



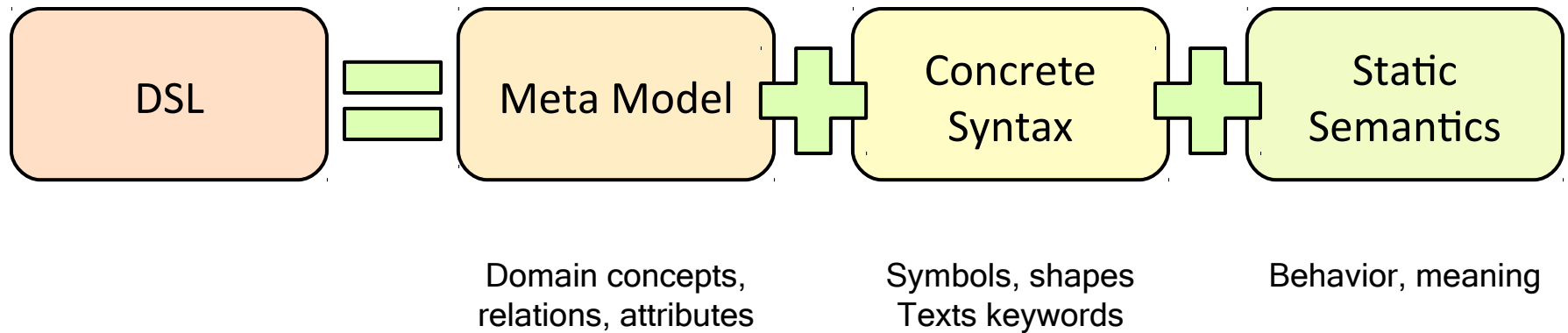
# 24) Exchange Syntax and Textual DSLs using EMFText

Florian Heidenreich, Jendrik Johannes,  
Sven Karol, Mirko Seifert, Christian  
Wende, Uwe Aßmann  
Version 11-0.2, 11/22/11

1. Introduction
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
  1. Mapping text to data types
  2. Reference resolving
  3. Syntax modules (Import and Reuse)
  4. Interpretation vs. Compilation
4. Integrating DSLs and GPLs
5. Other DSL examples
6. Conclusion

- <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.

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



# Productivity Gains with DSL

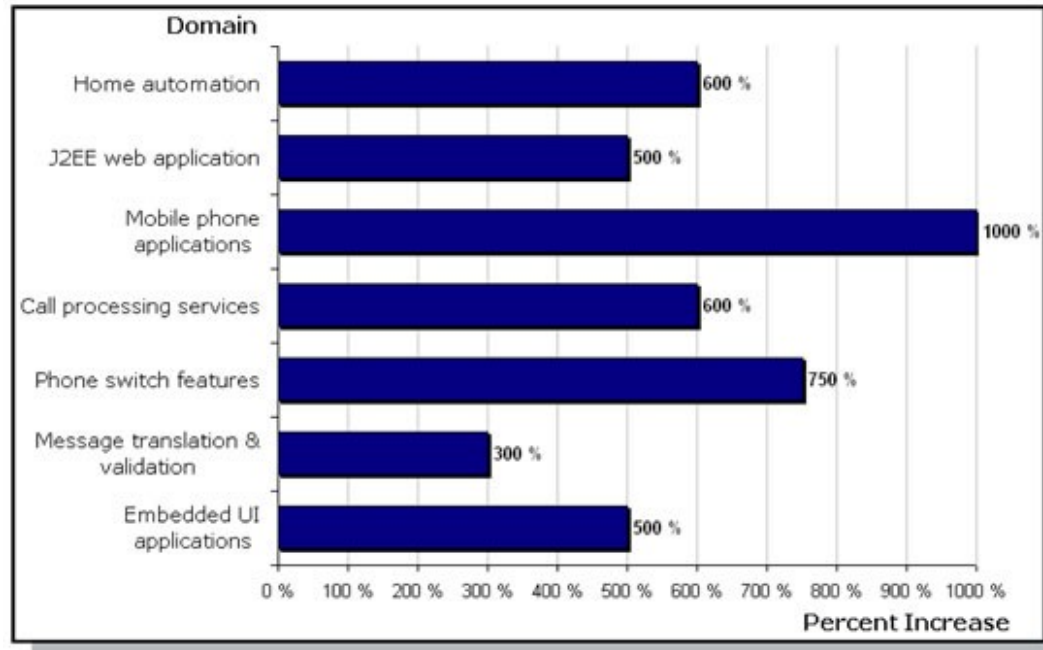
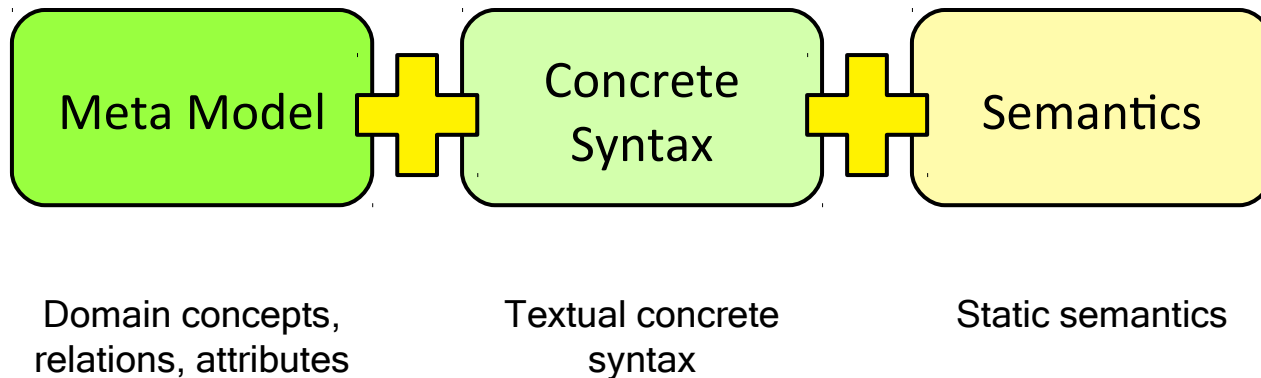


Figure 3: Measured productivity improvements in various domains

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

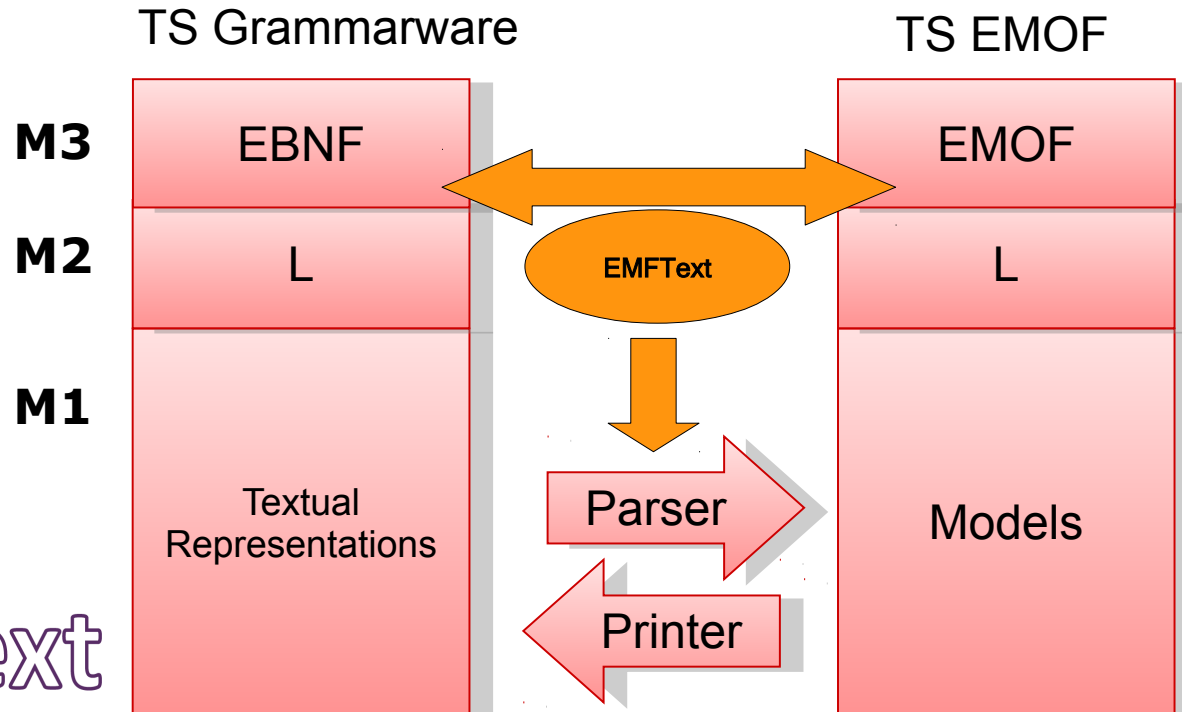
# What is a Textual Domain-Specific Language (DSL)?

- 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



- + 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/>



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

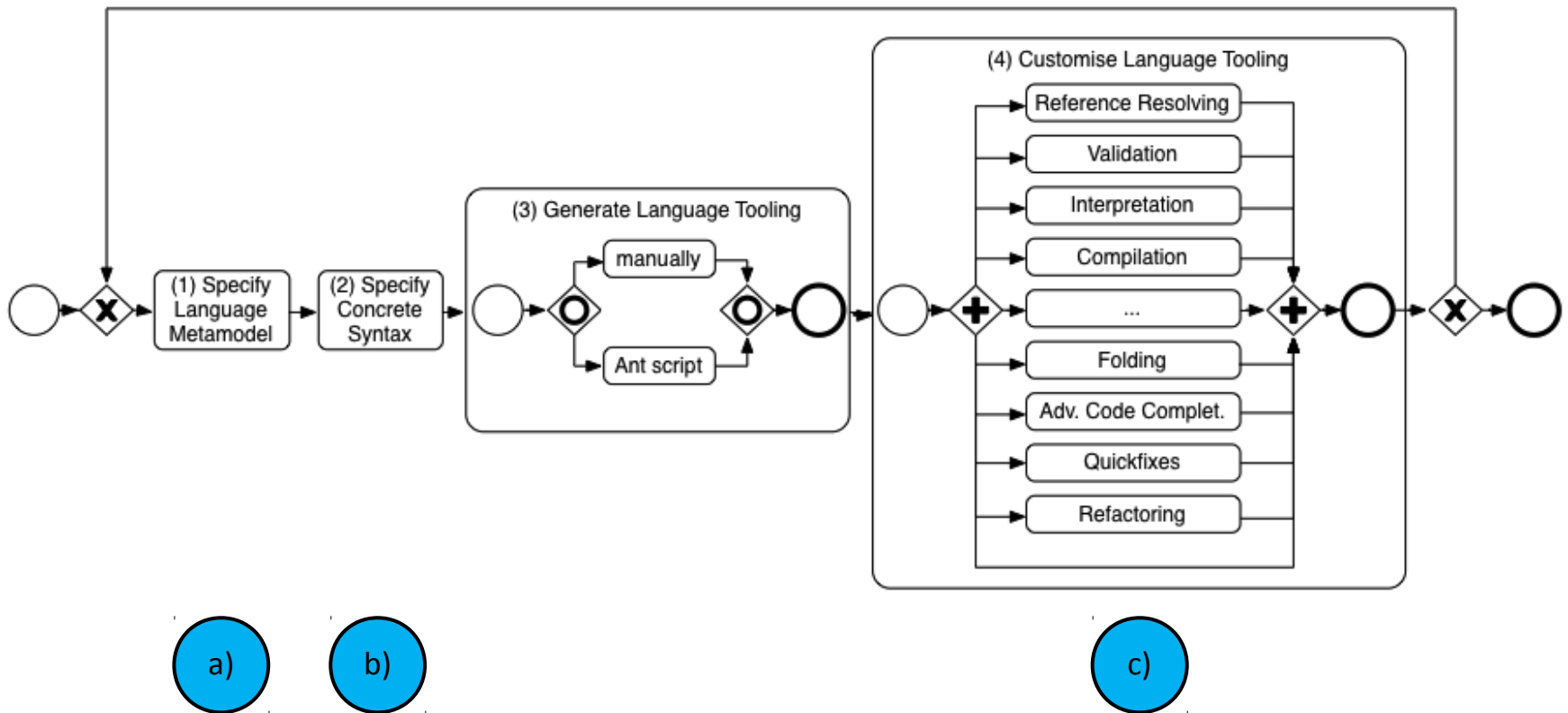
Design principles:

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

Syntax definition should be

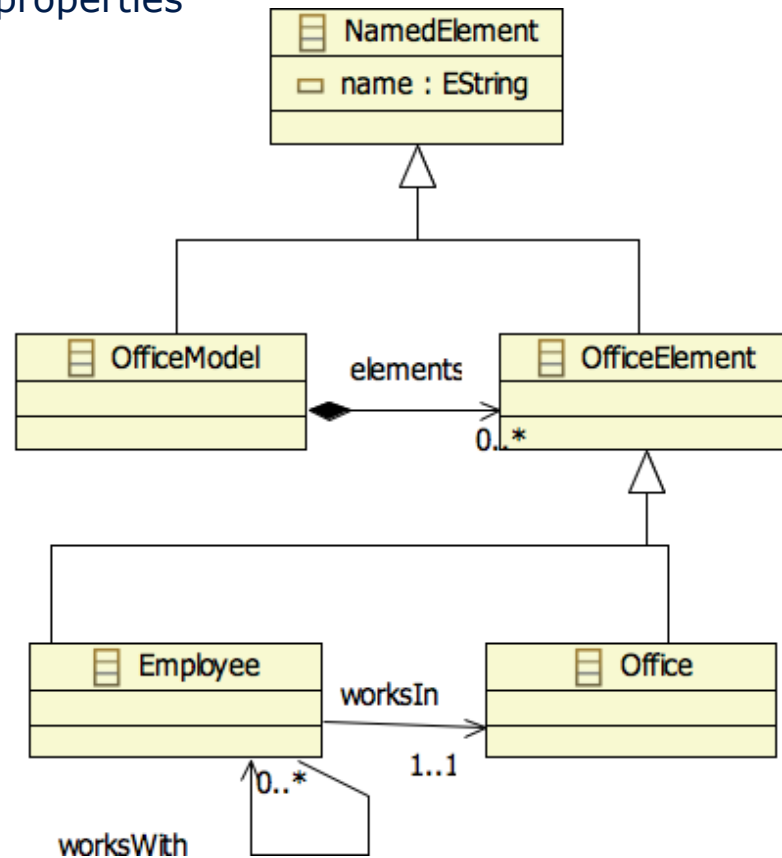
- Simple and easy for small DSLs
- Yet powerful for complex languages

- **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



Creating a new meta model:

- Define concepts, relations and properties in an Ecore model

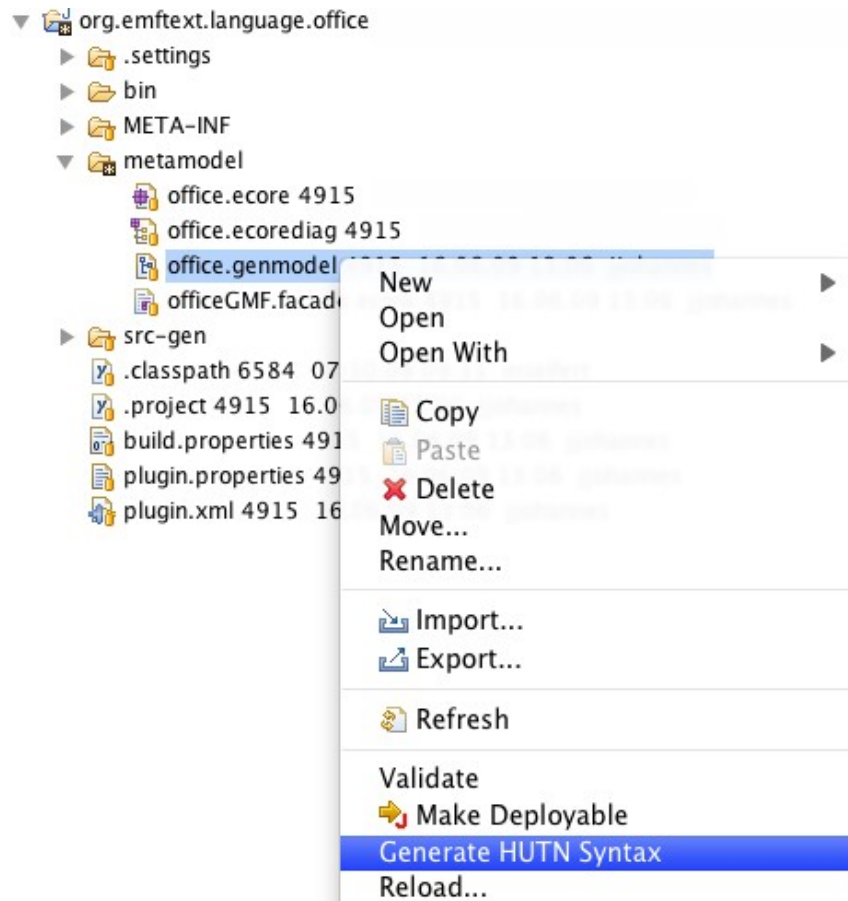
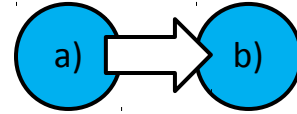


Existing meta models can be imported (e.g., UML, Ecore, ...)

Meta model elements:

- Classes
- Data Types
- Enumerations
  
- Attributes
- References (Containment, Non-containment)
- Cardinalities
  
- Inheritance

# Generate initial syntax (Human Usable Text Notation)



```
office.cs x
|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['"', '"] )* "}" ;
}
```



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

  elements :
    Office {
      name : "INF2080"
    }

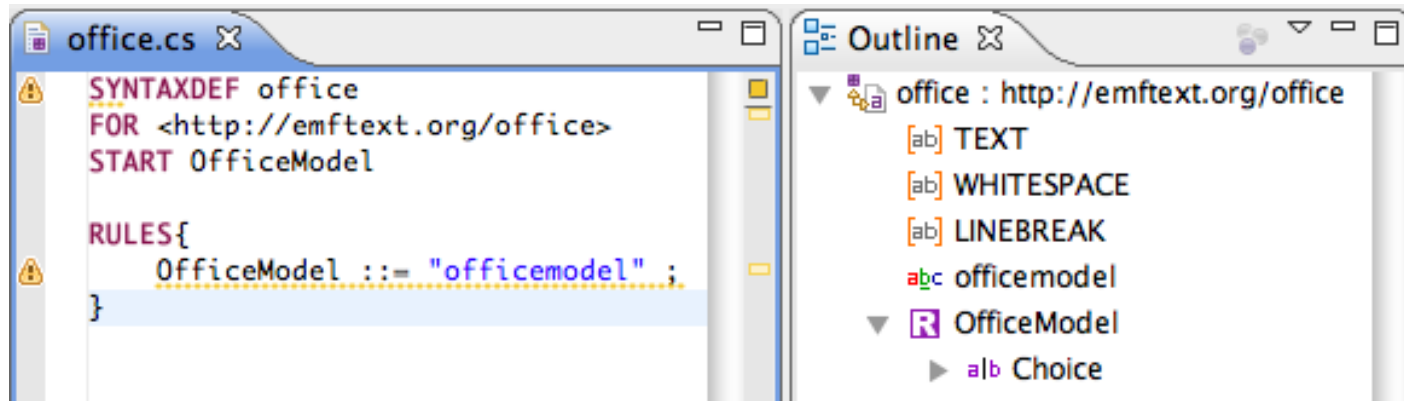
  elements :
    Office {
      name : "INF2084"
    }

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

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

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



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

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

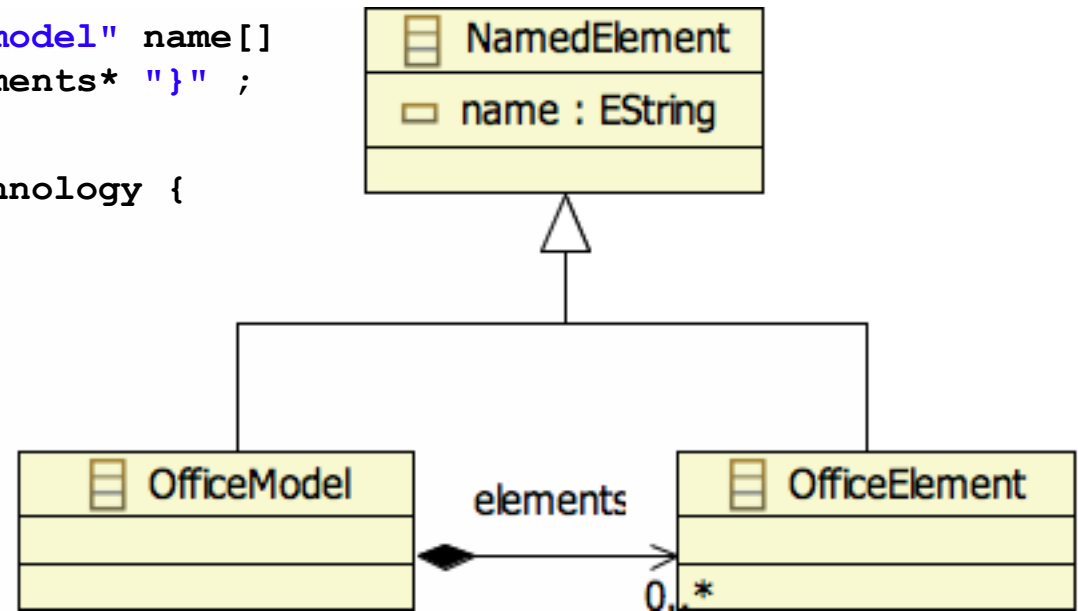
Outline view structure:

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

- One per meta class
  - Syntax: `MetaClassName ::= Syntax Definition ;`
- Definition elements:
  - Static strings (keywords)      `"public"`
  - Choices                              `a|b`
  - Multiplicities                        `+,*`
  - Compounds                            `(ab)`
  - Terminals                             `a[]`            (Non-containment references, attributes)
  - Non-terminals                        `a`              (Containment references)

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

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

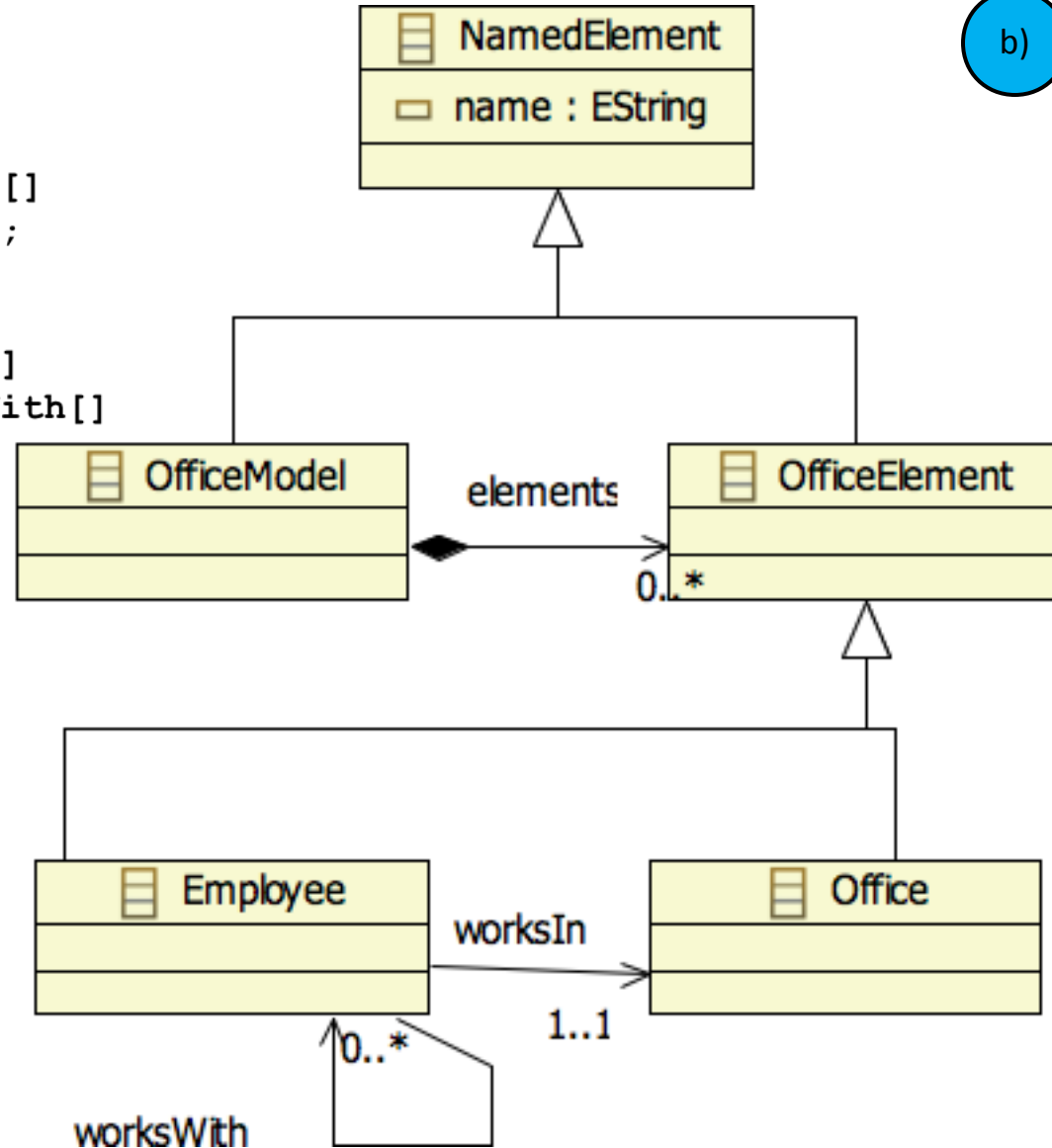


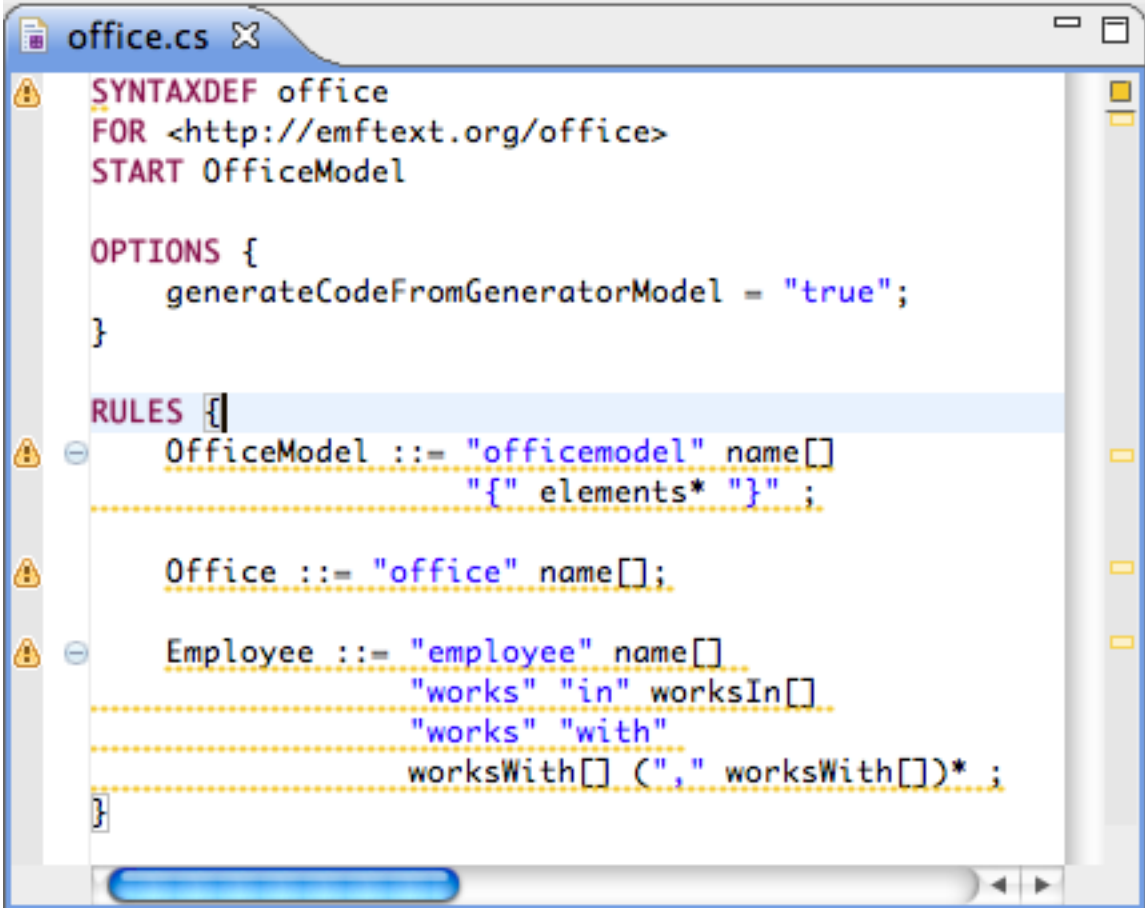
```
OfficeModel ::= "officemodel" name[]
              "{" elements* "}" ;
```

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

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

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





The screenshot shows a code editor window titled 'office.cs'. The code defines a syntax for an office model. It includes a SYNTAXDEF block, an OPTIONS block, and a RULES block. The RULES block defines three rules: OfficeModel, Office, and Employee. The OfficeModel rule defines a sequence of 'officemodel' followed by a list of 'elements'. The Office rule defines 'office' followed by a list of 'name'. The Employee rule defines 'employee' followed by a list of 'name', 'works in', 'works with', and 'works with' (repeated). The code is color-coded and has a light blue highlight on the 'RULES {' line.

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

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
}
```

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";
}
```

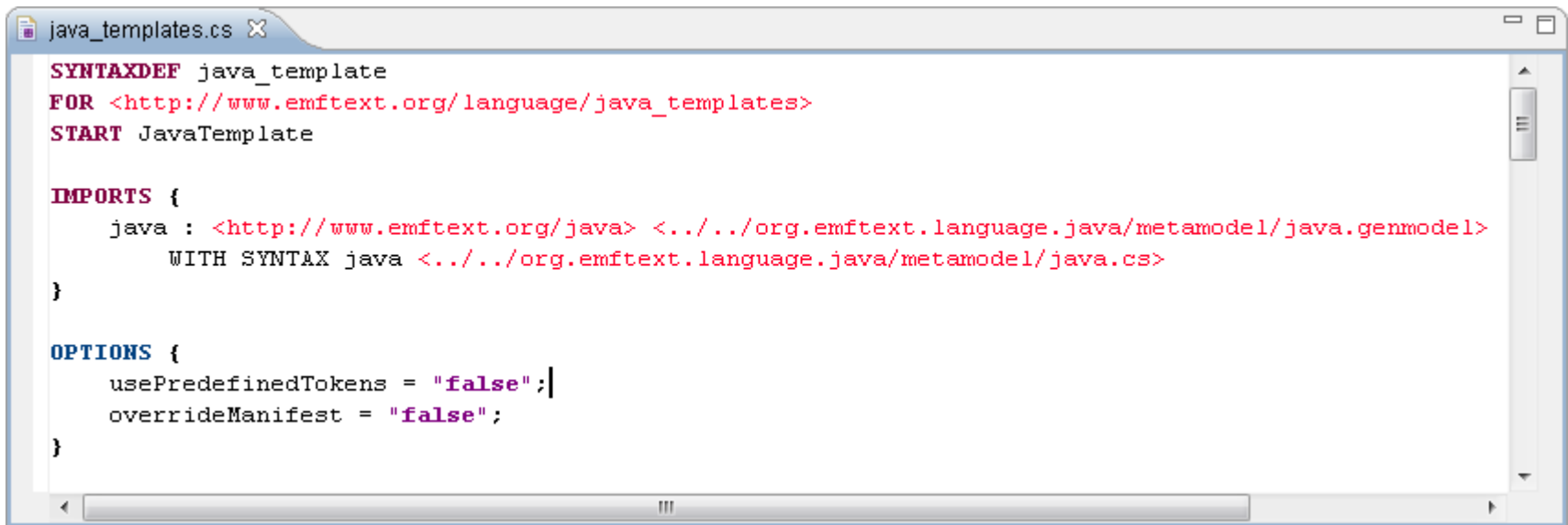


Well, quite similar to attribute mappings:

- Solution A: Default resolving  
Searches 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 resolving  
Change the generated resolver class  
(implements `IReferenceResolver<ContainerType, ReferenceType>`)

For examples see the resolvers for the Java language

- 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 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";
}
```

# 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

1. Interpretation  
Traverse the DSL document and perform appropriate actions
2. Compilation  
Translate the DSL constructs to another (possibly executable) language

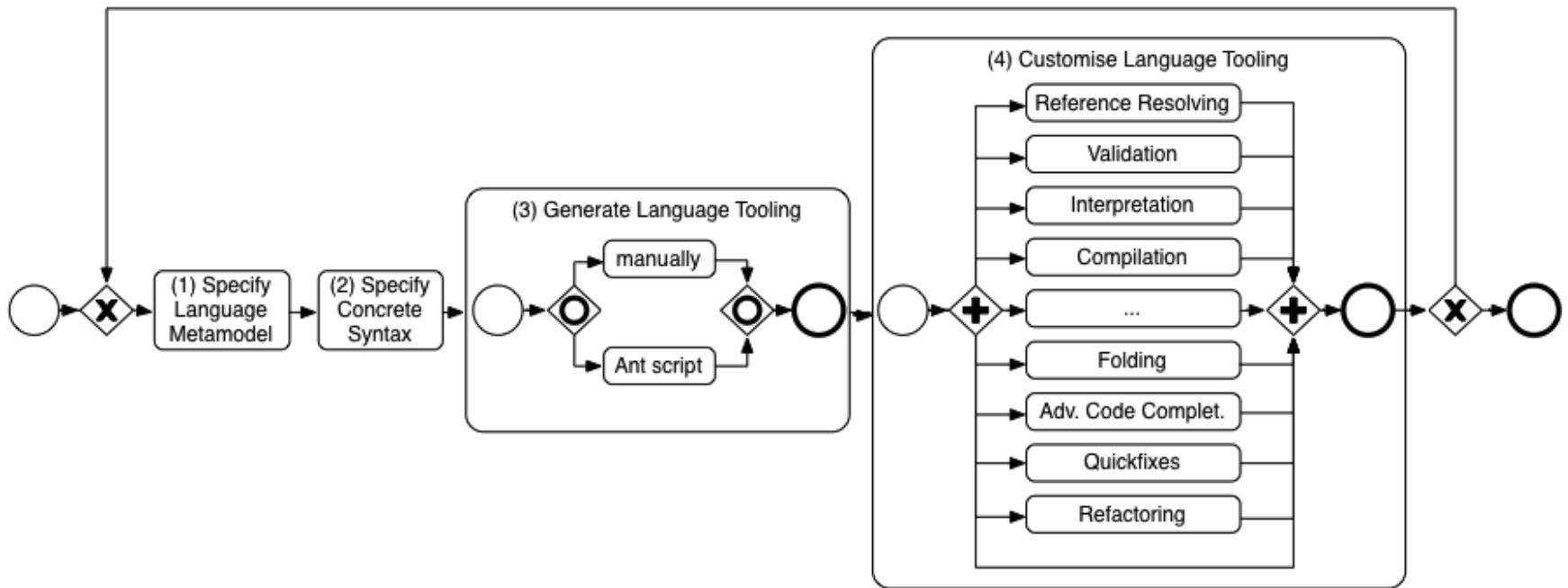
(In principle compilation is an interpretation where the appropriate action is to emit code of the target language)

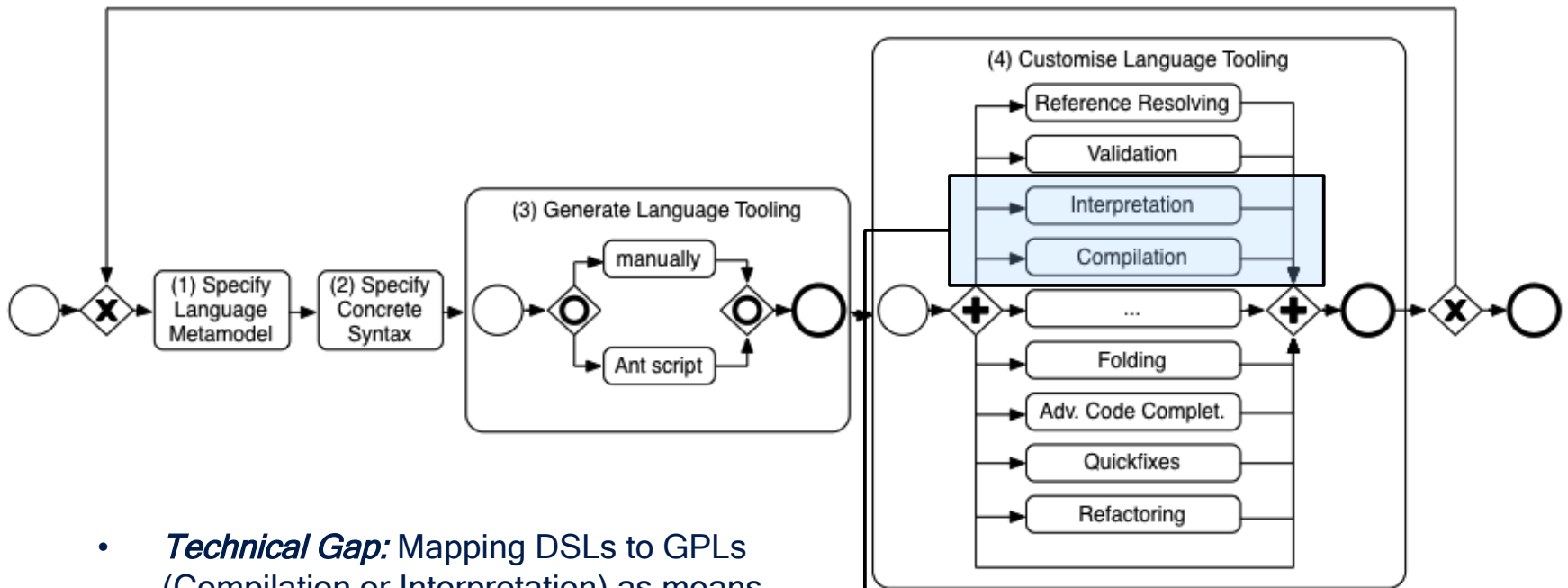
- Developers are required to use different tool machinery for DSLs and GPLs.
- Explicit references between DSL and GPL code are not supported. Their relations are, thus, hard to track and may become inconsistent
- DSLs can not reuse (parts of) the expressiveness of GPLs
- Naive embeddings of DSL code (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)

# Integrating DSLs and GPLs





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

## Approach

- (1) Use EMFText to *lift* GPLs to the technical space of DSLs
- (2) Language integration by metamodel and grammar inheritance



## Approach

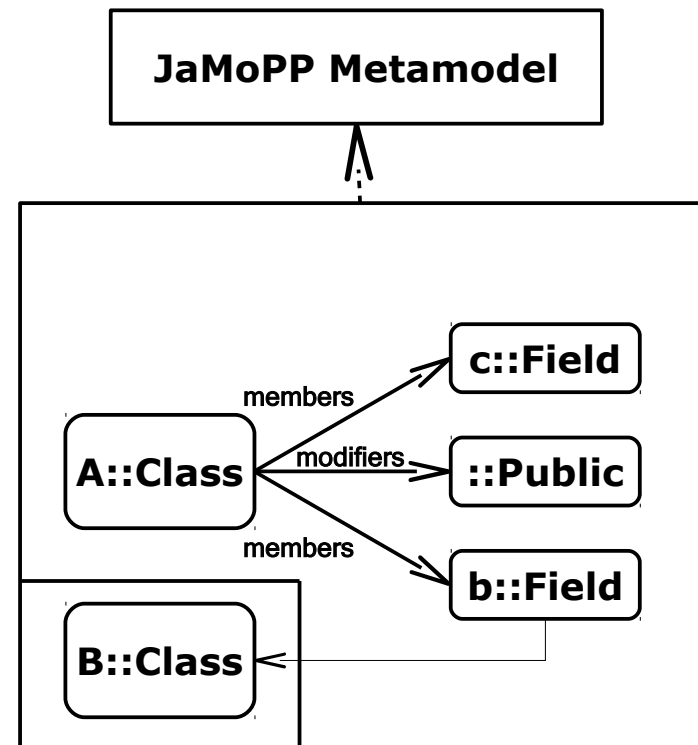
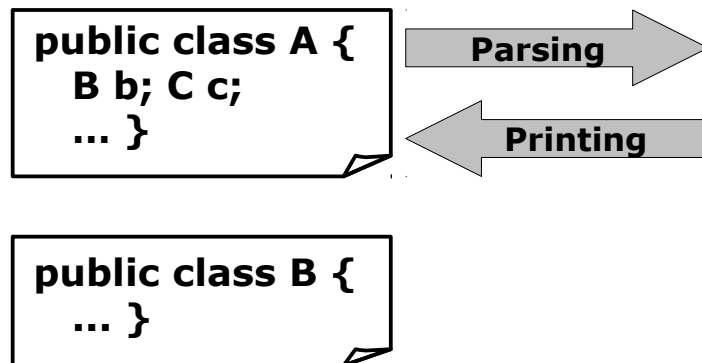
- (1) Use EMFText to *lift* GPLs to the technical space of DSLs**
- (2) Language integration by metamodel and grammar inheritance

- Ingredients:

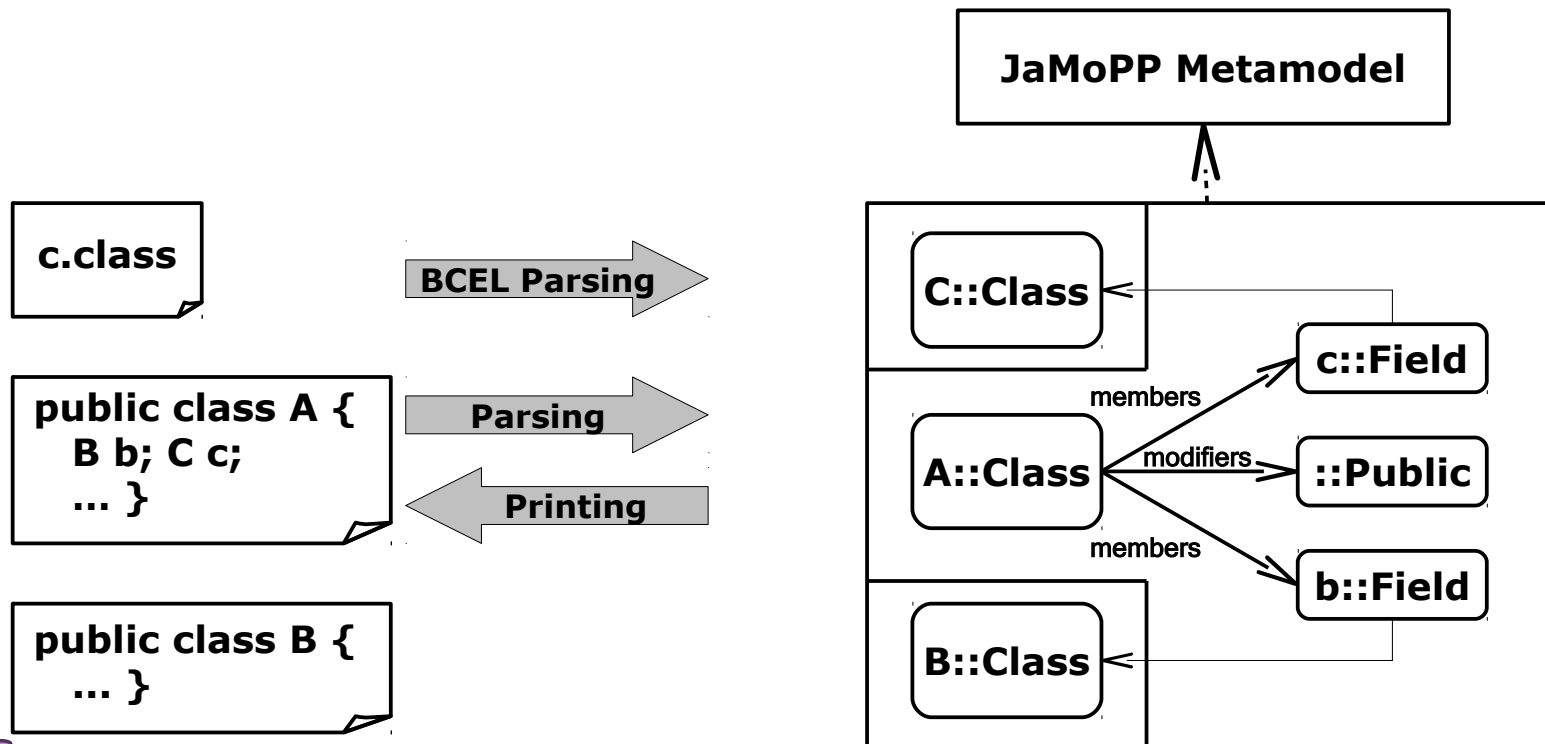
- Ingredients:
  - Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)

**JaMoPP Metamodel**

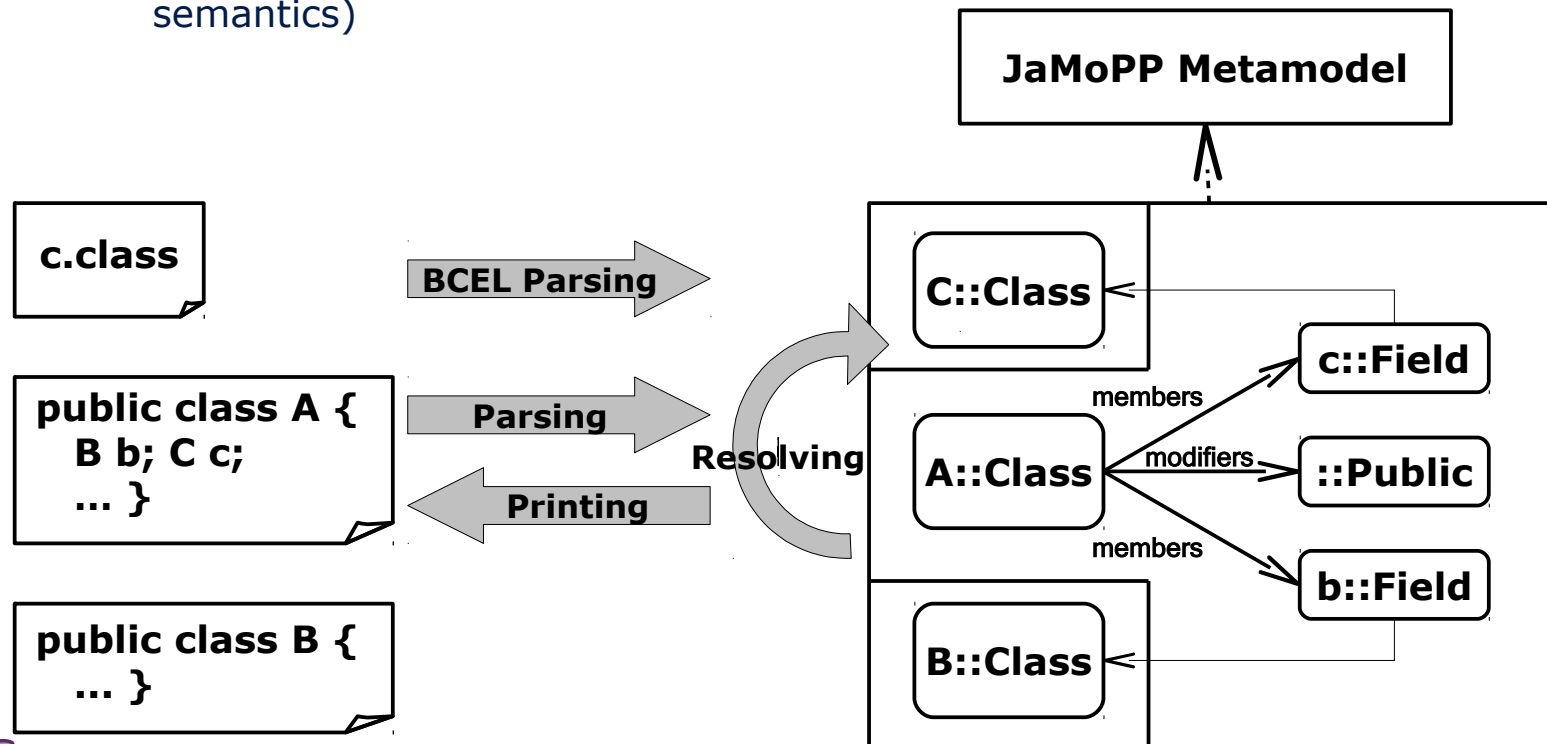
- Ingredients:
  - Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)
  - EMFText .cs definition for each concrete class

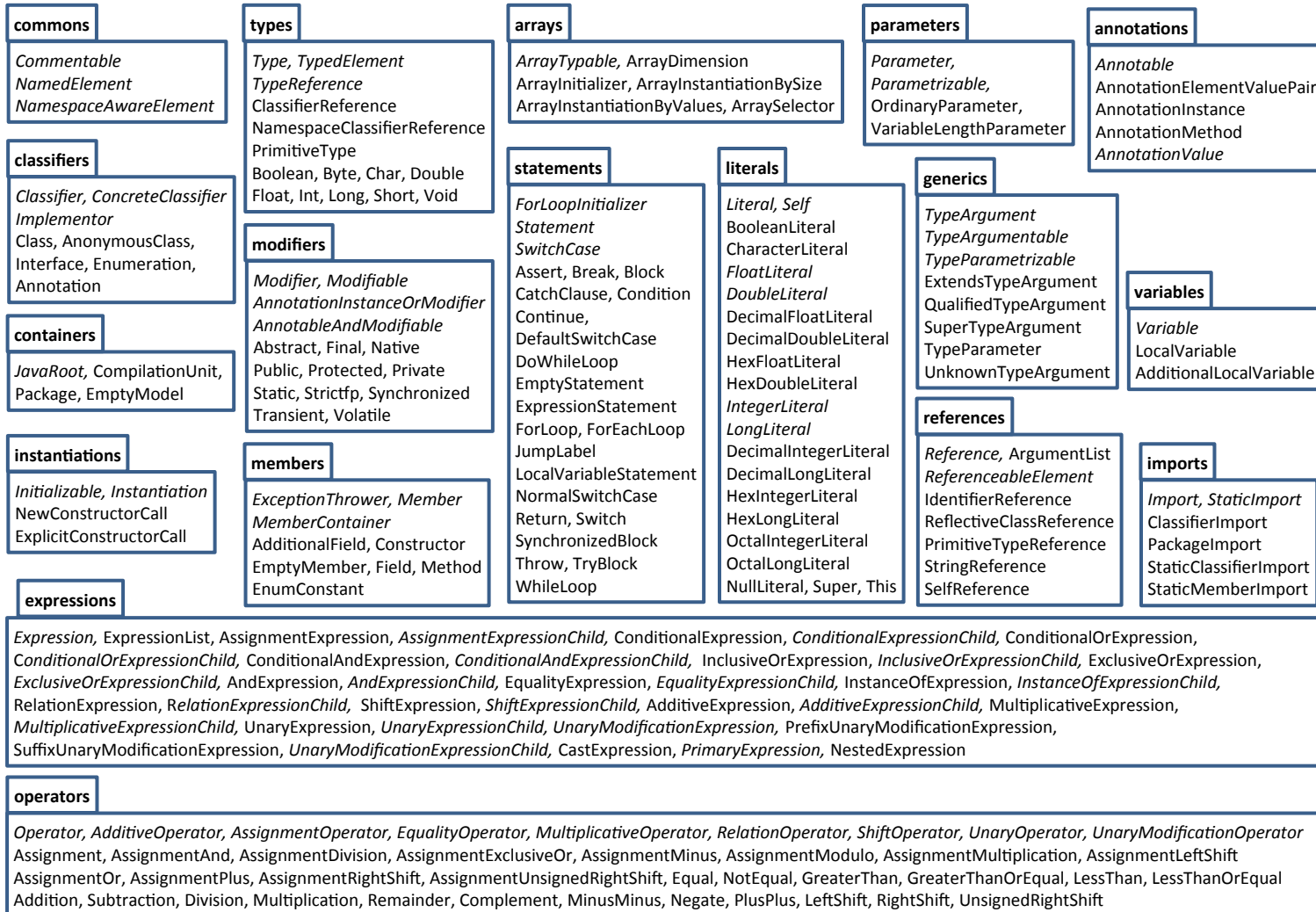


- Ingredients:
  - Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)
  - EMFText .cs definition for each concrete class
  - BCEL Bytecode-Parser – to handle third-party libraries

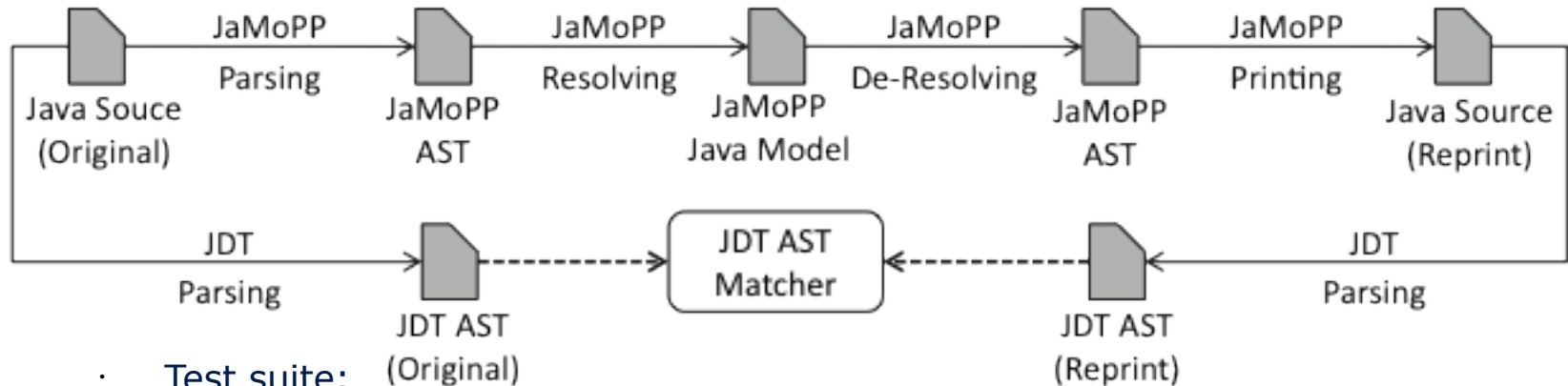


- Ingredients:
  - Ecore Metamodel for Java 5 (153 concrete, 80 abstract classes)
  - EMFText .cs definition for each concrete class
  - BCEL Bytecode-Parser – to handle third-party libraries
  - Reference Resolvers that implement java-specific scoping (static semantics)





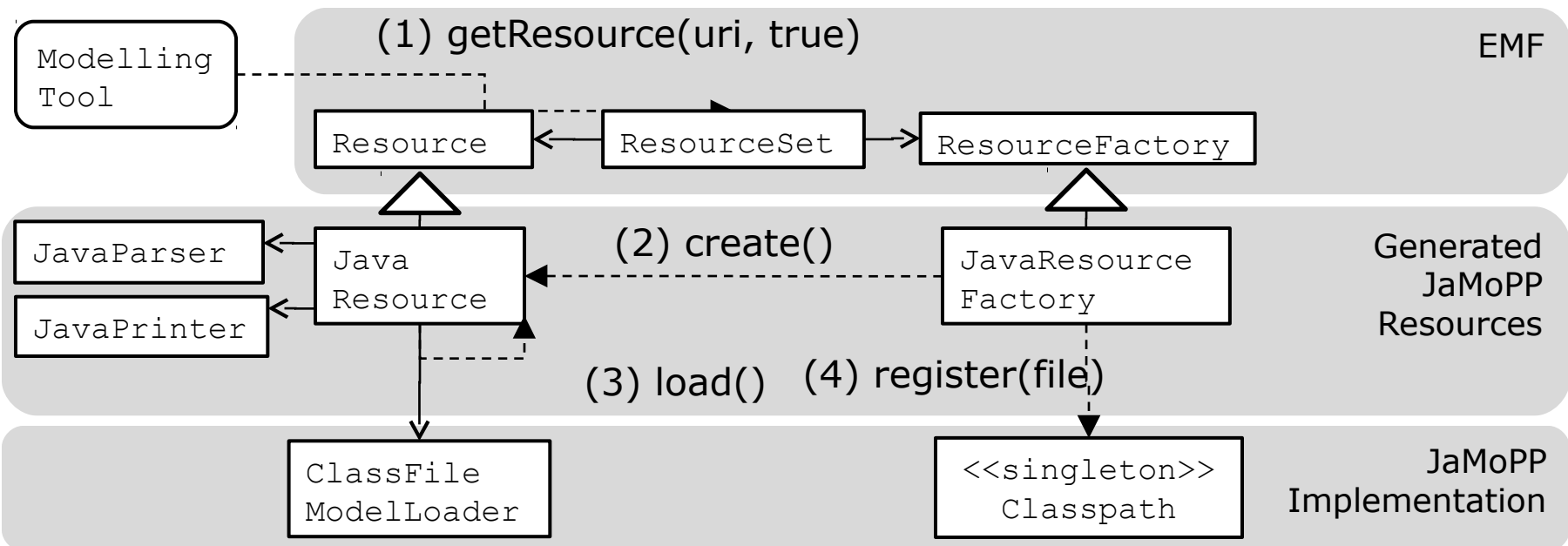
- Parsing public class A is easy, but parsing Java 5 is not (Unicode, Generics, Annotations and lots of weird things allowed by the JLS)
- We wanted JaMoPP to be complete



- 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

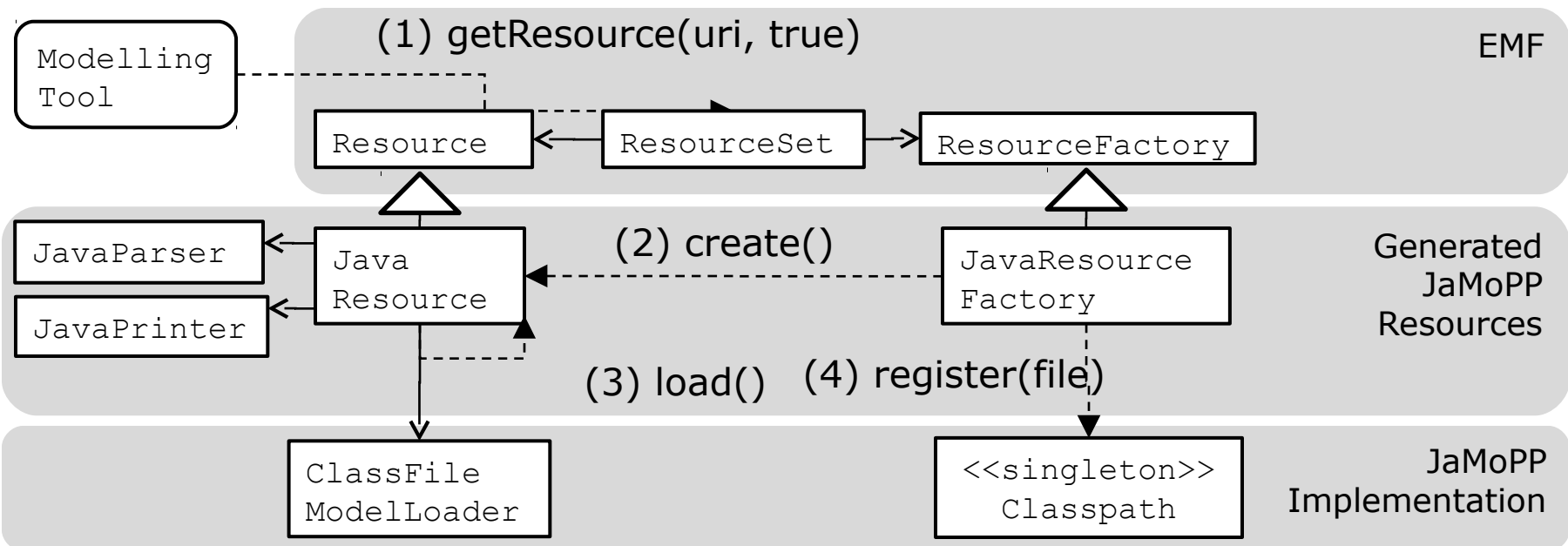


- JaMoPP seamlessly and transparently integrates with arbitrary EMF-based Tools



- Parsing Java files to models and Printing Java Files is simple

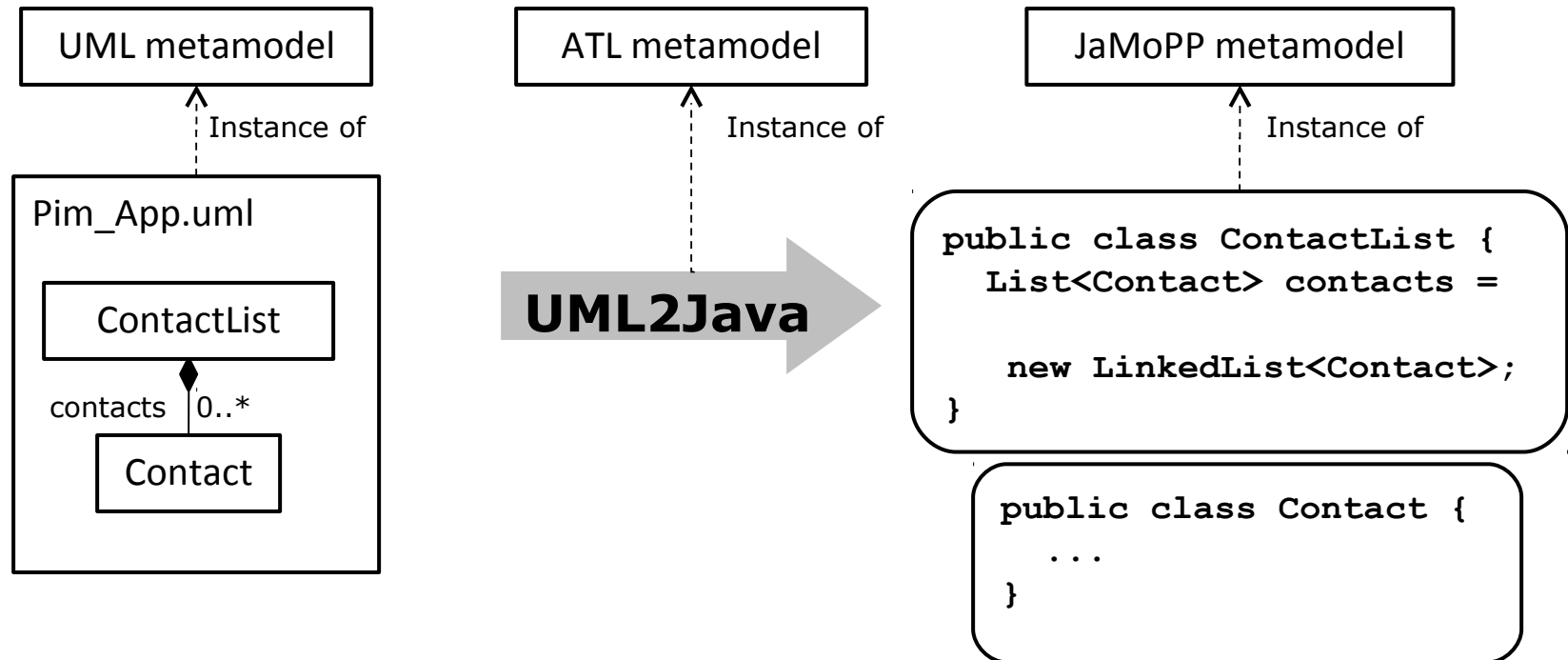
- JaMoPP seamlessly and transparently integrates with arbitrary EMF-based Tools



- 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
```

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



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

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

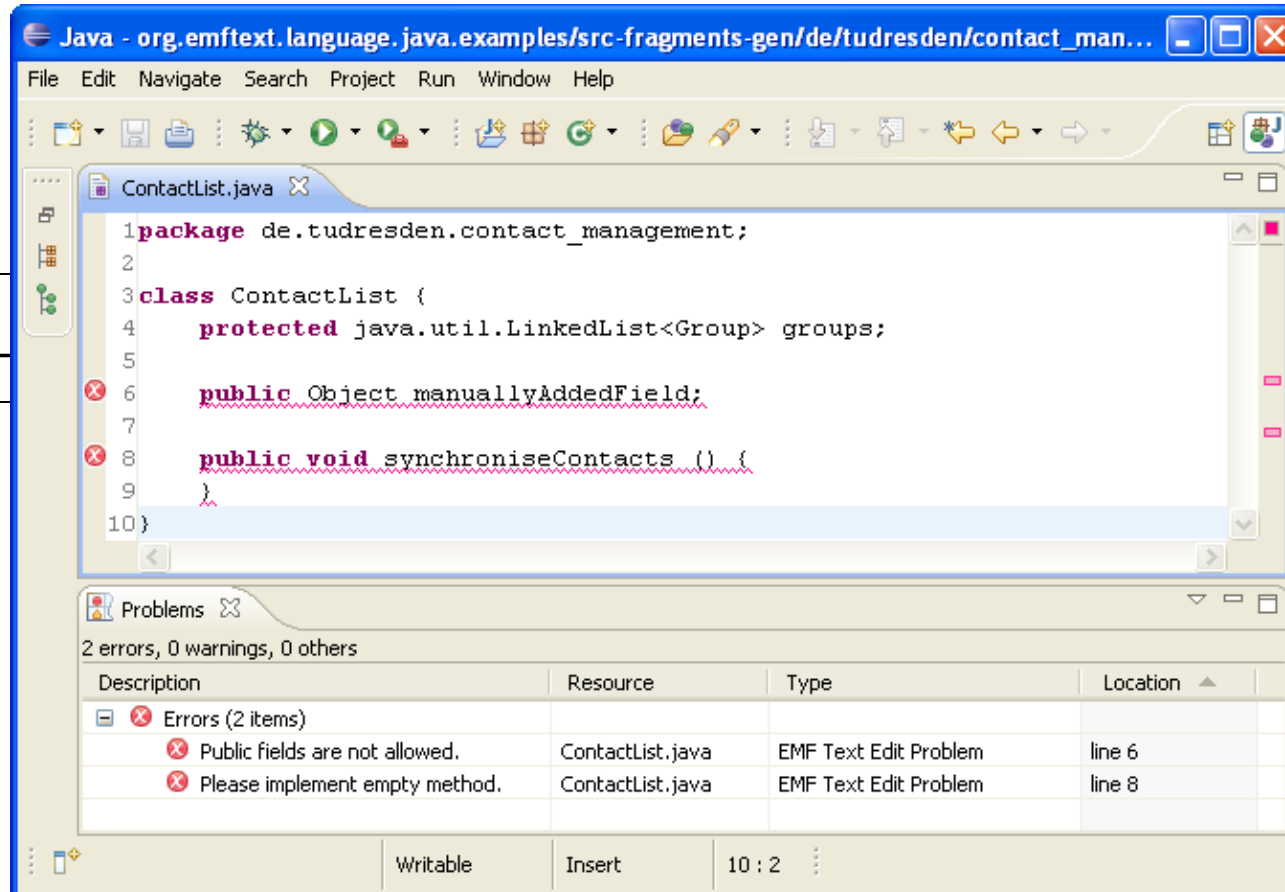
  typeReference : java!TypeReference (
    target <- if (umlProperty.upper = 1) then umlProperty.type
    else
      java!Package.allInstances()->any(p | p.name = 'java.lang').compilationUnits->collect(
        cu | cu.classifiers)->flatten()->any(c | c.name = 'LinkedList')
    endif,
    typeArguments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

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

- 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
```

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



The screenshot shows an IDE window titled "Java - org.emftext.language.java.examples/src-fragments-gen/de/tudresden/contact\_man...". The editor displays the following Java code:

```

1 package de.tudresden.contact_management;
2
3 class ContactList {
4     protected java.util.LinkedList<Group> groups;
5
6     public Object manuallyAddedField;
7
8     public void synchroniseContacts () {
9     }
10 }

```

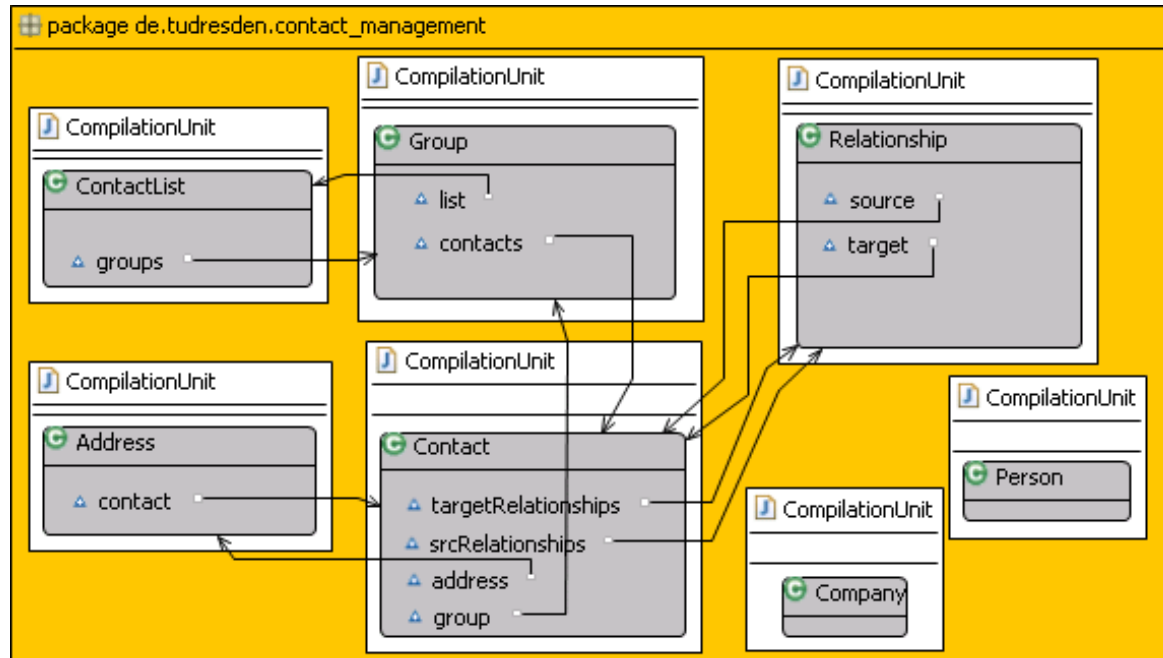
The Problems view at the bottom shows two errors:

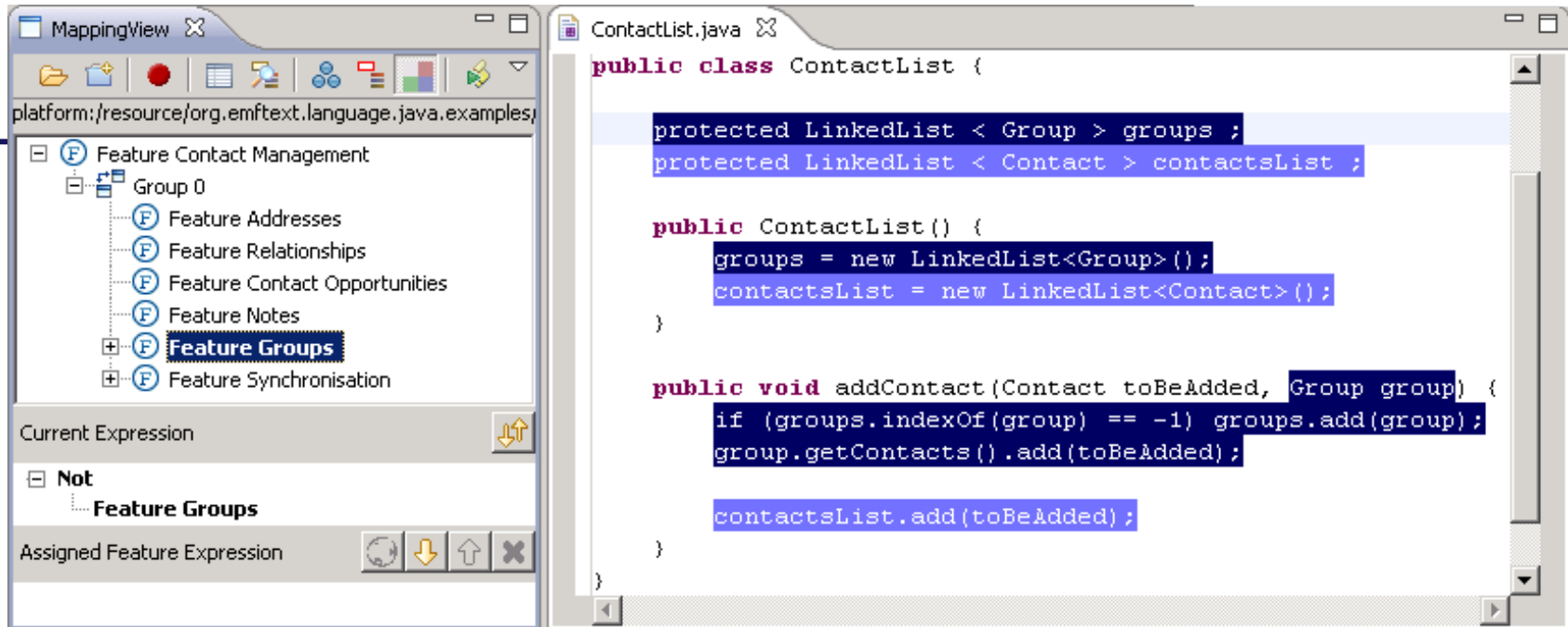
Description	Resource	Type	Location
Errors (2 items)			
Public fields are not allowed.	ContactList.java	EMF Text Edit Problem	line 6
Please implement empty method.	ContactList.java	EMF Text Edit Problem	line 8

context  
self-

ze () = 0

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





The screenshot displays two windows from the JaMoPP application. The left window, titled 'MappingView', shows a hierarchical feature model. The root node is 'Feature Contact Management', which contains a sub-node 'Group 0'. Under 'Group 0', there are several feature nodes: 'Feature Addresses', 'Feature Relationships', 'Feature Contact Opportunities', 'Feature Notes', 'Feature Groups' (highlighted with a blue selection box), and 'Feature Synchronisation'. Below the tree, there are sections for 'Current Expression' (containing 'Not') and 'Assigned Feature Expression'. The right window, titled 'ContactList.java', shows the source 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) ;  
    }  
}
```

Several lines of code are highlighted in blue, corresponding to the 'Feature Groups' feature in the model. A blue circle with the number '1' is positioned to the left of the MappingView window, and a blue circle with the number '2' is positioned to the right of the ContactList.java window.

1 – Feature Model

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

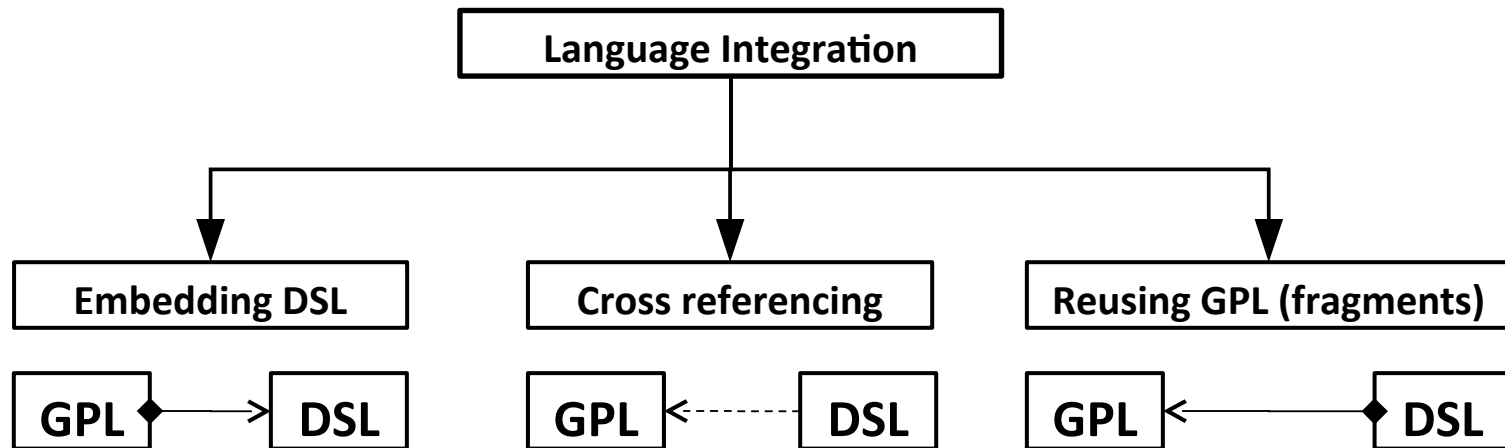


- Typesafe Template Languages
  - Same syntax as string-based templates
- Round-trip Support for template-based code generators
- Refactoring, Optimization using model transformations
- Traceability-related activities
  - Certification (Map code to the model elements)
  - Impact analysis (How much of the code will change if I do this?)
- Model-based compilation to byte code
- ...

## Approach

- (1) Use EMFText to *lift* GPLs to the technical space of DSLs
- (2) Language integration by metamodel and grammar inheritance**

- Different integration scenarios



- 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



- 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>

- 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?

**emftext**

**<http://www.emftext.org>**

**emftext**