

# 25. Model Synchronisation, Code Generation and Round-Trip Engineering for the Consistency of Macromodels

## Code Generation as Apps for RAG

Prof. Dr. U. Aßmann

Technische Universität Dresden

Institut für Software- und  
Multimediatechnik

[http://st.inf.tu-dresden.de/  
teaching/most](http://st.inf.tu-dresden.de/teaching/most)

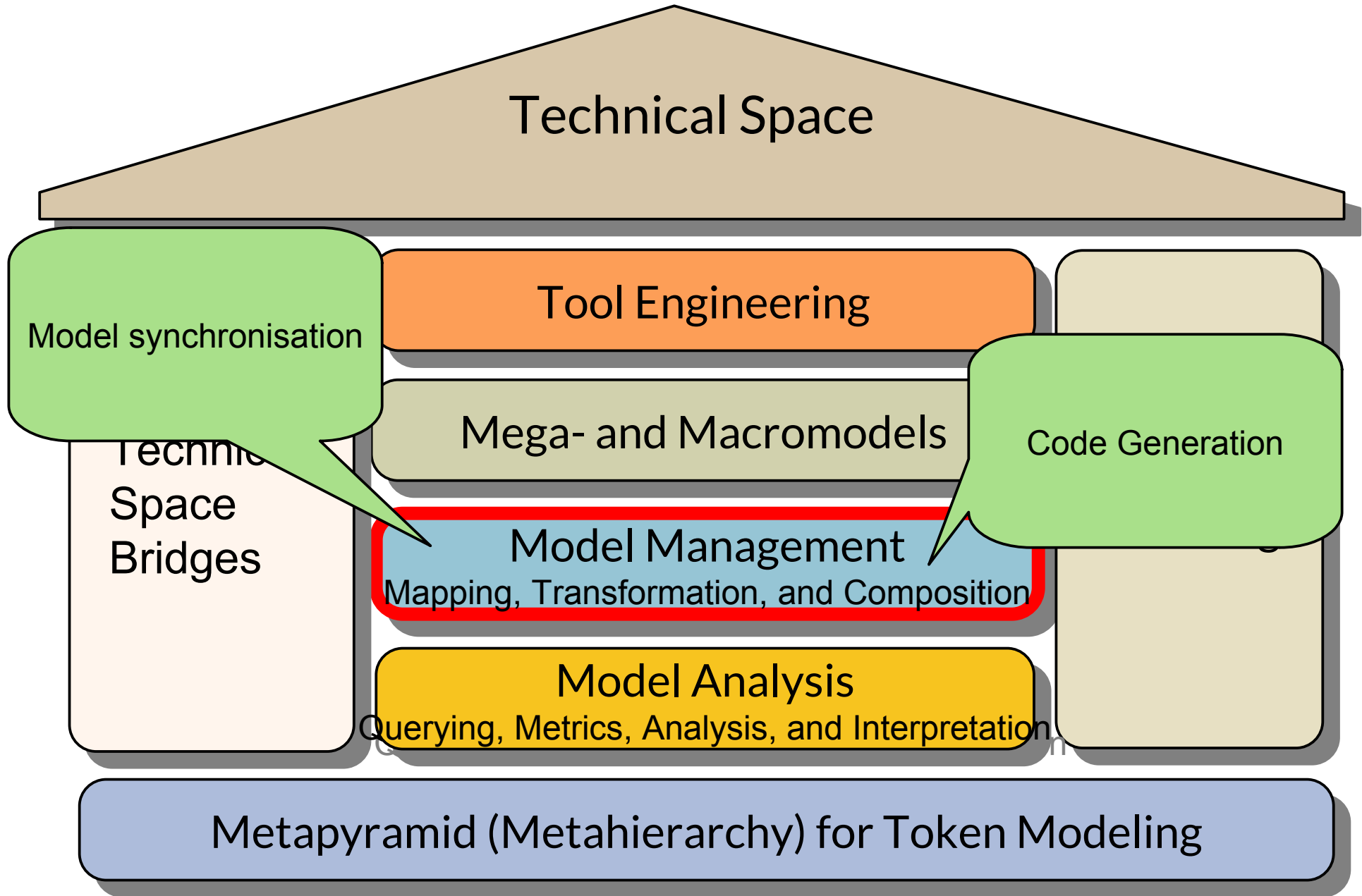
Version 19-0.4, 09.12.19

- 1) Single-source principle and macromodel principle
- 2) Code generation techniques
  - Template-based Code generation
- 3) Re-parsing



DRESDEN  
concept  
Exzellenz aus  
Wissenschaft  
und Kultur

# Q10: The House of a Technical Space



- ▶ <http://www.codegeneration.net/>
- ▶ [www.programtransformation.org](http://www.programtransformation.org)
- ▶ [http://www.codegeneration.net/tiki-read\\_article.php?articleId=65](http://www.codegeneration.net/tiki-read_article.php?articleId=65)
- ▶ Paul Bassett. Frame-based software engineering. IEEE Software, 4(4):9-16, 1987.
  - <http://doi.ieeecomputersociety.org/10.1109/MS.1987.231057>
- ▶ Chris Holmes, Andy Evans. A review of frame technology. University of York, Dept. of Computer Science, 2003  
<ftp://www-users.cs.york.ac.uk/reports/2003/YCS/369/YCS-2003-369.pdf>
- ▶ Daniel Weise and Roger Crew. Programmable syntax macros. In Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, pages 156-165, Albuquerque, New Mexico, June 23-25, 1993.
- ▶ Optional
  - Völter, Stahl: Model-Driven Software Development, AWL 2005.
  - Falk Hartmann. Safe Template Processing of XML Documents. PhD thesis, Technische Universität Dresden, Fakultät Informatik, July 2011.
    - <http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-75342>

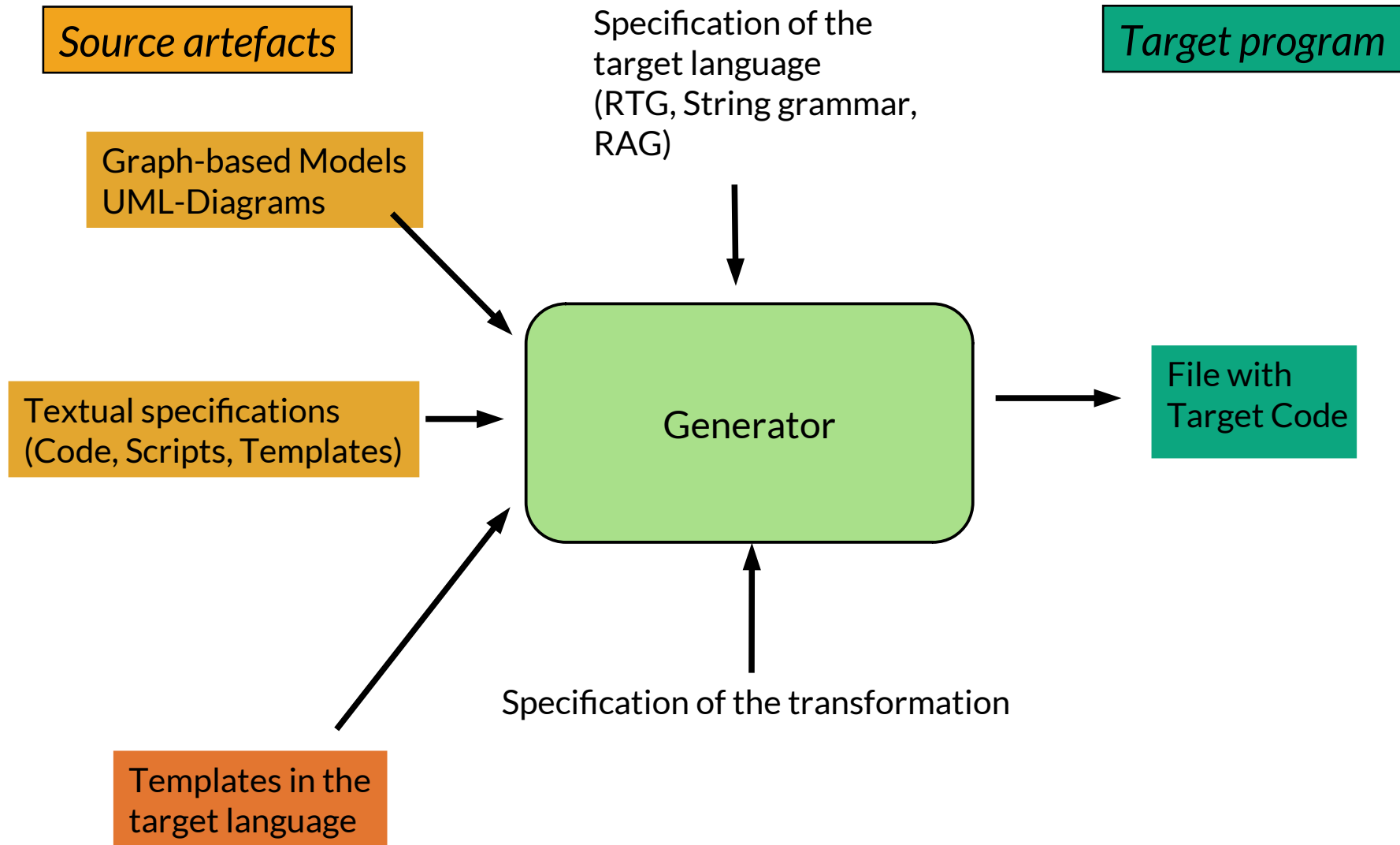
## 25.1 Model2Code Transformation (Code Generation)

Transforming models into code (Programmüberführung)

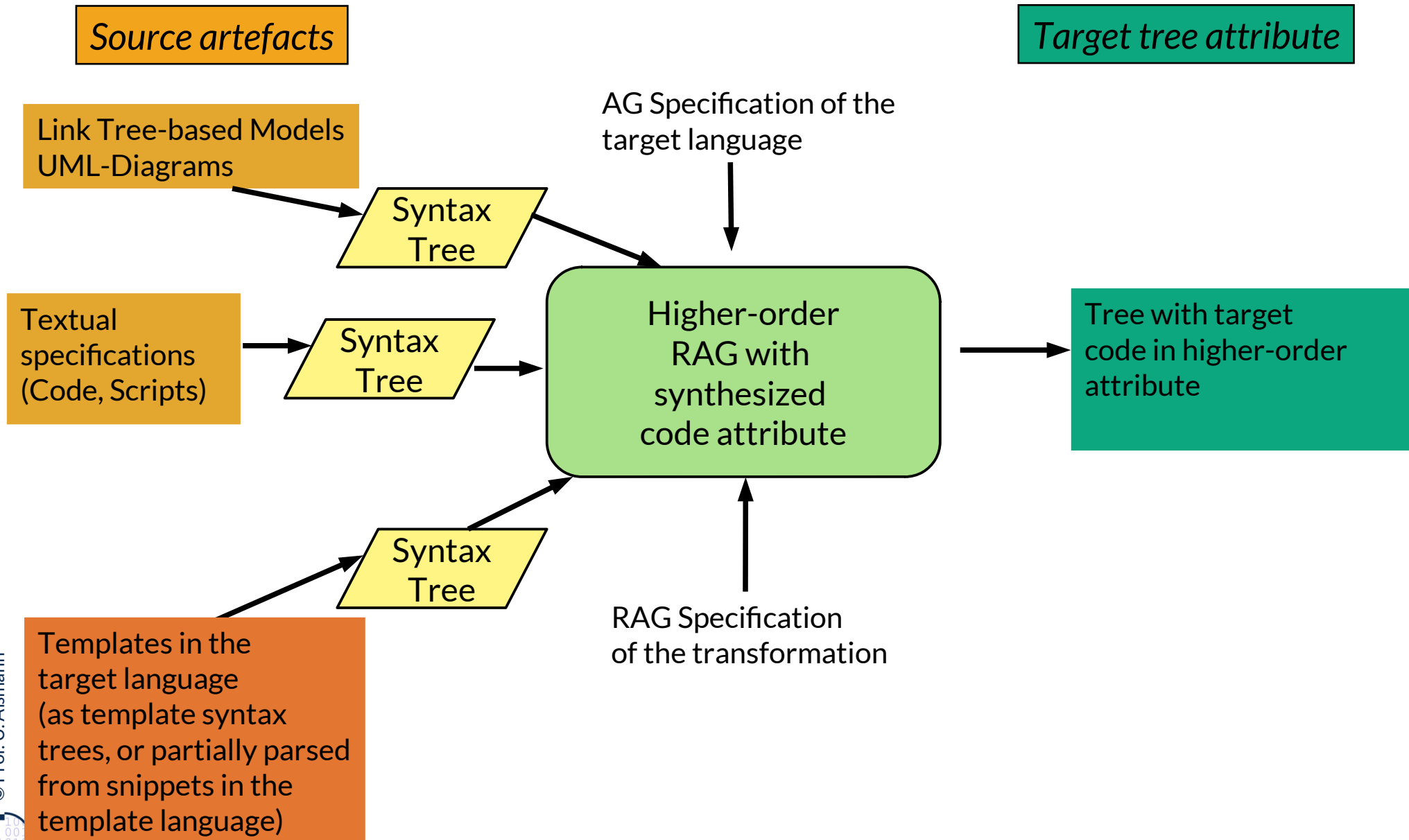


DRESDEN  
concept  
Exzellenz aus  
Wissenschaft  
und Kultur

# MDSD-Code-Generators



# MDSD-Code-Generators as Attributors of Syntax Trees

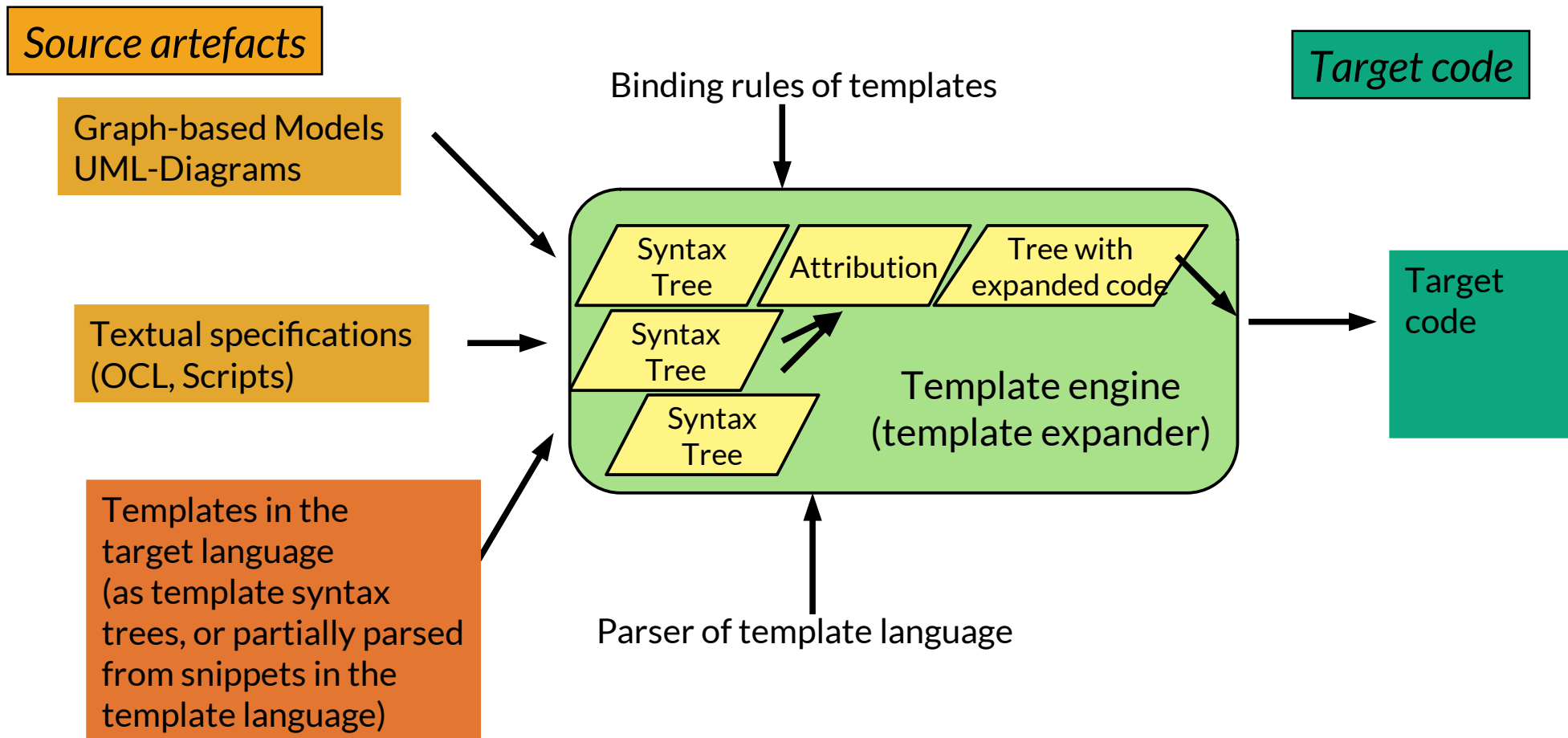


# Kinds of Code Generators

- ▶ A **code selector** is a transformation system (term, link trees, graphs) covering the input models with rules (**code coverage**) transforming the model elements once
- ▶ A **code scheduler** orders instructions in an optimized manner
  - Code scheduling runs after code selection
- ▶ **Metaprogramming code generators:**
  - A **template expander** generates code by filling code templates with *inset snippets*
  - An **invasive fragment composer (invasive software composition)** composes templates in a typed and wellformed way (→ CBSE)

# MDSD-Code-Generators as Template Expanders

- ▶ A **template engine** hides the tree construction, attribution with code attributes, and pretty-printing under a simple interface
- ▶ It provides a function `templparse(): String in TemplateLanguage → Tree`
- ▶ Template engines are *apps* of higher-order RAG





## 25.1.2 Code Generation in RAGs

- ▶ With higher-order (tree-generation) attributes and special functions
  - partial parsing
  - template expansion



# Code Generation with RAGs

- ▶ Attribution functions may generate code syntax trees
- ▶ Suppose a *partial parse function* `pparse(): String->LinkTree`

```
eq Constant.Code() {
  if (AsBoolean())
    if (AsValue() == 1)
      return pparse("(boolean)1");
    else if (AsValue() == 0)
      return pparse("(boolean)0");
    else return EmptyTree;
  else {
    if (AsValue() == 1)
      return pparse("new Integer(1)");
    else if (AsValue() == 0)
      return pparse("new Integer(0)");
    else
      return pparse("new Integer("+AsValue()+")");
  }
}
```

# Template-Based Code Generation with RAGs

- ▶ Attribution functions may expand code templates to code trees
- ▶ Done with the *template processing function*  
templparse(): String, List(ID)->LinkTree that expands variable names into attribution functions, e.g., TypeParameterName → TypeParameterName()
- ▶ templparse() is called a *template processor*, String is of a *template Language*

```
eq GenericClassInstantiation.Code() {
  return templparse(
    "public class GenClass$TypeParameterName$ extends Object {
      private int myId;
      public GenClass$TypeParameterName$() { // constructor
      }
      public int getId() { return myId; }
    }"
    , List(pparse("Person"))
  );
}
```

# Template-Based Code Generation with RAGs

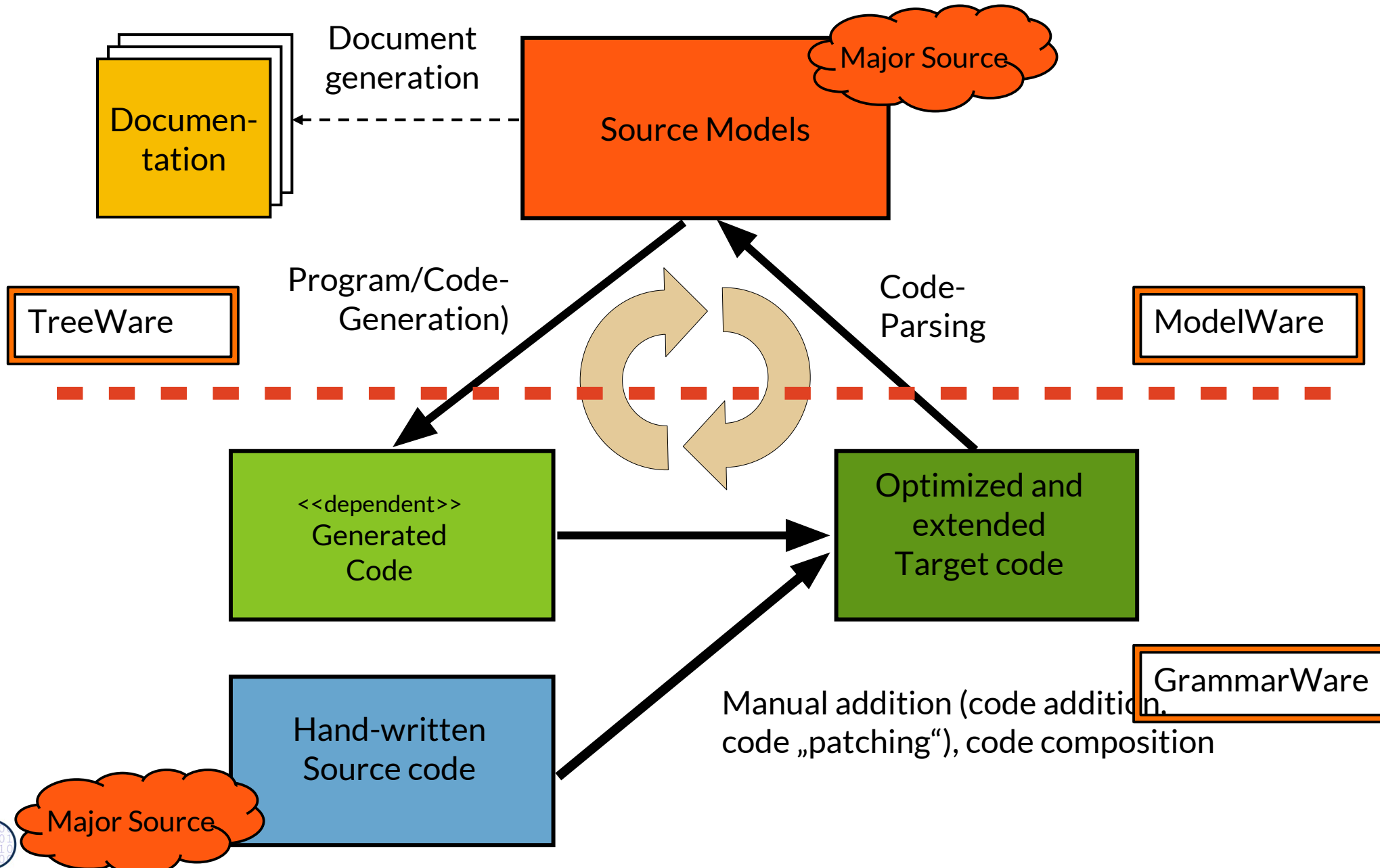
- ▶ The *template processing function* can be made generic in terms of grammars:  
templparseGeneric(): CSGrammar, RTG, String, List(ID) → LinkTree that expands variable names into attribution functions, e.g., TypeParameterName → TypeParameterName()
- ▶ templparse() is called a *template processor*, String is of a *template language*

```
eq GenericClassInstantiation.Code() {
  CSGrammar CSAceleo;
  RTGrammar RTGAceleo;
  return templparseGeneric(CSAceleo, RTGAceleo,
    "public class GenClass$TypeParameterName$ extends Object {
      private int myId;
      public GenClass$TypeParameterName$() { // constructor
      }
      public int getId() { return myId; }
    }"
    , List(pparse(„Person“))
  );
}
```

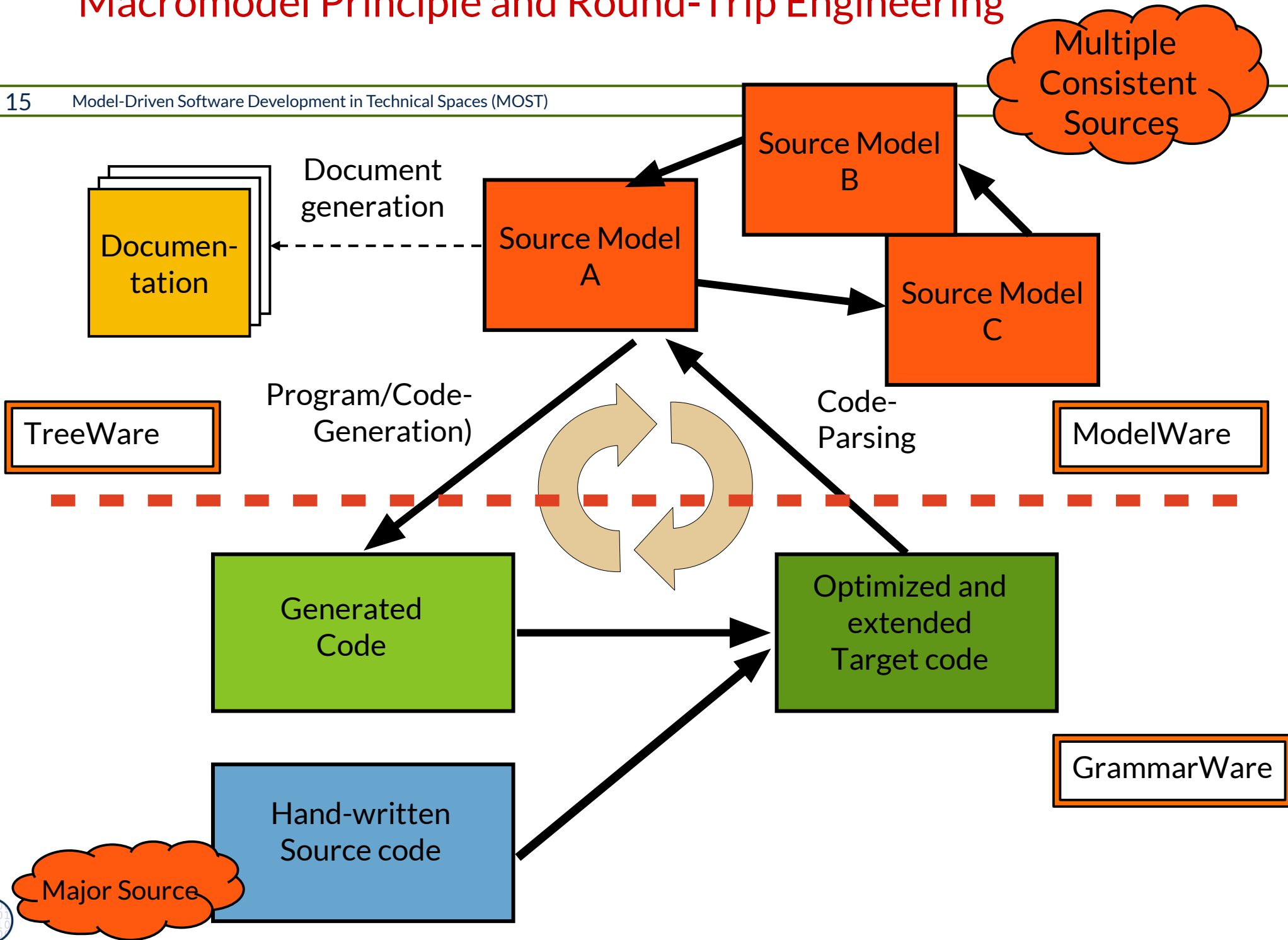
## 25.1.1 Single-Source Principle and Macromodels



# Single-Source-Principle, Major-Source, Code Addition, and Round-Trip Engineering



# Macromodel Principle and Round-Trip Engineering



# Single-Source Principle, Major-Source Principle and Macromodel Principle

- ▶ A **Single-Source-Technology** with automatic synchronisation and consistency between one model (single source), code, tests
  - 1997 introduced by Peter Coad in the Together-CASE-tool, now in all CASE tools
- ▶ A **Macromodel Technology** with automatic synchronisation and consistency between ALL (major-source) models, code, tests, and documentation (all models of a megamodel)
- ▶ In a macromodel, there are always *derived models*
  - Generated code (this chapter)
  - Generated documentation (Chapter on documenation)
  - Generated test suites and data



- ▶ Technically, the Single-Source-Principle and the Macromodel principle needs **Round-Trip-Engineering** (RTE) between ModelWare and GrammarWare, to achieve
  - **Model-to-code synchronisation** with
    - **Codegeneration** into several programming languages
    - **Template-based codegeneration** inserts code snippets into code templates
    - **Code reparsing** of the changed source code into models
  - **Model-to-model synchronization** with
    - **Bidirectional transformations** (with TGG)
    - **View based transformations** (with SUM)

# Example: Round-Trip Engineering in Together (P. Coad, Borland)

- ▶ In 1998, the CASE tool Together was the first to provide a Single-Source-Technology with automatic synchronisation and consistency between UML model, code and documentation
- ▶ Supported Programming Languages: Java, Visual Basic, VisualBasic.Net, CORBA IDL, C++, C#
  - Synchronisation by reparsing of generated, modified and extended code
- ▶ Round-trip Engineering:
  - Changes of class diagrams will be transformed to code
  - Changes of code reparsed to class diagrams
  - Reverse Engineering of entire projects

[http://www.borland.com/downloads/download\\_together.aspx](http://www.borland.com/downloads/download_together.aspx)

<http://www.borland.com/de-DE/Products/Requirements-Management/Together/Testimonials>

[https://en.wikipedia.org/wiki/Borland\\_Together](https://en.wikipedia.org/wiki/Borland_Together)

# Together Screenshot

19

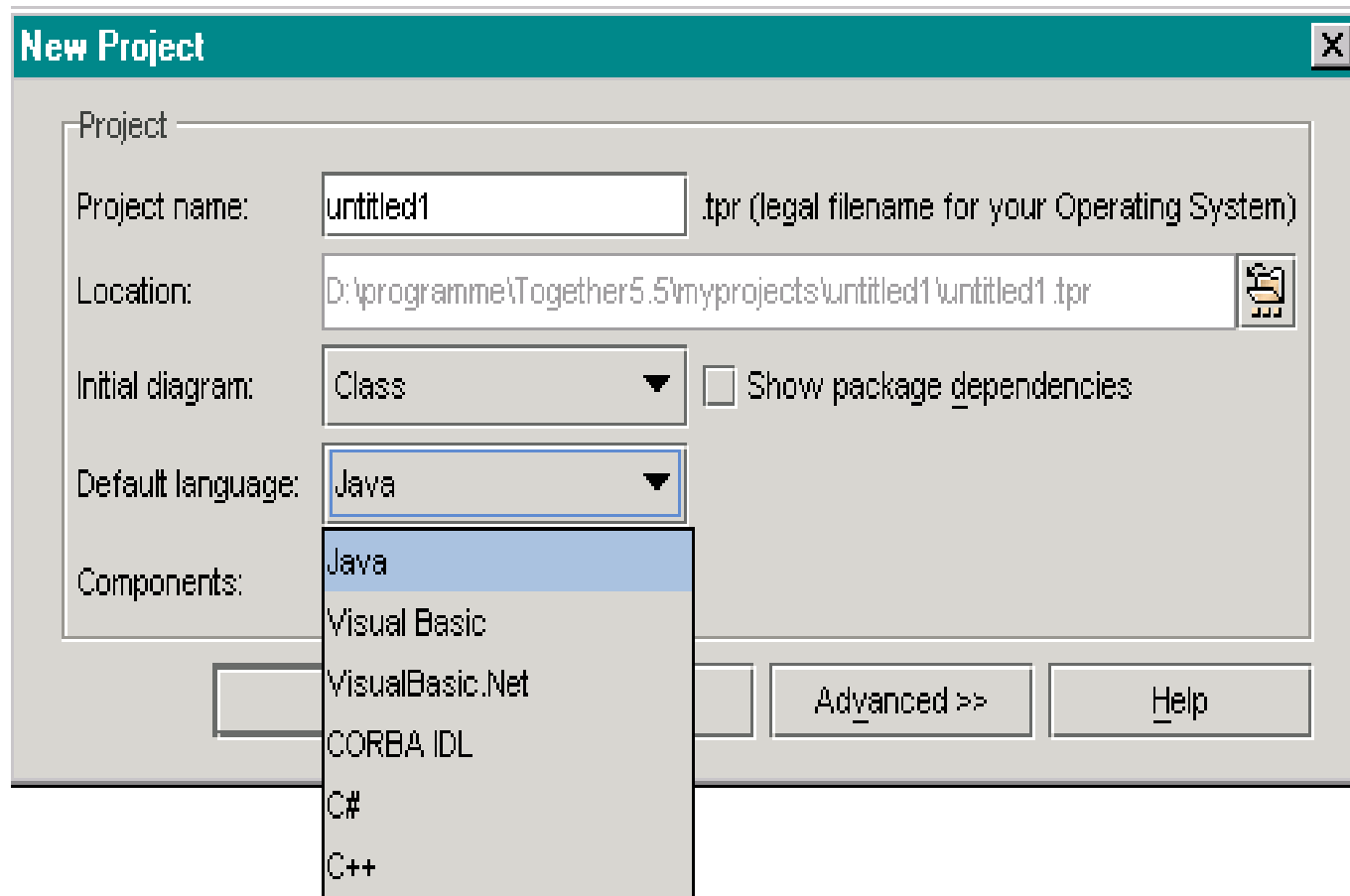
Model-Driven Software Development in Technical Spaces (MOST)

The screenshot displays the Together IDE interface. On the left, a project tree shows a package named 'Demo' containing two classes: '<default>' and 'Teilnehmer'. Below the tree is the 'Properties of <default>' inspector, which shows the diagram type as 'Class Diagram' and the name as '<default>'. The main workspace is divided into two panes. The top pane shows a UML class diagram with two classes: 'Person' and 'Teilnehmer'. Both classes have a private attribute '-attribute1:int' and a public operation '+operation1:void'. 'Teilnehmer' inherits from 'Person', as indicated by a solid line with an open arrowhead pointing to 'Person'. The bottom pane shows the generated Java code for the 'Teilnehmer' class, which extends 'Person' and implements the 'operation1()' method. The code is as follows:

```
/* Generated by Together */  
  
public class Teilnehmer extends Person {  
    public void operation1() {  
    }  
  
    private int attributel;  
}
```

At the bottom of the IDE, a tab labeled 'Teilnehmer.java' is visible. A status bar at the very bottom indicates 'Press Ctrl+Enter to finish editing and close Inspector'.

# Code Generation in Different Languages in Together



- Supports roles: Business Modeler, Designer, Developer and Programmer
- Appropriate views can be configured
- Code template based code generation

# 25.2 Technologies for Model-2-Code Generation

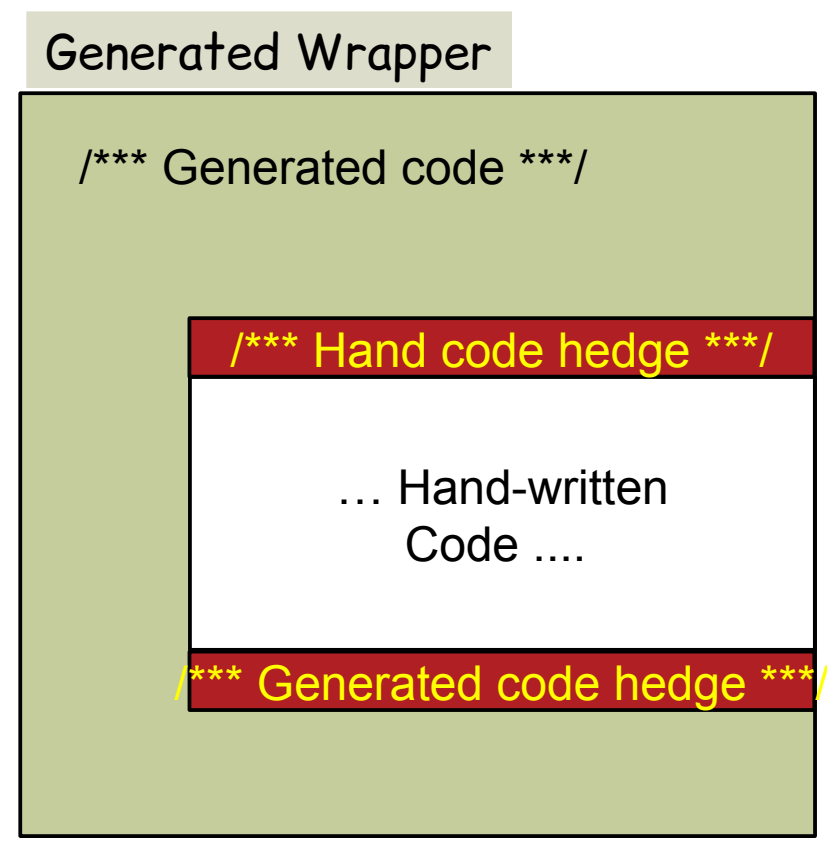
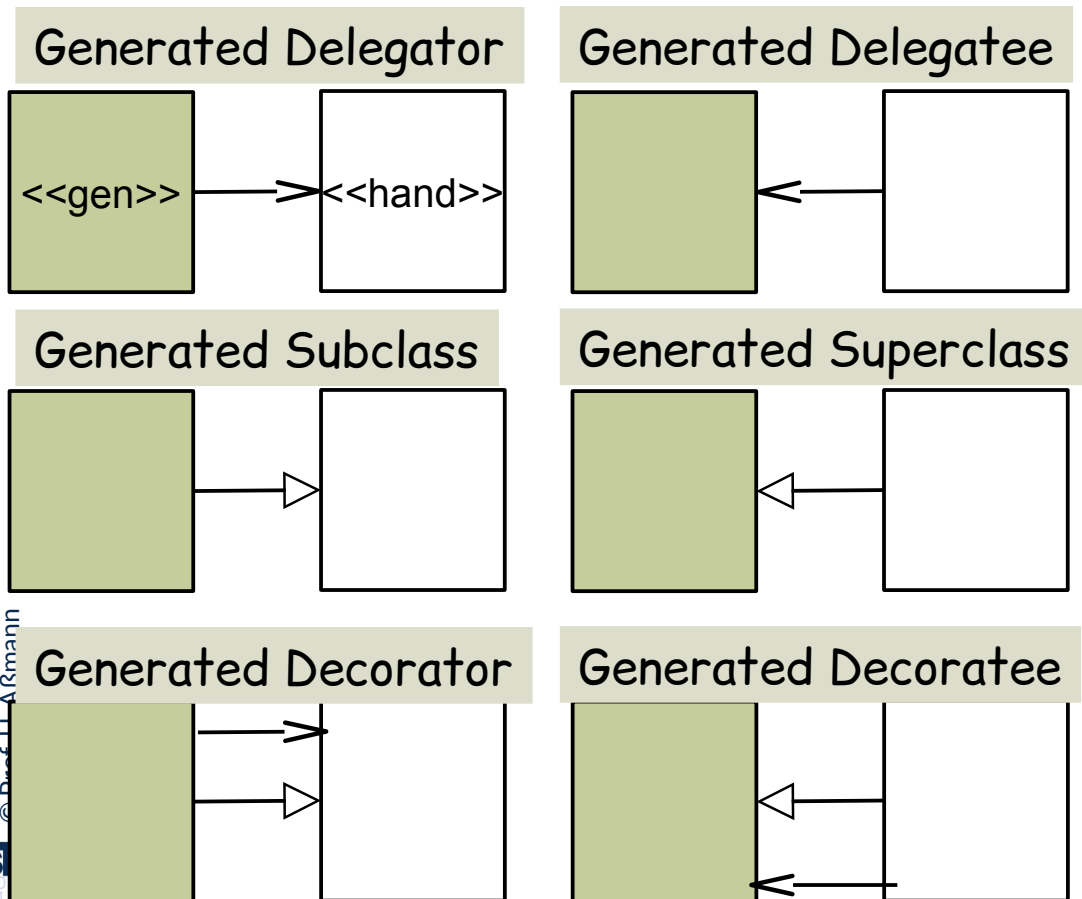


# Composition of Separated Hand-Written and Generated Code

- ▶ **In separate files:** Coupling by implementation pattern [Völter/Stahl]
- ▶ Use class composition like delegation, TemplateMethod, Composite, Decorator, etc

**In one file:**

Coupling with **hedges** (Trennmarkierung)



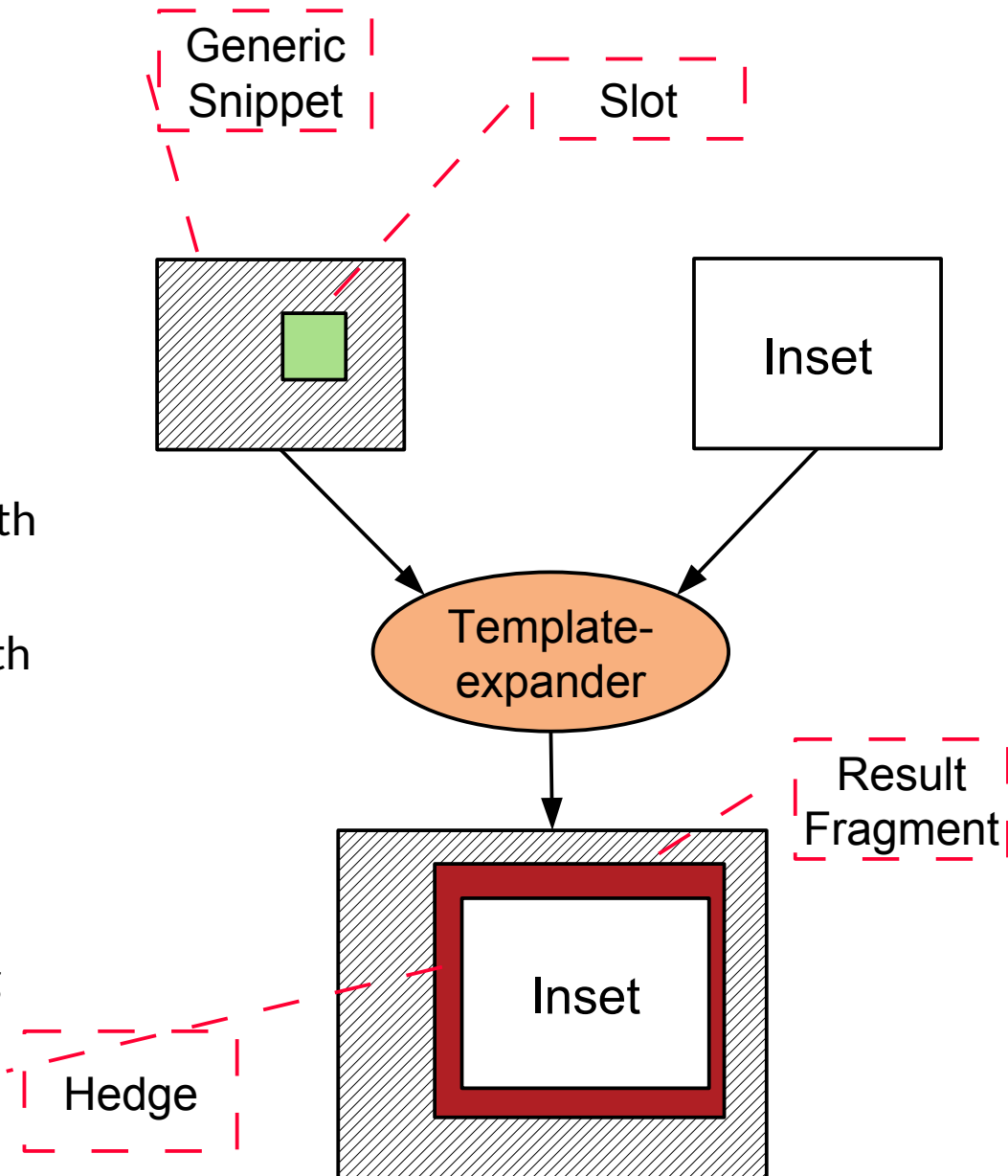
# Composition of Generated and Hand Written Code in an RAG

- ▶ Fine-grained glueing possible

```
eq Procedure.Code() {  
    return Head.Code()+  
        „/** HEDGE BEGIN */“+  
        GeneratedBody.Code()+  
        „/** HEDGE END */“;  
}  
eq Head.Code() {  
    return pparse(“public „+Head.name+“()”);  
}  
eq GeneratedBody.Code() {  
    return Body.Code();  
}
```

# Snippet Programming with RAG

- ▶ A **fragment (snippet)** is a incomplete sentence of a language, derived from a nonterminal of the grammar, or described by a metaclass
- ▶ A **generic fragment (template, form, frame)** is a fragment with **slots (holes, code parameters, variation points)**, which can be *bound (filled, expanded)* with an **inset fragment** to a **result fragment**
- ▶ A **extensible fragment** is a fragment with **hooks (extension points)**, which can be *extended* to a fragment
- ▶ **Generic programming** is programming with generic fragments (templates).
- ▶ **Invasive programming** is programming with generic and extensible fragments (templates with hooks)
- ▶ → CBSE course



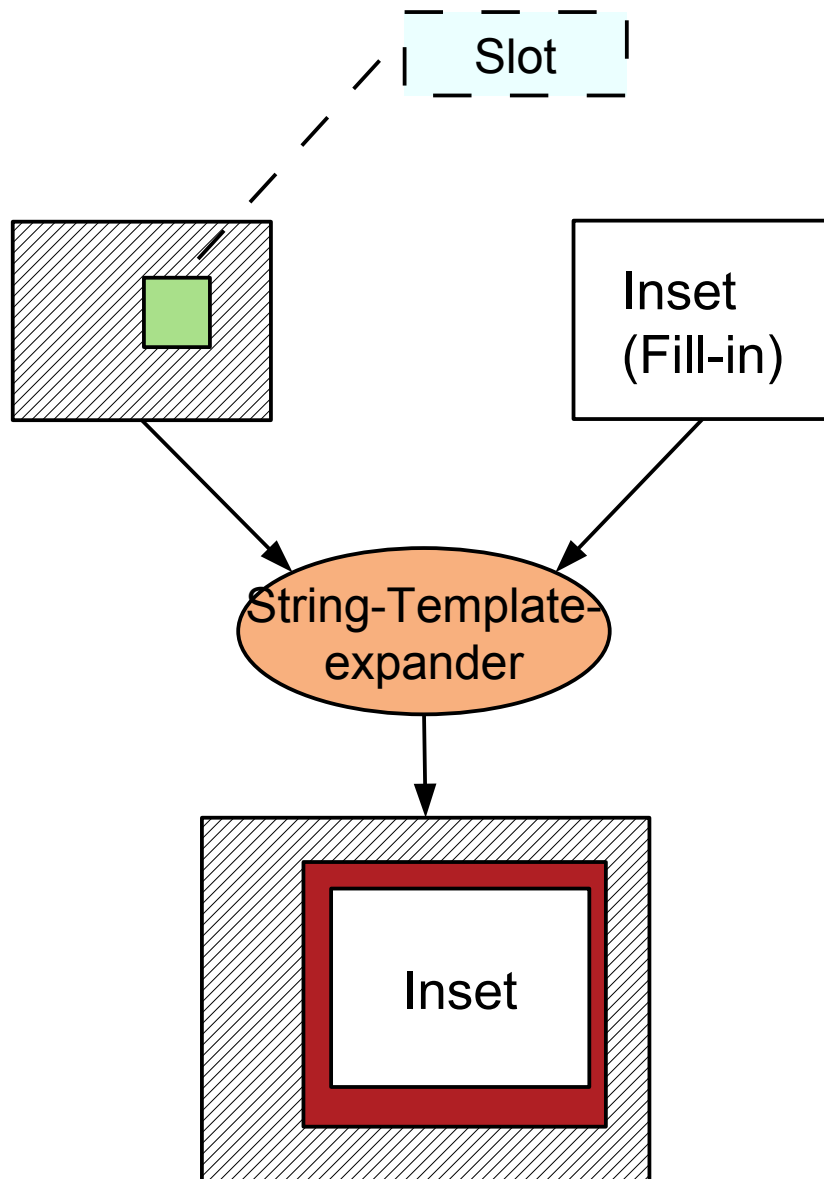


## 25.2.1 Template-based Code Generation (Schablonenbasierte Programmüberführung)

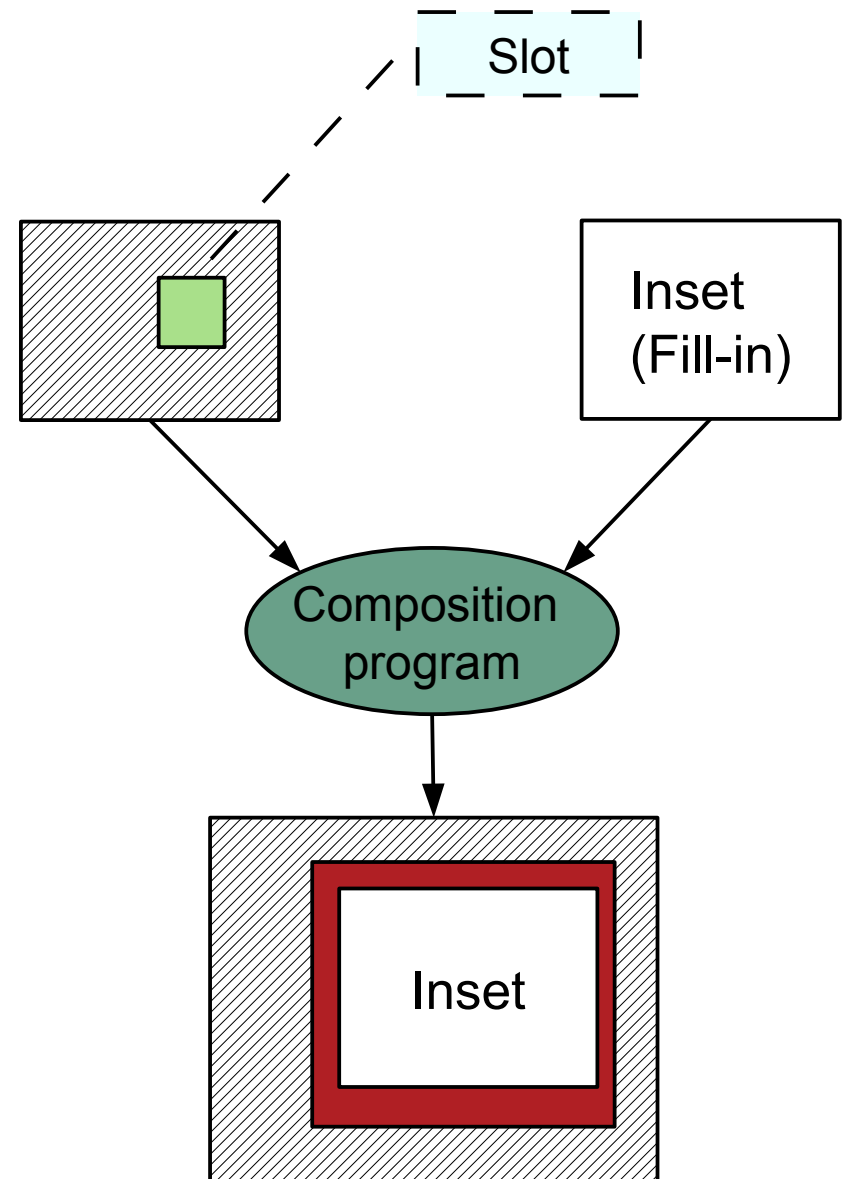


# Template Expansion by Composition of Insets and Hedging

### Coupling by string expansion



### Coupling by composition program



# Slots are Marked by Hedges

- ▶ **Hedges** are delimiters that do not occur in the base nor in the slot language
- ▶ **Slot hedges** are template2slot hedges marking the transition from the code language to the slot language
- ▶ **Inset hedges** are metaprogramming2code hedges marking the transition from the metaprogramming language to the code language

```
// code slot hedges << >>
template (superclass:CLASS, t:TYPE) {
    class Worker extends << superclass >>
    {
        <<t>> attr = new <<t>>();
        <<t>> getAttr();
        void setAttr(<<t>>);
    }
}
```



# Tools for Untyped Template Expansion

- ▶ **Frame processing** was invented in [P. Bassett] as an *untyped string template expansion technology*, universal for all textual languages [Holmes/Evans]
  - Frame processing is the main technology for web engineering today: it organizes reuse of page templates
  - The original frame processor used \$ as a hedge symbol for slots (slot variables)
- ▶ **Macro processing** is not much different
  - Because only slot variables hold insets, macro parameters correspond to slot variables
- ▶ **XML template processing engine XVCL** [Jarzabek] is an XML-controlled frame processor
  - <http://sourceforge.net/projects/fxvcl/files/XVCL%20Specification/Version%202.10/>
- ▶ **String template engines** in use today
  - Apache Velocity <http://velocity.apache.org/>
  - Parr's template engine StringTemplate
  - Jenerator for Java <http://www.voelter.de/data/pub/jeneratorPaper.pdf>

# Velocity String Template Language

- ▶ Velocity Template Language (VTL) is a frame processing language with
  - metaprograms in slots, written in a **slot language (blue)**
- ▶ {#, \$} are slot hedges
- ▶ < (from XML) is the inset hedge

```
<html>
<body>
#set( $foo = "Velocity" )
Hello $foo World!
</body>
</html>
```

```
<HTML>
<BODY>
Hello $customer.Name!
<table>
#foreach($mud in $mudsOnSpecial)
  #if
  ( $customer.hasPurchased($mud) )
    <tr>
      <td>
        $flogger.getPromo( $mud )
      </td>
    </tr>
  #end
#end
</table>
```

# Velocity Template Language

- ▶ Velocity Template Language (VTL) is a simple scripting language in the spirit of TCL
- ▶ It has control structures (if, switch, foreach), assignments (set), and macros
- ▶ Similar: Acceleo (in exercises)

<http://velocity.apache.org/engine/releases/velocity-1.7>

```
#macro( inner $foo )
  inner : $foo
#end

#macro( outer $foo )
  #set($bar = "outerlala")
  outer : $foo
#end

#set($bar = 'calltimelala')
#outer( "#inner($bar)" )
```

**Problem: the result of string template expansion may not be syntactically correct, nor well-formed, target language (error-prone)**

# Typed Template Expansion

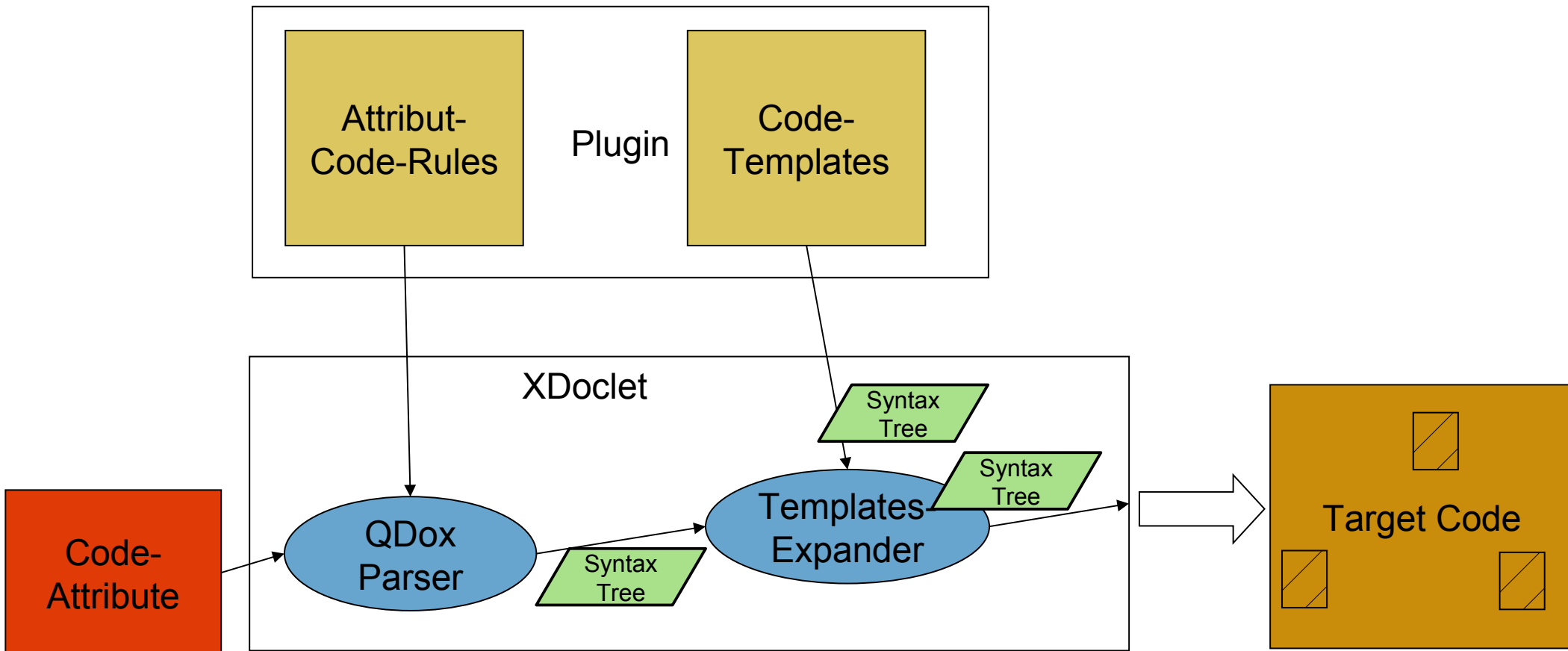
- ▶ Metamodel-controlled template engines
  - EMF: Xtend and Xpand scripting languages
  - XML slot markup language
  - Acceleo code generating system (see exercises)
- ▶ Invasive Software Composition provides fully typed and wellformed template expansion (see CBSE course)
  - Typed template expansion **and** -extension **and** weaving
  - Can be instantiated for arbitrary languages
  - <http://www.the-compost-system.org> (obsolete now)
  - <http://www.reuseware.org>
  - <https://bitbucket.org/svenkarol/skat/wiki/Home>

- ▶ **Semantic Macros** are metaprogramming procedures which are typed parameters and results.
  - A semantic macro is compiled to a fragment tree
    - which is instantiated by fragment parameters, type-checked on the metamodel, and copied to the instantiation spot
  - They allow for type-safe static metaprogramming.
  - In an higher-order RAG, a semantic macro can be instantiated in a higher- order attribute
- ▶ Examples:
  - Scheme
  - Scala <http://scalamacros.org/>
  - <http://docs.scala-lang.org/overviews/macros/overview.html>



# Xdoclet (xdoclet.sf.net) for Metadata-Based Code Generation

- ▶ Xdoclet transforms attributes (metadata) into helper code (aka boilerplate code)
  - With template-based code generation
  - Metadata attributes *trigger* the filling of templates, used from a library



- ▶ A **slot markup language** is a special template language for *any* XML dialect
- ▶ The slot language is represented as an XML dialect itself (XSD schema) [Hartmann]
  - Uniform syntax for templates
  - XML tools are usable
  - Filling templates is
    - type-safe
    - and wellformed, because OCL constraints can be defined that are checked

# The End

- ▶ Why is code generation a good application for RAG?
- ▶ How would you generate code with Xcerpt?
- ▶ Explain the difference of the code generation patterns GeneratedDelegatee, GeneratedDelegator, GeneratedSuperClass, GeneratedSubclass!
- ▶ Why does code generation most often use synthesized attributes?
- ▶ What is the difference of a metadata attribute (annotation), and an attribute in an RAG?
- ▶ Why are template engines apps for RAGs?
- ▶ Think about GOTO statements in machine code, or in C programs.
  - How would you represent them in an RAG?
  - Why are AG not really appropriate for representing GOTOs?

# A.1. Code Modification and Reparsing (Codemodifikation und -rückführung)



# Example of Code Reparsing Technique

- ▶ Code-Reparsing in Fujaba:

[http://www.fokus.fraunhofer.de/en/fokus\\_events/motion/ecmda2008/\\_docs/rs01\\_t03\\_ManuelBork\\_EMCD2008\\_slides.pdf](http://www.fokus.fraunhofer.de/en/fokus_events/motion/ecmda2008/_docs/rs01_t03_ManuelBork_EMCD2008_slides.pdf)

- ▶ Parallel Parsing of Template and Generated Code, with comparison to resolve indeterministic situations of re-parsing