Part III: Tool Integration Architectures and Macromodels

Prof. Dr. U. Aßmann

Technische Universität Dresden

Institut für Software- und Multimediatechnik

http://st.inf.tu-dresden.de/teaching/most

Version 21-0.4, 18.12.21

30. 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

0) Tool Integration Architectures

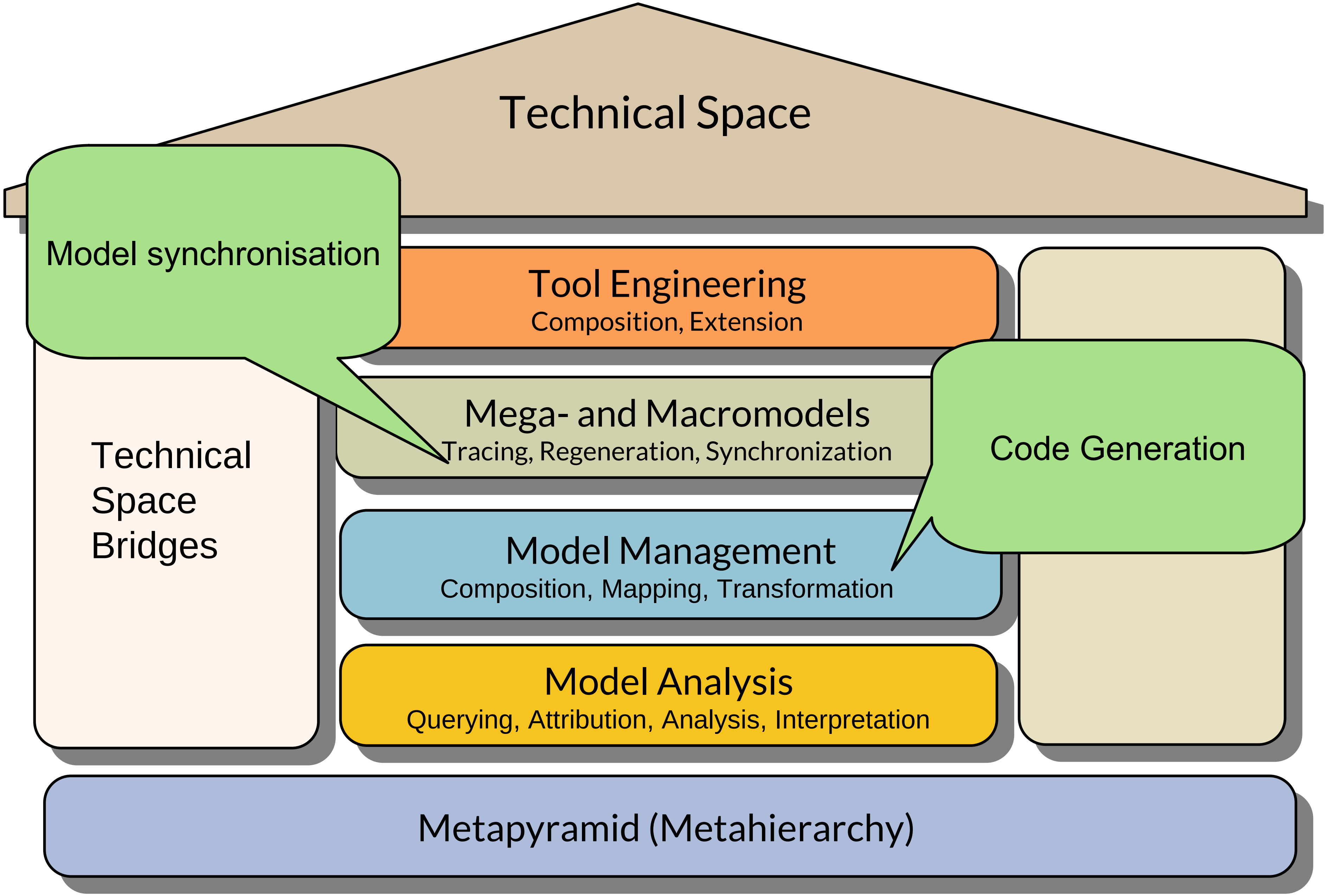
1) Single-source principle and macromodel principle

2) Code generation techniques

- Template-based Code generation

3) Re-parsing

Q10: The House of a Technical Space



Literature

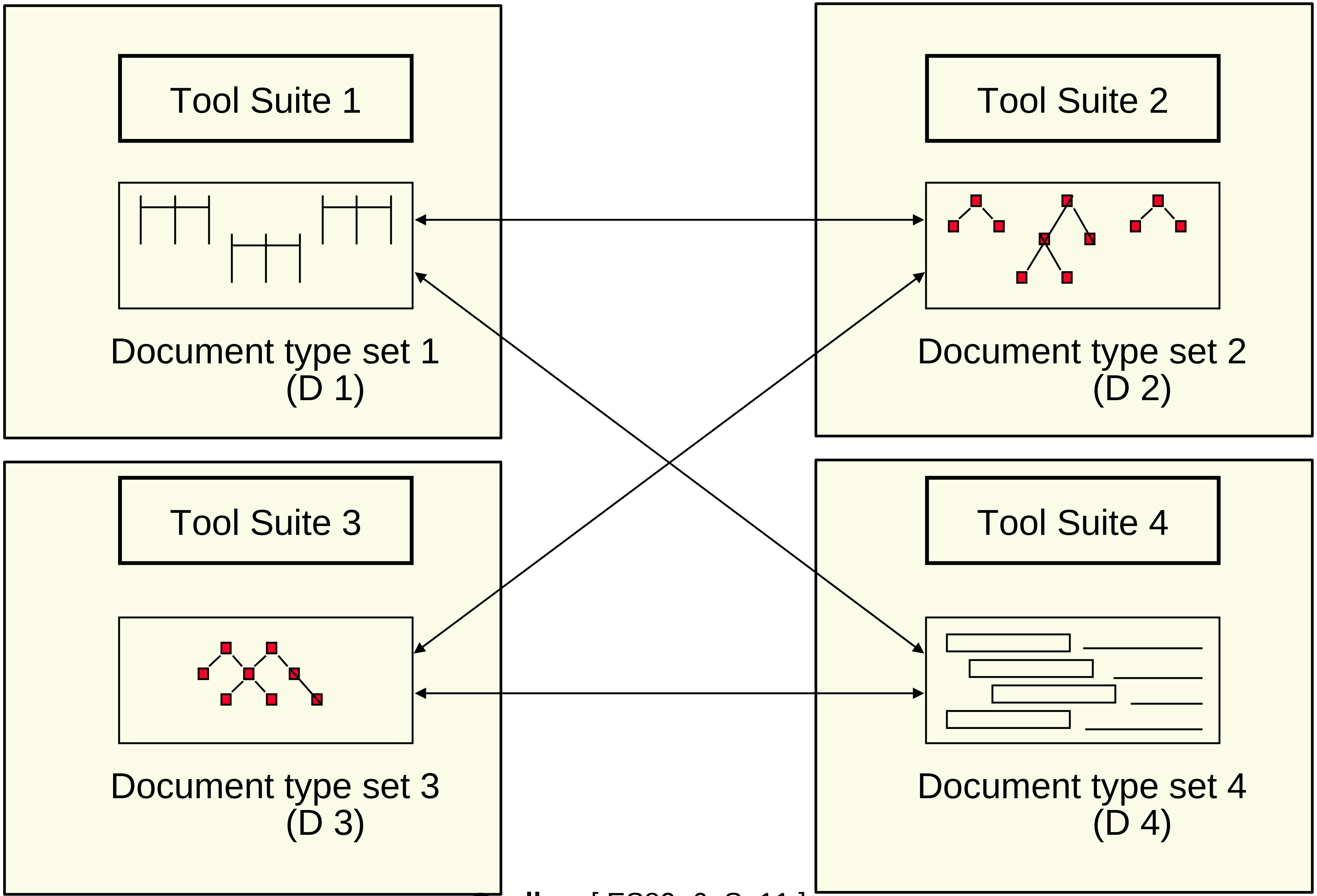
- http://www.codegeneration.net/
- www.programtransformation.org
- http://www.codegeneration.net/tiki-read_article.php?articleId=65
- Paul Bassett. Frame-based software engineering. IEEE Software, 4(4):9-16, 1987.
  - http://doi.ieeecomputersociety.org/10.1109/MS.1987.231057
- Chris Holmes, Andy Evans. 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

30.0 Concepts of Tool Integration for Software Factories

The following integration techniques for standalone tools can be used for

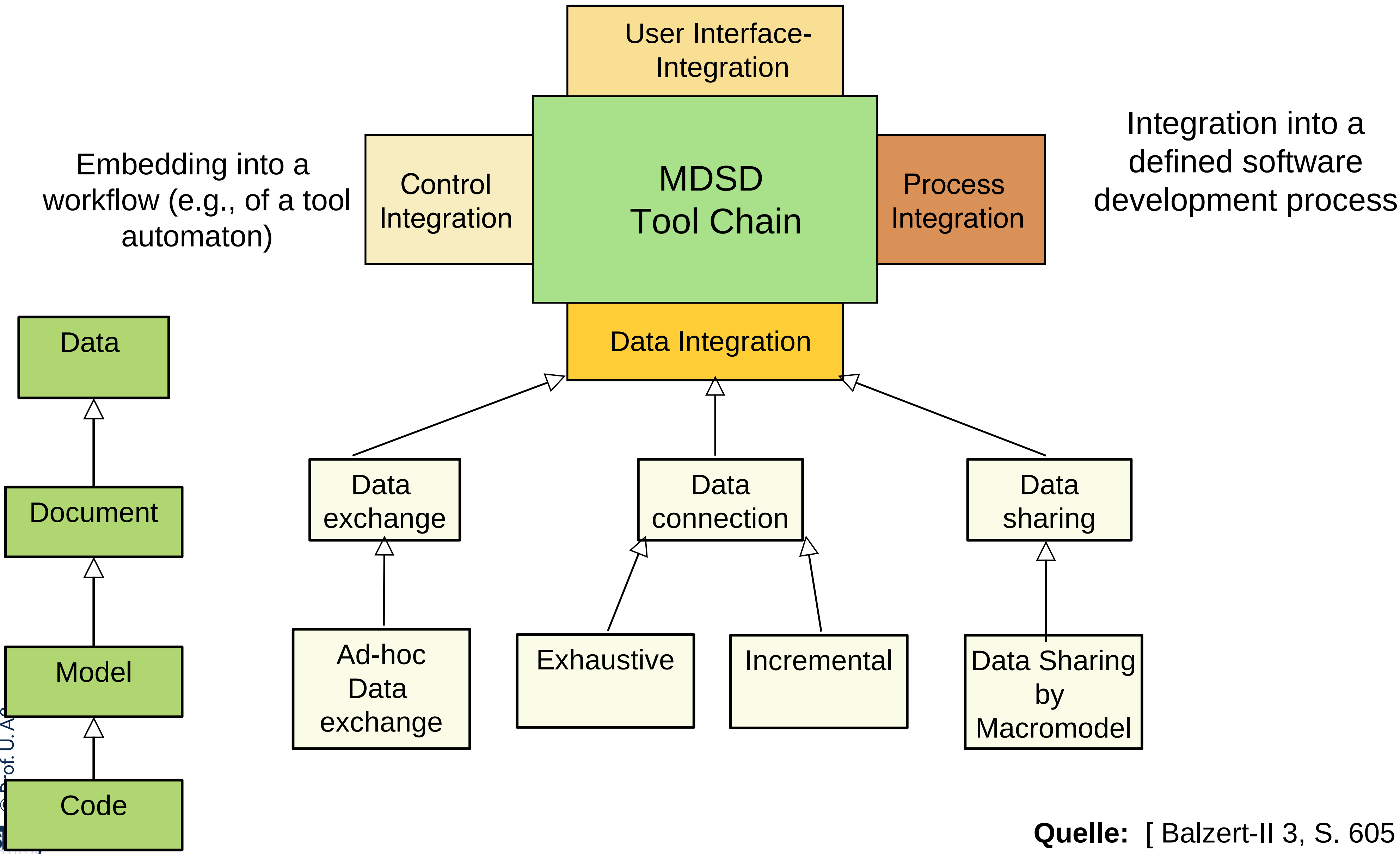
- Enterprise Application Integration (EAI)
- Distributed Software Systems
- Software Factories, MetaCASE tools, IDE
- Heterogeneous Software Factories

Integration of Standalone, Persistent Tool Suites by Data Connection



Quelle : [ES89, 6, S. 11]

Tool Integration in 4 Dimensions

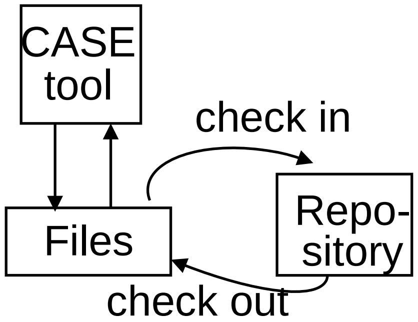
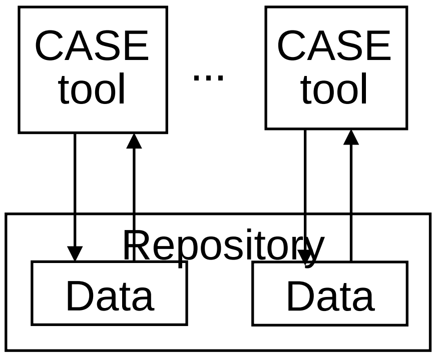
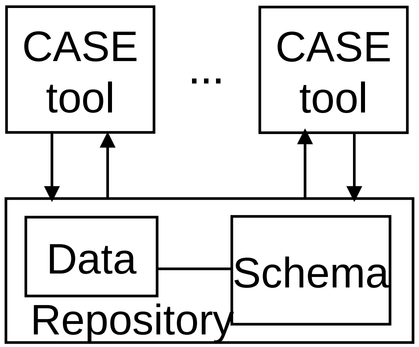


Quelle: [Balzert-II 3, S. 605]

53.2 Data Integration

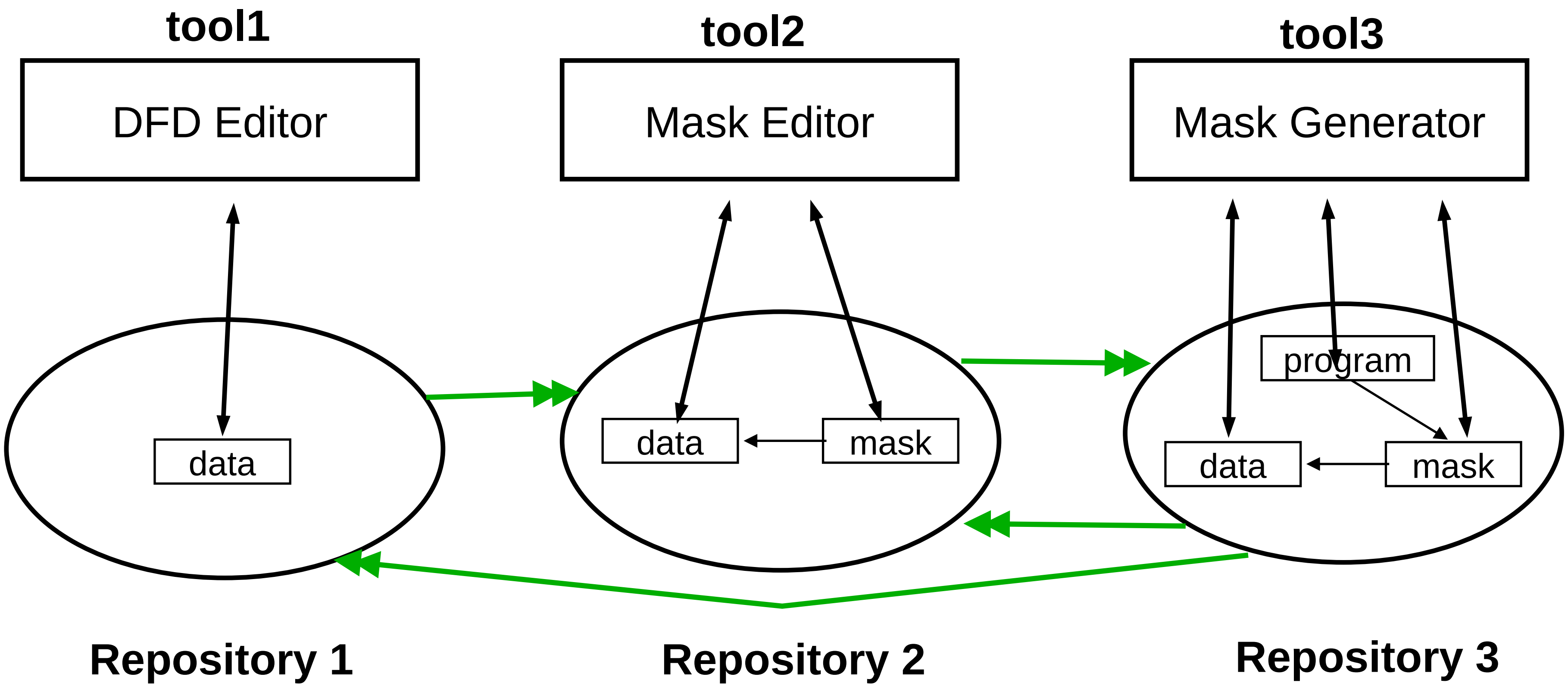
Levels of Data Integration

[Balzert-II, S. 608]

Integration Kind	Schema	Characteristics
Black-box-Integration (weak)		• Tool works isolated on own data • Repositoriy coordinates data with check in and check outl • mostly directory based, e.g., git
Grey-box-Integration (medium)		• Separate repositoriy, but common access interfaces • Access layer to repositories
White-box-Integration Data sharing (strong)		• Definition of uniform data schema (material metamodels) for all tools • Integration of data schemata of tools by • Role-based integration • Inheritance-based integration • dependent on Technical Space

a) Ad-hoc Data Exchange (Simple)

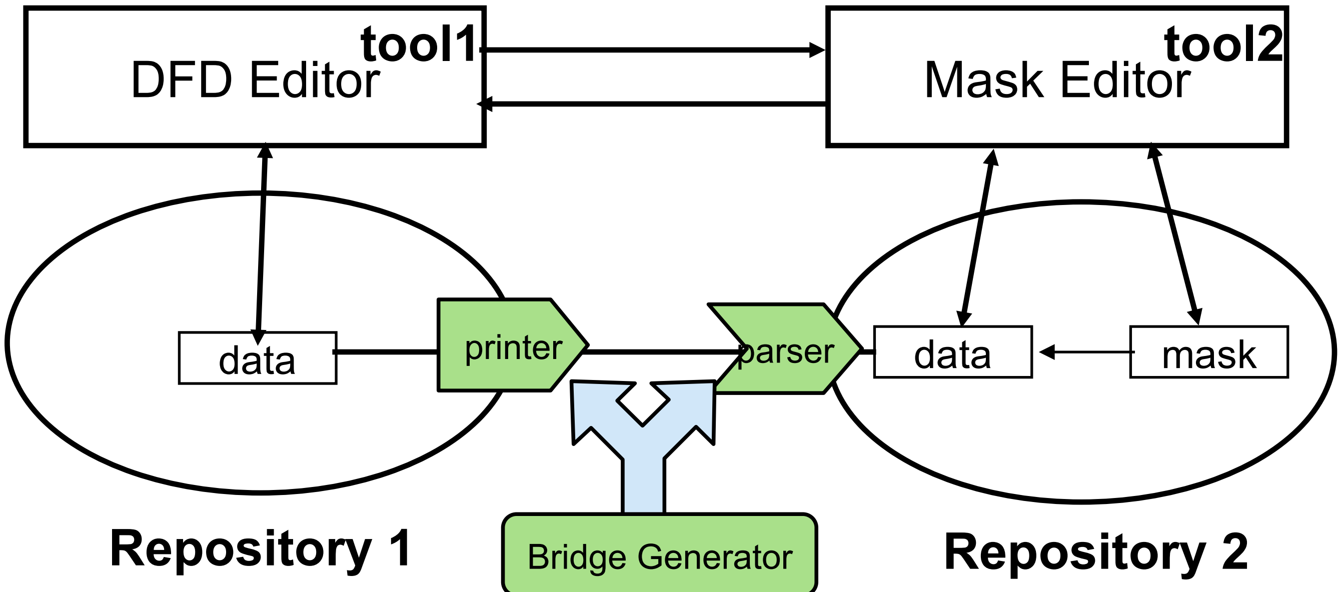
- No common data, high cost for exchange
- Exchange with Streams and Data-Flow Architecture (Example: UNIX shell)
  - Queries filter data, transformations change data
  - Data formats are defined in an exchange language
- Tools are rather independent



Quelle: nach [HMF. S. 196]

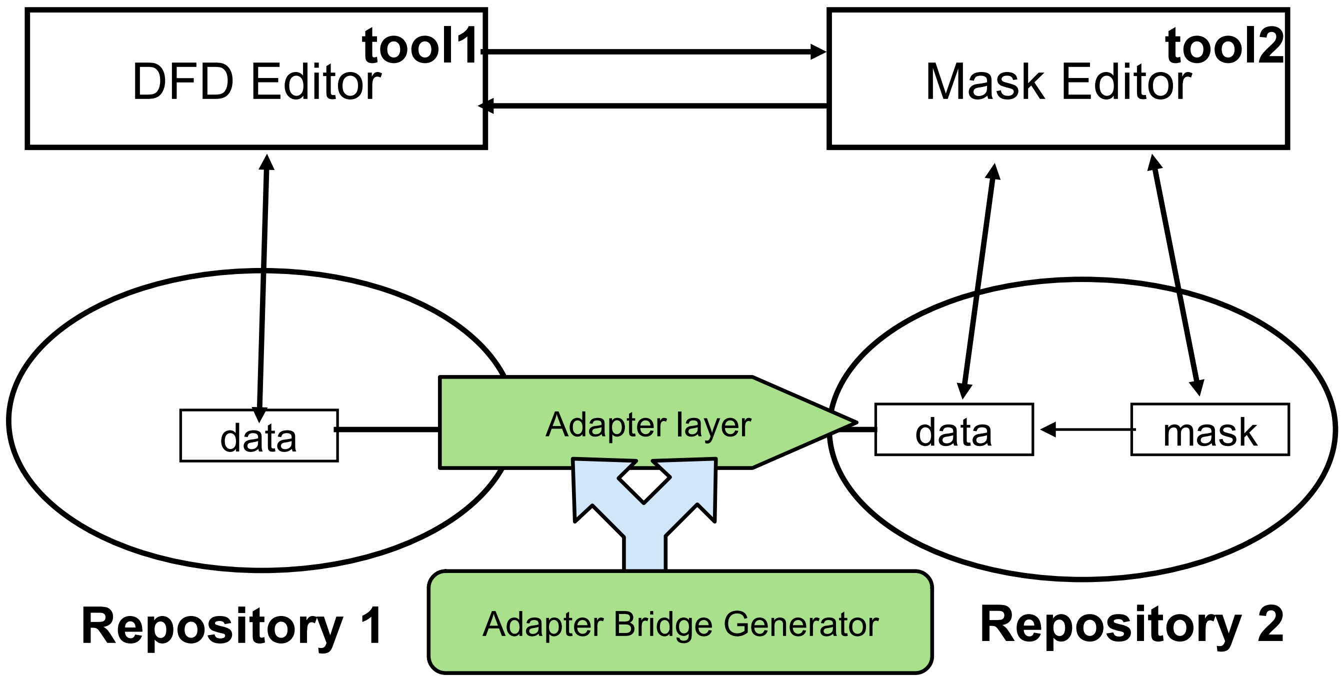
b) Data Connection with Systematic Data Exchange (Transformation Bridges)

- **Data connection (Datenverbindung)** relies on the definition of *mappings between the data schemata (material metamodels)*
- **Automated data exchange** in standardised **exhange formats (Exchange formats,** such as ASN, XMI, JSON, CDIF, XML)
  - Automation (generation of parsers and printers) relies on mappings between the material metamodels (language mappings)
  - Use as a technical space for the exchange format a link-tree metalanguage (XML, RAG)
- **Transformation bridge:** a prettyprinter transforms the internal representation of a repository into an external exchange format
  - Parser reads the exchange format again and transforms it into the internal representation of the other repository
  - Query and transformation languages filter the data



c) Data Connection with Incremental Data Exchange (Adapter Bridges)

- An **Adapter Bridge** is a layer between two repositories allowing for *incremental access* from one to the other
  - Transformation of data incrementally with each access
  - Direct transformation without exchange format



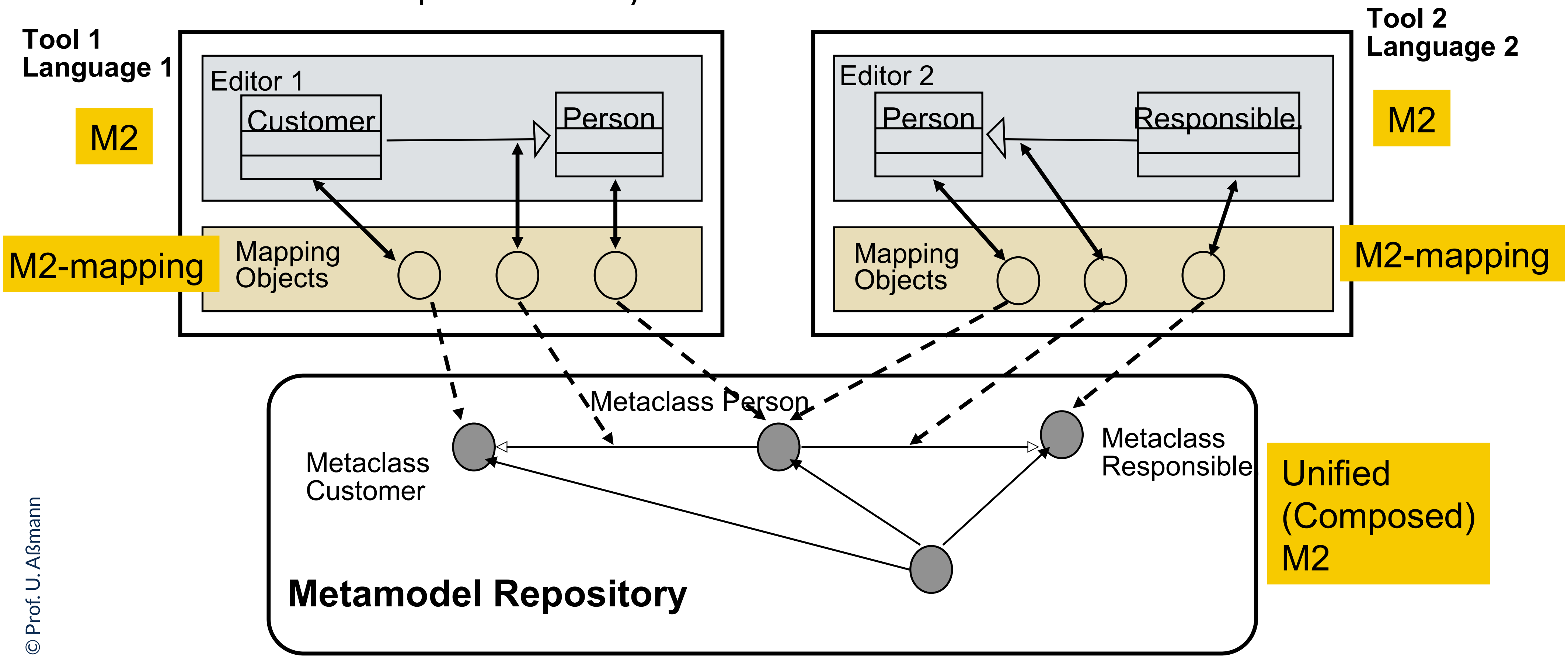
Current Exchange Formats for Data Connection

An ***exchange format*** is a standardized format for the exchange of data between tools, also of different vendors

- Comma-separated values (CSV): simple text-based exchange format for tools on relation, text algebra, and tables (Excel, TeX, ...)
  - No metalanguage but simple table schema http://tools.ietf.org/html/rfc4180
  - http://en.wikipedia.org/wiki/Comma-separated_values
- XML Metadata Interchange (XMI) for exchange of UML-diagrams in XML-format
  - Meta Object Facility (MOF) as metalanguage
  - http://www.omg.com/technology/documents/formal/xmi.htm
- JSON hierarchic maps
  - TOML, YAML variants are less wordy
- ASN.1 Standard is a metalanguage based on BNF
  - http://de.wikipedia.org/wiki/Abstract_Syntax_Notation_One
- RDF/RDFS Resource Description Format – Models as graphs, stored in elementary tripels http://www.w3c.org
- GXL Graph Exchange format: Open Source Format for exchange of graphs
  - http://www.gupro.de/GXL/
- Historic:
  - CASE Tool Data Interchange Format (CDIF) - metalanguage ERD for data definition as well as
  - Data Flow Model, State Event Model, Object Oriented Analysis and Design
  - http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-270.pdf

d) Data Sharing (With Common Repository)

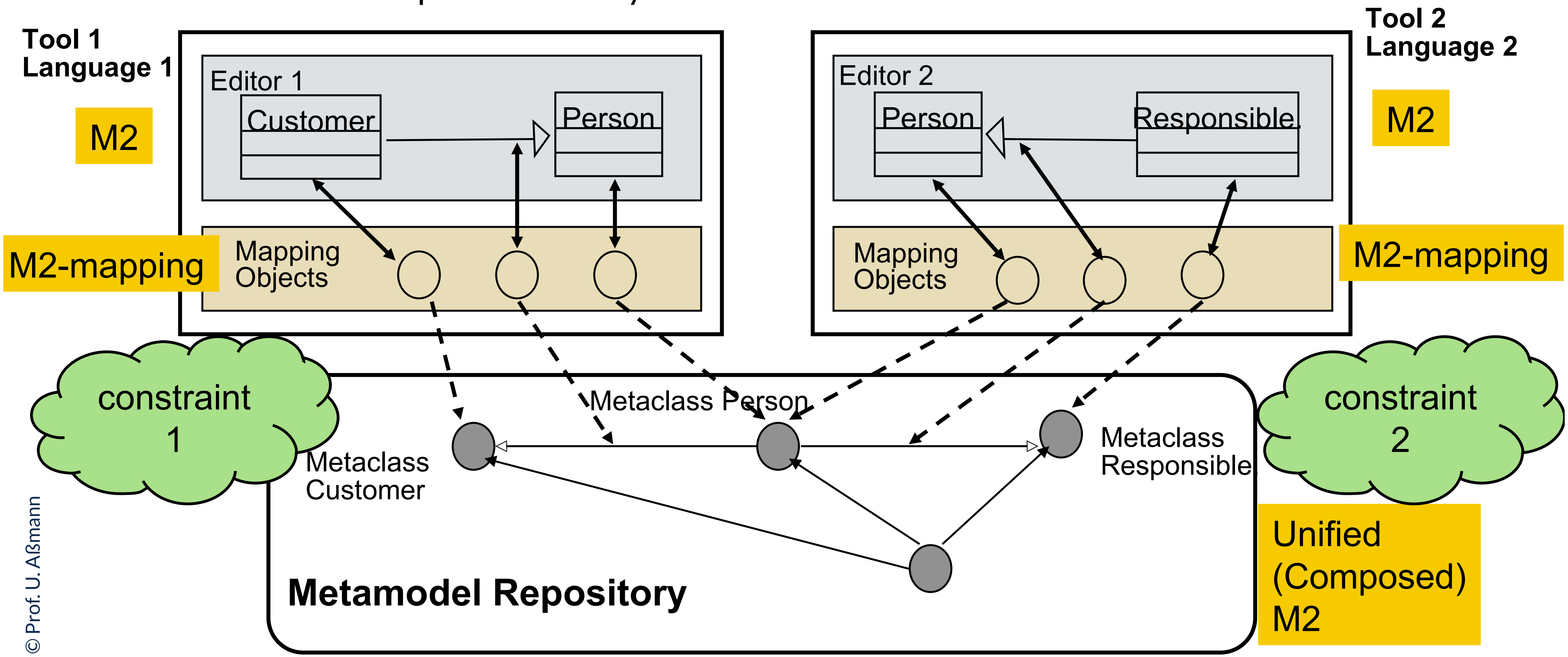
- With **data sharing (Datenteilung)** all tools share common data with a uniform schema (material metamodel)
- Metaclass mappings control the integration of the repositories and metamodels
- Metaclass compositions unify different views of metaclasses



Quelle: Platz, D.,Kelter, U.: Konsistenzerhaltung of Fensterinhalten in SEU; http://pi.informatik.uni-siegen.de

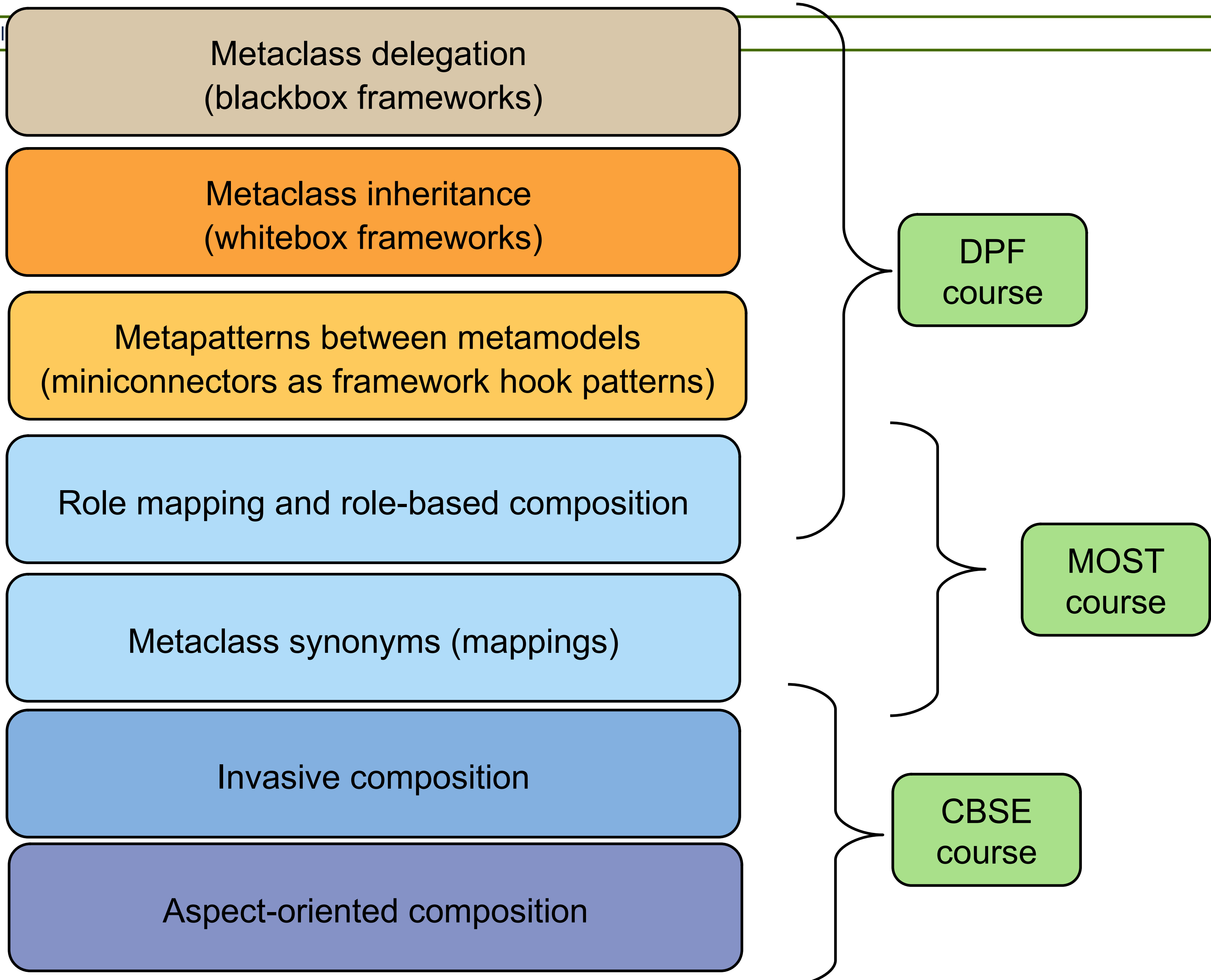
e) Data Sharing with Macromodels

- With **data sharing by macromodels** all tools share common model with a uniform metamodel **underlying wellformedness constraints**
- Metaclass mappings control the integration of the repositories and metamodels
- Metaclass compositions unify different views of metaclasses



Quelle: Platz, D.,Kelter, U.: Konsistenzerhaltung of Fensterinhalten in SEU; http://pi.informatik.uni-siegen.de

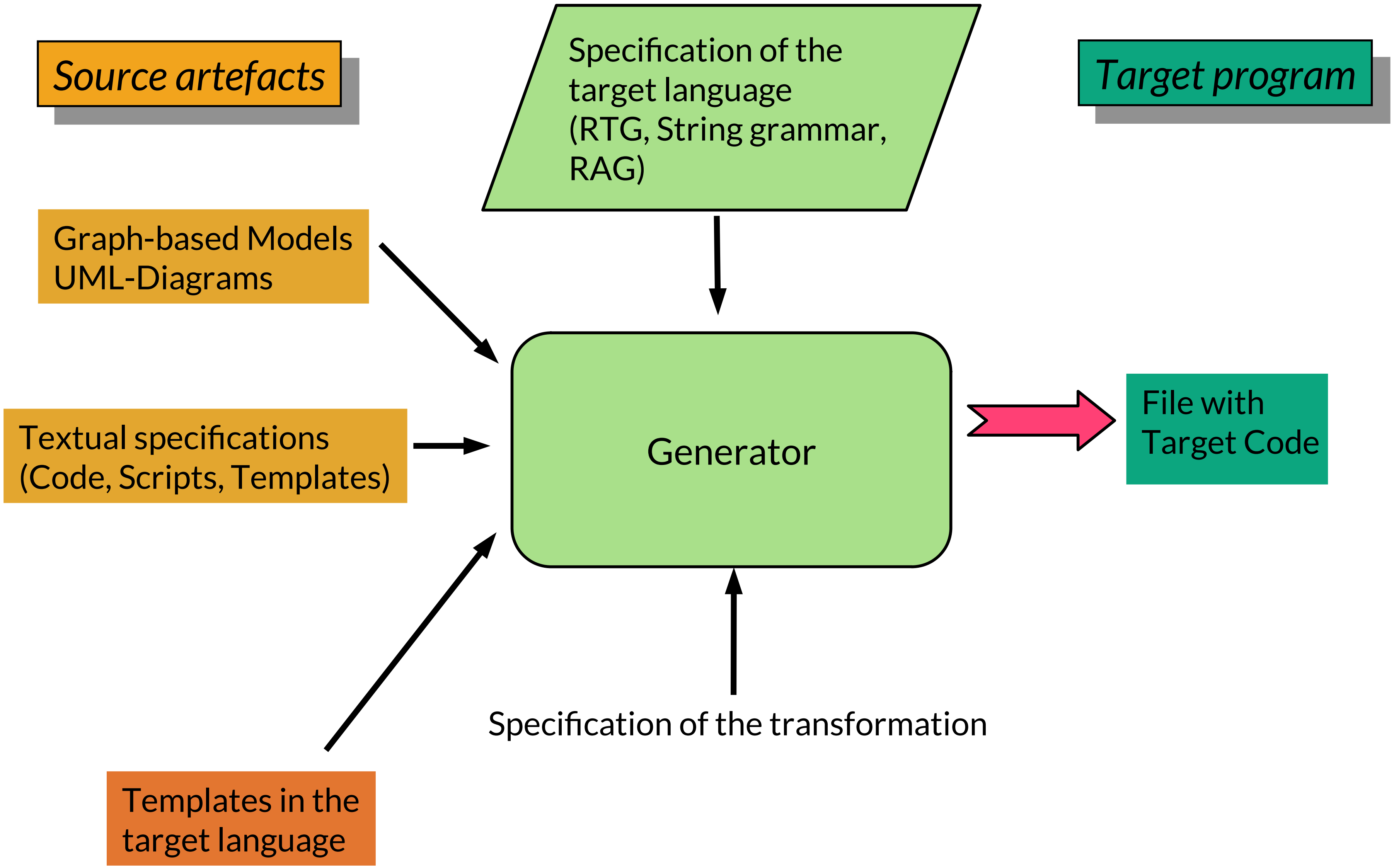
How to Compose and Relate Metaclasses and Metamodels for Data Sharing?



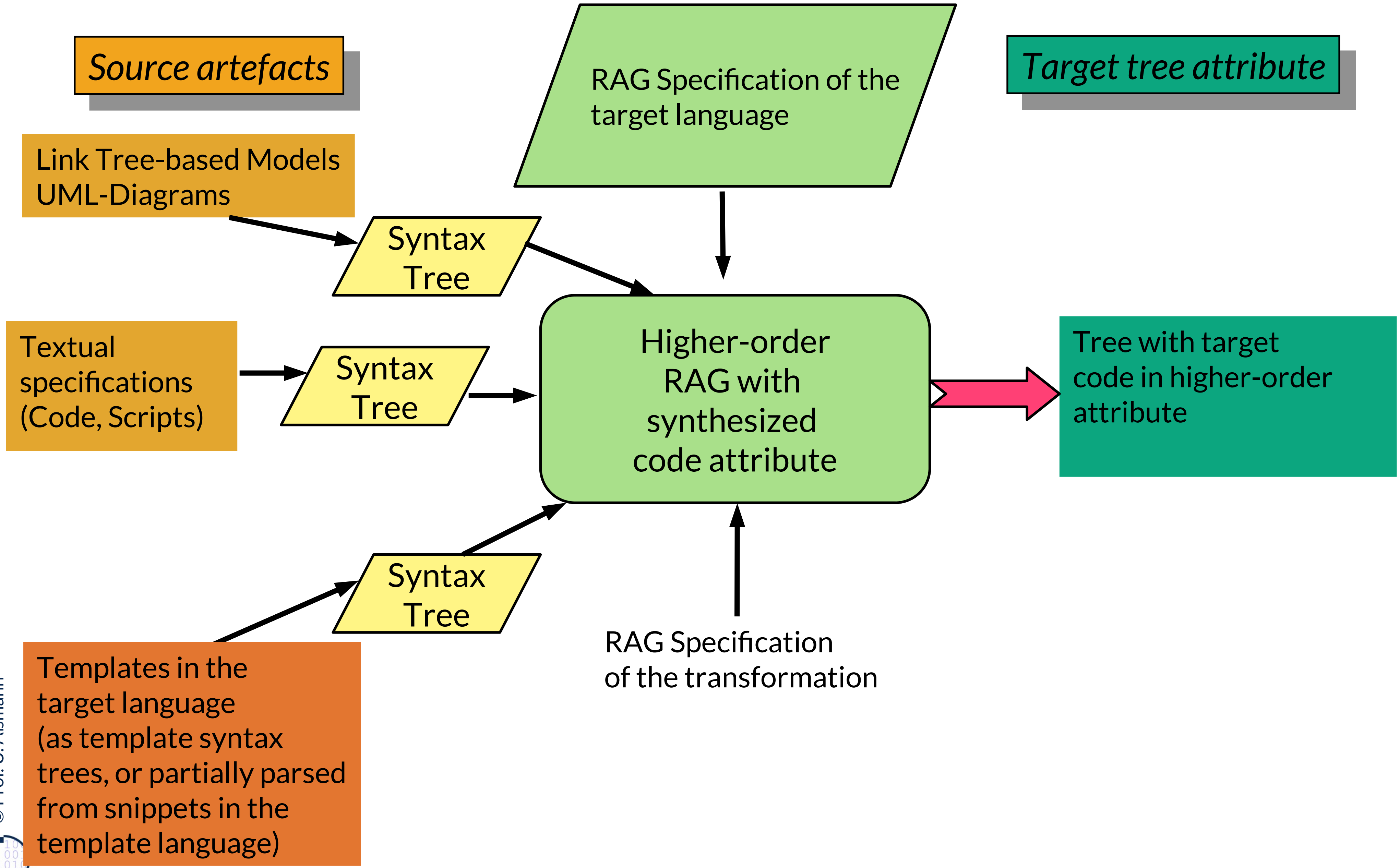
30.1 Model2Code Transformation (Code Generation)

Transforming models into code (Programmüberführung)

MDSD-Code-Generators



MDSD-Code-Generators as Attributors of Syntax Trees

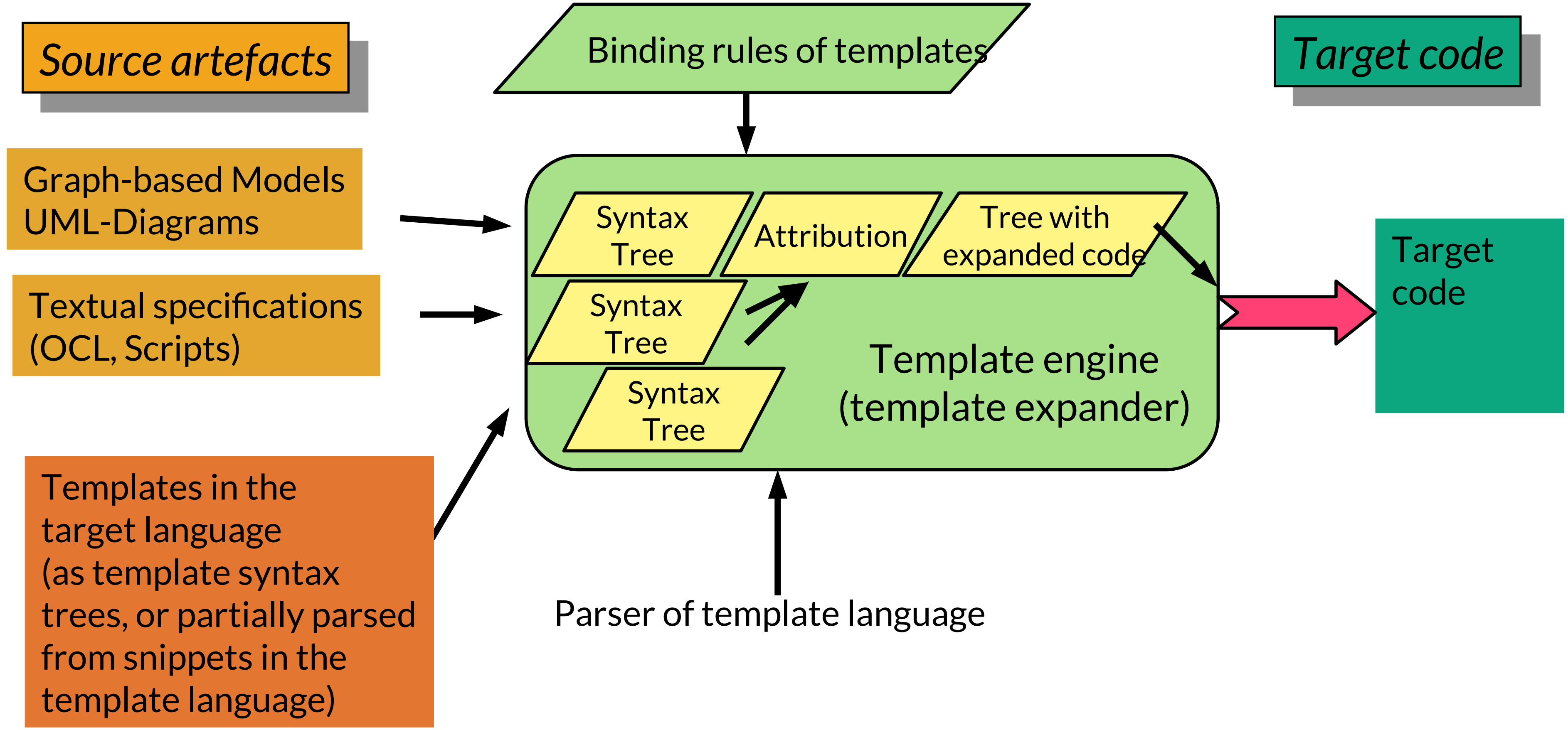


Different Kinds of Code Generators

- A **code selector** is a transformation system (on strings, terms, 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 functions
  - `templparse(): String in TemplateLanguage → Tree`
  - `pparse():      String in BaseLanguage → Tree`
- Template engines are *apps* of higher-order RAGs



30.1.2 Code Generation in RAGs

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

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

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"))
  );
}
```

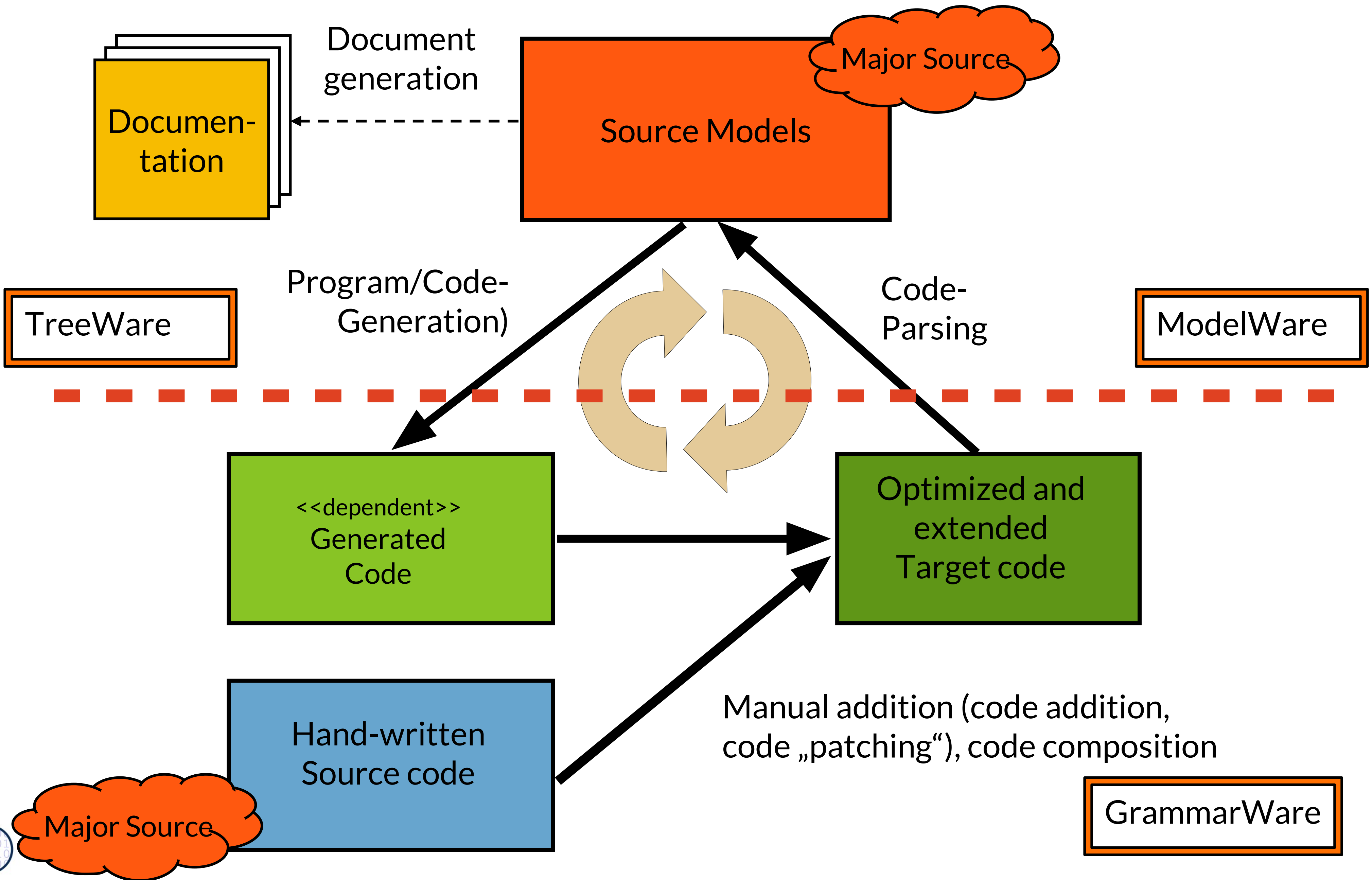
Template-Based Code Generation with RAGs

- 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`*

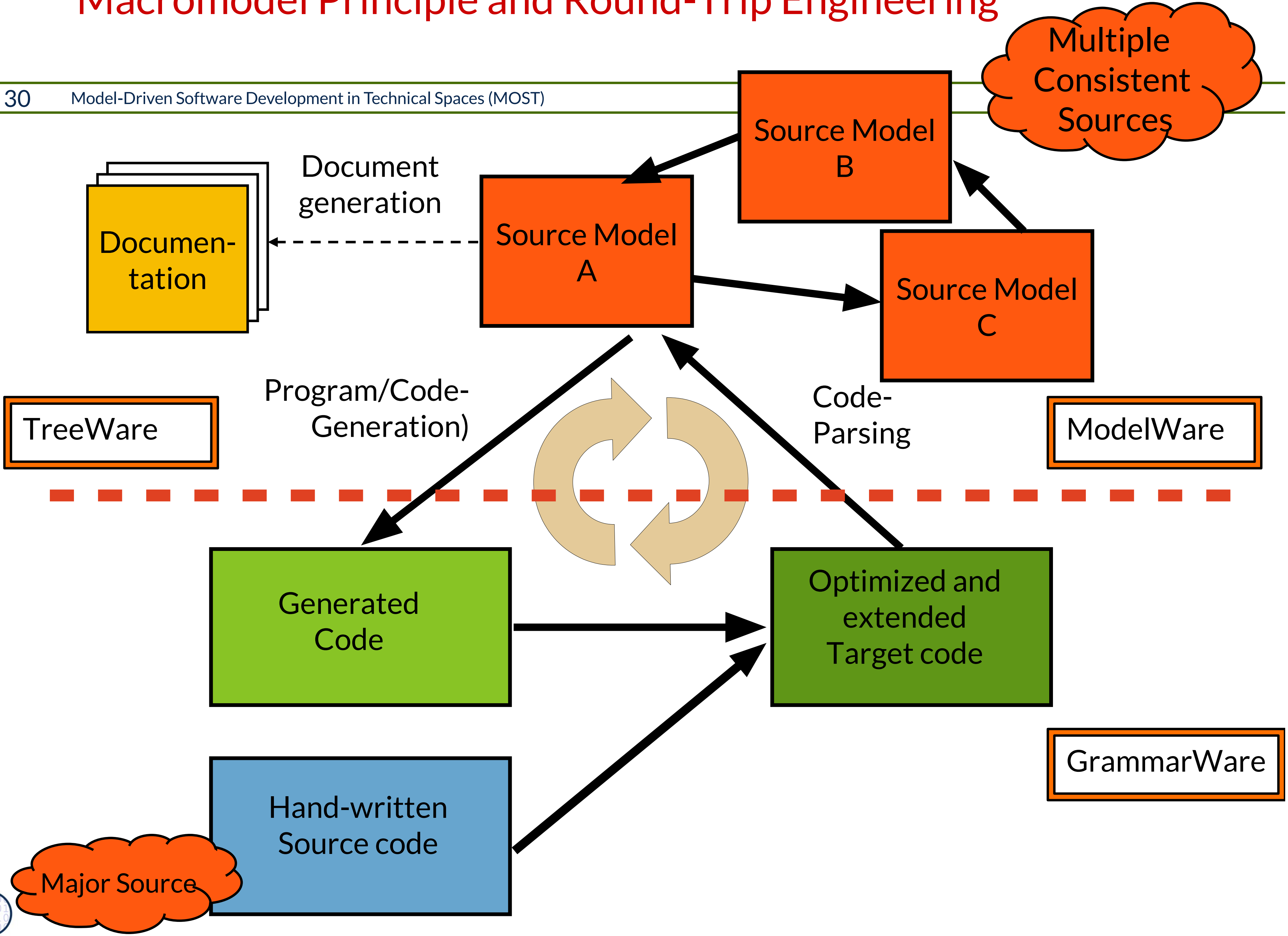
```
eq GenericClassInstantiation.Code() {
  CSGrammar CSAceleo;
  RTGrammar RTGAceleo;
  return templparseGeneric(CSAceleo, RTGAceleo,
    "public class GenClass$TypeParameterName$ extends Object {
         private int myId;
         public GenClass$TypeParameterName$() { // constructor
         }
 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

30.1.3 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** uses a *single source* (one set of *major-source* models), from which code, tests (derived models) are derived, with automatic synchronisation and consistency
- A **Macromodel Technology** is a single-source technology with automatic synchronisation and consistency between ALL (major-source) models, code, tests, and documentation (all models of a multi-model)
- In a macromodel, there are always ***derived models***
  - Generated code (this chapter)
  - Generated documentation (Chapter on documentation)
  - Generated test suites and data

Synchronization is Used in Round-Trip Engineering

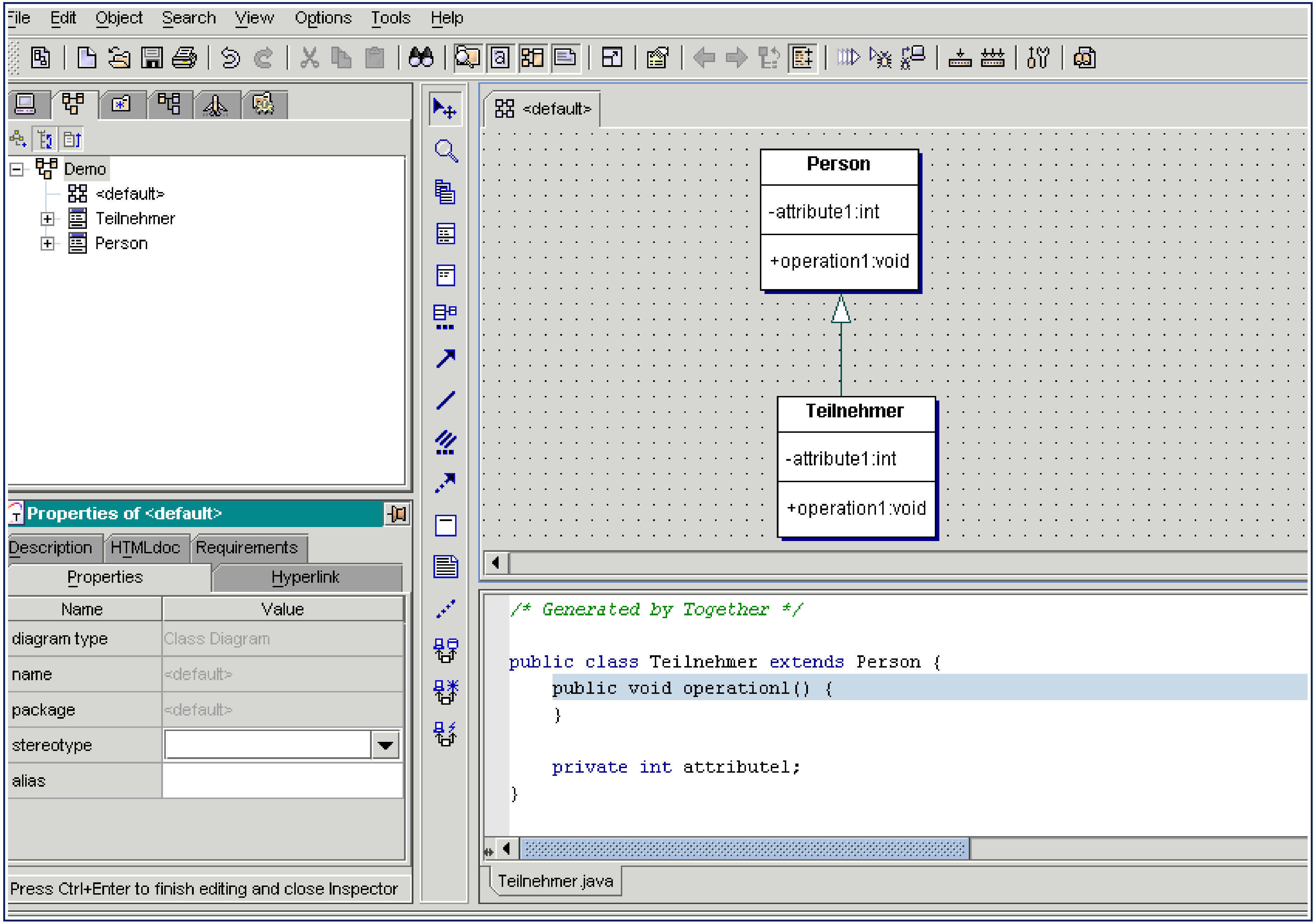
- Technically, the Single-Source-Principle and the Macromodel principle needs **Round-Trip-Engineering** (RTE) between ModelWare and GrammarWare, to achieve
  - **Model-to-code synchronisation** with
    - **Codegeneration** into several programming languages
    - **Template-based codegeneration** inserts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

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

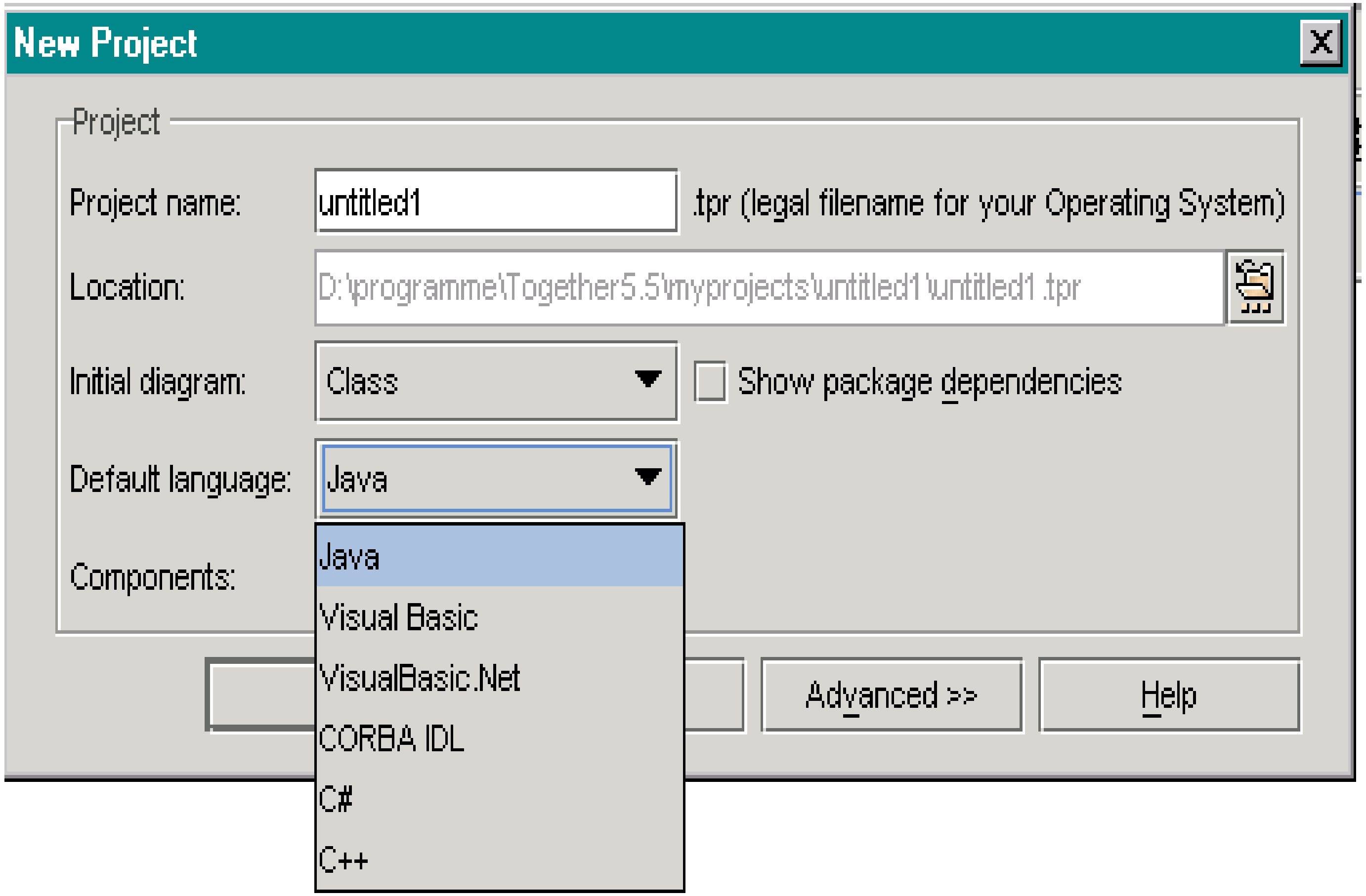
- In 1997, the CASE tool ***Together*** was the first to provide a Single-Source-Technology with automatic synchronisation and consistency between UML model, code and documentation
- Supported Programming Languages: Java, Visual Basic, VisualBasic.Net, CORBA IDL, C++, C#
  - Synchronisation by reparsing 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/de-DE/Products/Requirements-Management/Together/Testimonials
https://en.wikipedia.org/wiki/Borland_Together

Together Screenshot



Code Generation in Different Languages in Together



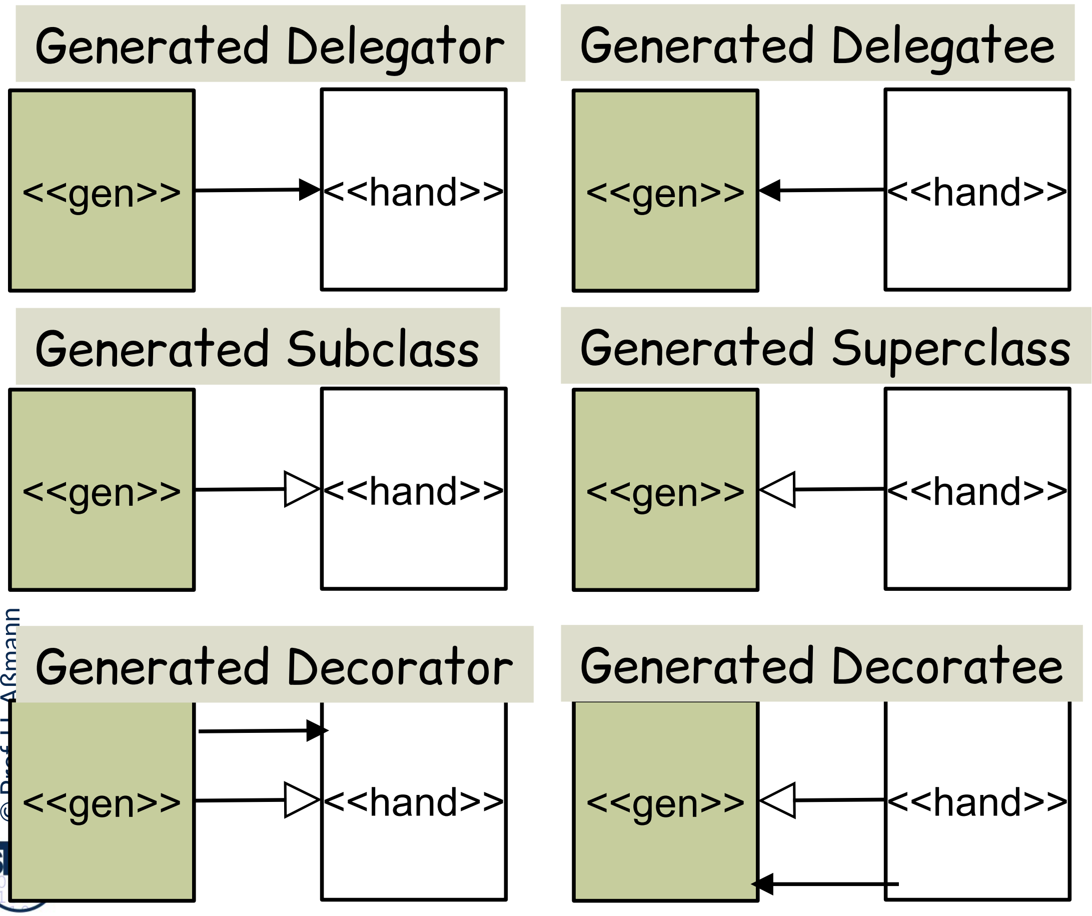
- Supports roles: Business Modeler, Designer, Developer and Programmer
- Appropriate views can be configured
- Code template based code generation

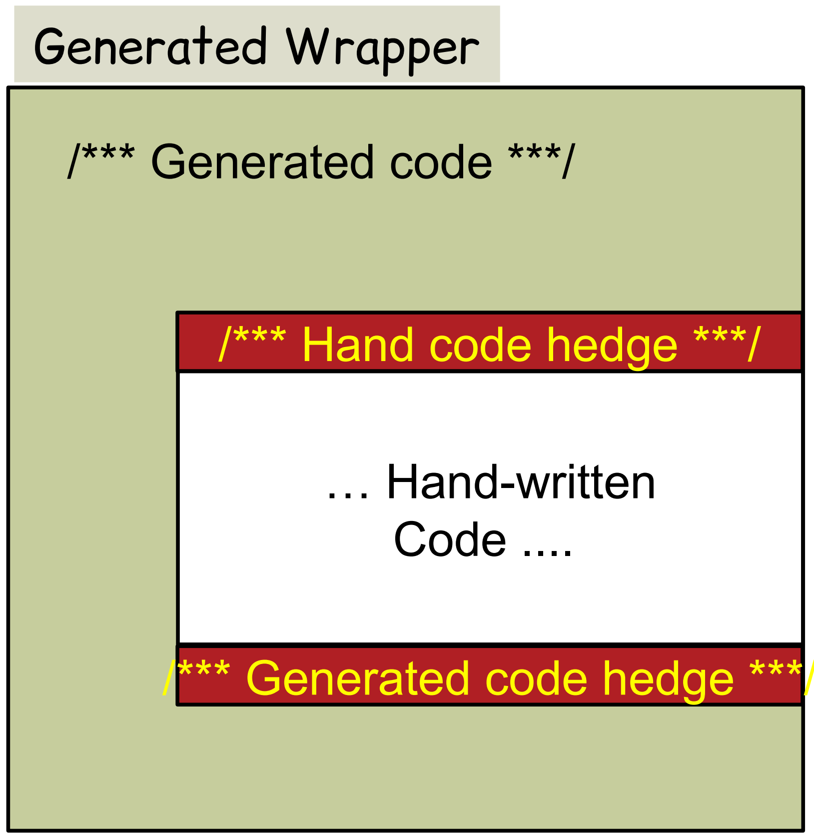
30.2 Technologies for Model-2-Code Generation and Synchronization

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
- Synchronization is easy: do not touch generatees

- **In one file:** Coupling with **hegdes (Trennmarkierung)**
- Synchronization should stay out of hedged areas





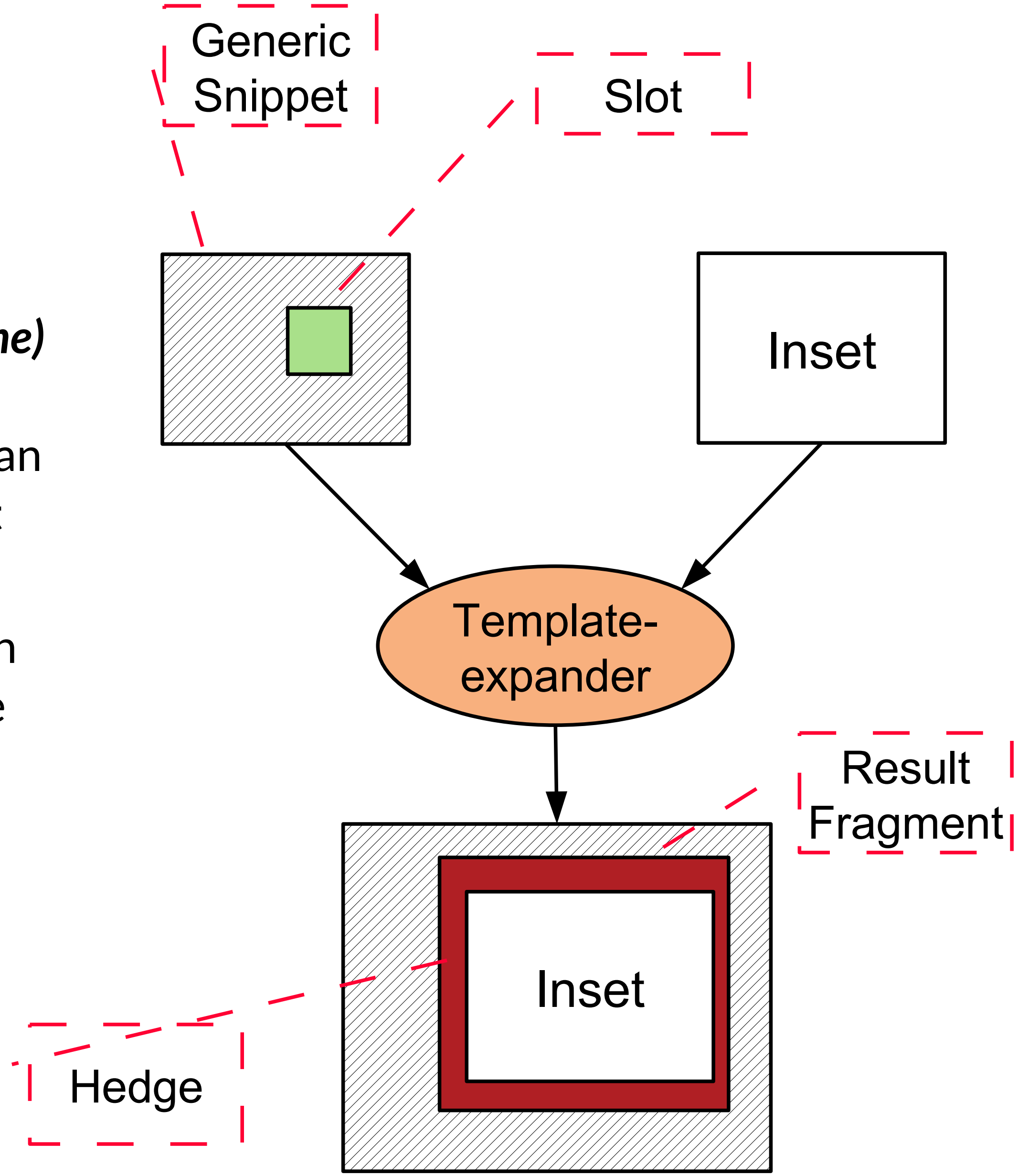
Composition of Generated and Hand Written Code in an RAG

- Fine-grained glueing possible
- Wrapping hedges around synthesized snippets

```
eq Procedure.Code() {
   return Head.Code()+
          "/*** HEDGE2Gen BEGIN */"+
          GeneratedBody.Code()+
          „/*** HEDGE2Gen END */";
}
eq Head.Code() {
   return pparse("public „+Head.name+"()");
}
eq GeneratedBody.Code() {
   return Body.Code();
}
```

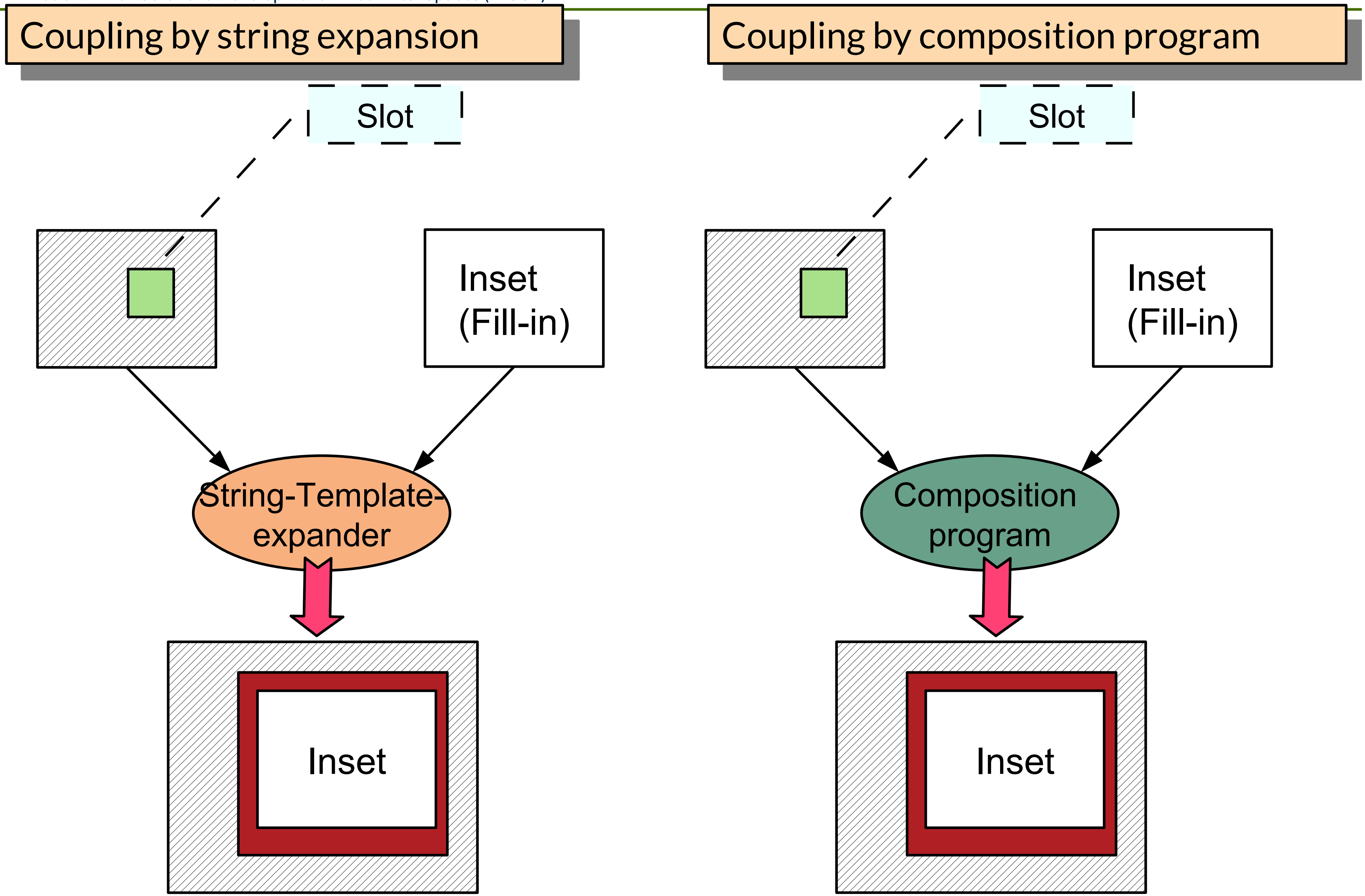
Snippet and Template Programming with RAG

- **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



30.2.1 Template-based Code Generation (Schablonenbasierte Programmüberführung)

Template Expansion by Composition of Insets and Hedging



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>> a);
   }
}
```

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
  - Acceleo https://www.eclipse.org/acceleo/

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>
```

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

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

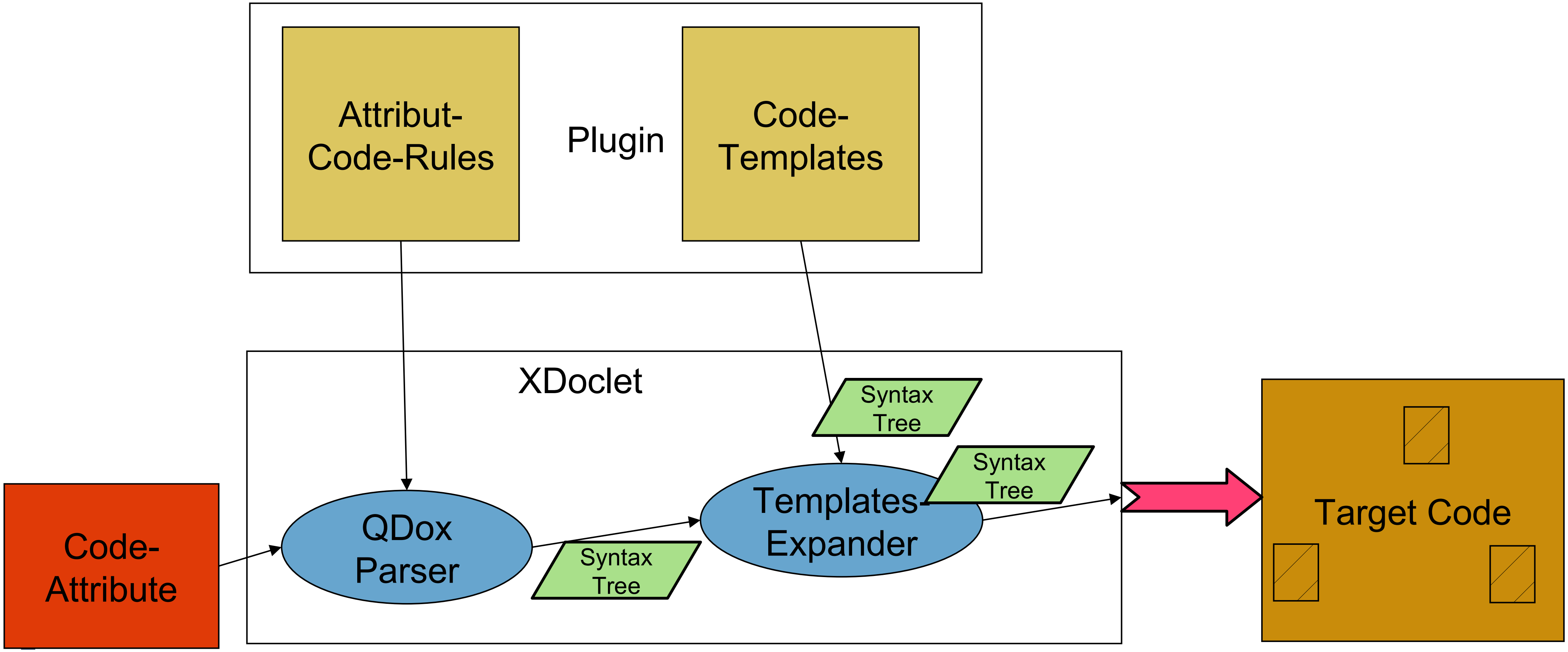
- Metamodel-controlled template engines
  - EMF: Xtend and Xpand scripting languages
  - XML slot markup language
  - Acceleo code generating system (see exercises)
- Invasive Software Composition provides fully typed and wellformed template expansion (see CBSE course)
  - Typed template expansion **and** -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

- **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



Slot Markup Languages

- 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?

30.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_EMCDA2008_slides.pdf

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