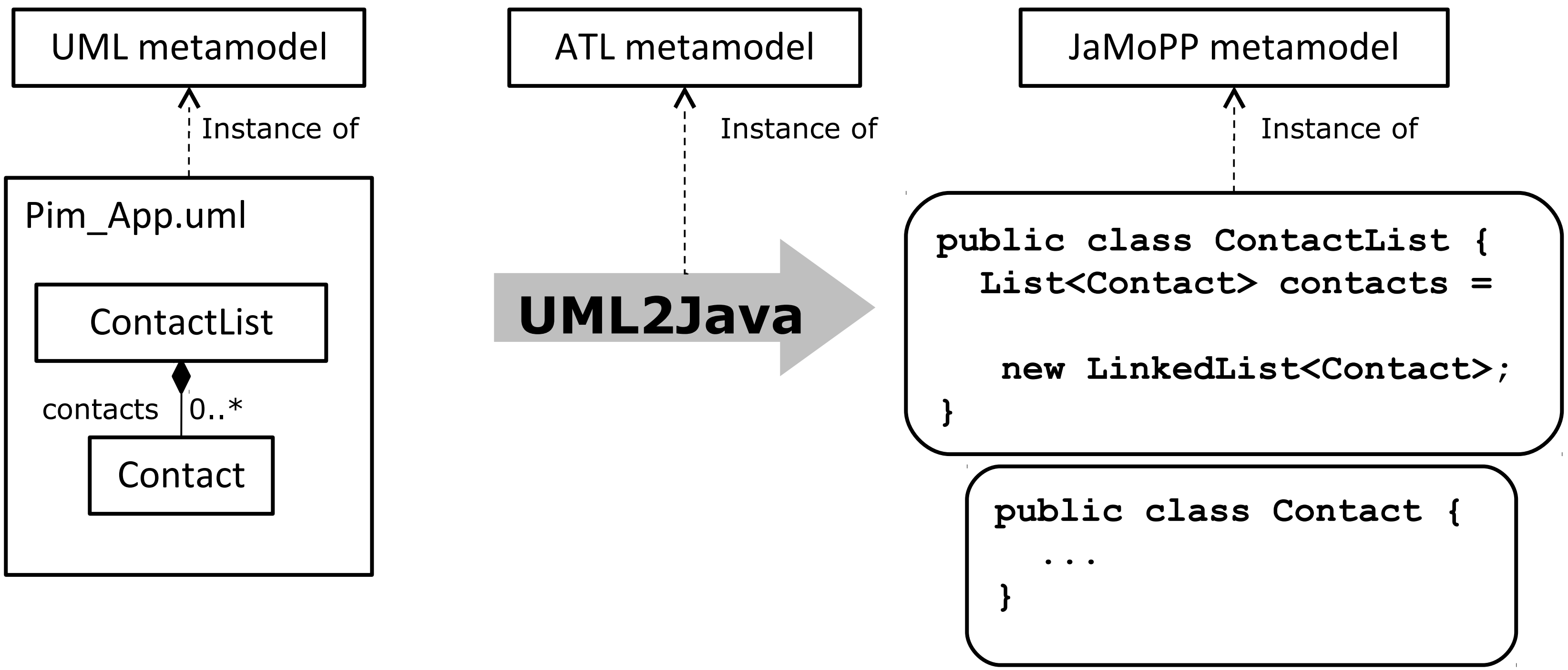
- JaMoPP seamlessly and transparently integrates with arbitrary EMF-based Tools
- Parsing Java files to models and Printing Java Files is simple




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

JaMoPP Application: Code Generation (ATL)

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



JaMoPP Application: Code Generation (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
   )
 }
```

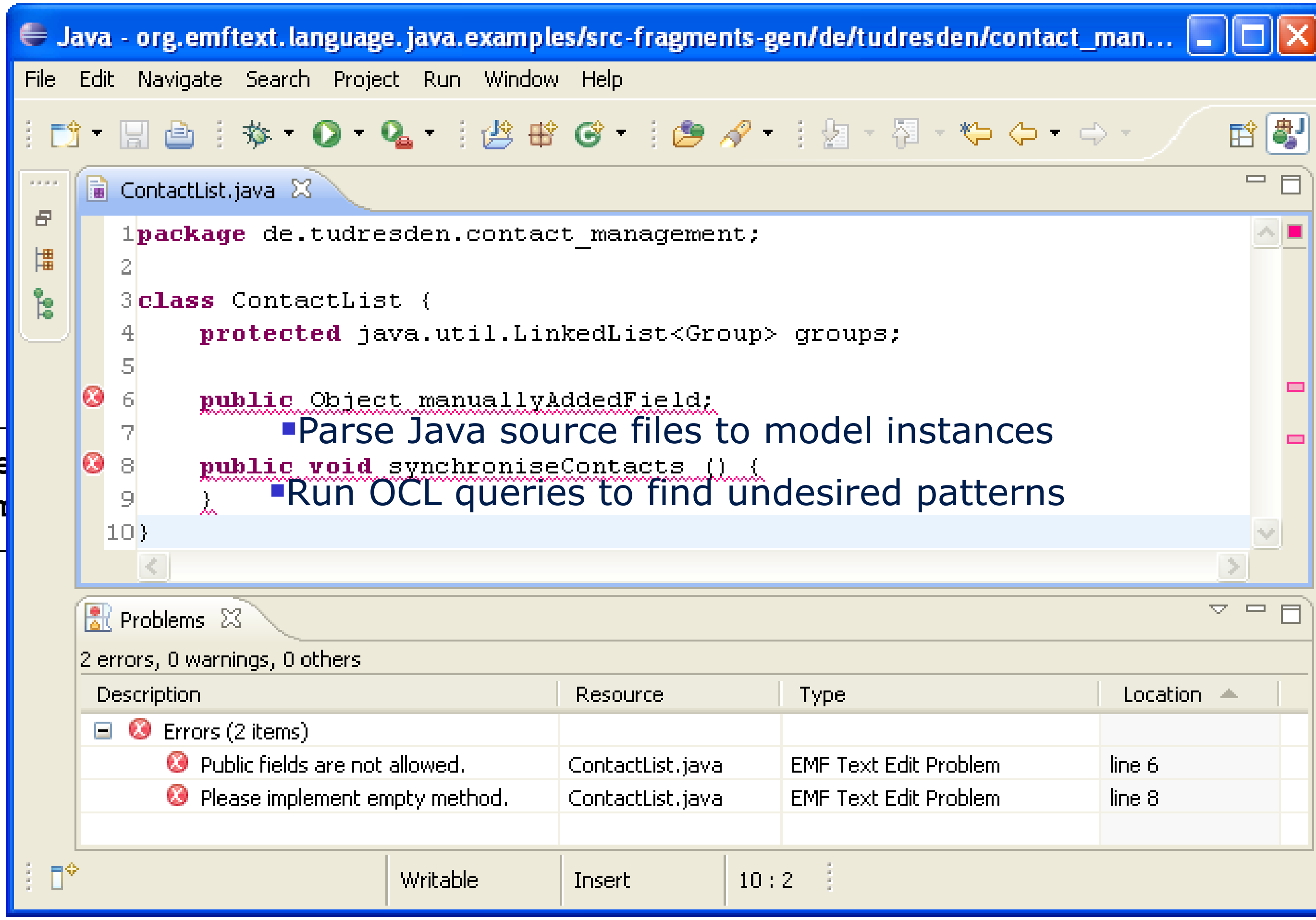
JaMoPP Application: Code Analysis (OCL)

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

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

JaMoPP Application: Code Analysis (OCL)

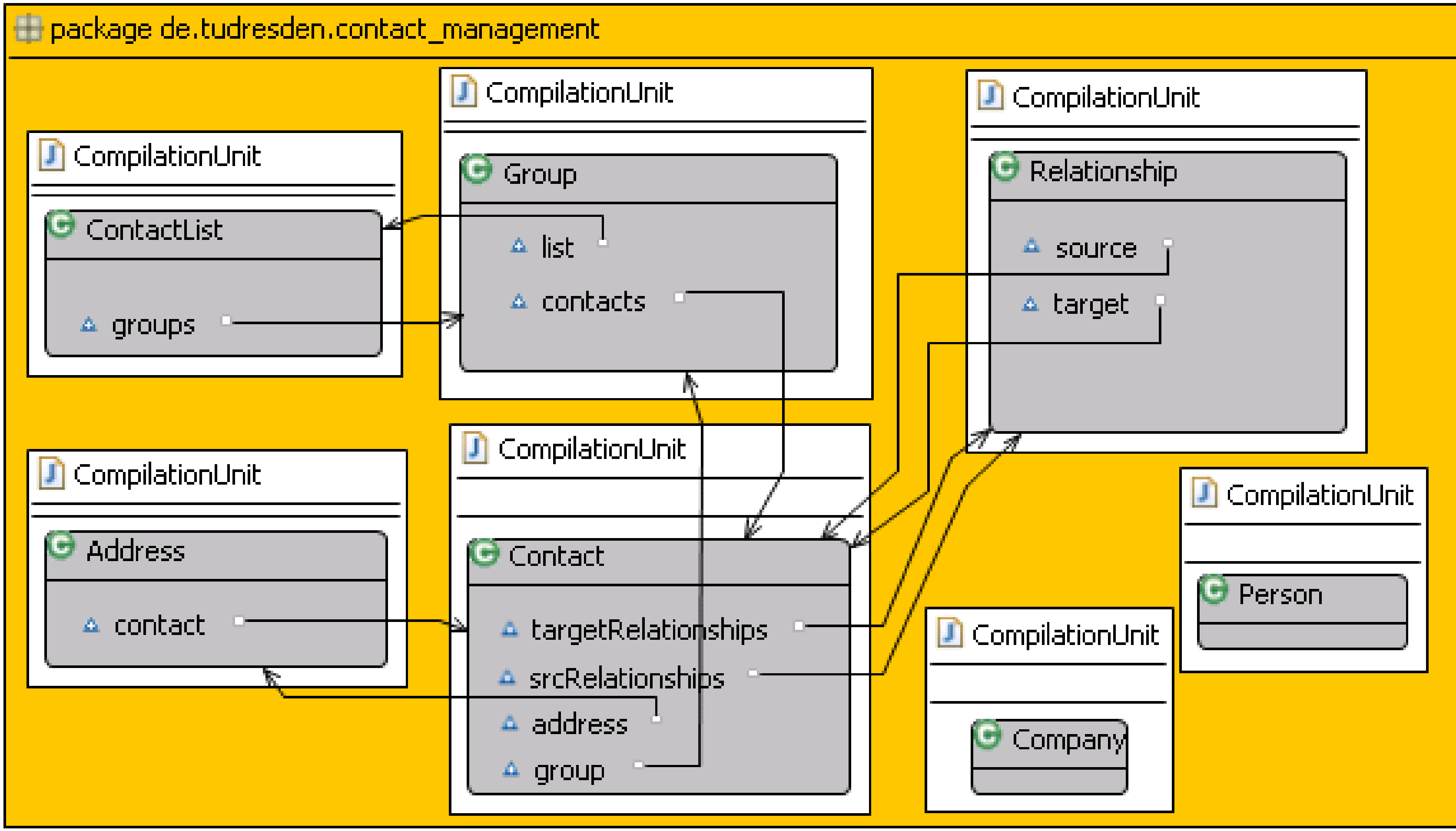
```
context me
   self->m                                         = 0
```



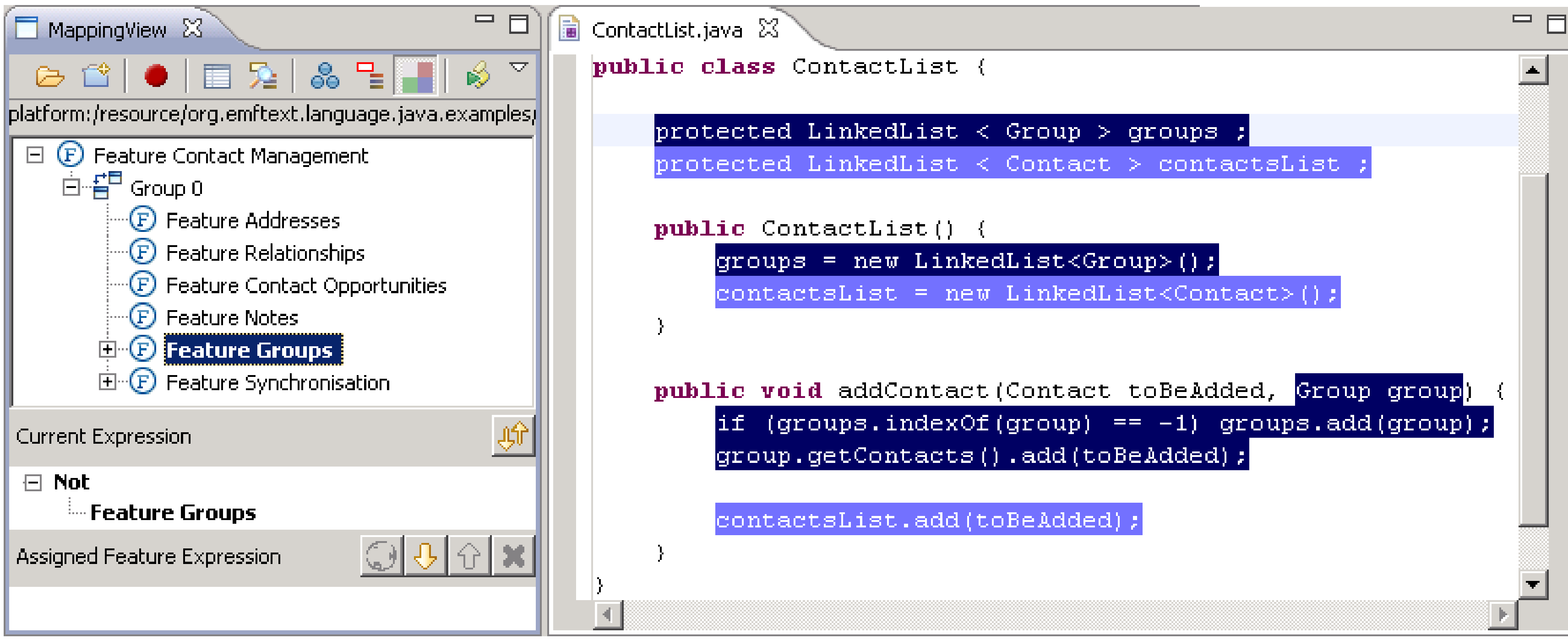
- Parse Java source files to model instances
- Run OCL queries to find undesired patterns

JaMoPP Application: Code Visualization (GMF)

- Create .gmfgraph, gmftool, and gmfmap model
- Generate Graphical Editor for Java



JaMoPP Application: Software Product Line Engineering (FeatureMapper)



JaMoPP Applications: What else?

- Typesafe Template Languages
  - Same syntax as string-based templates
- 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?)
- Model-based compilation to byte code
- …

40.4.2 Integrating DSLs and GPLs

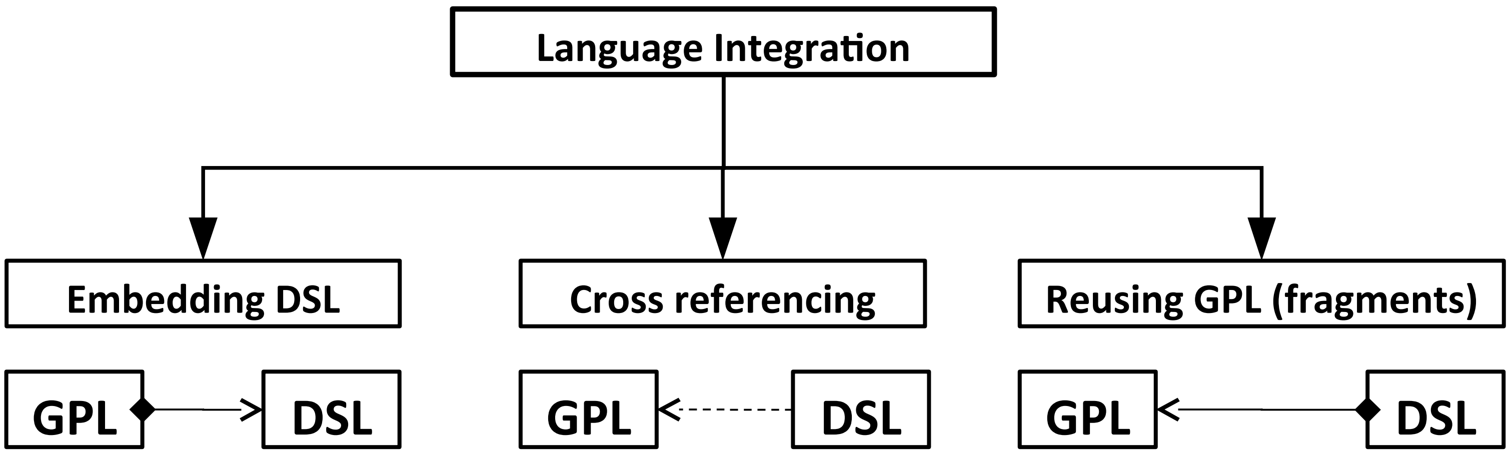
Approach

(1) Use EMFText to *lift* GPLs to the technical space of DSLs

(2) Language integration by metamodel and grammar inheritance

Integrating DSLs and GPLs

- Different integration scenarios



Language Integration Examples

- FormsExtension

- FormsEmbedded

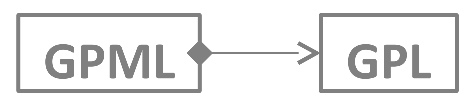
- JavaForms

- 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



40.5. 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)

Thank you!

Questions?

http://www.emftext.org