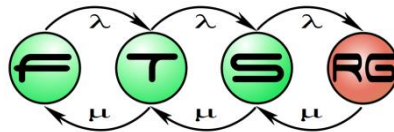


Code Generation

Ákos Horváth
Gábor Bergmann
Dániel Varró

Model Driven Systems Development
Lecture 8



Agenda

- Code Generation in general
- Approaches
- Advanced Text Generation Issues
- Example template languages
 - JET, Velocity, Xpand and XTend

Code Generation

(text synthesis)

Why?

- Let's shorten Development time!
- Use our **models/requirements/plans** to derive...
 - Documentation
 - Source code
 - Configuration descriptors
 - Communication messages
 - Object Serialization
 - ...
- Need to support designing „text” synthesis

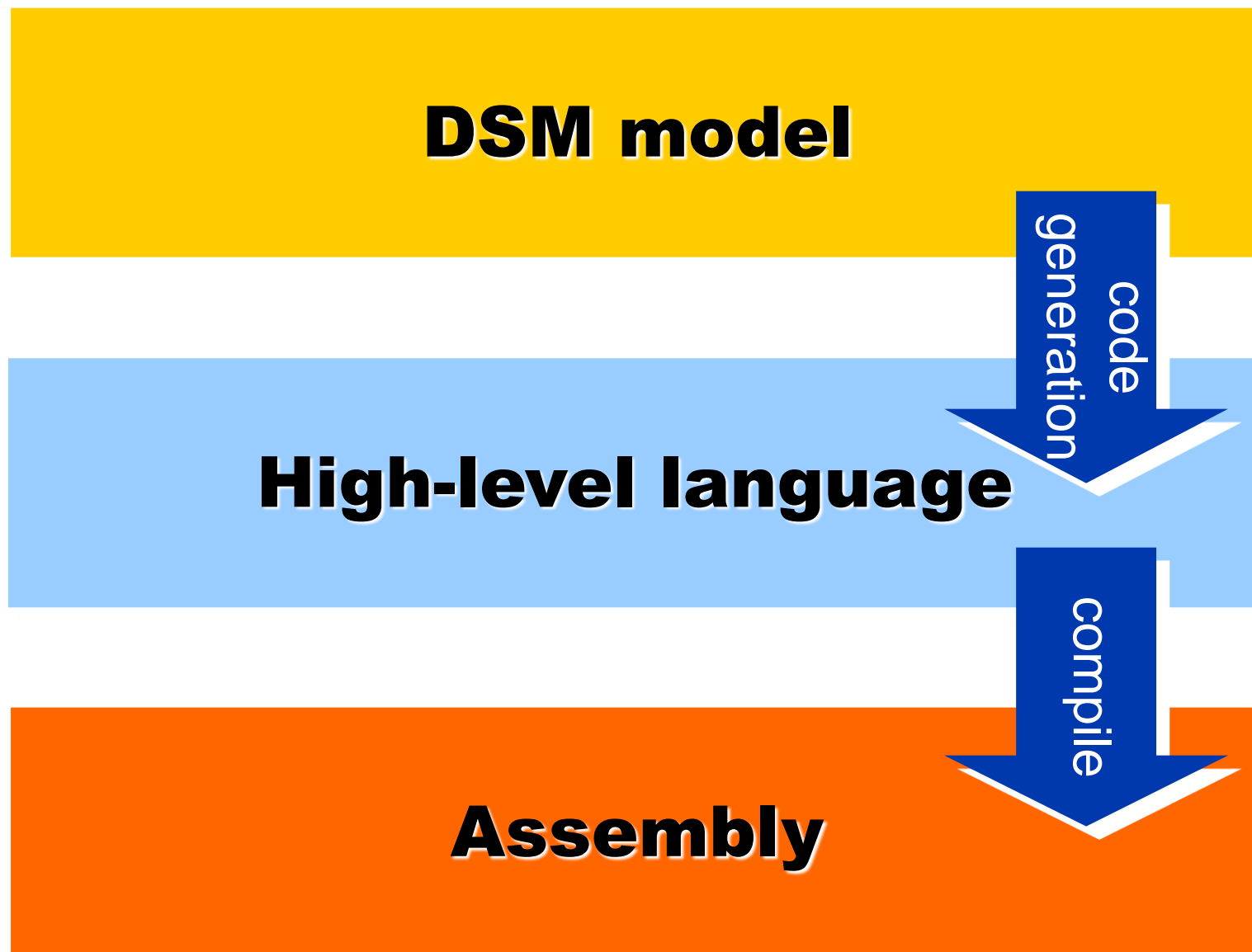
Text synthesis

- The realization of a high-level model on an implementation platform
- A choice between certain attributes – compromise between:
 - Compatibility
 - Performance
 - Maintainability
 - Reusability

Similarity with compilers

- Mapping between abstraction levels
 - e.g., From C to assembly
- Usage of design patterns
 - e.g., function calls in C
- Many similarities, NOT a strict separation
 - pl. C++ templates, automatically generated ctor+dtor
- Prediction:
 - yesterday's design pattern → today's code generation feature → tomorrow's language element
- Domain-specific instead of universal languages

Example: Source Code generation in MDD



Approaches

Approaches

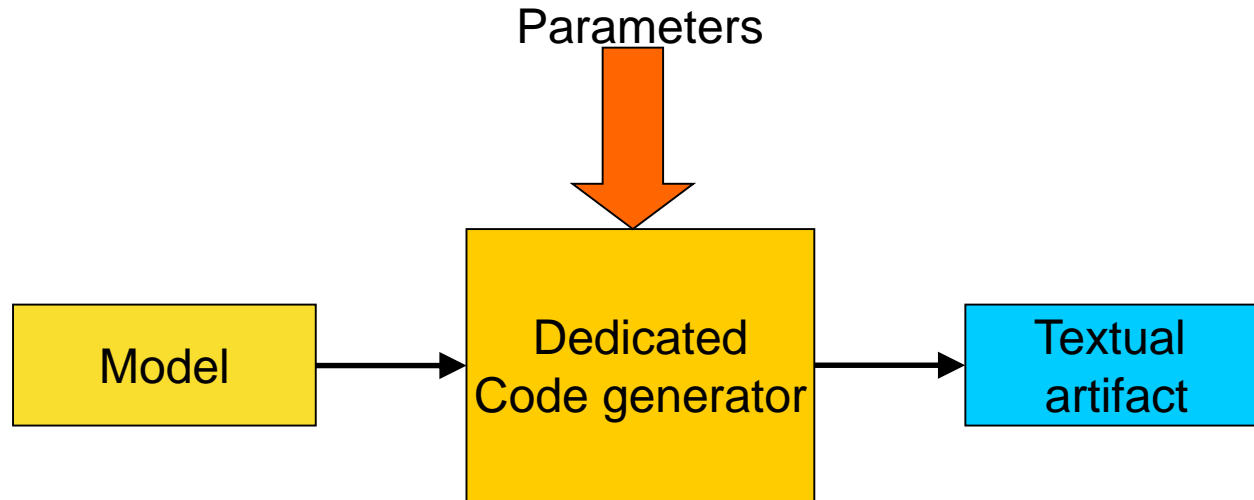
- Dedicated
 - Specific, ad-hoc
 - Using a dedicated code generator
- Template based

Specific, ad-hoc

```
sourceFile.write("    temp = ((AIDA_PARTITION_TYPE*) selfModule.partitions.elements);\n" )
i = 0
for partition in partitions:
    numPorts = getNumberOfAllCommPorts_Partition(currModuleComm, interPartitionComm, partition.partitionName)
    sourceFile.write("    temp[" + str(i) + "].partition_id = " + str(partition.partitionID) + ";\n" )
    sourceFile.write("    strcpy( &temp[" + str(i) + "].partition_name[0], \"" + str(partition.partitionName) + "\");\n")
    sourceFile.write("    temp[" + str(i) + "].ports.type = CONST_AIDA_PORTS_TYPE;\n")
    sourceFile.write("    temp[" + str(i) + "].ports.elements = &mem_ports_" + str(partition.partitionName) + "[0];\n")
    sourceFile.write("    temp[" + str(i) + "].ports.numOfElements = " + str(numPorts) + ";\n")
    sourceFile.write("\n")
    i = i + 1
## end for
sourceFile.write("\n")
```

- Designed for the specific problem domain:
 - Best performance
 - Quick and dirty
 - Long development, hard maintainability
 - Zero reusability
 - Dedicated problem domains
 - Minimal changes during support cycle (safety critical embedded system, defense)
 - Certifiability
 - Example:
 - ARINC653 Multistatic configuration generator (python script)

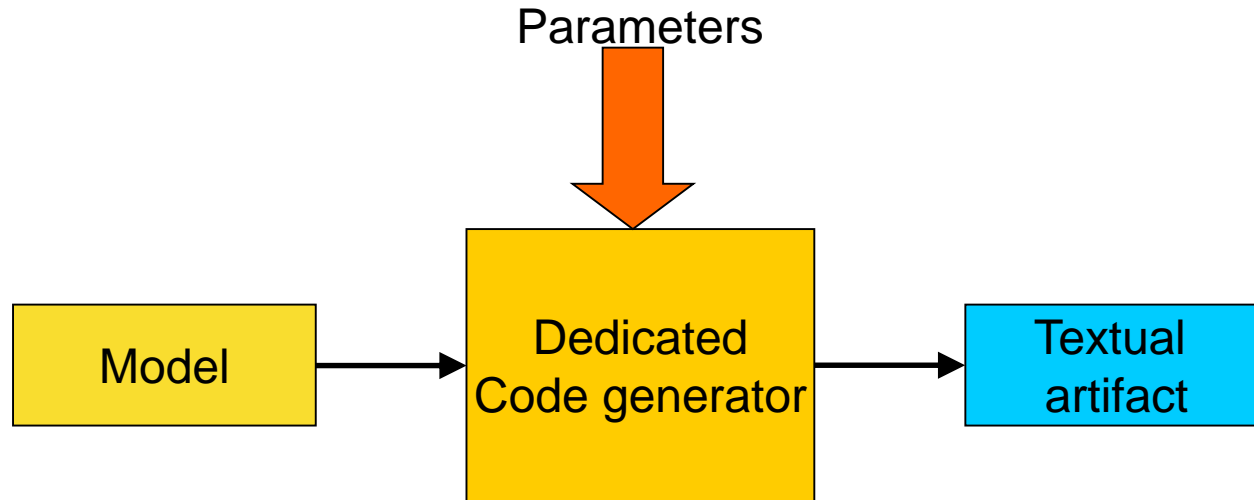
Dedicated code generator



- Based on a framework:

- Faster development time
- Slower performance, better reusability
- Embedded systems, moderate changes during project lifecycle

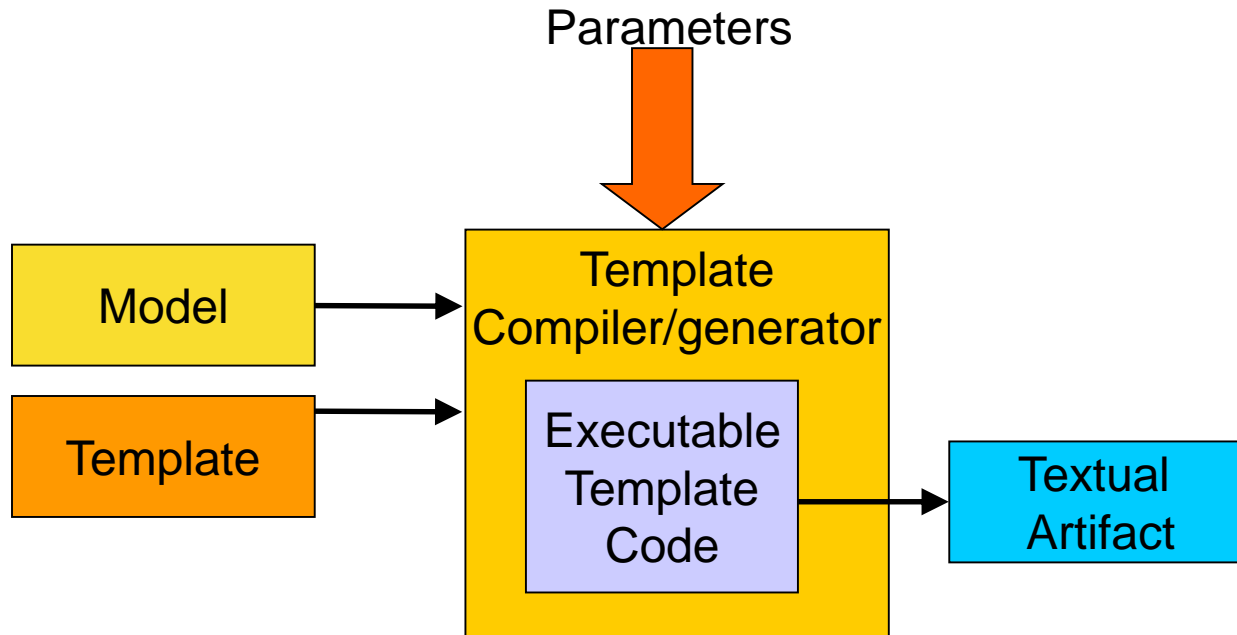
Dedicated code generator



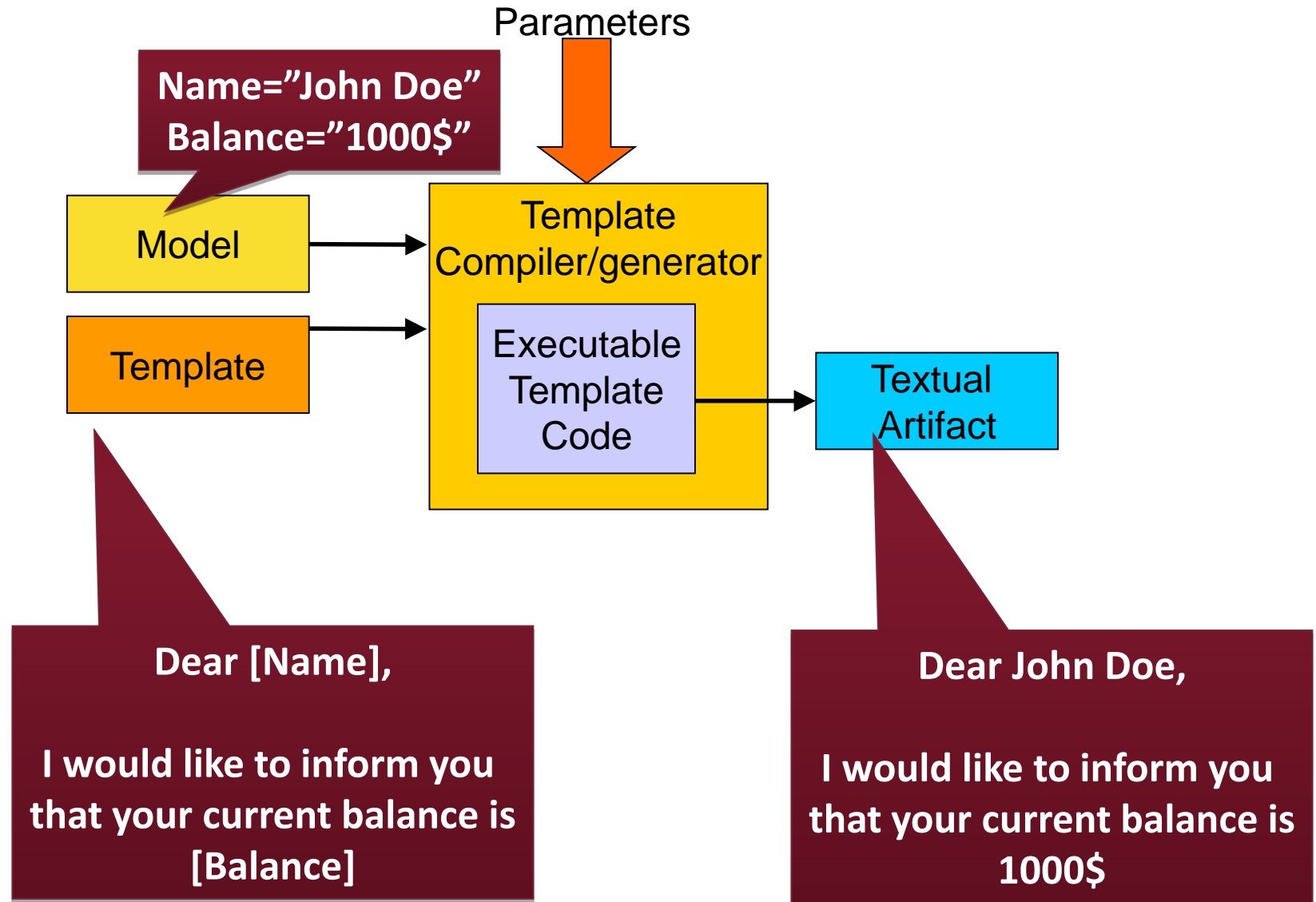
■ Examples:

- IBM Rational Software Architect
- VASP (DO-178B Level A) Display graphics in avionics
- Mathworks
- Matlab Simulink
- Esterel Scade suite

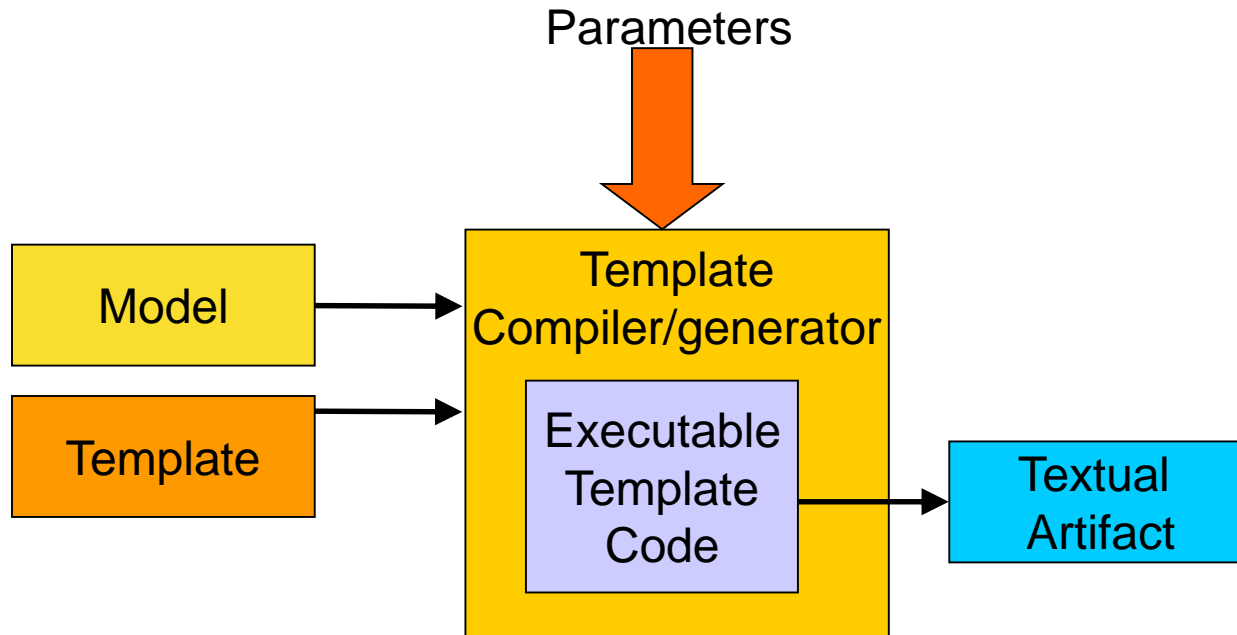
Template Based



Template Based

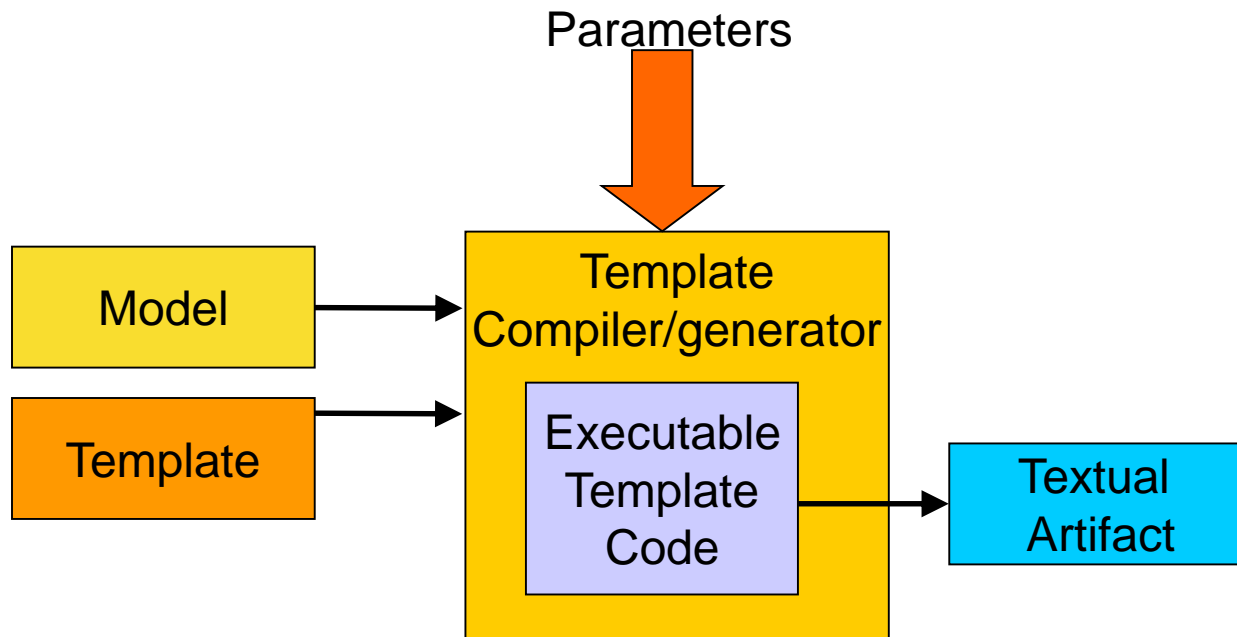


Template Based



- Fastest development time
- „Slowest” performance, highest reusability
- Fast changing environments (e.g., web based technologies)
- Complex changes during project lifecycle
 - Models and templates can be changed independently

Template Based



■ Examples:

- JET (for EMF models)
- Velocity (/JSP)
- OpenArchitectureWare/ XPand (MDD approach)
- AutoFilter (Kalman filters)
- Smarty (php)

Advanced Code/Text Generation Issues

Direct source code generation

- Direct source code generation
 - Simple structure
 - Low complexity
 - Fast development
 - Linear output generation (single pass)
 - Problematic formatting
 - Problematic M2C synchronization

Output:

```
package hu.bme.mit.pimpsm.diana.editors;

import hu.bme.mit.pimpsm.api.editors.PimPsmEditor;

/**
 * @author Akos Horvath
 *
 */
public class DianaPimPsmEditor extends PimPsmEditor
{

    public static final String PLUGIN_ID = "hu.mit.bme.pimpsm.diana";

    /* (non-Javadoc)
     * @see hu.bme.mit.pimpsm.api.editors.PimPsmEditor#createModelManager()
     */
    @Override
    public PimPsmModelManager createModelManager() {
        // TODO Auto-generated method stub
        return new DianaPimPsmModelManager(this);
    }

    /** Have to return the exact id of the project for the
     * in order to be able to include the icons
     */
}
```

AST generation

Output:

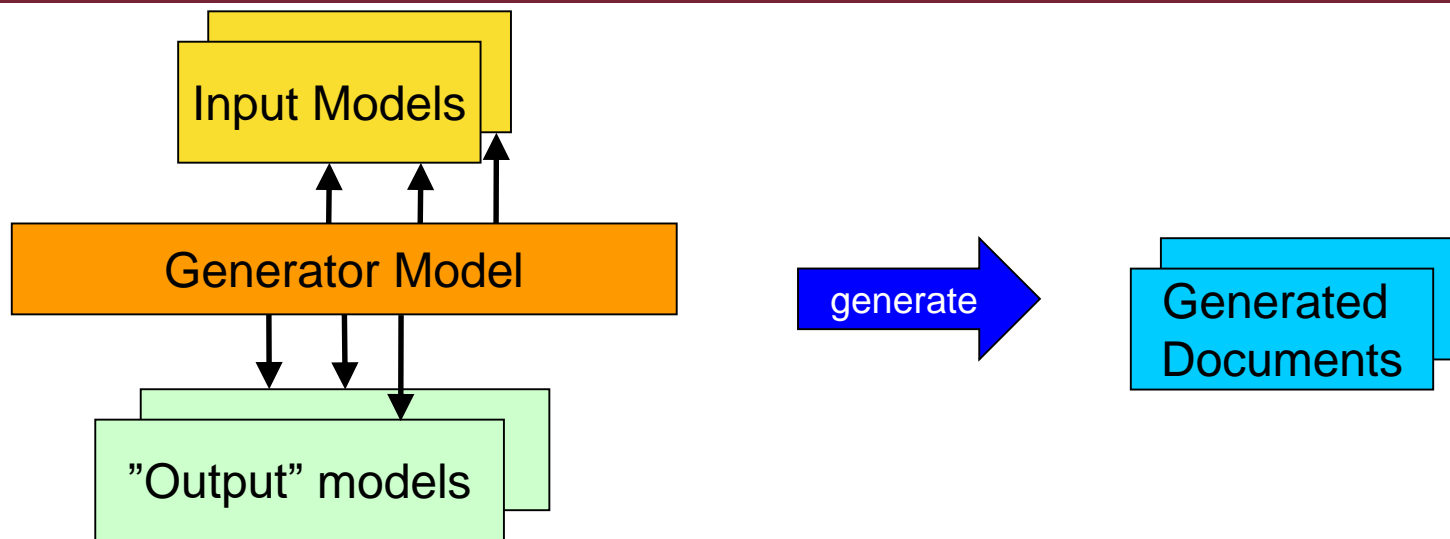
```
⊕ PACKAGE
⊖ IMPORTS (2)
  ⊖ ImportDeclaration [59, 50]
    ⊕ > type binding: hu.bme.mit.pimpsm.api.editors.PimPsmEditor
    --- STATIC: 'false'
    ⊕ NAME
    --- ON_DEMAND: 'false'
  ⊕ ImportDeclaration [111, 57]
⊖ TYPES (1)
  ⊖ TypeDeclaration [172, 817]
    ⊕ > type binding: hu.bme.mit.pimpsm.diana.editors.DianaPimPsmEditor
    ⊕ JAVADOC
    ⊖ MODIFIERS (1)
      ⊖ Modifier [211, 6]
        --- KEYWORD: 'public'
        --- INTERFACE: 'false'
    ⊕ NAME
    --- TYPE_PARAMETERS (0)
    ⊖ SUPERCLASS_TYPE
      ⊖ SimpleType [250, 12]
        ⊕ > type binding: hu.bme.mit.pimpsm.api.editors.PimPsmEditor
        ⊕ NAME
    --- SUPER_INTERFACE_TYPES (0)
    ⊖ BODY_DECLARATIONS (4)
      ⊖ FieldDeclaration [270, 65]
        --- JAVADOC: null
```

- AST generation
 - Represents the program structure (PSM)
 - Can be very complex
 - Slower development
 - Non-linear generation process
 - Support for M2C synchronization
 - Incremental output generation
 - "pretty formatting"
 - E.g., Eclipse JDT

Direct source code generation vs. AST

- Direct source code generation
 - Simple structure
 - Low complexity
 - Fast development
 - Linear output generation (single pass)
 - Problematic formatting
 - Problematic M2C synchronization
- AST generation
 - Represents the program structure (PSM)
 - Can be very complex
 - Slower development
 - Non-linear generation process
 - Support for M2C synchronization
 - Incremental output generation
 - "pretty formatting"
 - E.g., Eclipse JDT

Generator model



- Multiple source models → **"generator" model**
- Stores all additional information
- References to both Input Models and „Outputs” (prettyprintable)
- Helps code generation by
 - Multiple output streams
 - Traceability between models → cross references
 - Support for Non-linear **"Multi Pass"** traversals and model build
 - Support for complex model hierarchies (multiple AST-s, packages etc.)

Model to code synchronization

- What if the output text is changed? → M2C synchronization
- Works only with AST based approaches
- Requires
 - Traceability between model and text
 - Model compare
 - Change localization
- Incremental model building for better performance
- Example
 - Eclipse JDT: java source and its AST
 - EMF: model generator

Manual and generated parts

- Don't overwrite manual extensions upon re-generation
- Where to put non-changing parts
 - Model
 - Allows better reusability
 - Increases complexity
 - Template
 - Works well for simple cases
 - AST
 - Manual markings in AST → the rest is generated
 - Directly to code
 - Java → no support ☹️
 - Use generalization
 - C# partial classes 😊

Code formatting

- Where to include?
 - Model
 - Does not follow typical MVC design paradigm
 - Templates
 - Simple formatting element
 - AST
 - Can store all relevant information
 - Makes it very complex
- Best solution: Code formatting as **separate step**
 - a new step in the generation workflow
 - Can be handled with 3rd party code formatters
 - Eclipse JDT formatter
 - XML DOM serializer

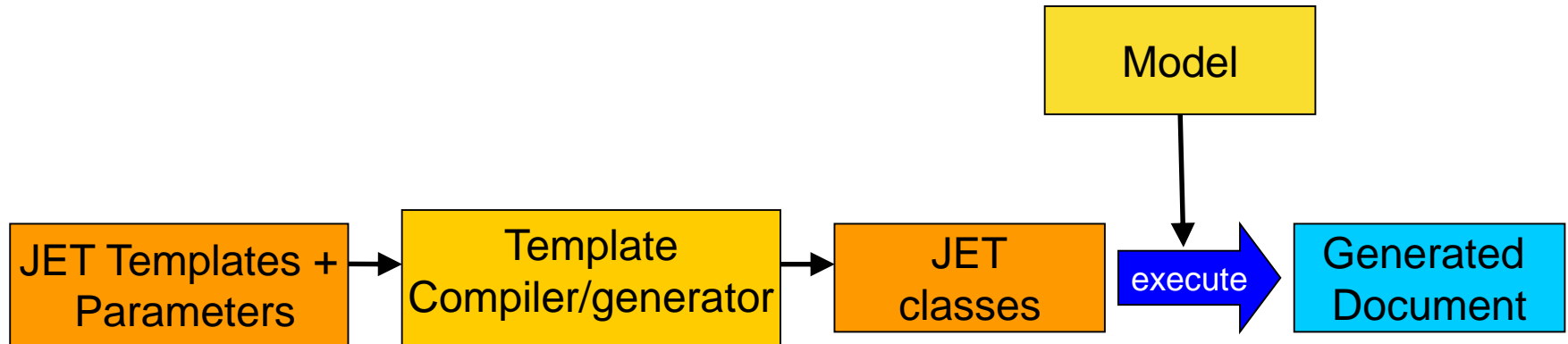
Keywords and special characters

- Restricted keywords in the target language
 - Java: abstract, class
 - XML: '<', '>'
 - etc.
- Needs to validate the model before generation
 - Can be very complex → separate step before code generation
 - Example
 - Java simple support: isJavaIdentifierStart() (in Character)
 - EMF validation
- Escaping
 - On the model (in separate generator model?)
 - Only at code generation time

Java Emitter Templates (JET)

Velocity, Xpand, Xtend

Java Emitter Templates



■ Java Emitting Templates (JET)

- JSP-like template language using Java as its control sequence
- **Compiled** to Java
- Open output format (Text)
- Parameters as Java objects
- Part of EMF
- Eclipse uses JET as its own template language

JET example

```
<%@ jet package="hello" imports="java.util.*"
class="XMLDemoTemplate" %>
<% List elementList = (List) argument; %>

<?xml version="1.0" encoding="UTF-8"?>
<demo>

<% for (Iterator i = elementList.iterator();
i.hasNext(); ) { %>
<element><%=i.next().toString() %></element>

<% } %>

</demo>
```

JET example

```
<%@ jet package="hello" imports="java.util.*"
class="XMLDemoTemplate" %>
<% List elementList = (List) argument; %>
<?xml version="1.0" encoding="UTF-8"?>
<demo>
  <% for (Iterator it = elementList.iterator();
  i.hasNext(); %>
    <element><%=i.next().toString()%></element>
  <% } %>
</demo>
```

Jet Header

Package of representing class

Packages to import

Name of the Class representing the Template

JET example

```
<%@ jet package="hello" imports="java.util.*"
class="XMLDemoTemplate" %>
<% List elementList = (List) argument; %>
<?xml version="1.0" encoding="UTF-8"?>
<demo>
  <% for (Iterator i = elementList.iterator();
i.hasNext(); ) { %>
    <element><%=i.next().toString()%></element>
  <% } %>
</demo>
```

Start of code section

Input parameter

End of code section

JET example

```
<%@ jet package="hello" imports="java.util.*"  
class="XMLDemoTemplate" %>
```

```
<% List elementList = (List) argument; %>
```

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<demo>
```

```
<% for (Iterator i = elementList.iterator();  
i.hasNext(); ) { %>
```

```
<element><%=i.next() %> </element>
```

Start of target document

```
<% } %>
```

```
</demo>
```

JET example

```
<%@ jet package="hello" imports="java.util.*"  
class="XMLDemoTemplate" %>
```

```
<% List elementList = (List) argument; %>
```

Loop with the input parameter

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<demo>
```

```
<% for (Iterator i = elementList.iterator();  
i.hasNext(); ) { %>
```

```
<element><%=i.next().toString() %></element>
```

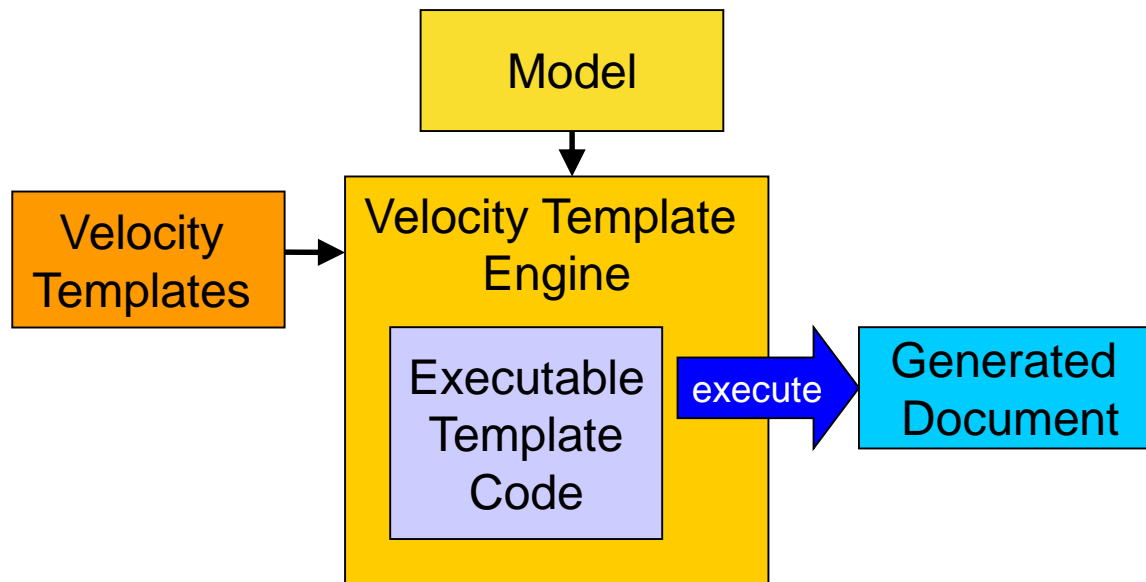
```
<% } %>
```

Loop body

```
</demo>
```

Returns value of
the argument

Apache Velocity



■ Apache Velocity

- JSP like template language with limited control sequence
- **Interpreted**
- Open output format (Text)
- Parameters as a Map

Velocity example

```
<?xml version="1.0" encoding="UTF-8"?>
<demo>
#set( $tempString = " Element")
#foreach( $element in $elementList)
    <element> ${element.toString()} <element>
#end

</demo>
```

Velocity example

Start of target document

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<demo>
```

New value of tempString

```
#set( $tempString = "Element")
```

```
#foreach( $element in $elementList)
```

```
  <element> ${element.toString()} <element>
```

```
#end
```

```
</demo>
```

Setting values

New variable

Velocity example

```
<?xml version="1.0" encoding="UTF-8"?>
<demo>
#set( $tempString = "Element")
#foreach( $element in $elementList)
  <element> ${element.toString()} <element>
#end
</demo>
```

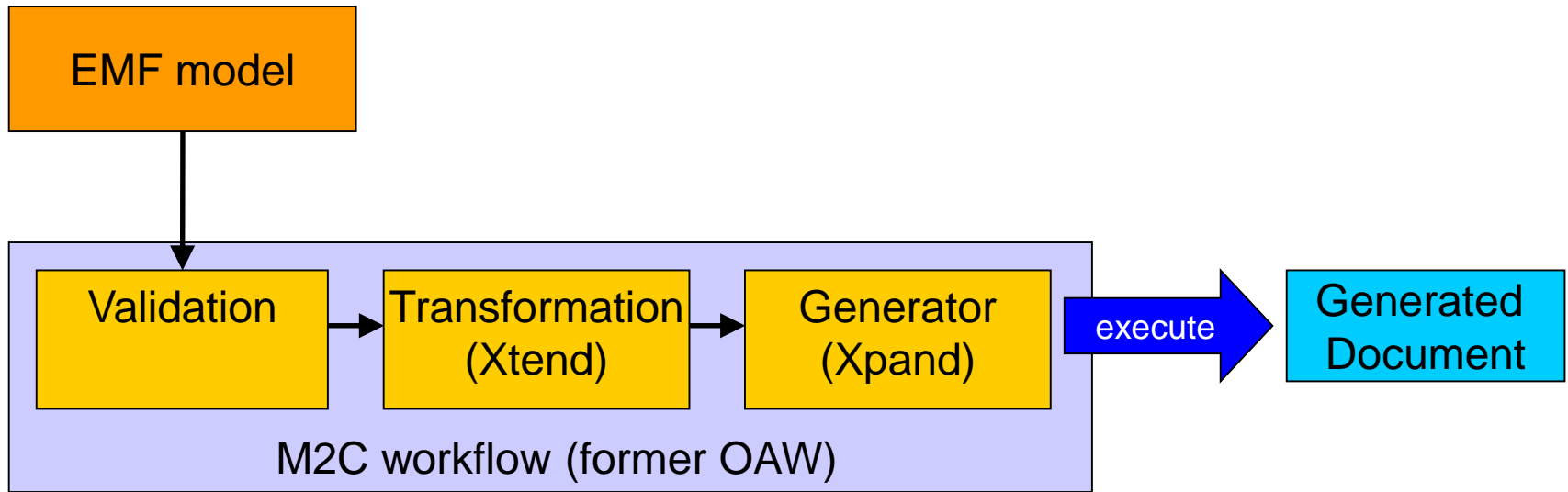
Input parameter

For loop

New running variable

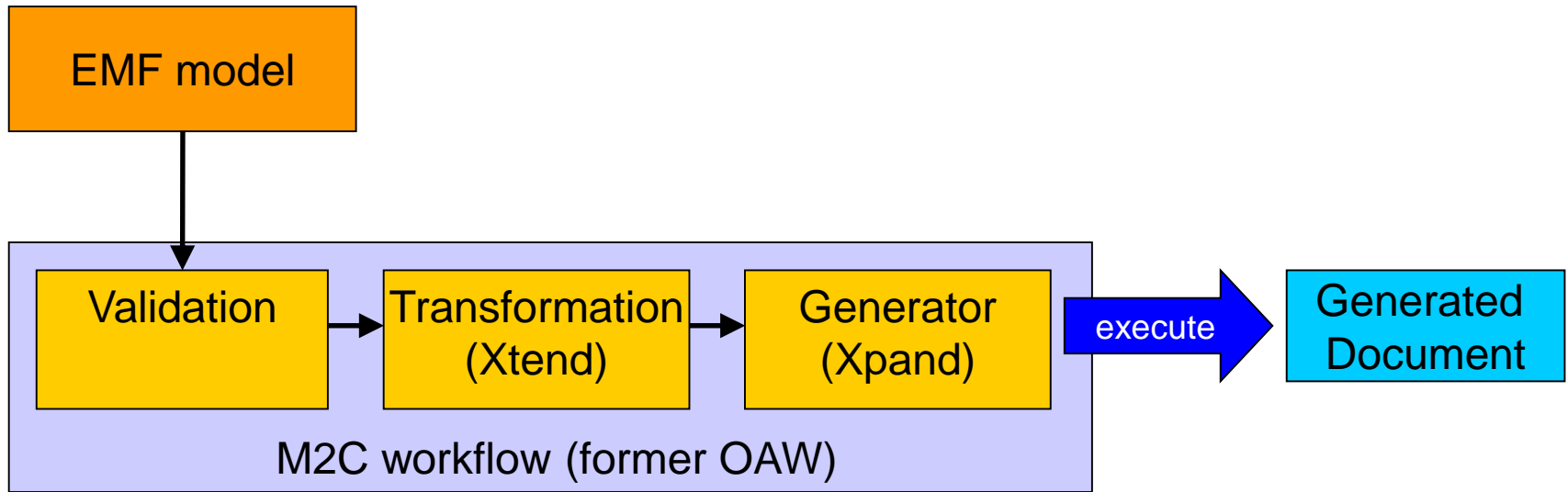
Arbitrary Java method call

Xtend



- **Eclipse M2C (former OAW)**
 - Complete M2C workflow
 - Validation
 - Transformation (Xtend language)
 - Code generation (Xpand language)
- **Mainly for EMF model based transformation**
- **Flexible workflow definition**

Xtend



- **Interpreted**
- Statically-typed template language
- Polymorphic template invocation
- Support for AOP programming
- Error handling
- Support for whitespace generation/definition 😊

Xpand

```
«IMPORT XMLmetamodel»
«DEFINE main FOR Model»
  «FILE this.name + ".myxml"»
  <?xml version="1.0" encoding="UTF-8"?>
  <demo>
    «EXPAND listElement FOREACH elements»
  </demo>
  «ENDFILE»
«ENDDEFINE»

«DEFINE listElement FOR Element»
<element> «this.toString()»</element>
«ENDDEFINE»
```

Xpand

```
«IMPORT XMLmetamodel»
```

Import EMF metamodel

```
«DEFINE main FOR Model»
```

```
  «FILE this.name + ".xml"»
```

```
  <?xml version="1.0" encoding="UTF-8"?>
```

```
  <demo>
```

Define template for specific type

```
  «EXPAND listElement FOREACH elements»
```

```
  </demo>
```

```
  «ENDFILE»
```

```
«ENDDEFINE»
```

```
«DEFINE listElement FOR Element»
```

```
<element> «this.toString()»</element>
```

```
«ENDDEFINE»
```


Xpand

```
«IMPORT XMLmetamodel»  
«DEFINE main FOR Model»  
  «FILE this.name + ".myxml"»  
  <?xml version="1.0" encoding="UTF-8"?>  
  <demo>  
    «EXPAND listElement FOREACH elements»  
  </demo>  
  «ENDFILE»  
«ENDDEFINE»
```

Output file definition

Start of target document

```
«DEFINE listElement FOR Element»  
<element> «this.toString()»</element>  
«ENDDEFINE»
```

Xpand

```
«IMPORT XMLmetamodel»  
«DEFINE main FOR Model»  
  «FILE this.name + ".myxml"»  
  <?xml version="1.0" encoding="UTF-8"?>  
  <demo>  
    «EXPAND listElement FOREACH elements»  
  </demo>  
  «ENDFILE»  
«ENDDEFINE»  
«DEFINE listElement FOR Element»  
<element> «this.toString()»</element>  
«ENDDEFINE»
```

EReference holding the elements

Invoke other template with type definition

Xtend overview

- Foundation of Xtext2
 - Original purpose: compile Xtext2 DSLs to Java
- A JVM-based language
 - Imperative, statically typed, compiles to Java
 - Incorporates functional programming constructs
- Advanced features
 - Type inference
 - Properties
 - Everything is an expression
 - Operator overloading
 - Power switch
 - Lambda expressions
 - Templates

Xtend example

```
import com.google.inject.Inject
```

Java Import

```
class DomainmodelGenerator implements IGenerator {
```

```
    @Inject extension IQualifiedNameProvider nameProvider
```

?

```
    override void doGenerate(Resource resource, IFileSystemAccess fsa) {
```

```
        for(e: resource.allContentsIterable.filter(typeof(Entity))) {
```

```
            fsa.generateFile(
```

```
                e.fullyQualifiedName.toString.replace(".", "/") + ".java",
```

```
                e.compile)
```

Built-in function

```
        }
```

```
    }
```

```
    def compile(Entity e) ""
```

```
        «IF e.eContainer != null»
```

```
            package «e.eContainer.fullyQualifiedName»;
```

```
        «ENDIF»
```

Syntactic sugar for first parameter

"" =generated text is the default

```
        public class «e.name» «IF e.superType != null
```

```
            » extends «e.superType.fullyQualifiedName» «ENDIF»{
```

```
            «FOR f:e.features»
```

```
                «f.compile»
```

```
            «ENDFOR»
```

```
        }
```

Summary

Code generation - Summary

- Started from source code generation
 - UML -> Java, C++,
- Used in many other text based artifacts
 - document generation (web)
 - report generation (XML, XLS, CSV, print)
 - Configuration (wsdl)
- Strong tool support
 - Xpand
 - Xtend
 - (CodeDOM)
- There are some use cases outside of the MDE field