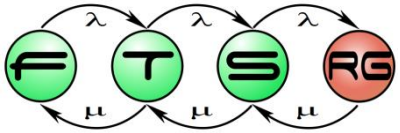
Code Generation

Ákos Horváth
Gábor Bergmann
Dániel Varró

Model Driven Systems Development



Agenda

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

Motivation

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

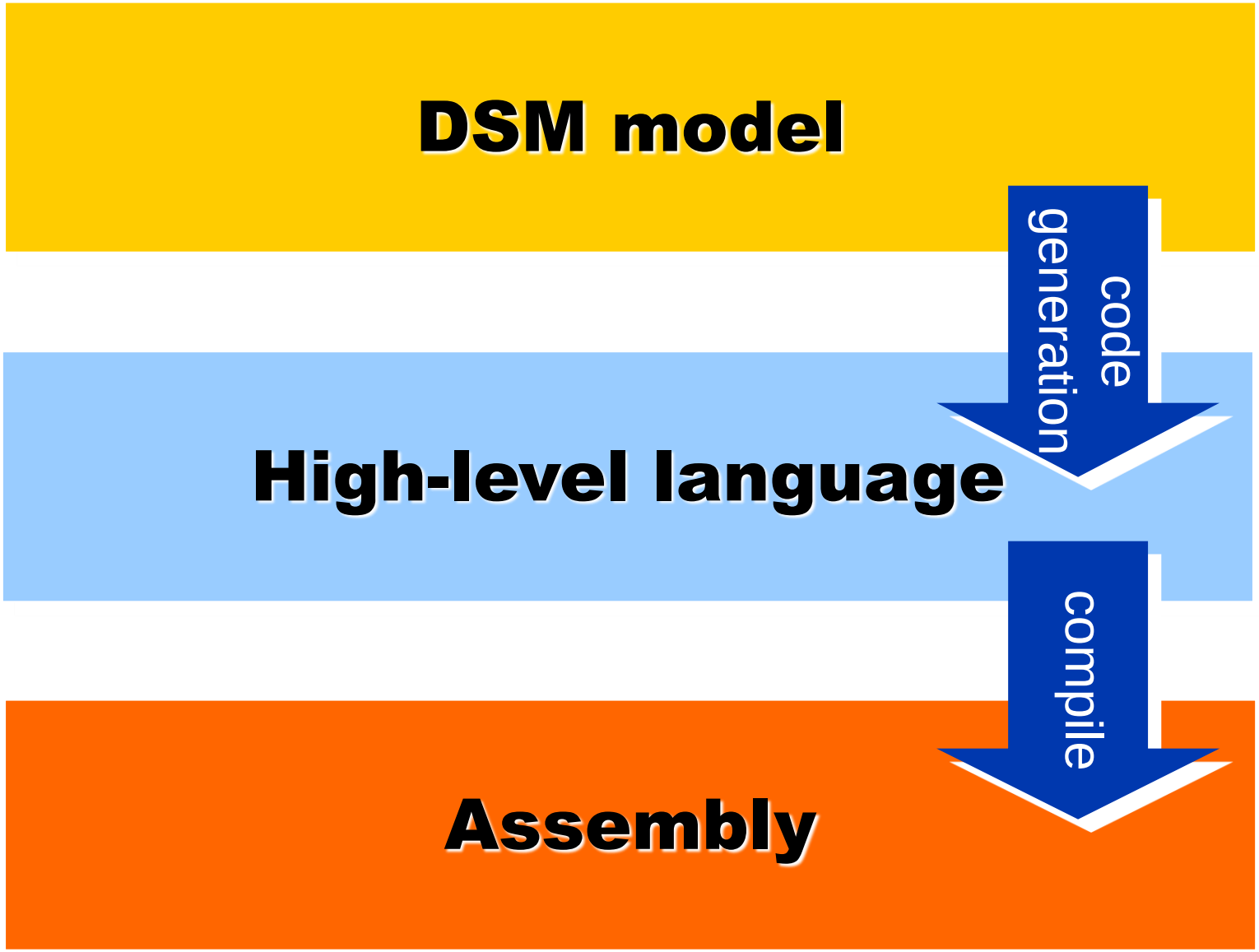
Similarity with compilers

- 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

Comparison against interpreters

- Dynamic Interpreter
  - Quick startup, short feedback
  - Typically introduces overhead (time, mem)
  - Runtime dependency on interpreter
  - May support changing model during execution
  - Always adheres exactly to model
- Code Generation
  - Start after generation & compilation only
  - May pre-optimize for efficient runtime code
  - With or without runtime dependencies
  - Model change requires re-generation
  - Code may be manually modified (is this good?)

Example: Source Code generation in MDD



Conceptual Approaches

Approaches

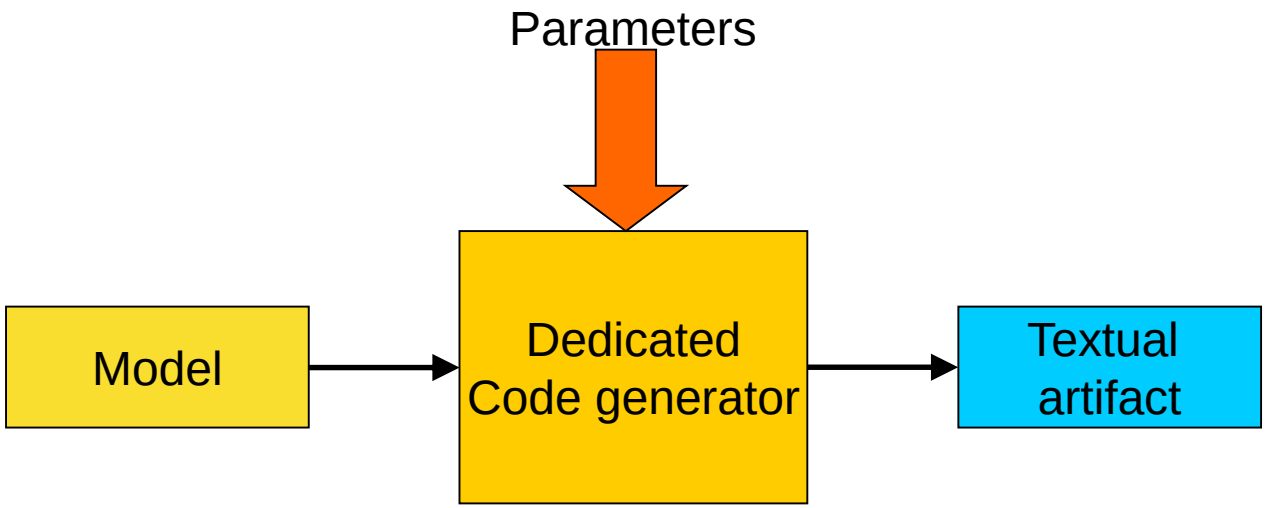
- Dedicated
  - Specific, ad-hoc
  - Using a dedicated code generator
- Template-based
- (Serializer-based)

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")
```

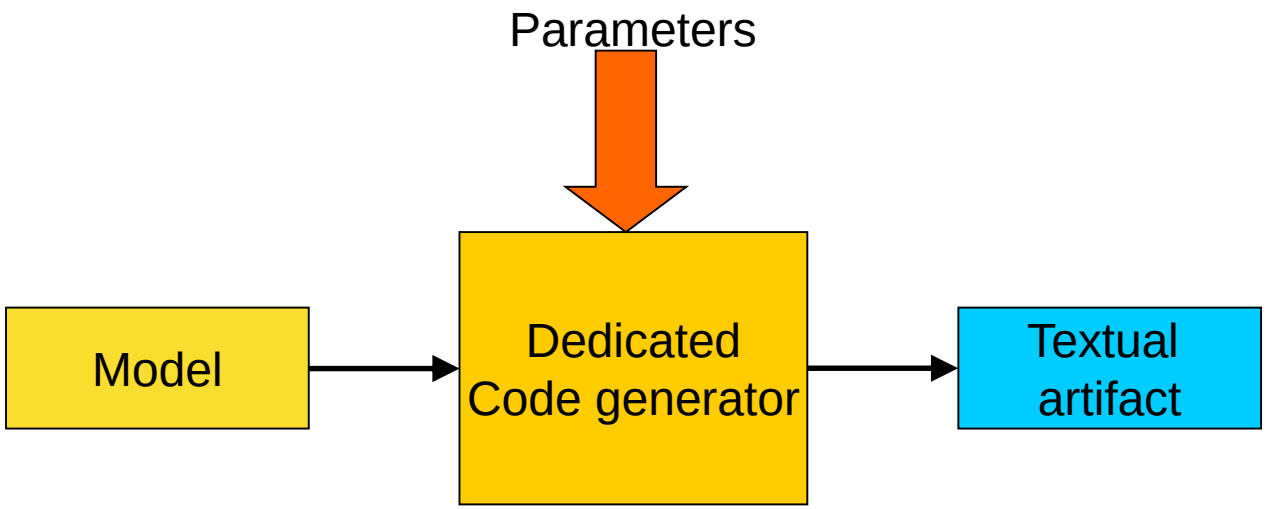
- Designed for the specific problem domain:
  - Best performance
  - Quick and dirty
  - Long development, hard maintainability
  - Zero reusability
  - Dedicated problem domains
    - Minimal changes during support cycle (safety critical embedded system, defense)
    - Certifiability
  - Example:
 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

Dedicated code generator



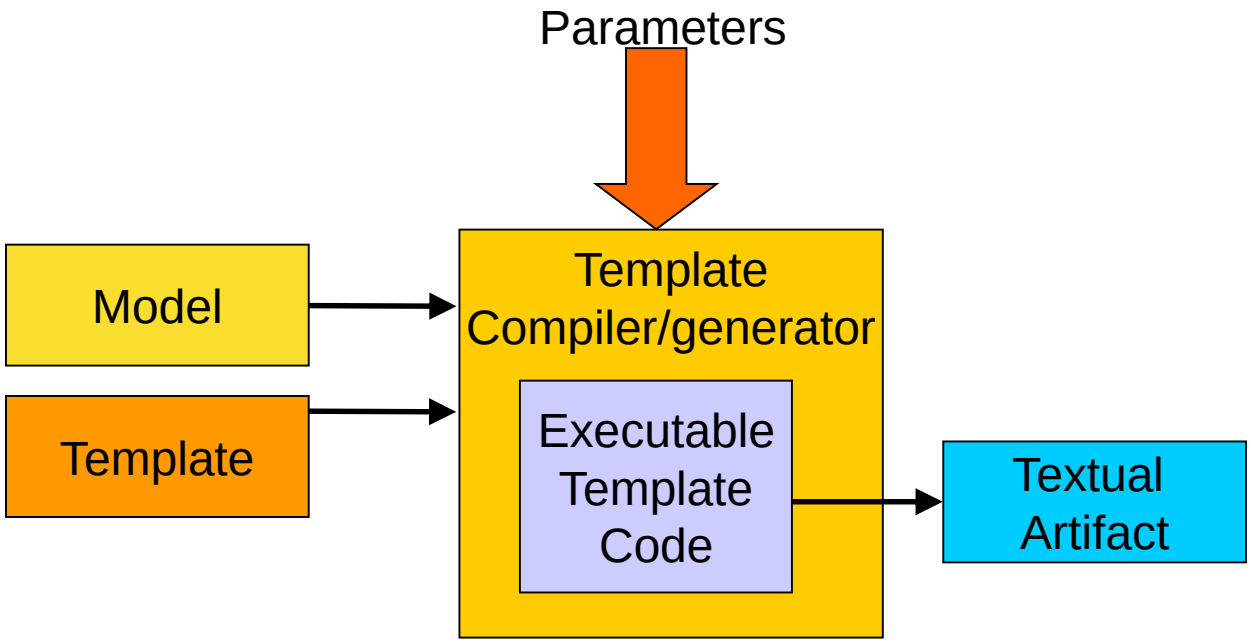
- Based on a framework:
  - Faster development time
  - Slower performance, better reusability
  - Embedded systems, moderate changes during project lifecycle

Dedicated code generator

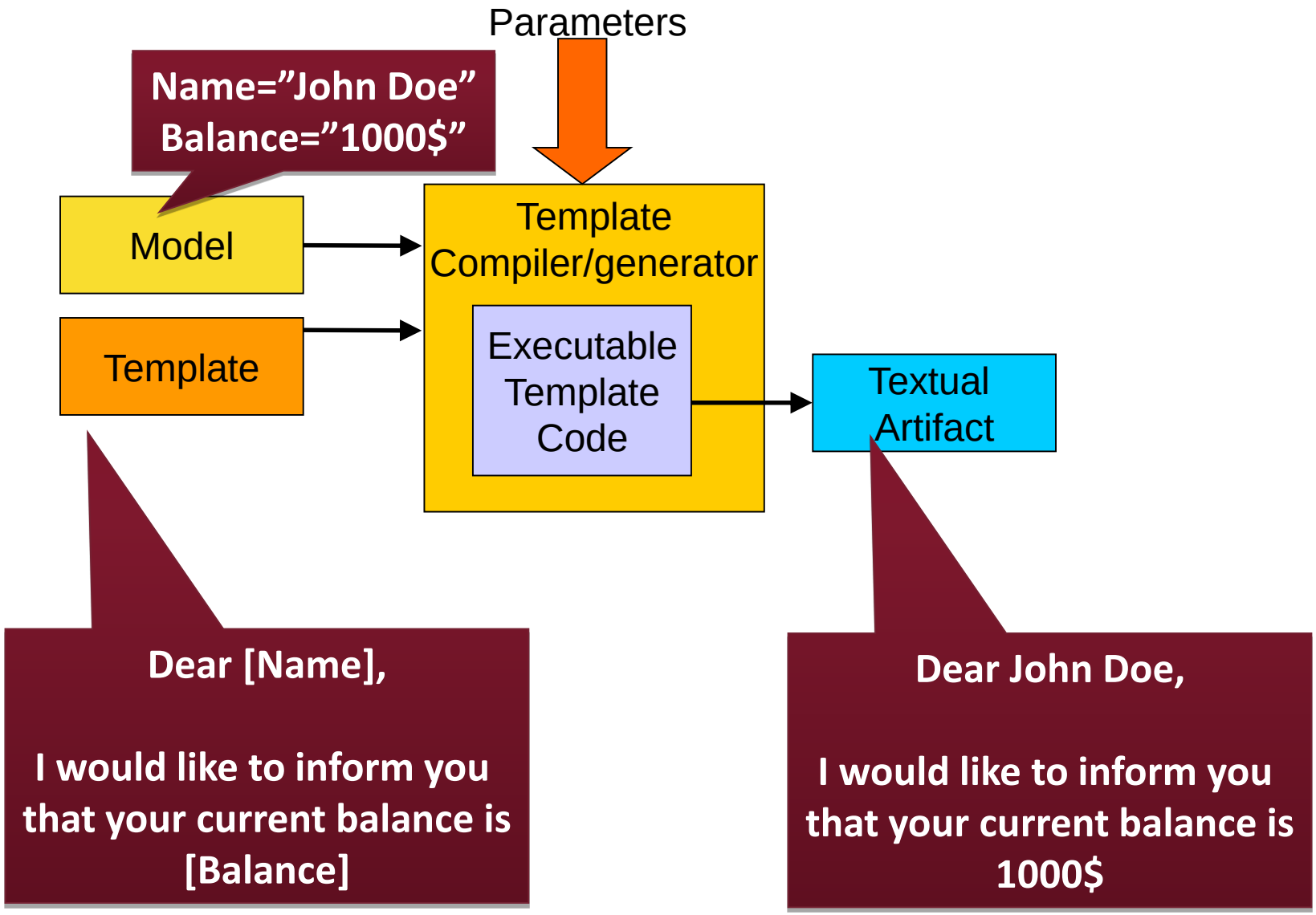


- Examples:
  - IBM Rational Software Architect
  - VASP (DO-178B Level A) Display graphics in avionics
  - Mathworks
  - Matlab Simulink
  - Esterel Scade suite

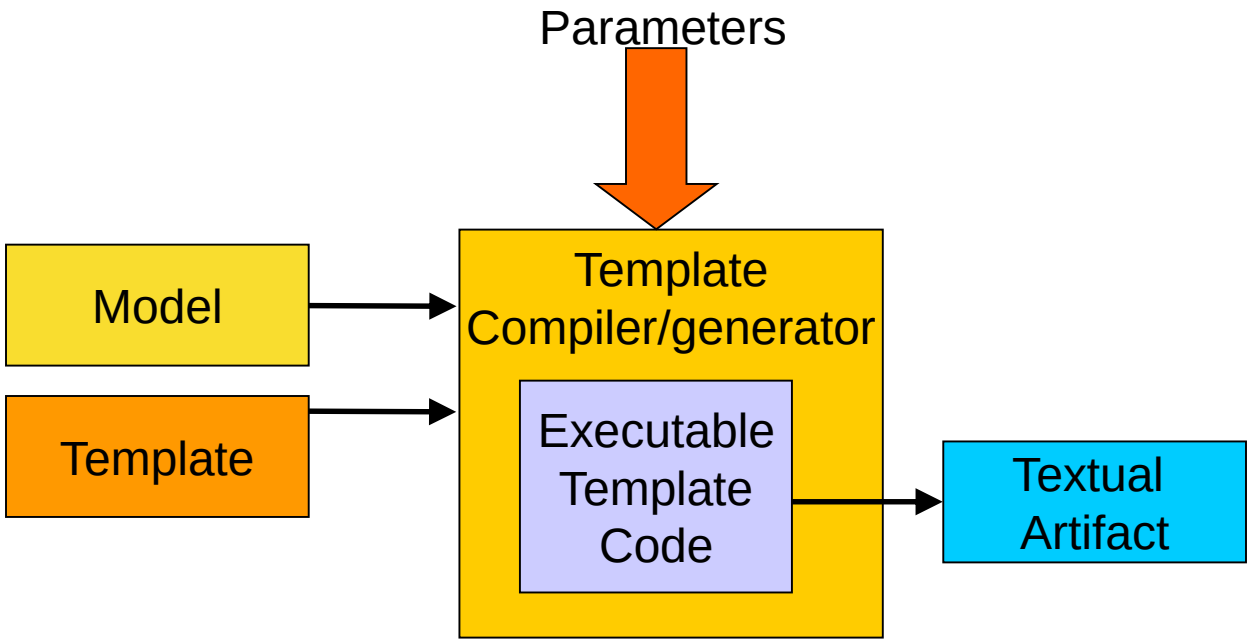
Template Based



Template Based

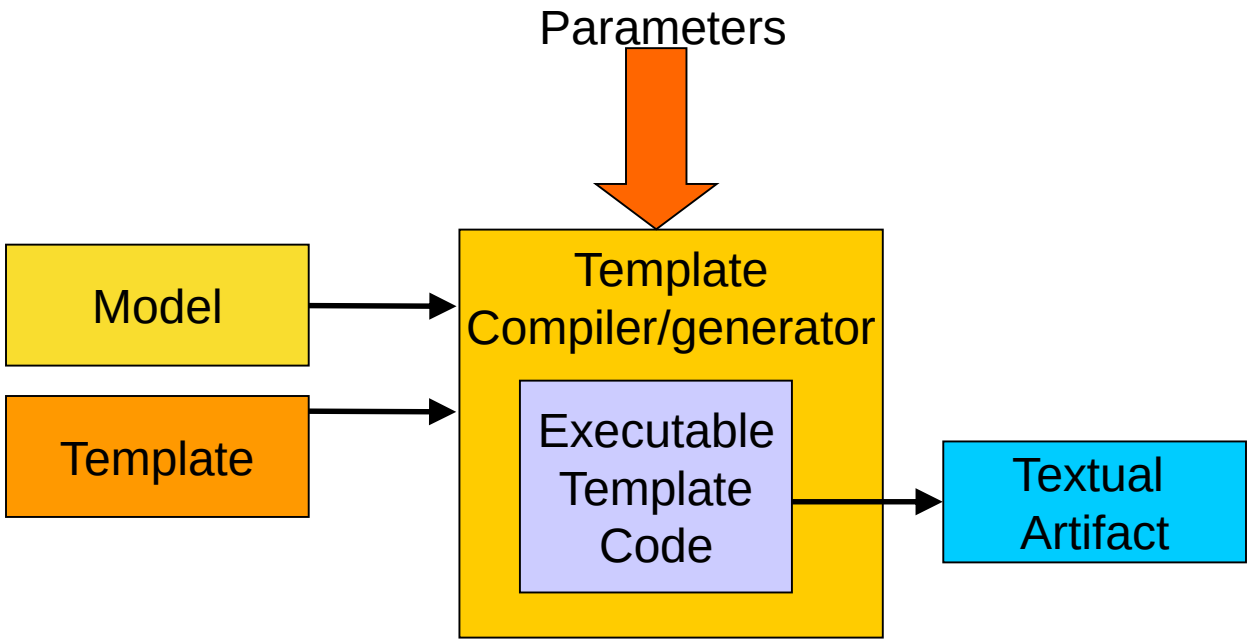


Template Based



- 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



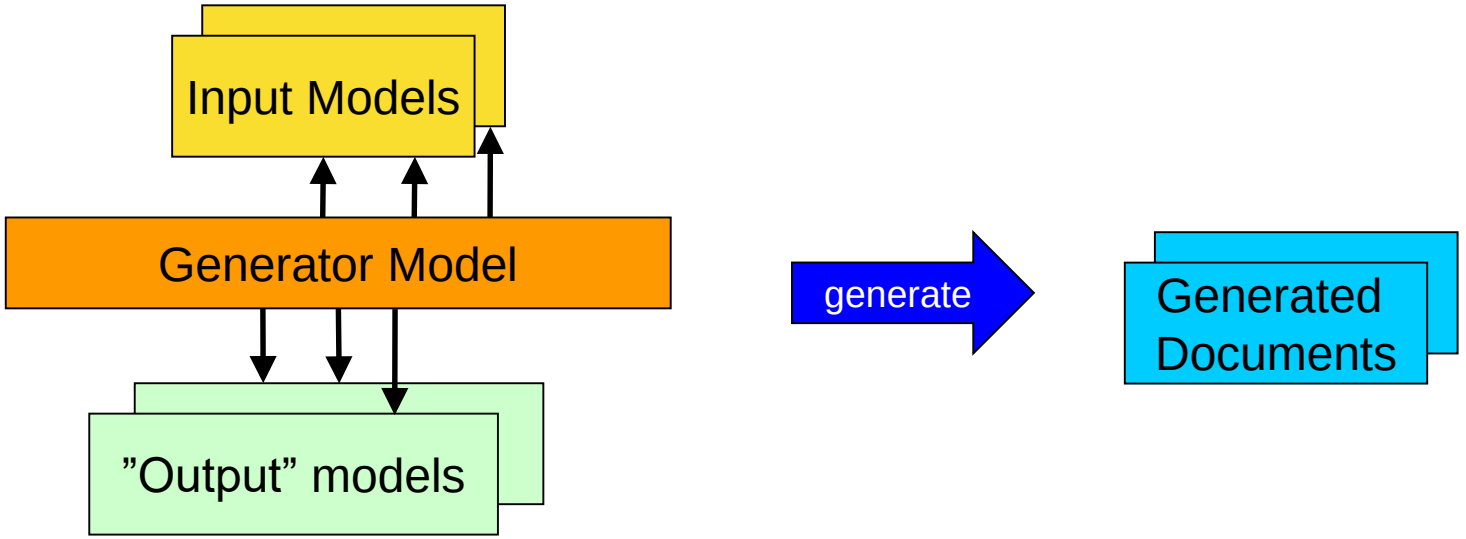
- Examples:
  - JET (for EMF models)
  - Velocity (/JSP)
  - **Xtend** (MDD approach)
  - AutoFilter (Kalman filters)
  - Smarty (php)

Serializer-based

- Idea: do not generate code, but DOM/AST instead
  - Needs DOM+serializer for the target textual language
    - Typically available for textual modeling languages
    - Also for many mainstream programming languages
  - Formatting + syntax difficulties handled by serializer ☺
    - Especially (qualified) cross-references, e.g. Java imports
    - Escaping (special characters) as well
  - Supports multiple iterations, non-linear logic, incrementality ☺
  - Not as practical for text-heavy target (e.g. documentation) ☹
- How to transform to DOM: see M2M lectures
  - Today's lecture focuses on actual text synthesis

Advanced Text Generation Issues

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 DOM/AST-based approaches
- Requires
  - Traceability between model and text
  - Model comparison (or change notifications)
  - 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 manual code parts (to be preserved)?
  - Model
    - Allows better reusability ☺
    - Increases complexity ☹
    - Text editing within model?
  - Template: for simple cases
  - DOM/AST: not user-facing ☹
  - Generated code files
    - Difficult, ad-hoc separation ☹
  - Separate code file / folder
    - VCS-friendly (diff, ignore) ☺
    - Cleaning is easy ☺

How to connect them?

- Interleave in same class
  - Java → no support ☹
  - C# partial classes ☺
- „Generation Gap” pattern ☺
  - Manual inherits from generated
- Manual invokes generated
  - Handling control difficult ☹

Code formatting

- Where to include?
  - Model
    - Does not follow typical MVC design paradigm
  - Templates
    - Simple formatting element
  - DOM/AST/CST
    - 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

Technologies

Xtend overview

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

https://www.eclipse.org/xtend/

Xtend example

```
import com.google.inject.Inject

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)
        }
 }
 def compile(Entity e) '''
     «IF e.eContainer != null»
      package «e.eContainer.fullyQualifiedName»;
     «ENDIF»
     public class «e.name»
      «IF e.superType != null»extends «e.superType.shortName» «ENDIF»
     {
     «FOR f:e.features»
      «f.compile»
     «ENDFOR»
     }
 '''
```

Full Java interop

Type inference

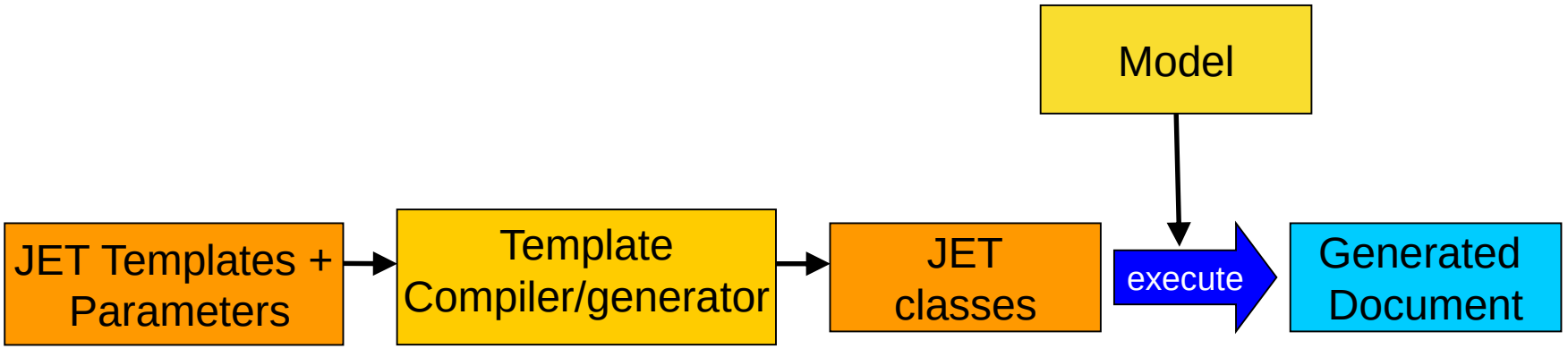
Syntactic sugar for first parameter

''' =template expression

String interpolation

In-template control structures

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 = (L      argument; %>

<?x   l version=    .0" encoding="UTF-8"?>
<de    >
<%  for (Ite                tList.iterator();
i.hasNext()    )  {
<element><%=i.next().toString()%></element>

<% } %>

</demo>
```

Package of representing class

Packages to import

Jet Header

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>

<% } %>

</demo>
```

Start of target document

JET example

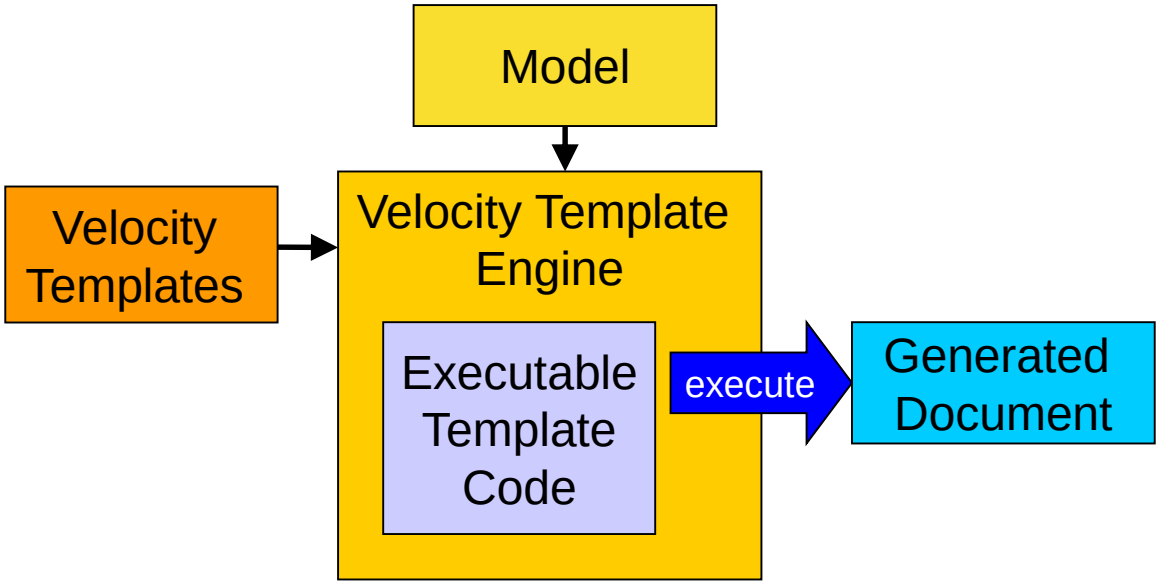
```
<%@ jet package="hello"  imports="java.util.*"
class="XMLDemoTemplate" %>
<% List elementList = (List) argument; %>

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

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

<% } %>

</demo>
```

Loop with the input parameter

Loop body

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>
#set( $tempString = "Element")
#foreach( $element in $elementList)
     <element> ${element.toString()} <element>
#end

</dem
```

New value of tempString

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

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
  - Xtend
  - (CodeDOM)
- There are some use cases outside of the MDE field