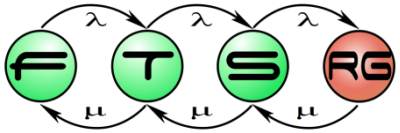
Code Generation

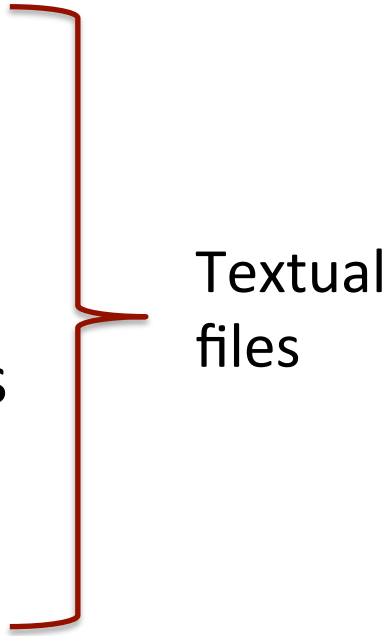


Designing modeling languages

- Metamodel: a model of models
  - Abstract syntax
  - Concrete syntax
  - Well-formedness rules
  - Behavioral (dynamic) semantics
  - **Translation to other languages**

Goals

- **Shorten development time**
- Based on model/requirement/plan
  - Documentation
  - Source code
  - Configuration files
  - Communication messages
  - Object serialization
  - …



Examples

Ecore model

- EMF model, edit and editor projects

GMF Diagram model

- GMF based editor

Xtext grammar (later)

- Textual editor

Generating Javadoc comments with JAutodoc

```
public void setNumberOfQuestions(int numberOfQuestions)
                                 throws IllegalArgumentException {
    if (numberOfQuestions < 0 ) {
        throw new IllegalArgumentException("numberOfQuestions < 0");
    }
    this.numberOfQuestions = numberOfQuestions;
}
```

Generating Javadoc comments with JAutodoc

```
/**
 * Sets the number of questions.
 *
 * @param numberOfQuestions the number of questions
 * @throws IllegalArgumentException the illegal argument exception
 */
public void setNumberOfQuestions(int numberOfQuestions)
                                 throws IllegalArgumentException {
    if (numberOfQuestions < 0 ) {
        throw new IllegalArgumentException("numberOfQuestions < 0");
    }
    this.numberOfQuestions = numberOfQuestions;
}
```

Generating Javadoc comments with JAutodoc

- Code generator (kind of)
  - Does not create knowledge
  - Only new representation
- Not the best example
  - See cost/benefit ration

```
/**
 * Sets the number of questions.
 *
 * @param numberOfQuestions the number of questions
 * @throws IllegalArgumentException the illegal argument exception
 */
public void setNumberOfQuestions(int numberOfQuestions)
                                 throws IllegalArgumentException {
    if (numberOfQuestions < 0 ) {
        throw new IllegalArgumentException("numberOfQuestions < 0");
    }
    this.numberOfQuestions = numberOfQuestions;
}
```

Generating Javadoc comments with JAutodoc

```
/**
 * Sets the number of questions.
 *
 * @param numberOfQuestions the number of questions
 * @throws IllegalArgumentException the illegal argument exception
 */
public void setNumberOfQuestions(int numberOfQuestions)
                                 throws IllegalArgumentException {
    if (numberOfQuestions < 0 ) {
        throw new IllegalArgumentException("numberOfQuestions < 0");
    }
    this.numberOfQuestions = numberOfQuestions;
}
```

Creating Textual Files

- Implementation of high level models
  - Runtime platform
- Decision points based on attributes (compromise)
  - Compatibility
  - Performance
  - Maintainability
  - Reusability

Creating Textual Files

- Implementation of high level models
  - Runtime platform
- Decision points based on attributes (compromise)
  - Compatibility
  - Performance
  - Maintainability
  - Reusability

Same design rules shall apply to both generated and manually written code

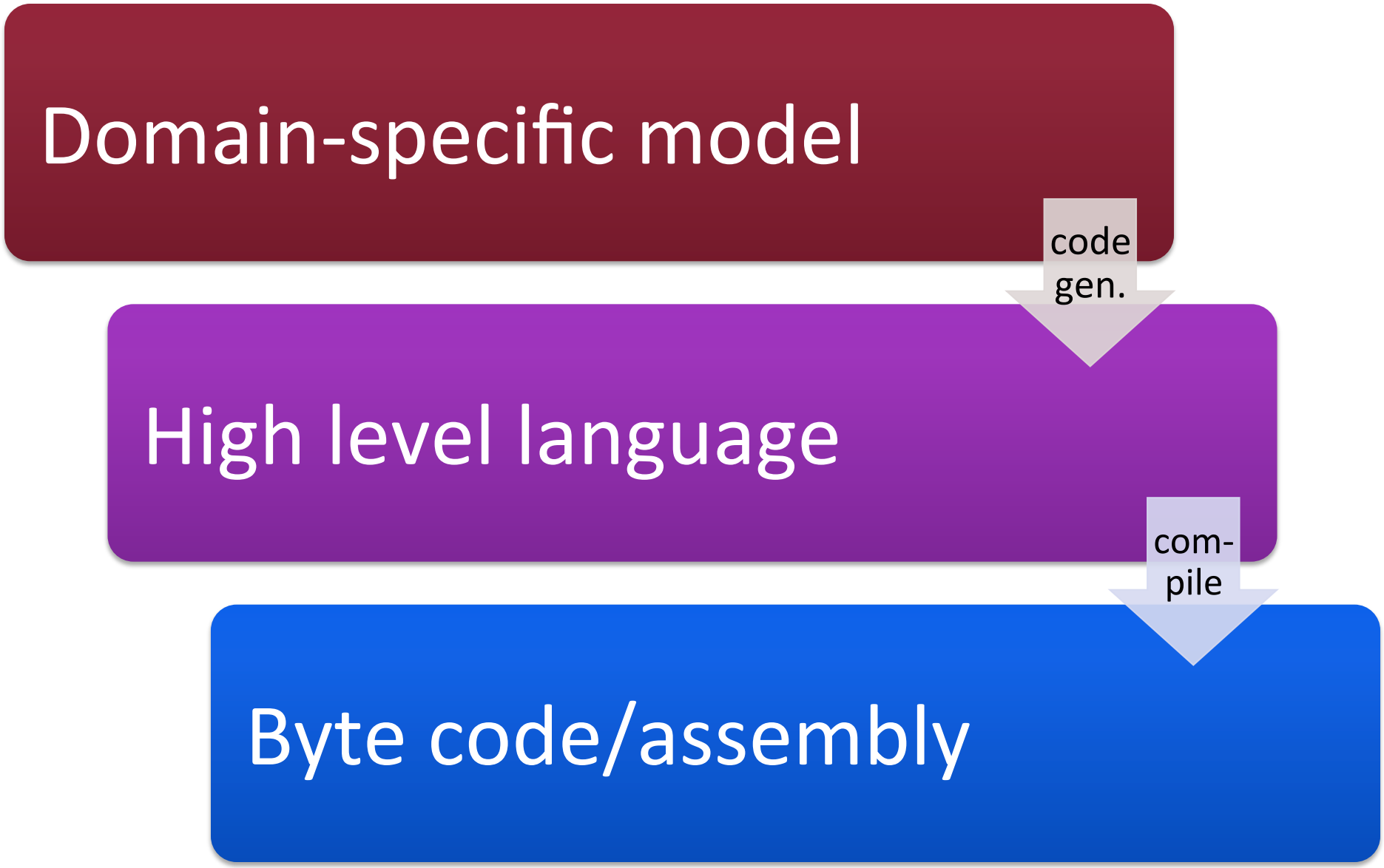
Creating Textual Files

- Implementation of high level models
  - Runtime platform
- Decision points based on attributes (compromise)
  - Compatibility
  - Performance
  - Maintainability
  - Reusability

Compilers

- Bridge between abstraction levels
  - E.g. C and assembly
- Reuse of design patterns
  - E.g. function calls in C
- Very similar to code generation, e.g.,
  - C preprocess
  - C++ templates
  - Automatically generated constructors/destructors

Code generation vs compiling



Code Generation Approaches

Approaches

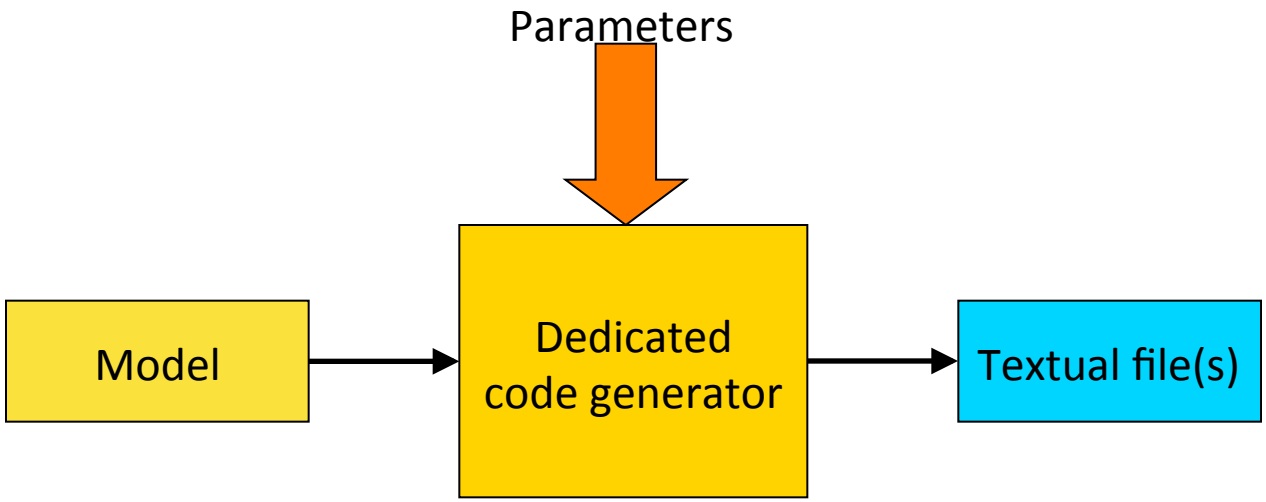
- Dedicated
  - Special, ad-hoc
  - Generic code generator based
- Template based

Special, Ad-hoc Generator

```
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")
```

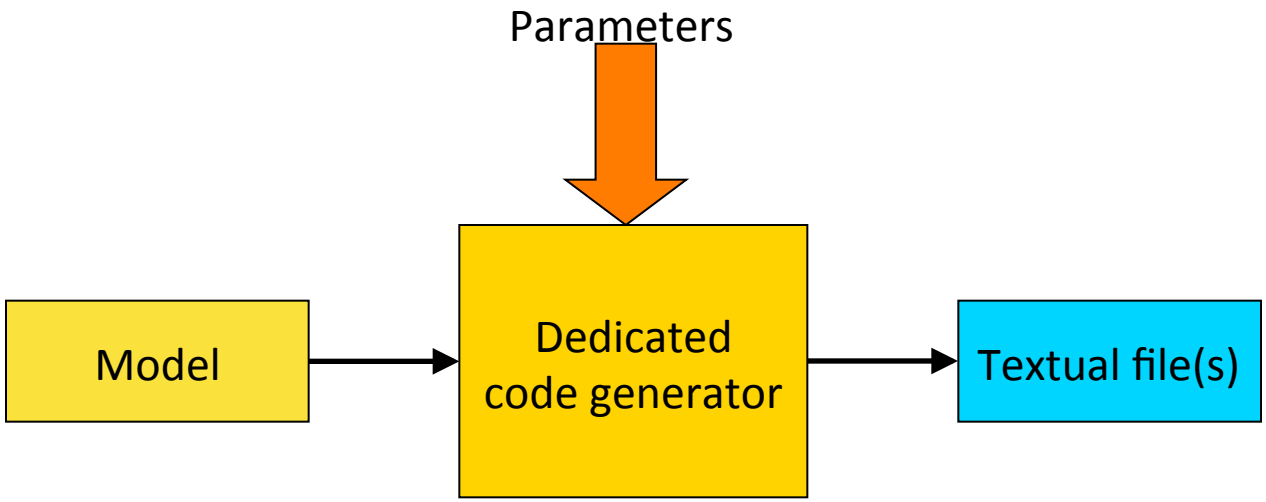
- Optimized for domain
- Best performance
- „Quick-and-dirty” solution
- Long term: problematic maintenance, no re-use
- For special domains
  - Minimal changes during life (safety critical embedded systems, security)
  - Certifiability
  - Example: ARINC653 Multistatic configuration generator (Python script)

Dedicated Code Generator



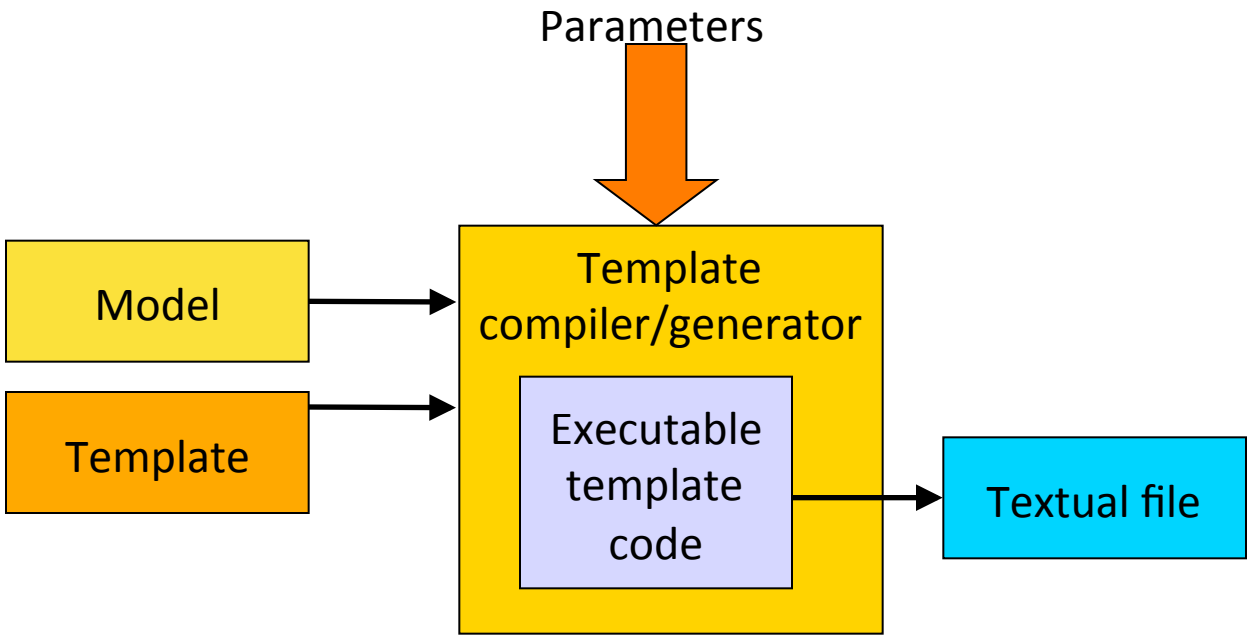
- Based on code generation framework
  - Faster development
  - Worse performance, better reusability
  - Medium amount of change during lifecycle
    - E.g. embedded systems

Dedicated Code Generator

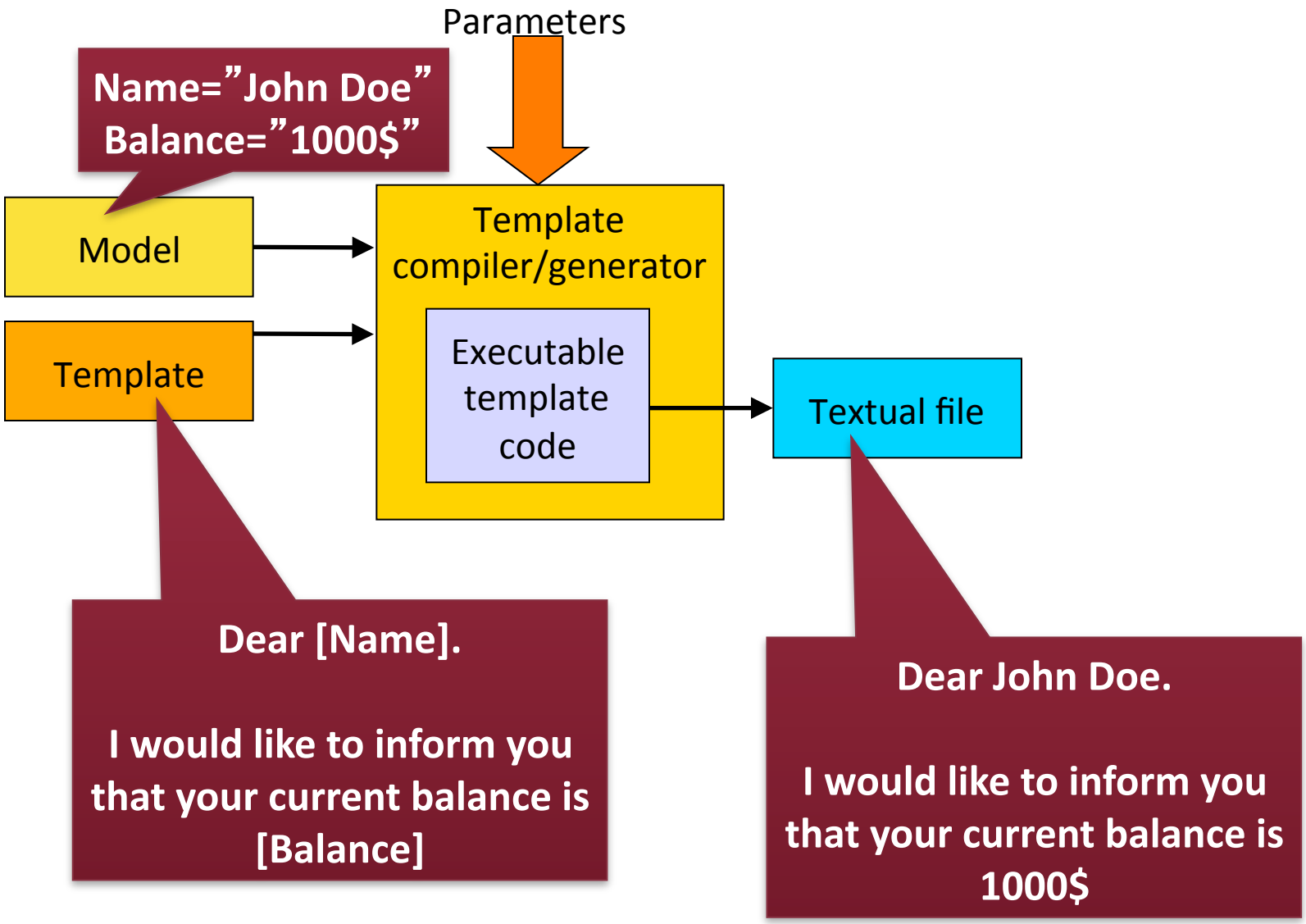


- Examples:
  - IBM Rational Software Architect
  - VASP (DO-178B Level A) Display graphics in avionics
  - Mathworks
  - Matlab Simulink
  - Esterel Scade suite

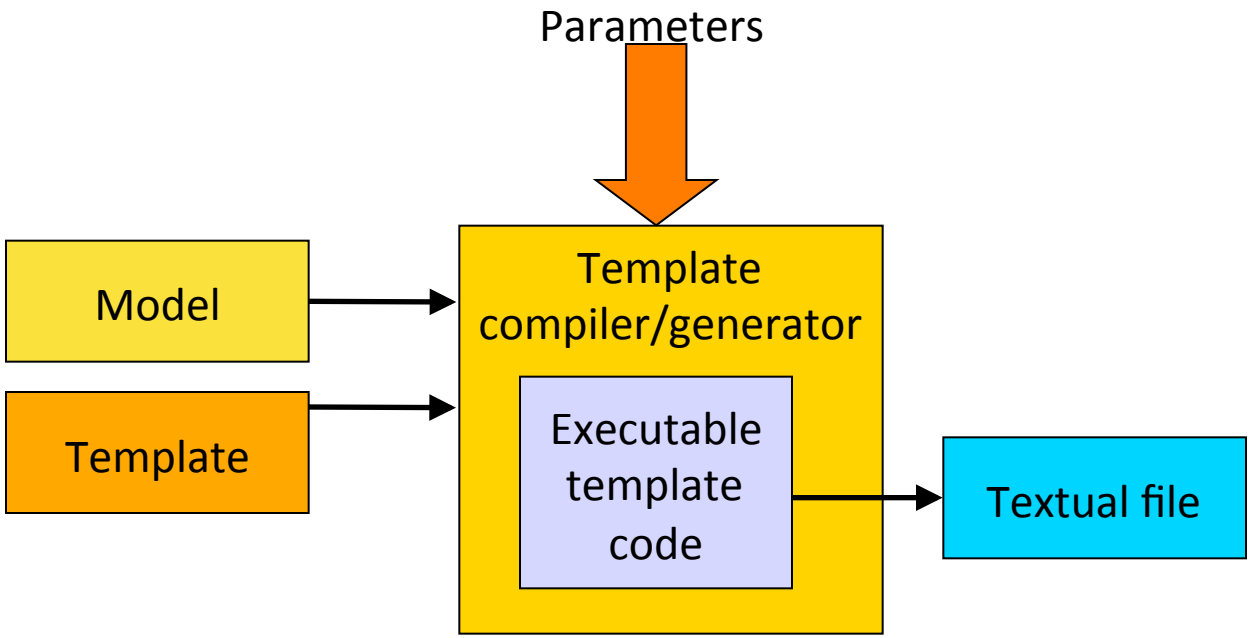
Template based



Template based

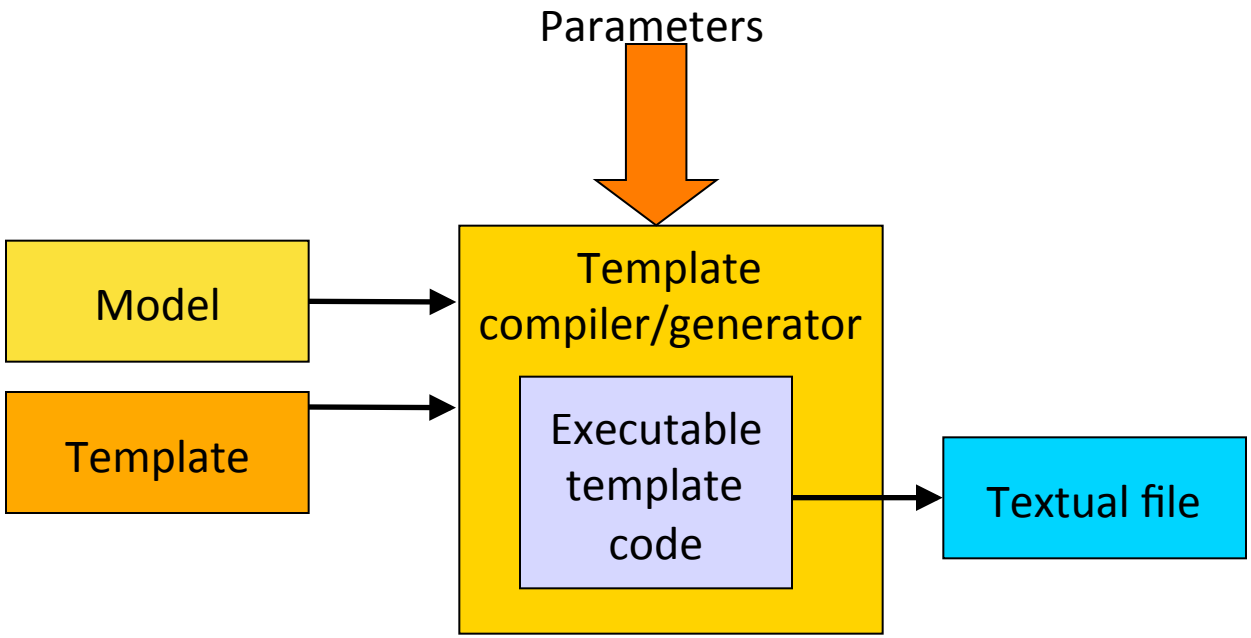


Template based



- Fastest development
- Worst performance, best reusability
- Rapidly changing environment (e.g. web technologies)
- Complex changes during lifecycle
  - Model and template can be changed separately

Template based

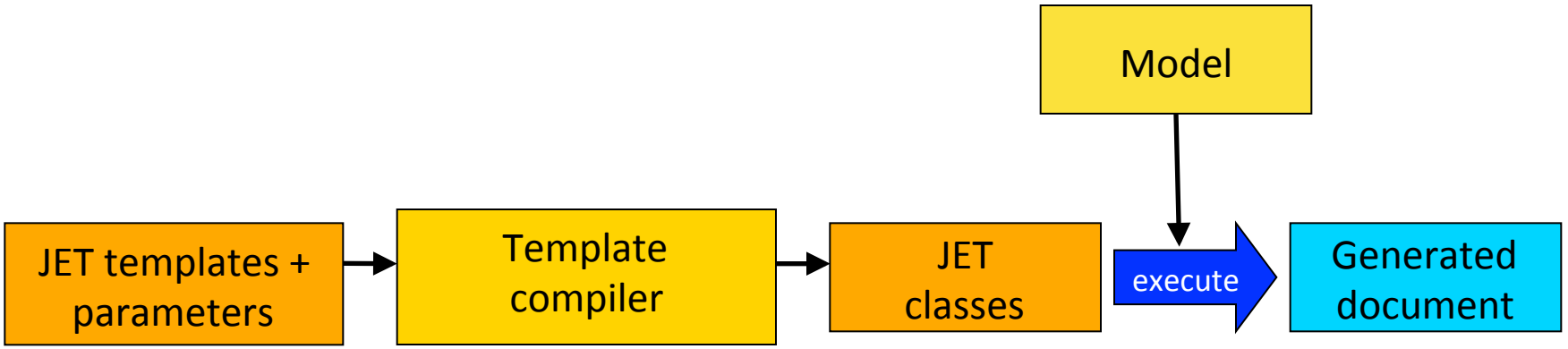


- Examples:
  - JET (for EMF models)
  - Velocity (/JSP)
  - OpenArchitectureWare/ XPand (MDD approach)
  - Xtend
  - AutoFilter (Kalman filters)
  - Smarty (php)

Code generators for EMF models

Java Emitter Templates (JET),
Acceleo és Xtend

Java Emitter Templates (JET)



- JSP-like template language with Java control structure
- **Translated to Java code**
- Output format: text
- Parameters: Java objects
  - Created for EMF, but not EMF-specific
  - EMF code generation uses JET

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

<% } %>

</demo>
```

Jet Header

JET example

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

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

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

<% } %>

</demo>
```

Package of representing class

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 = elementL
i.hasNext(); ) { %>
<element><%=i.next().toString()%></element>

<% } %>

</demo>
```

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>
```

Packages to import

JET example

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

<?x                    1.0" encoding="UTF-8"?>
<demo>

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

<% } %>

</demo>
```

Start of code section

JET example

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

<?xml version="1.0" encod"?>
<demo>

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

<% } %>

</demo>
```

Input parameter

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.iterato;
i.hasNext(); ) { %>
<element><%=i.next().toString()%></element>

<% } %>

</demo>
```

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" encoding="UTF-8"?>
<demo>

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

<% } %>

</demo>
```

Loop with the input parameter

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>
```

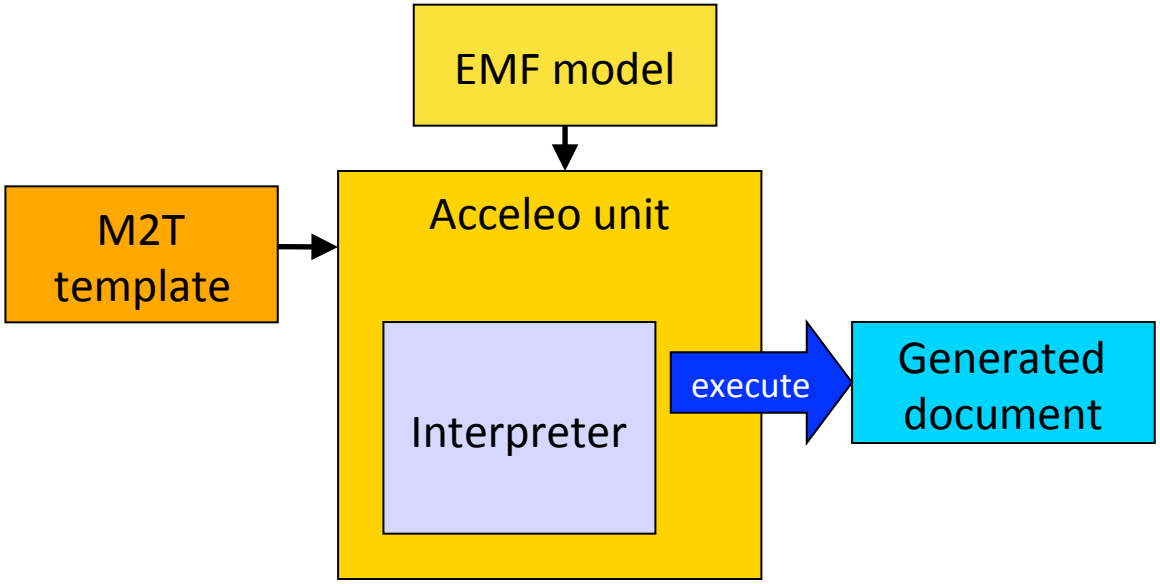
Returns value of the argument

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>
```

Loop body

Acceleo



- OMG M2T implementation
  - EMF model
  - OCL evaluation (EMF-OCL dependency)
- Interpreted template
- Simple control flow
  - if-else, for and let

Acceleo example

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/emf/2002/Ecore')/]

[template public generate(e : EClass)]

[comment @main /]
[file (e.name, false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
  <demo name=”[e.name/]">
    [for (it : EAttribute | e.eAttributes)]
      <attr name="[it.name/]" />
      [/for]
  </demo>
[/file]

[/template]
```

Acceleo example

Import EMF metamodel

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/emf/2002/Ecore')/]

[template public generate(e : EClass)]

[comment @main /]
[file (e.name, false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
  <demo name=”[e.name/]">
    [for (it : EAttribute | e.eAttributes)]
      <attr name="[it.name/]" />
      [/for]
  </demo>
[/file]

[/template]
```

Acceleo example

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/emf/2002/Ecore')/]

[template public generate(e : EClass)]

[comment @main /]
[file (e.name, false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
  <demo name=”[e.name/]">
    [for (it : EAttribute | e.eAttributes)]
      <attr name="[it.name/]" />
      [/for]
  </demo>
[/file]

[/template]
```

Template

Acceleo example

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/emf/2002/Ecore')/]

[template public generate(e : EClass)]

[comment @main /]
[file (e.name, false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
  <demo name=”[e.name/]">
    [for (it : EAttribute | e.eAttributes)]
      <attr name="[it.name/]" />
      [/for]
  </demo>
[/file]

[/template]
```

Entry point

Acceleo example

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/emf/2002/Ecore')/]

[template public generate(e : EClass)]

[comment @main /]
[file (e.name, false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
  <demo name=”[e.name/]">
    [for (it : EAttribute | e.eAttributes)]
      <attr name="[it.name/]" />
      [/for]
  </demo>
[/file]

[/template]
```

File information

Acceleo example

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/emf/2002/Ecore')/]

[template public generate(e : EClass)]

[comment @main /]
[file (e.name, false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
  <demo name=”[e.name/]">
    [for (it : EAttribute | e.eAttributes)]
       <attr name="[it.name/]" />
       [/for]
  </demo>
[/file]

[/template]
```

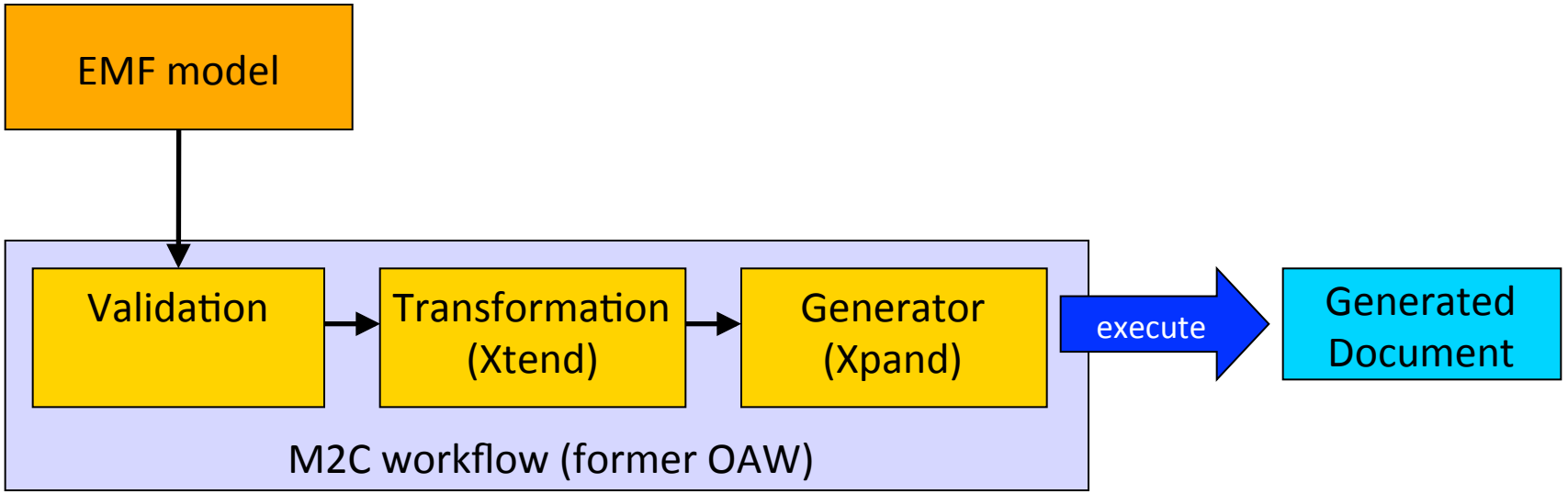
Reference

Acceleo example

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/emf/2002/Ecore')/]

[template public generate(e : EClass)]

[comment @main /]
[file (e.name, false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
  <demo name=”[e.name/]">
    [for (it : EAttribute | e.eAttributes)]
      <attr name="[it.name/]" />
      [/for]
  </demo>
[/file]

[/template]
```

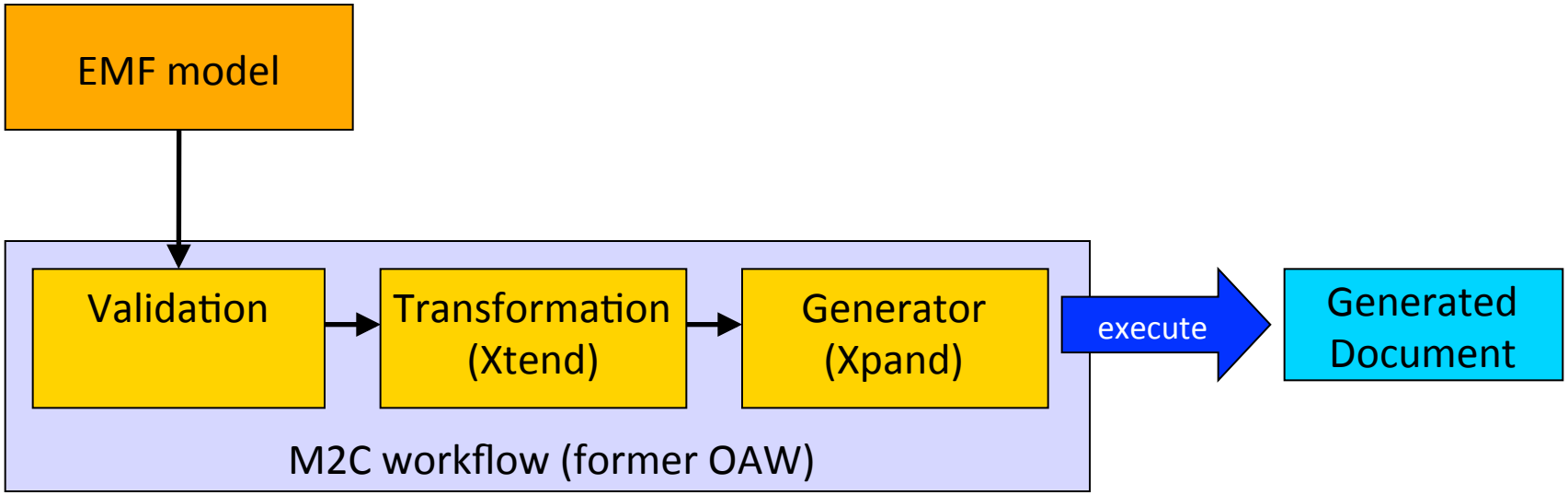
For structure

Xpand (v1)



- Eclipse M2C (formerly OAW)
  - Full M2C workflow
    - Validation
    - Transformation (Xtend(1) language)
    - Code generation (Xpand language)
- Focuses on EMF models
- Flexible workflow definition

Xpand (v1)



- **Interpreted**
- Statically typed template language
- Polimorph template calls
- Support for aspect-oriented programming
- Error handling
- Specifies non-visible characters ☺

Xpand example

```
«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 example

```
«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»
```

Import EMF metamodel

Xpand example

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

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

Define template for specific type

Xpand example

```
«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»
```

Output file definition

Xpand example

```
«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»
```

Start of target document

Xpand example

```
«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

Xpand example

```
«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»
```

Invoke other template with type definition

Xtend2

- Designed as Java-extension language
  - Compiles to Java classes
  - Multiple functions
- Especially well usable for
  - EMF model processing
  - Code generation

Xtend2

- New language features
  - Type inference
  - Lambda expressions (closures)
  - Extension methods
  - Template strings

Lambda expressions

- Example: Collect all person names!
- *Java*

```
List<String> names = new ArrayList<String>();
for (String name : persons) {
 names.add(name);
}
```

- *Xtend*

```
persons.map( p | p.name )
```

Example: Template strings

```
@GET
@Produces("text/html")
def sayHighToEverybody(@PathParam("names") String names) '''
    <!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//EN">
    <HTML>
       <HEAD>
          <TITLE>
             Hello «names»!
          </TITLE>
       </HEAD>
    <BODY>
       «FOR name : names.split(',')»
          «greet(name)»
       «ENDFOR»
    </BODY>
    </HTML>
'''

def private greet(String name) '''
        <H1>Hi «name»!</H1>
'''
```

More complex Xtend example

```
html [
  head [
   title [$("XML encoding with Xtend")]
  ]
  body [
   h1 [$("XML encoding with Xtend")]
   p [$("this format can be used as an alternative to XML")]

    // an element with attributes and text content
   a("http://www.xtend-lang.org") [$("Xtend")]

    // mixed content
   p [
     $("This is some")
     b[$("mixed")]
     $("text. For more see the")
     a("http://www.xtext.org")[$("Xtext")]
     $("project")
   ]
   p [$("some text")]

   // content generated from arguments
   p [
    for (arg : args)
      $(arg)
   ]
  ]
]
```

Xpand 1 vs Xtend 2

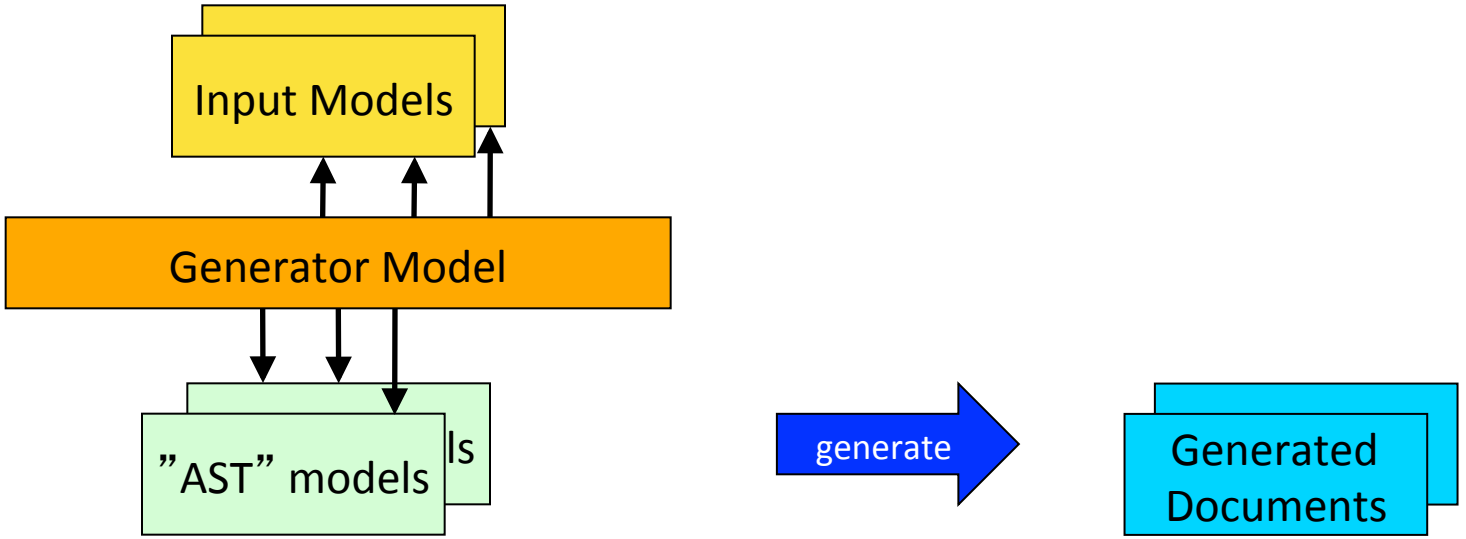
- Xpand 1 is older
  - Explicit workflow support
  - Weaker editor support
- Xtend 2
  - Does not focus on code generation
  - Compiles to Java
    - Easier to integrate
    - No direct code generation workflow support

Advanced issues

Error handling

- Debugging the code generator
  - Requires the knowledge of source and target language
  - Avoid it (at all costs)!
- Source level validation
  - Static
    - Language-specific well-formedness constraints
  - During generation
    - E.g. target language keyword handling

Generator model



- Multiple source models → **”generator” model**
- Code generation settings
- Refers to input (and possibly to generated AST models)
- Handling model hierarchies (multiple models, packages, etc.)
- Non-linear, multirun code generation
- Can store traceability links
- Multiple output streams

Target: source code or AST

	Source code	AST
Output	Text	Program structure
Complexity	Low	Complex
Development	Fast	Slow
Execution	One step Linear	Multiple steps Non-linear
Incrementality	Only on file level	Possible even inside files
“Pretty printing”	Postprocessing	During AST serialization
Model-code synchronization	?	AST synchronization
Evaluation	**Quick solution** **For simple cases**	**For complex cases**

Target: source code or AST

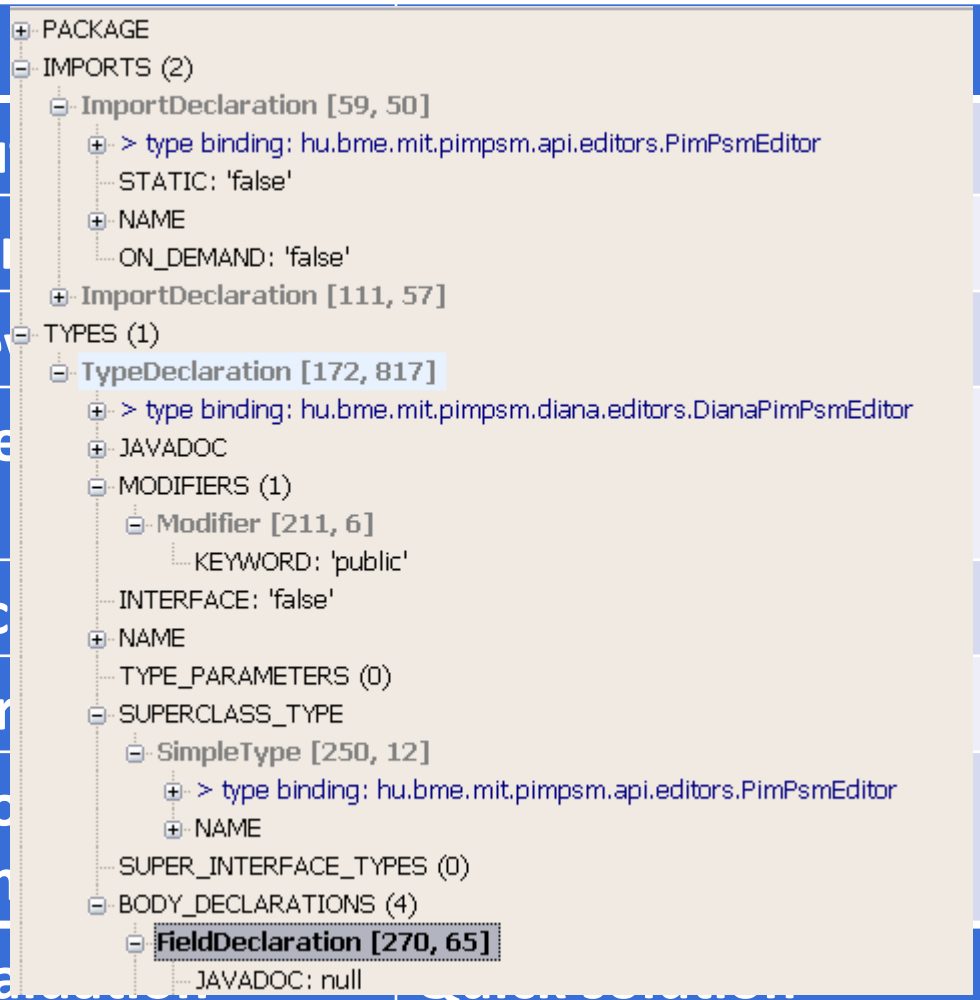
	Source code	AST
Output	Text	
Complexity	Low	
Development	Fast	
Execution	One step Linear	
Incrementality	Only on file level	
“Pretty printing”	Postprocessing	
Model-code synchronization	?	
Evaluation	Quick solution For simple cases	For complex cases

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

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

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

    public static final String PLUGIN_ID = "hu.mi

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

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

Target: source code or AST

	Source code	AST
Output	Text	Program structure
Complexity	Low	Complex
Development	Fast	Slow
Execution	One step Linear	Multiple steps Non-linear
Incrementality	Only on file level	Possible even inside files
“Pretty printing”	Postprocessing	During AST serialization
Model-code synchronization	?	AST synchronization
Evaluation	**Quick solution** **For simple cases**	**For complex cases**

Target: source code or AST



	AST
Ou	Program structure
Co	Complex
De	Slow
Exe	Multiple steps Non-linear
Inc	Possible even inside files
“Pr	During AST serialization
Mo syn	AST synchronization
Eva	
For simple cases	**For complex cases**

Target: source code or AST

	Source code	AST
Output	Text	Program structure
Complexity	Low	Complex
Development	Fast	Slow
Execution	One step Linear	Multiple steps Non-linear
Incrementality	Only on file level	Possible even inside files
“Pretty printing”	Postprocessing	During AST serialization
Model-code synchronization	?	AST synchronization
Evaluation	**Quick solution** **For simple cases**	**For complex cases**

Modell-code synchronization

- Problem:
  - If generated code changes, update source model
  - M2C synchronization
  - Requires AST generation
- Preconditions
  - Traceability information between model and AST (and text)
  - Model comparison support
  - Change localization support
- Allows incremental model building for better performance
- Examples
  - Eclipse JDT: Java source and AST
  - EMF: generator model

Code formatting

- Where to define formatting rules
  - In model
    - Not an MVC paradigm
  - In template
    - Element-level formatting
  - In AST
    - Stores all information
    - Can be complex
- Or a better alternative
  - Extra step in code generation workflow
  - Use external tool possible
    - Eclipse JDT formatter
    - XML DOM serializer

Keywords and special characters

- Reserved keywords in target language
  - Java: abstract, class, …
  - XML: ‘<‘ , ‘>’
  - …
- Validate model before code generation
  - If complex → extra workflow step
  - Examples
    - Java: isJavaIdentifierStart() (in Character)
    - EMF validation
- Escaping
  - In model, or
  - Only in generated code

Initializing code generation

- Manually
  - So far everything
- Automatically on model changes
  - Eclipse builder mechanism
- Which one to choose?

Initializing code generation

Manually

- Complex steps
- Workflow driven approach
- Might be slow (relatively)

On model change

- Simpler steps
- Change detection
  - More complex control
- **Must** be fast

Initializing code generation

Manually

- Complex steps
- Workflow driven approach
- Might be slow (relatively)

This approach goes for build automatization

In IDEs this is practical

On model change

- Simpler steps
- Change detection
  - More complex control
- **Must** be fast

Projects with generated code

Model-driven Projects (Utopia)

Model

Generated code

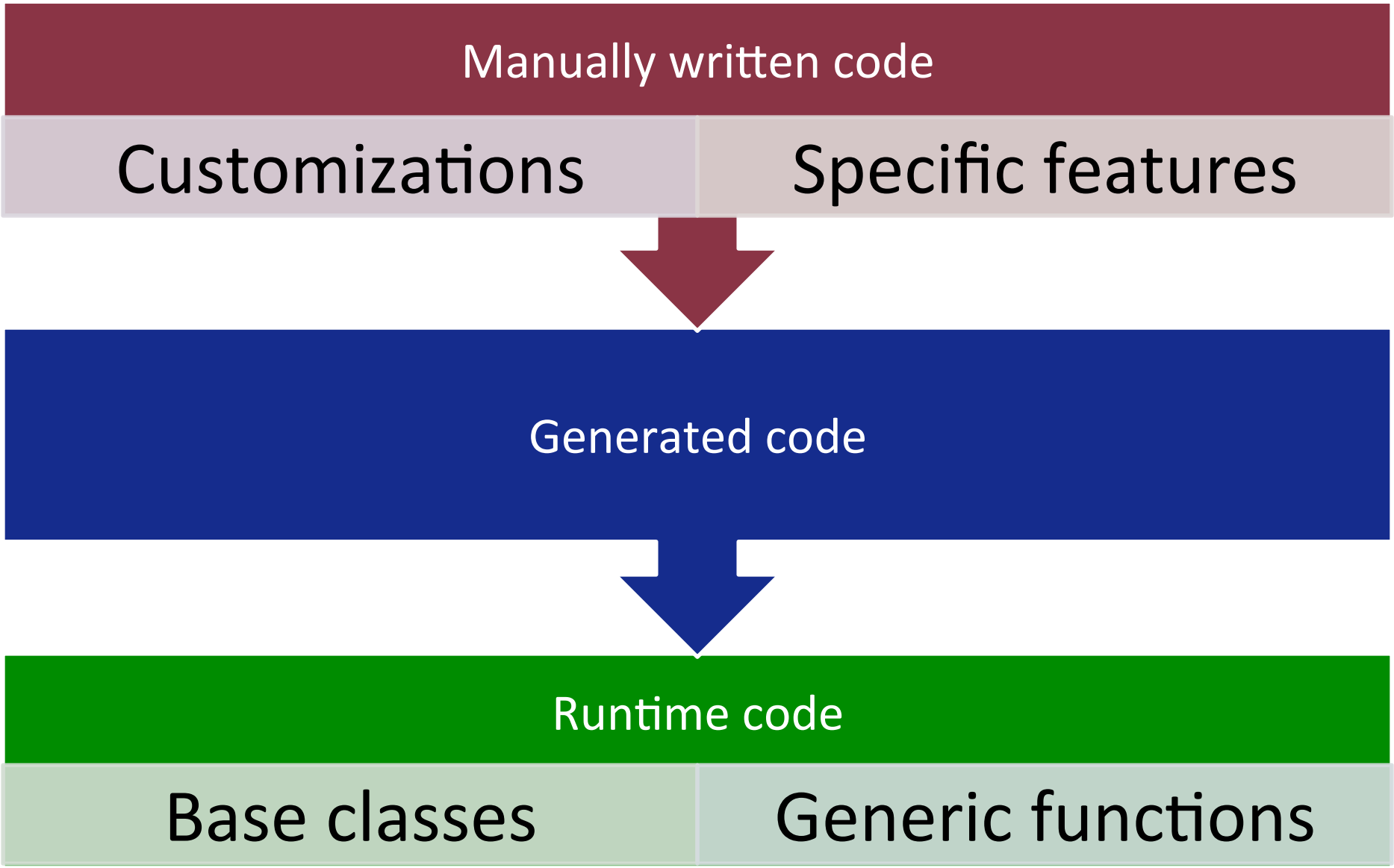
Model-driven Projects – Take 2

Model

Generated Code

Manually written code

Even more realistic architecture



Code Quality

- Generated code indistiguishable from manual
  - Even if not written, we read it!
  - Formatted code
  - Code documentation
- Use a well-designed target runtime
  - Less generated code is required
  - Own runtime: abstract base classes help

Code placement strategy

- Eclipse Java project
  - Multiple source folders
    - Manual code
    - One for each code generator
  - Eclipse compiler will merge them

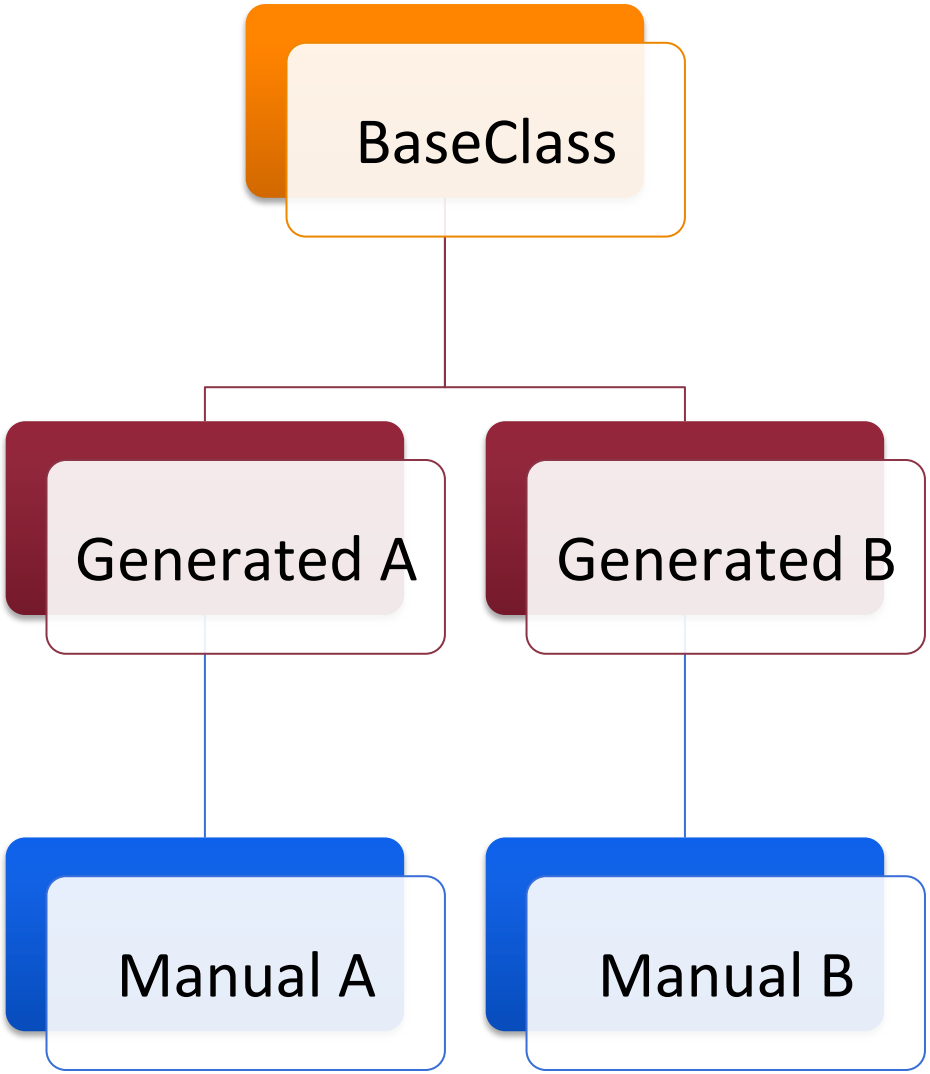
Rules of thumb – 1.

- Do not store generated code in version control
  - Reproducible
  - Unnecessary conflicts
    - E.g. formatting, line endings…
- Code generation is part of the build
  - Including automatic builds
  - Model and code should always be synchronized

Rules of thumb – 2.

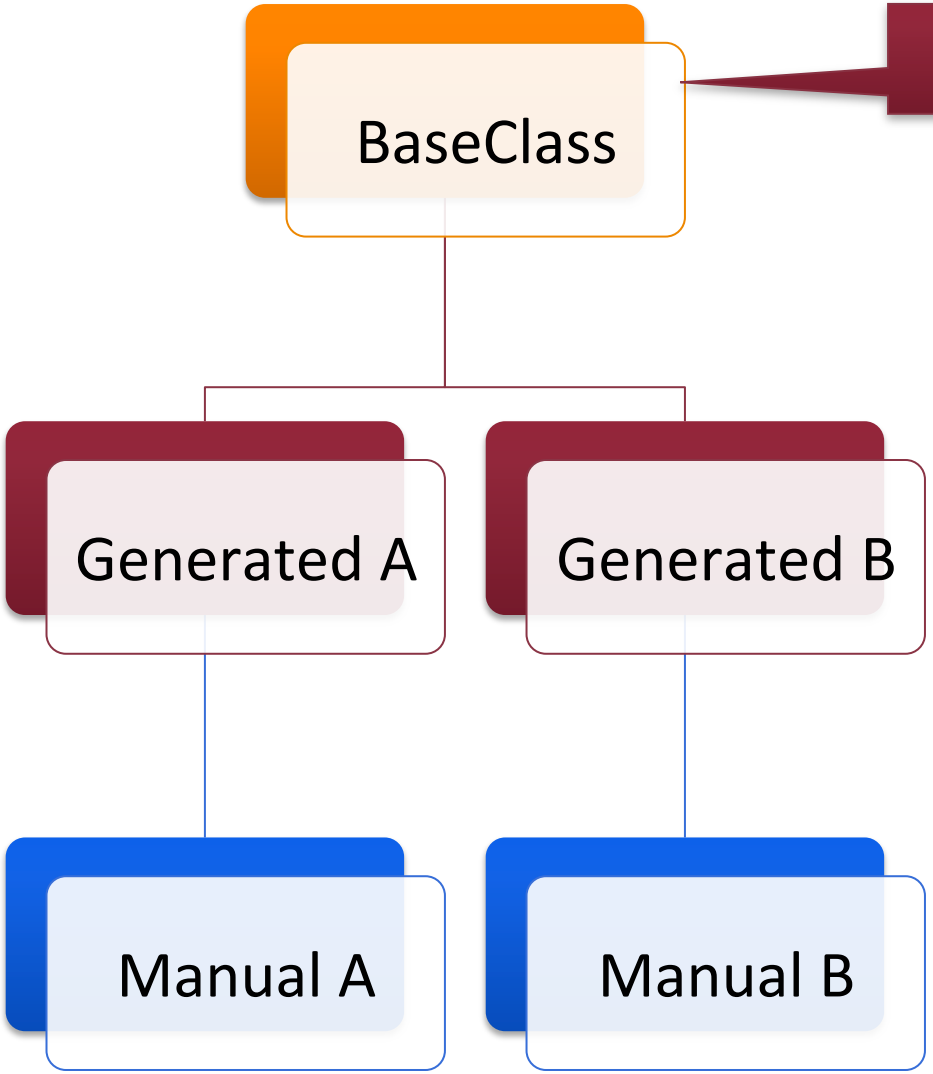
- New file: either written or generated
  - Code generation can overwrite manual code ☺
    - See: EMF
  - Manageable
  - Easier to separate (e.g. for version control)
- How to store customizations?
  - In C#: partial classes help
  - In Java: some design patterns should be followed

Generation Gap pattern – 1.



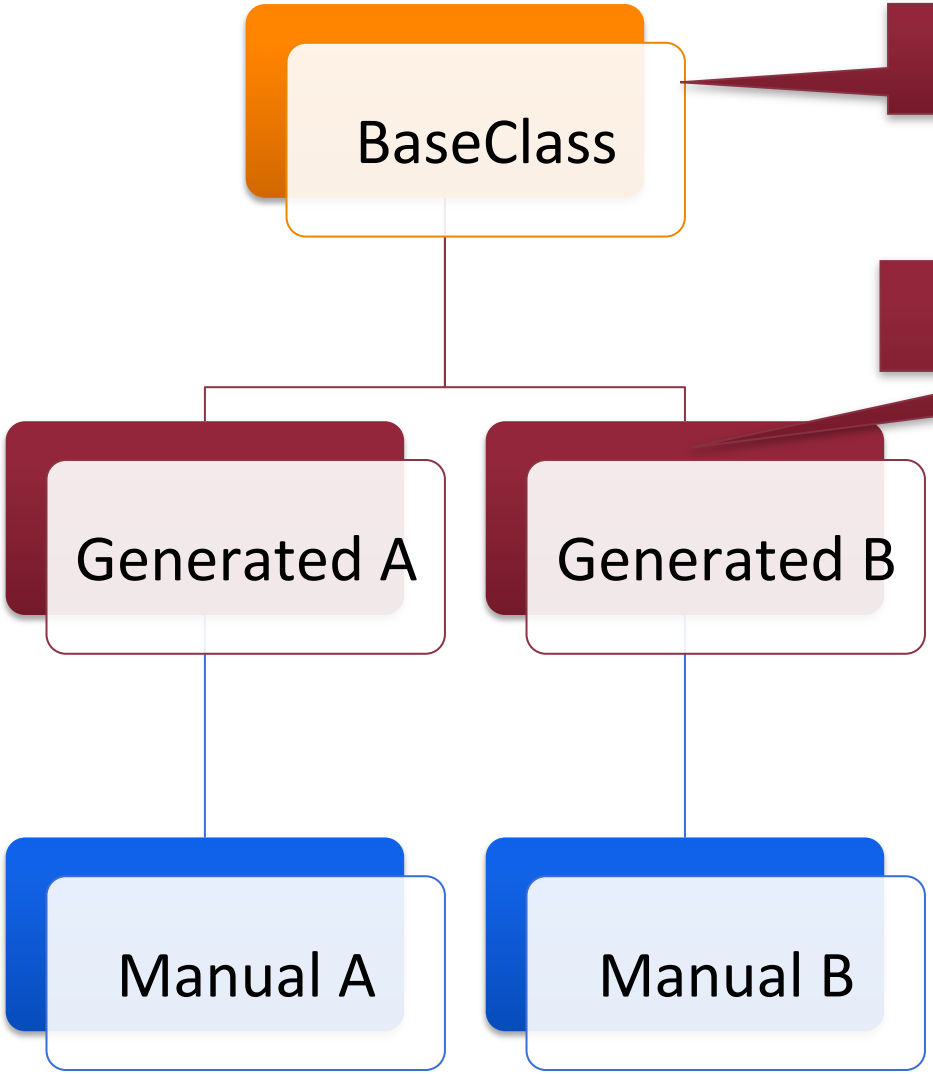
- Customization
  - Use inheritance
  - Customizations in descendants
    - Changes will transfer
- Generated code needs not to be touched

Generation Gap pattern – 1.



- Use inheritance
- Customizations in descendants
  - Changes will transfer
- Generated code needs not to be touched

Generation Gap pattern – 1.



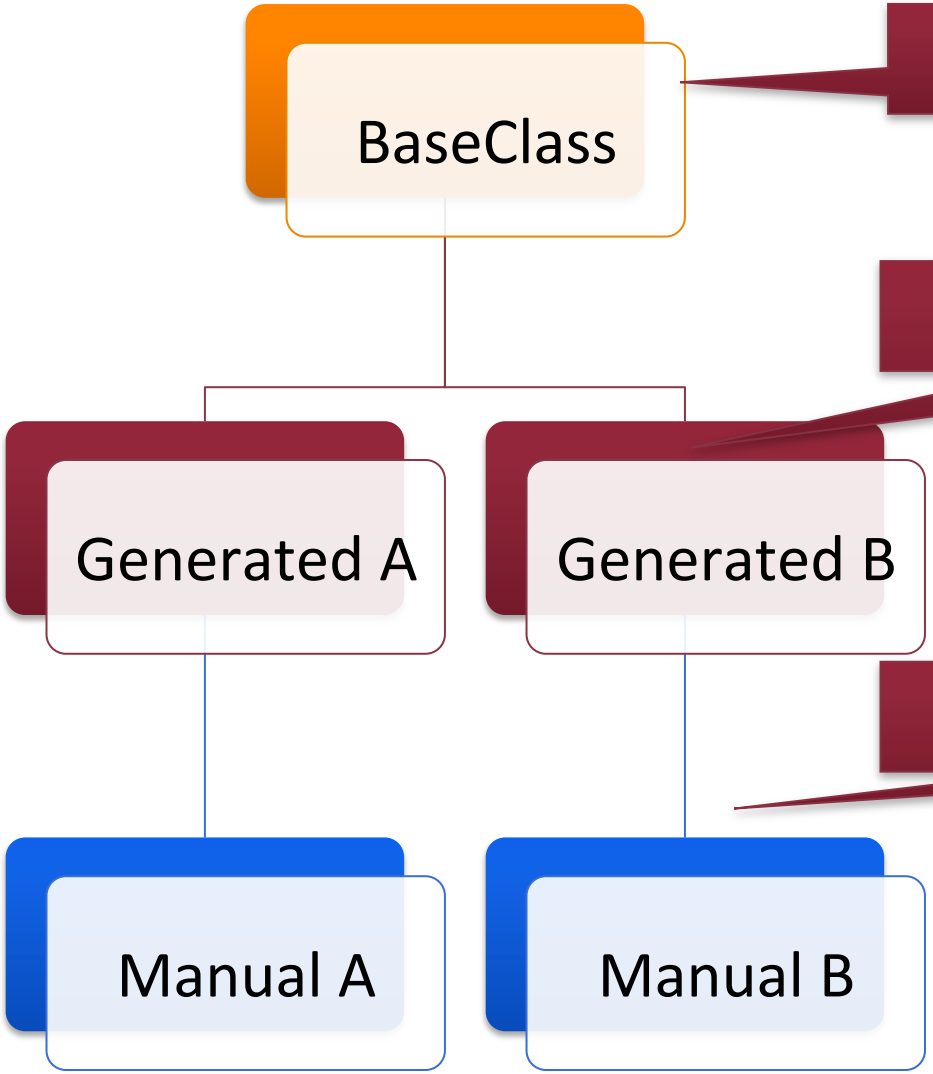
Framework class

- Use inheritance
- Customizations in

Regenerated on each build

- Generated code needs not to be touched

Generation Gap pattern – 1.



Framework class

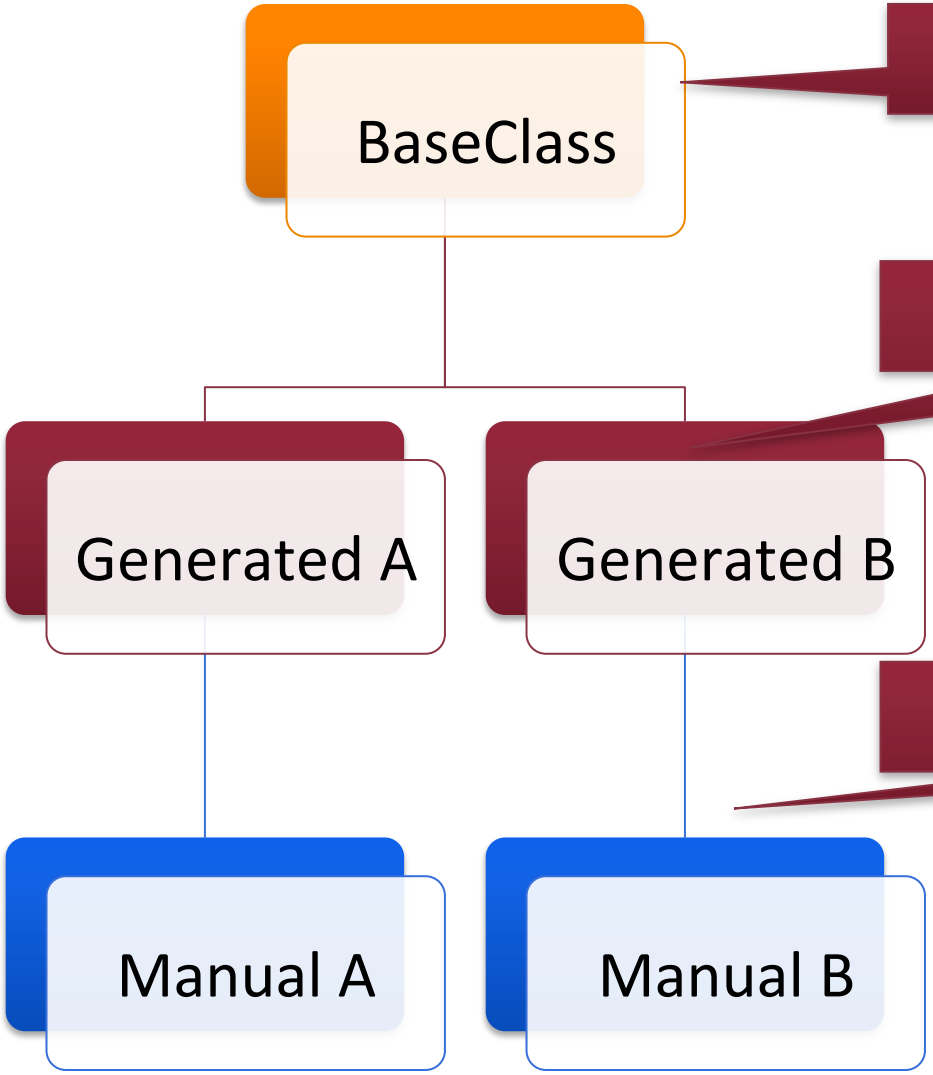
- Use inheritance
- Customizations in

Regenerated on each build

- Changes will transfer
- Generated code needs not to be touched

Manually written

Generation Gap pattern – 1.



Framework class

- Use inheritance
- Customizations in

Regenerated on each build

- Generated code needs not to be touched

Manually written

Can this approach be used with EMF models?

Generation Gap pattern – 2.

- Application conditions
  - Code generation
  - One or more classes
  - Generated interfaces remain compatible (usually)
  - Generated classes are not referenced/instantiated directly by the framework

Generation Gap pattern – 2.

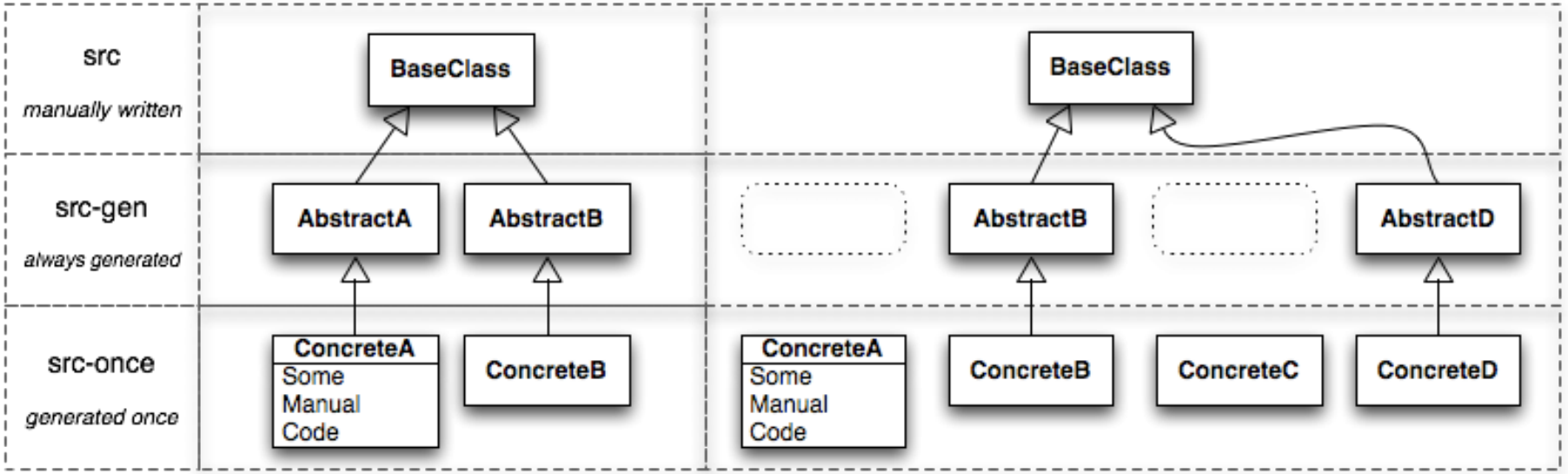
- Application conditions
  - Code generation
  - One or more classes
  - Generated interfaces remain compatible (usually)
  - Generated classes are not referenced/instantiated directly by the framework

EMF breaks the last assumption

Generation Gap pattern – 3.

- Drawback: Extra set of classes required
  - One for each generated class
    - Lot of classes
    - May be generated
    - Must not be regenerated
- Some alternative code generation strategies help
  - Generate-once
  - Conditional generation

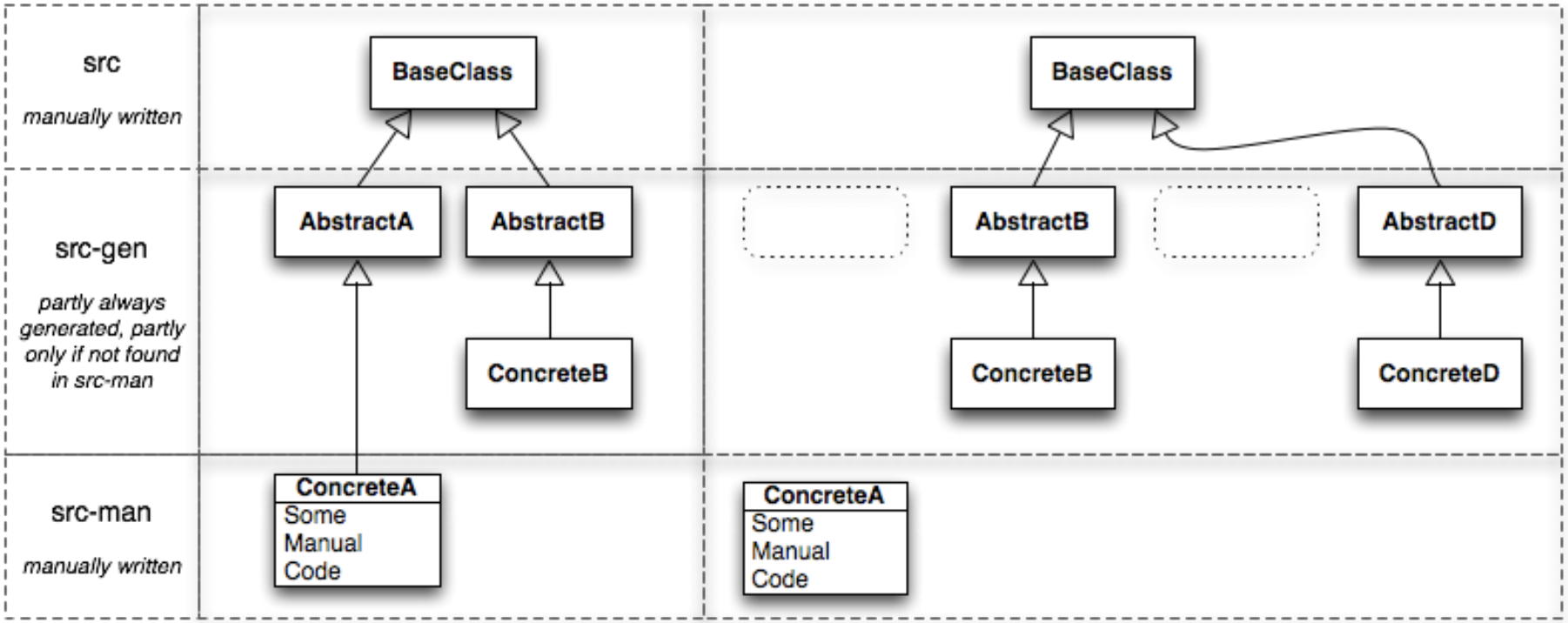
“Generate once”



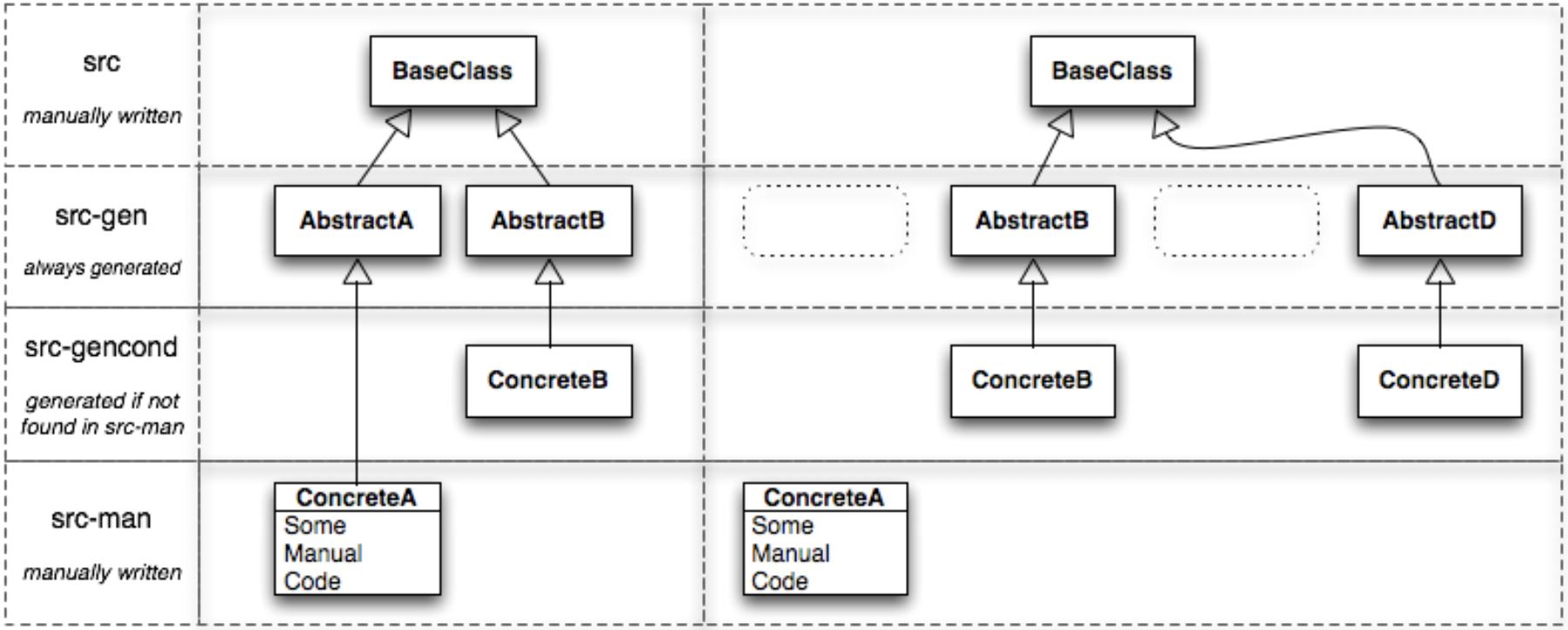
Generate once

- Generate an initial implementation
  - Never regenerated
  - Safe to modify
- Examples
  - Generating extended classes (once)
  - Sample code

Conditional generation – 1.



Conditional generation – 2.



Conditional generation

- Idea: Check whether a replacement is created
  - In an alternative source folder
  - If yes, do not generate code
- Advantages
  - Fewer classes
  - Works in more cases (may work for EMF)
- Disadvantages
  - More complex code generator
  - Generated code updates will not be available

Implementing code generation strategies

- JET
  - Controller Java code
- Acceleo
  - No direct support
- Xpand
  - MWE constructs available
- Xtend(2)
  - Integration code manages this

Summary

Code generation

- Started as source code generation
- Various extensions possible
  - Not as simple as it seems
  - Definitely possible
- Good tool support
  - Various approaches
  - Keep them in mind