Code Generation
Model-to-text transformation languages

Juan de Lara, Elena Gómez, Esther Guerra
{Juan.deLara, MariaElena.Gomez, Esther.Guerra}@uam.es

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Outline

- Introduction
- Acceleo
- Issues in code generation
- Other languages

Introduction
Definition

- Generation of textual software artifacts from a source model
  - Input: one or more models
  - Output: one or more textual files
- Code generation languages
  - Model-to-text (M2T) or Model-to-code (M2C)
  - Usually, template languages

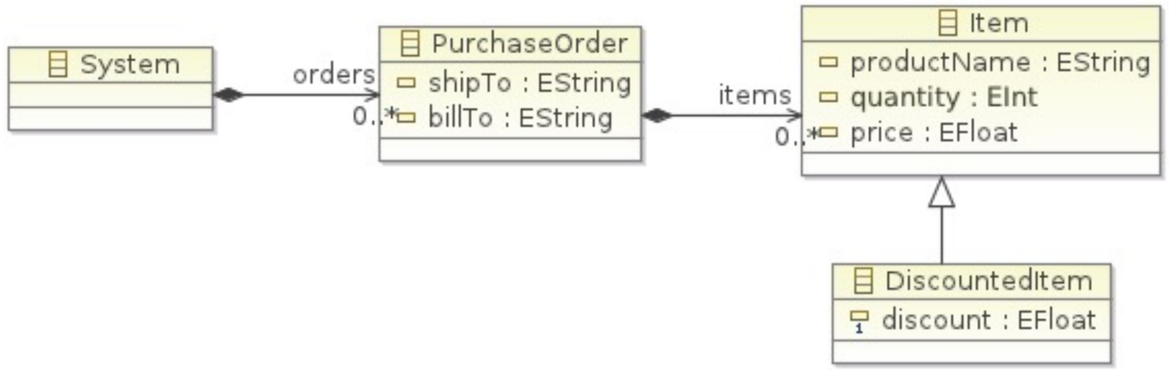
Introduction
Some application scenarios

- DSL → GPL
  - Generate code for a DSL (e.g., built with xText)
  - Full code generation could be achieved in restricted domains
- UML → GPL
  - Create some scaffolding code from a UML model
- Recovered model → GPL
  - Reverse engineering a system
  - Regenerate the system from the recovered info.

Introduction

Basics of code generation – by example (I)

- Generate HTML from the “purchase order model”



Introduction
Basics of code generation – by example (II)

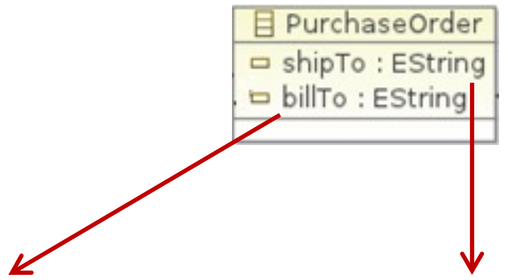
- Text we want to generate

```
<html>
<body>
    <h3>Orders listing</h3>
    <ul>
        <li>Order billed to Juan and sent to UAM</li>
        <li>Order billed to Esther and sent to Home</li>
        <li>Order billed to Elena and sent to Office</li>
    </ul>
  </body>
</html>
```

Introduction

Basics of code generation – by example (III)

- Identify placeholders

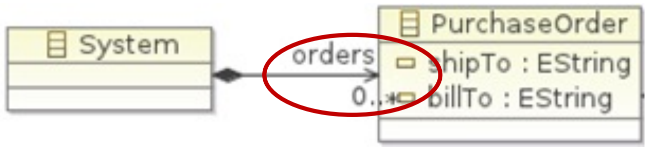


```
<html>
<body>
    <h3>Orders listing</h3>
    <ul>
        <li>Order billed to Juan and sent to UAM</li>
        <li>Order billed to Esther and sent to Home</li>
        <li>Order billed to Elena and sent to Office</li>
    </ul>
  </body>
</html>
```

Introduction

Basics of code generation – by example (IV)

- Identify iterations



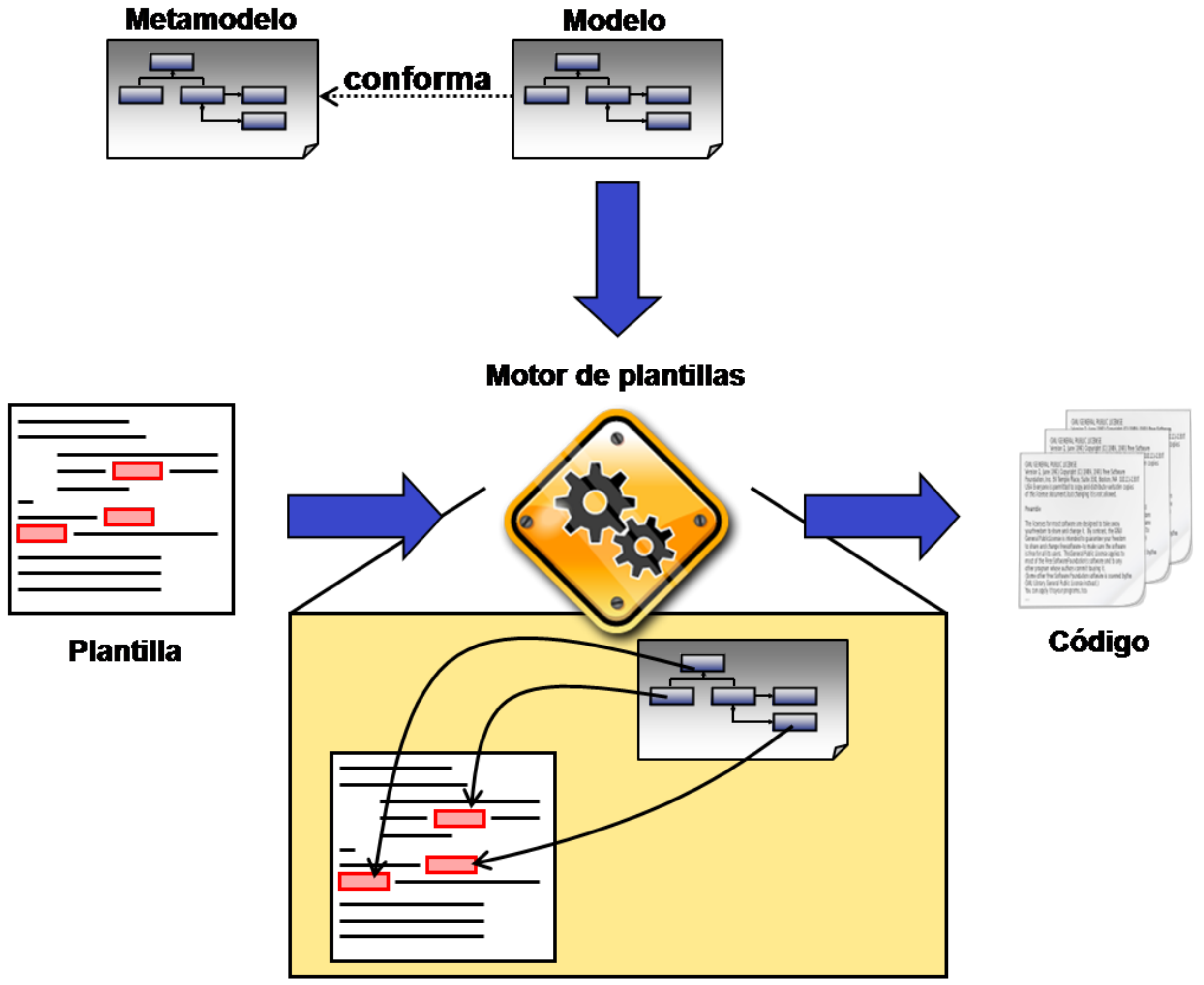
One <li> per *PurchaseOrder*

```
<html>
<body>
    <h3>Orders listing</h3>
    <ul>
       <li>Order billed to Juan and sent to UAM</li>
       <li>Order billed to Esther and sent to Home</li>
       <li>Order billed to Elena and sent to Office</li>
    </ul>
  </body>
</html>
```

Introduction
Basics of code generation

- Code generation template
  - Filling placeholders with model element values
  - Navigating (includes iterating) over the model
- Elements of a template
  - Fixed text
  - Placeholders within the text
  - Navigation expressions / statements

Introduction
Execution of a template language



Introduction
Issues in code generation

- Hand-written code
  - Incremental consistency
- Compiling generated code
- Pretty printing

Introduction
Template languages

- EGL – Epsilon Generation Language
  - http://eclipse.org/gmt/epsilon/doc/egl/
- MOFScript
  - http://www.eclipse.org/gmt/mofscript/
- M2T Project
  - http://www.eclipse.org/modeling/m2t/
  - JET
  - Acceleo
  - xPand

Introduction
Template languages (II)

- MOF Model to Text Transformation Language
  - OMG Standard
  - http://www.omg.org/spec/MOFM2T/1.0/
- Acceleo
  - An implementation of the standard
  - Very similar, but not complete

Outline

- Introduction
- Acceleo
- Issues in code generation
- Other languages

Acceleo
Introduction

- Template-based code generator
  - Fixed code is plain text
  - Variable parts written in OCL
- Advanced features
  - Preserving handwritten sections (language-independent merging engine)
  - Traceability mechanisms
  - IDE support
- Modularity mechanisms

Acceleo *Elements*

- Elements
  - **Module**
    - Mechanism for structuring transformation definitions.
    - Imports the metamodels for the input models.
    - Contains one or more templates.

Acceleo
Elements

- Elements
  - **Template**
    - Defined for a particular **meta-class.**
    - Templates may call each other.
    - Templates may extend each other.
    - Text with placeholders (`[ expr /]`)
    - May contain blocks
  - **Blocks**
    - `For, If, Let, etc.`

Acceleo
Defining templates

```
[comment encoding=UTF-8 /]
[module generate('http://master/mde/orders/version_model2text')]

[template public generateSystem(s : System)]
[comment @main/]

<html>
<body>
    <h2>Orders listing</h2>
    <ul>

[for (o : PurchaseOrder | aSystem.orders)]
<li>Order billed to [o.billTo /] and shipped to [o.shipTo /]</li>
[/for]

    </ul>
  </body>
</html>

[/template]
```

Acceleo
Generating files

- File block
  - Everything directly or indirectly contained in the block is added to the file

```
[file ('orders.html', false)]
   ...
[/file]
```

  - Open mode: true = append, false = truncate
  - File blocks can be nested
    - `[file('stdout', false)] processing shipment...[/file]`

Acceleo
Invoking templates

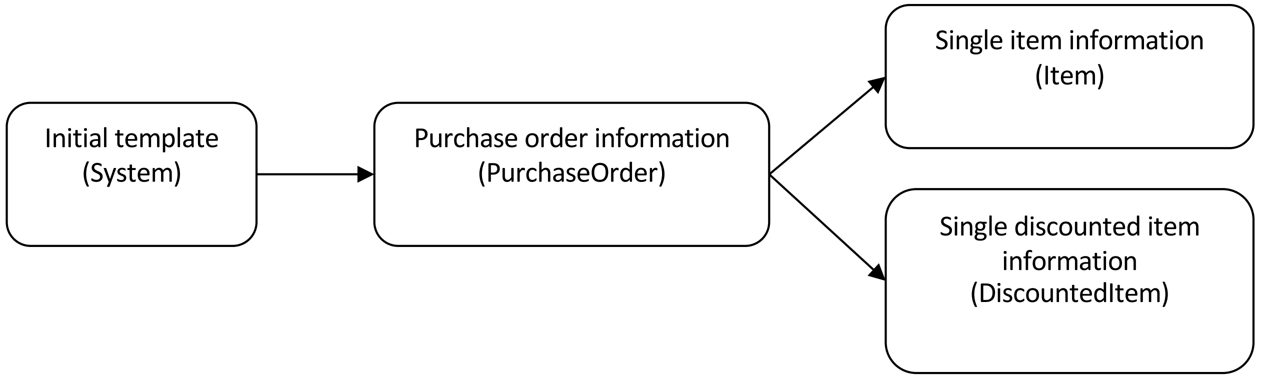
- Split a transformation in several templates
  - Example: generate detailed order information

```
<div>
<h3>Order 1</h3>
  <p>Bill to: Juan</p>
  <p>Ship to: UAM</p>
  <p>Items of the order:</p>
  <ul>
    <li>3 Book – 20 euros = 60 euros</li>
    <li><b>Offer 10%!</b> 2 CD – 10 euros = 18 euros</li>
  </ul>
<h3>Order 2</h3>
...
</div>
```

Acceleo

Invoking templates

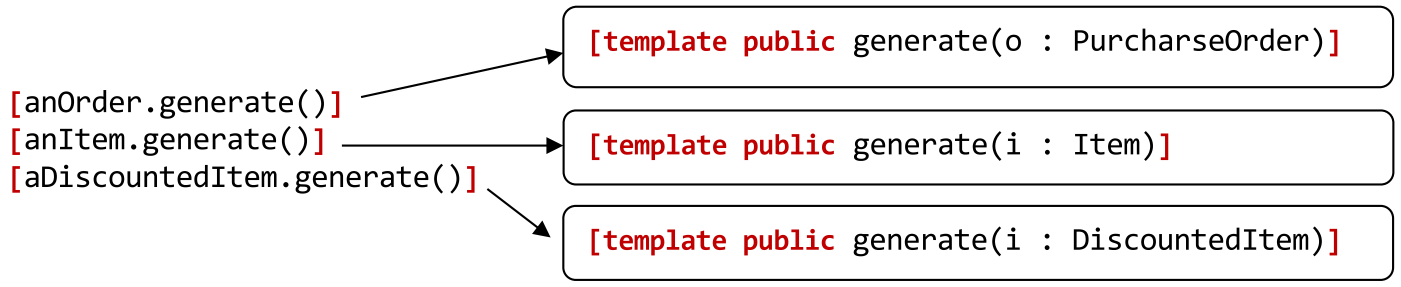
- Split a transformation in several templates
  - Example: generate detailed order information
  - Steps:
    - Traverse the list of orders of the system
    - Output information about the current order
    - Traverse the list of items of the current order
    - Output information the current item


Initial template (System)
Purchase order information (PurchaseOrder)
Single item information (Item)
Single discounted item information (DiscountedItem)

Acceleo

Invoking templates

- Invocation mechanism
  - Dot notation
  - First formal parameter of the template will be receptor object
  - Rest are normal parameters
  - Dynamic dispatch applies



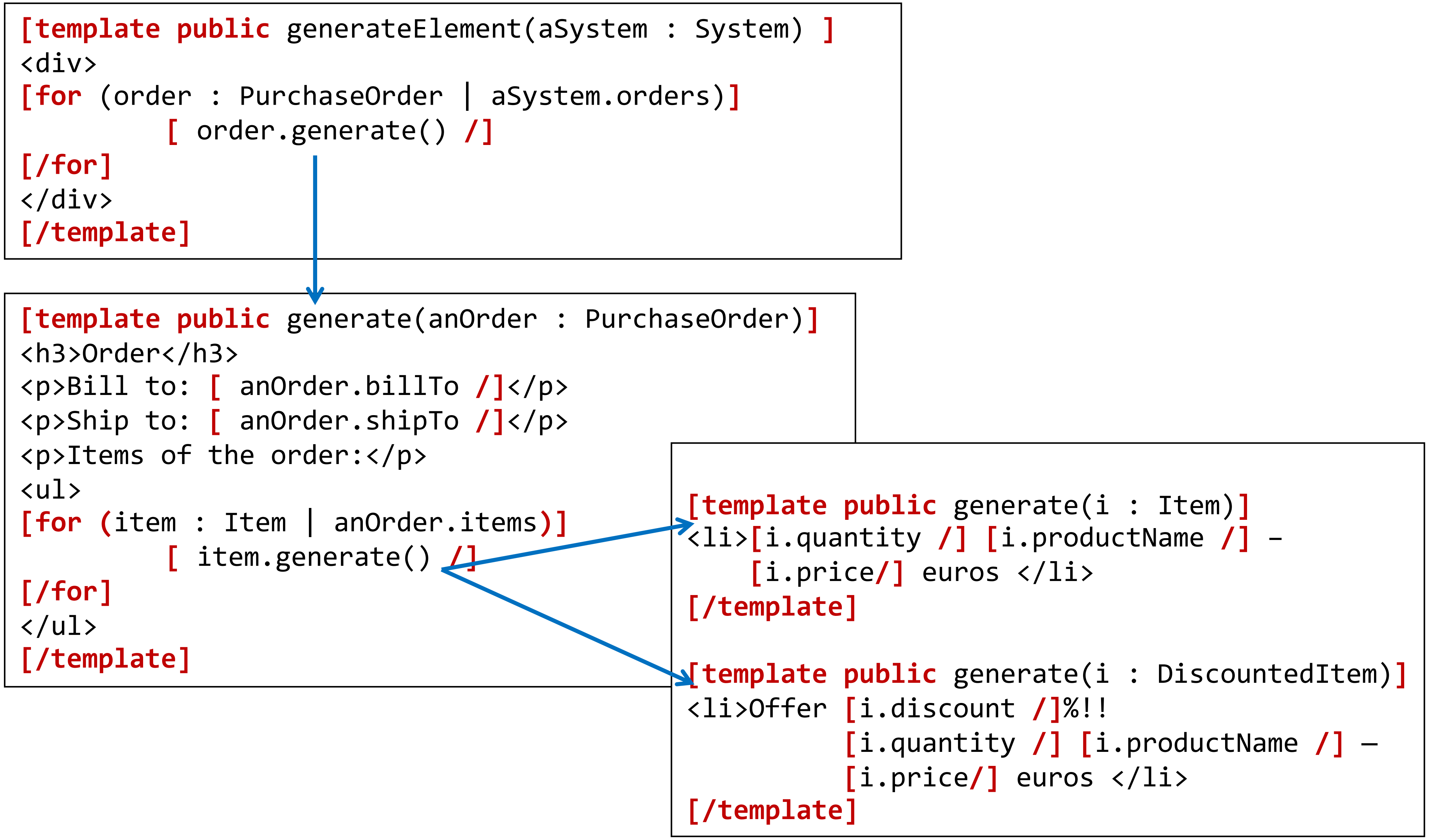
```
[anOrder.generate()]
[anItem.generate()]
[aDiscountedItem.generate()]
```

```
[template public generate(o : PurcharseOrder)]
```

```
[template public generate(i : Item)]
```

```
[template public generate(i : DiscountedItem)]
```

Acceleo
Invoking templates



```
[template public generateElement(aSystem : System) ]
<div>
[for (order : PurchaseOrder | aSystem.orders)]
         [ order.generate() /]
[/for]
</div>
[/template]
```

```
[template public generate(anOrder : PurchaseOrder)]
<h3>Order</h3>
<p>Bill to: [ anOrder.billTo /]</p>
<p>Ship to: [ anOrder.shipTo /]</p>
<p>Items of the order:</p>
<ul>
[for (item : Item | anOrder.items)]
         [ item.generate() /]
[/for]
</ul>
[/template]
```

```
[template public generate(i : Item)]
<li>[i.quantity /] [i.productName /] –
    [i.price/] euros </li>
[/template]

[template public generate(i : DiscountedItem)]
<li>Offer [i.discount /]%!!
      [i.quantity /] [i.productName /] –
      [i.price/] euros </li>
[/template]
```

Acceleo

User-defined operations

- Methods associated to a type
  - Similar to templates
  - Return a value of some type (e.g., String, Item)

```
[query public totalPrice(o : PurchaseOrder) : Real =
      o.items->collect(i | i.totalPrice() )->sum() /]

[query public totalPrice(i : Item) : Real = i.quantity * i.price /]

[query public totalPrice(i : DiscountedItem) : Real =
      i.quantity * (i.price - i.price * i.discount / 100.0) /]
```

Acceleo
Organizing transformations

- Split a generator into several modules
  - Reusing templates
  - Reusing queries
- Two forms of reuse
  - Module (single) inheritance
  - Import another module

```
[comment encoding = UTF-8 /]
[module generate('http://master/mde/orders/version_model2text')]
[import master::orders::acceleo::main::orderDetails /]
```

Acceleo
More sentences

- If – elseif – else
  - Output text depending on some condition

```
[template public myTemplate(i: Item)]
[if (i.price < 10)]
       Cheap product
[elseif( i.price < 20 )]
       Regular product
[else]
       Expensive product
[/if]
[/template]
```

Acceleo
More sentences

- Let
  - Declare a variable
  - Useful to improve code comprehension

```
[template public myTemplate(po: PurchaseOrder)]
[let maxDiscount : Real = anOrder.items->
        selectByKind(DiscountedItem)->collect(i | i.discount )->max() ]

       Maximum Discount [maxDiscount \]

[/let]
[/template]
```

  - Also to simplify “`instanceOf`” + “casting”

Outline

- Introduction
- Acceleo
- Issues in code generation
- Other languages

Issues
Dealing with hand-written code

- Frequently, generated code must be completed
  - How to ensure preservation of changes?
- Two alternatives:
  - Protected regions
    - Requires support of the M2T language
  - Framework structure
    - Requires more careful design of the generated code
    - Rely on target language modularisation mechanisms

Issues

Dealing with hand-written code (in Acceleo)

- Protected regions
  - `[protected(id)] … [/protected]`

```
<!--
[protected ('header-customization')]
-->
        <head>
        <!-- Fill meta-data for this site -->
        <head>
<!-- [/protected] -->
```

```
<!-- Start of user code header-customization
-->
        <head>
        <link href="myStyle.css" rel="stylesheet" type="text/css" />

        <!-- Fill meta-data for this site -->
        <head>
<!-- End of user code →
```

Issues
Dealing with hand-written code (in Acceleo)

- Annotations + JMerge
  - Add @generated to methods, classes, etc.
  - @generated NOT == Not overwrite an entity
  - Used by EMF generator

```
public class PurchaseOrderImpl extends EObjectImpl
                               implements PurchaseOrder {
  /**
   * @generated NOT
   */
  public void setBillTo(String newBillTo) {
    System.out.println("Manually added this sentence");
    String oldBillTo = billTo;
    billTo = newBillTo;
    if (eNotificationRequired())
      eNotify(new ENotificationImpl(this, Notification.SET,
PedidosPackage.PURCHASE_ORDER__BILL_TO, oldBillTo, billTo));
  }
```

Issues
Dealing with hand-written code

- General rules for generation code:
  - Don’t modify generated code.
  - Keep generated code clearly separate from hand-written code.
- Split a class in multiple files.
- …but Java doesn't handle with classes split in multiple files (others do).

Issues
Generation Gap Pattern

- *Separate generated code from non-generated code by inheritance.*
  - The handwritten class was a subclass of the generated class.
- Disadvantage: Relaxation in visibility rules.

Issues
Pretty-printing

- Beautification of generated code
  - Not required by compiler. Then, why?
    - Need of customisation
    - Developers mistrust generated code
- Alternatives
  - Not to pretty-print
  - In the code generator
    - Hard with template-languages
  - Use a pretty-printer

Issues
Compiling generated code

- Generate ANT, Maven, Make spec.
- Generate code within an Eclipse project
  - Create “sibling” project
    - Generate code in `src` folder.
  - Compilation happens after refreshing

Outline

- Introduction
- Epsilon Generation Language (EGL)
- Issues in code generation
- Other languages

Other languages
MOFScript

- “No template-based”
- Featuring
  - OCL-like navigation language
  - Polymorphic rules ≈ methods
  - Pretty-print control
    - newline, tab, space
- Two modes
  - Print statements
  - Escaped output

Other languages
MOFScript

- Advanced features
  - Transformation inheritance
  - Aspect-oriented extension

Other languages
MOFScript

```
texttransformation UML2Java(in uml:"http://simpleUML") {

uml.Class::main() {
  file(self.name + '.java')
  'package test.generation; '
  newline(1)
  'class ' self.name '{'
  tab(1) self.features->forEach(f) { f.mapFeature() }
  '}'
}

uml.Property::mapFeature() {
 'private' self.type.name space(1) self.name
}

uml.Operation::mapFeature() {
 'public' self.type.name space(1) self.name '() {'
    ...
 '}'
}
```

Other languages
EGL

- Part of the Epsilon platform
- Template-based
- Similar to JSP templates
- Imperative style with EOL (OCL-like lang.)