Dimensions of Transformations

1. vertical vs. horizontal
2. exogenous vs endogenous
3. model to model, text to model or model to text

What is a model to model transformation?

Automatic creation of target models out of source models (1-1,1-m,n-1 or n-m)

Transformation strategies?

1. Out-place transformations (build new model from scratch) eg. ATL
2. In-Place transformations (change parts in the model) eg. Graph Transformation

ATL

- Source-models are ready-only, Target models are write only
- ATL is a declarative-imperative hybrid
    - Declarative part: Matched rules
        - Source pattern to be matched in source model
        - Target pattern to be created in target model for EACH match during rule application
        - Optional action block
    - Imperative part: Called rules and lazy rules, Action blocks (do-block), global variables an helpers
        - Procedure called by its name
        - May takes arguments
        - Contains a target declarative pattern and optional action block
        - Called/Lazy rules are applied AS MANY TIMES as they are called from other rules
- Recommended style: declarative

Applying a rule in ATL means:

- Creating a target element and assign the values/properties

ATL Syntax

create OUT : Book from IN : Publication;

Variable:
helper def : VARNAME : TYP = Anfangswert
helper def: id : INTEGER = 0

Funktion:
helper context MATCHINGTYPE *def:* NAME[(PARAMS)] *:* RETURNTYPE = DEF

```
helper context Publication!Publication def : fromThisYear() : boolean =
	if self.year = '2014' then
		true
	else
		false
	endif;

rule NewPublication2Book {
	from p: Publication!Publication (p.fromThisYear())
	to b: Book!Book (
		book <- p.publication
	)
	do {

	}
}
```

Unterschied Called-Rule zu Lazy Rule:

Eine Called Rule, hat nur einen to Block, und hat als letztes Argument im do {} Block den return Wert.

Eine Lazy Rule, hat einen from und to Block, welcher über Patterns gematched wird, wird **nicht** automatisch aufgerufen.

In beiden fällen kann man Parameter übergeben.

ATL Transformation - Execution phases:

1. Module initialization phase: Variables and trace model are initialized
2. Matching phase: Matching patterns FROM => TO, each pattern/match must be unique
3. Target initialization phase: Elements in target are initialized based on <- and resolveTemp function evaluates based on traceability information. Imperative code is executed (do-Block)

Graph Transformations:

Graphs are Models.
The Type-Graph is the generalization of Graph elements.
The Type-Graph represents the Meta-Model, the Graph itself the Model.
A representation could be e.g. Graph = Objectdiagram, TypeGraph = Classdiagram

NAC:

Negative Application Condition defines what must not be existing to execute the rule. (Precondition which should not be satisfied to execute the transformation). It describes a forbidden subgraph structure.

Different kinds of model transformations:

1. Bidirectional Transformation
    a. Synchronization updates the model in case the integration mode has reported unsatisfied correspondences
    b. Integration (checks if elements produced by the transformation mode exist in the target model)
    c. Transformation (a -> b or b -> a, both directions possible). Forward and backwards.
2. Higher order Transformation (Takes model transformation as input and generates model transformation as output)
3. Batch transformation: Standard mode : Read complete input model, generate complete output model
4. Lazy- and Incremental Transformation:
    a. Incremental: An Output Model already exists, run for a given input model
    b. Lazy: Only a part of the model is needed by the customer
5. Transformation chains: Complex process, Divide and Conquer, Modeling orchestration of different model transformations

QVT:

Query: An expression evaluated over models
View: A model completely derived from another model
Transformations: Mapping from Source to Target Model (depended or completely independent)
Hybrid like ATL : Declarative and Imperative

Some Requirements for QVT:

1. Queries on models
2. Views on metamodels
3. Transformation on models
4. Complete generation of target model
5. Updates existing models
6. Bidirectional transformations
7. Checking model consistency

Unterschied WHEN und WHERE clause bei QVT:

Die WHEN-Clause (guard) sagt aus: Die aktuelle Regel muss nur halten, wenn die WHEN Clause ebenfalls hält. (Was muss vorher gelten)
A guard is a Boolean expression that can be seen as a pre-condition.

Die WHERE-Clause sagt aus: Wenn die aktuelle Regel gilt, muss auch die WHERE clause halten. (Was muss nachher gelten)
The where clause can be used to define a post-condition for a mapping operation

Unterschied zwischen CHECKONLY and ENFORCE bei QVT:

A QVT-R transformation can be used either in checkonly mode, to determine whether a target model is consistent with a given source model, or in enforce mode, to change the target model.

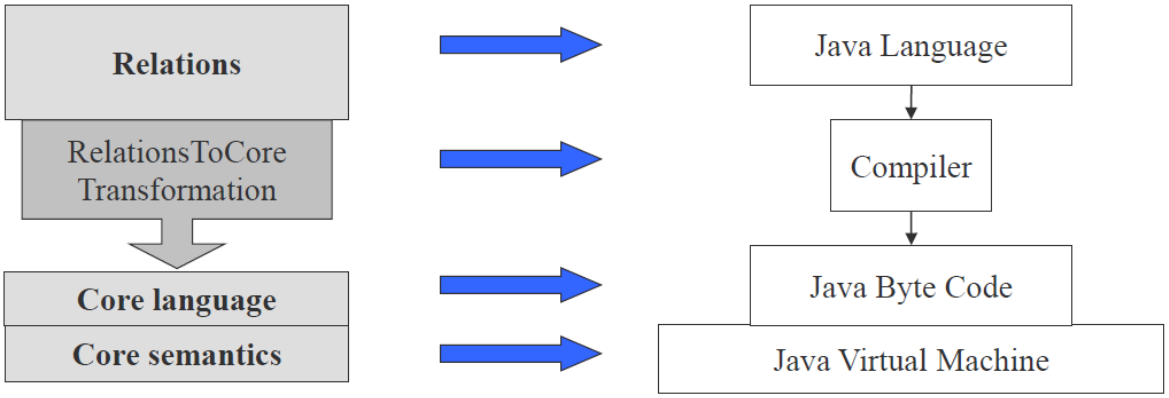
Wikipedia:
QVT-Relations is a declarative language designed to permit both unidirectional and bidirectional model transformations to be written. A transformation embodies a consistency relation on sets of models. Consistency can be checked by executing the transformation in **checkonly** mode; the transformation then returns True **if the set of models is consistent according to the transformation and False otherwise**. The same transformation can be used in **enforce** mode

to attempt to **modify one of the models so that the set of models will be consistent**. The QVT-Relations language has both a textual and a graphical concrete syntax.

QVT Relations

- Are defined on Metamodel M2 and applied on M1
- Execution of Relations supports Consistency Checking, Transforming and Updating models
- Declarative way of describing model transformations
- Textually and graphically expressions

QVT Analogy:



Unterschied CORE and RELATION Language bei QVT

CORE:

- Core is as powerful as Relation language
- Core is not so user friendly
- No graphical notation is provided for the core language
- Directly implemented
- Traces must be explicitly defined
- Pattern matching only over flat set of variables

Relation language must be first transformed to Core language (The relations language can not be executed directly).

Relation Language is like Code.
Core Language is like Byte-Code
Relations to Core lis like a Compiler

Unterschied declarative and imperative (QVT, etc.)

Declarative: Relation between source and target elements, focuses on the “WHAT” of a transformation (XSLT, Xquery, SQL)

Imperative: describes primarily the solution. Focuses on the “HOW” of a transformation (Java, PHP,..)

Development Process MDA Artifacts:

CIM (Computation Independent Model) => PIM (Platform Independent Model e.g. UML?) => PSM (Platform Specific Model e.g. SQL Model, Web Model) => CODE (DAO Code, SQL Code, …)

Why source code generation?

- Separation of application modeling and technical code
    - Increasing maintainability, extensibility, portability to new hardware, operating systems, and platforms
- Rapid prototyping
- Early and fast feedback due to demonstrations and test runs

Advantages, Disadvantages source code generation:

Advantages:
- No new transformation languages
- No new tool support necessary

Disadvantages:
- No separation of static output code and model information
- Structure of output code not nicely readable
- Similar problem like using Java Servlets

```
pw.println("<head><title>GuestBookServlet</title></head>");
```

- No declarative query language for model information

- Iterators, loops, and queries lead to lots of code
- Code for reading input models and persisting output code necessary

TESTFRAGEN:

(8pkt) Was ist der Unterschied zwischen einem deskriptiven und einem konstruktiven Modell?

Konstruktives Modell

ist eine Spezifikation des Systems, die mit Hilfe eines automatischen Generators zur Codegenerierung eingesetzt wird. Die Veränderung eines konstruktiven Modells hat die direkte Veränderung des Produkts zur Folge. Die Modellierungssprache wird auch als High-Level-Programmiersprache angesehen, da sie grundsätzlich ausführbar ist.

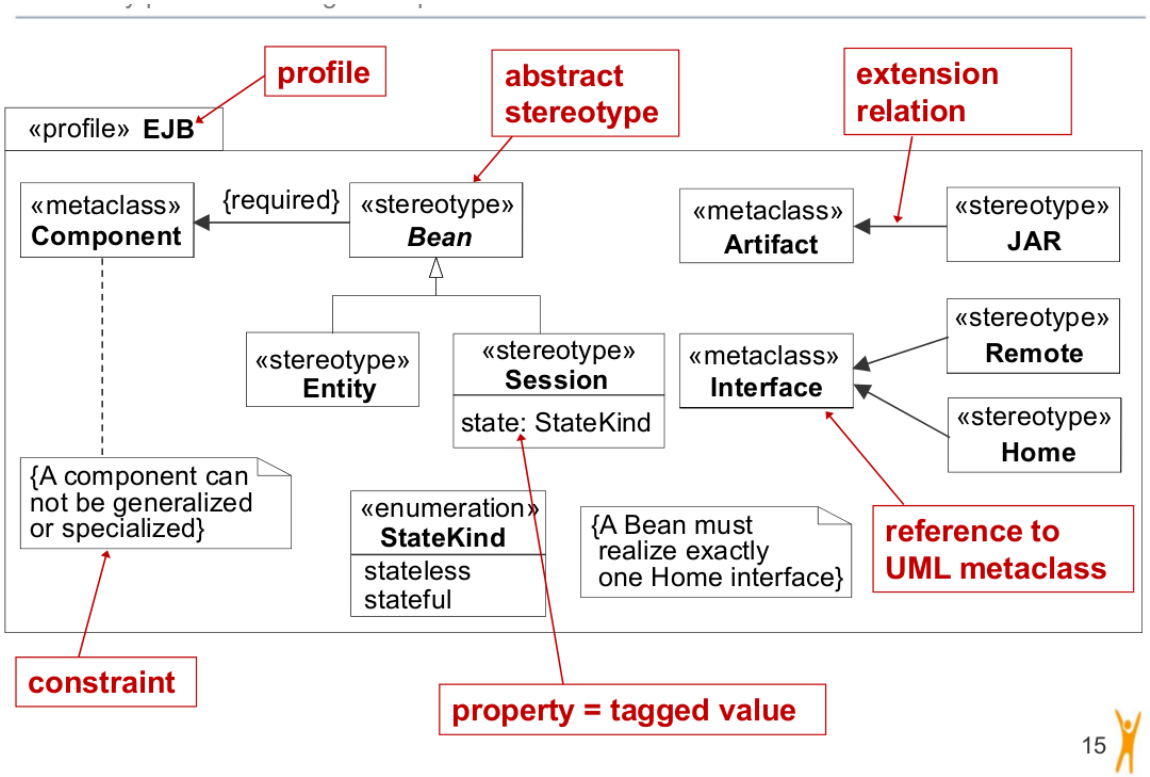
Deskriptives Modell

ist eine Spezifikation, die zur Beschreibung des Systems verwendet wird, ohne konstruktiv bei der Implementierung eingesetzt zu werden. Das sind typischerweise abstrakte und unvollständige Beschreibungen, also insbesondere Modelle, die als Vorlage für eine manuelle Implementierung dienen oder erst nach Systemerstellung als Dokumentation verfasst werden.

Quelle: http://mbse.se-rwth.de/book2/index.php?c=chapter4-1

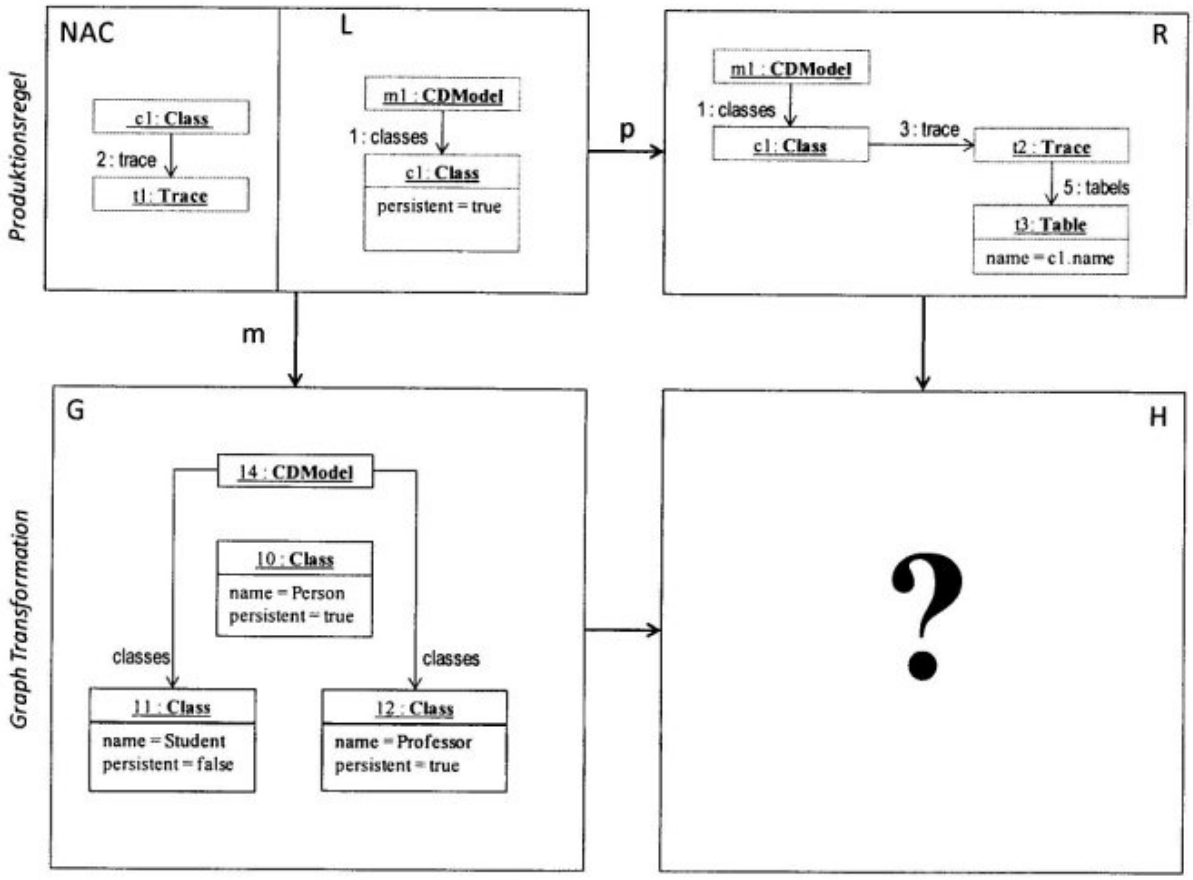
Anhand eines selbst gezeichnet Beispiels die 5 Notationselemente erklären. (10P)

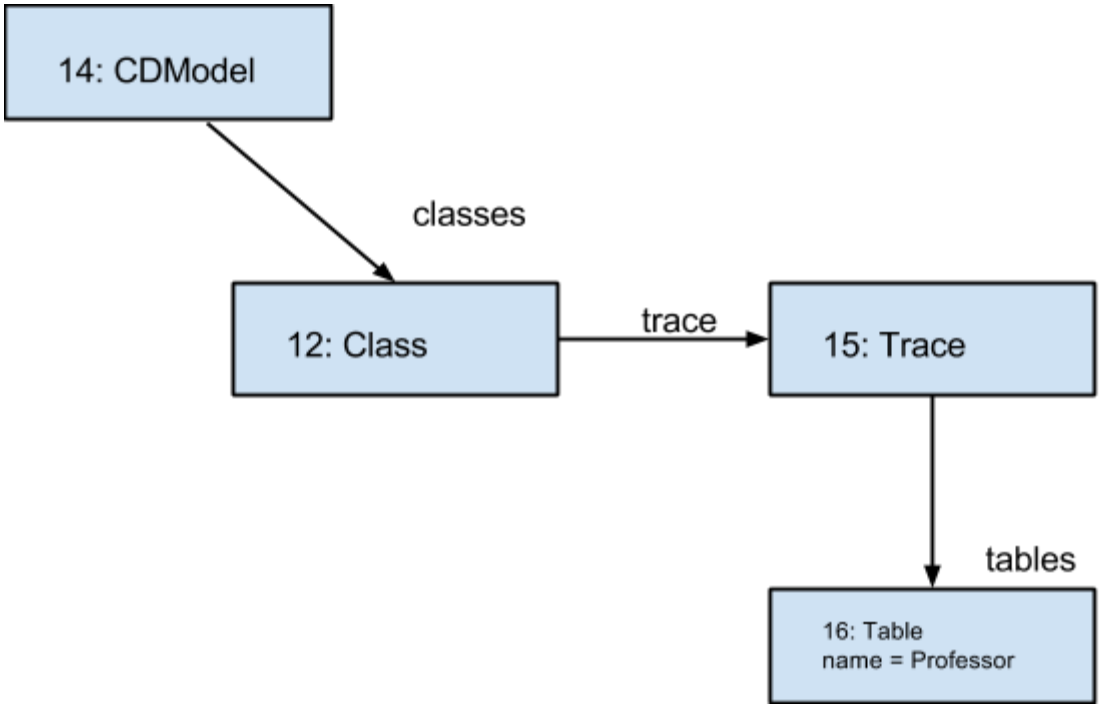
1. sterotype
2. constraints
3. extensions
4. tagged value (property)
5. reference to UML metaclass



Graphentransformation:

- Gegeben war ein Graph samt Produktionsregel. Der Ergebnisgraph war zu zeichnen.
- Frage: Was würde bei der Graphtransformation passieren wenn man NAC weglässt?


Produktionsregel
NAC
c1 : Class
2 : trace
t1 : Trace
L
m1 : CDModel
1 : classes
c1 : Class
persistent = true
p
R
m1 : CDModel
1 : classes
c1 : Class
3 : trace
t2 : Trace
5 : tabels
t3 : Table
name = c1.name
m
Graph Transformation
G
14 : CDModel
10 : Class
name = Person
persistent = true
classes
classes
11 : Class
name = Student
persistent = false
12 : Class
name = Professor
persistent = true
H
?


14: CDModel
classes
12: Class
trace
15: Trace
tables
16: Table
name = Professor

Nichts würde passieren, weil dies nicht vorkommt

Expand und Define anhand von selbst gewählten Bsp erklären?

DEFINE blocks represent templates and consist of a template name, a parameter list, and a name of the meta class, for which the template is defined, i.e., to which this template should be applied

«DEFINE templateName(parameterList) FOR metaClass»
    Sequence of statements and arbitrary text
«ENDDEFINE»

«EXPAND templateName [(parameterList)] [FOR exp | FOREACH exp [SEPARATOR exp]]»

EXPAND block calls a DEFINE block
The called DEFINE block adds its text output to the calling block

3 Erweiterungsmöglichkeiten für UML nennen + Vorteile/Nachteile???!

1. Variant1: New Metamodel
2. Variant2: Heavy-weight Extension (uncontrolled 1st class extension of the UML Metamodel)
3. Variant3: Light-weight Extension (controlled extension of the UML Metamodel on base of inherent extension mechanisms): Stereotypes and Profiles

Domain Spec. Models vs UML Profiles:

UML:

+ Lightweight language extension (no need to change metamodel and framework)
+ dynamic model extension (just extend existing models)
+ preventing metamodel pollution (metamodels represent from modeling domain, specific is defined in profile)
+ model-based representation (reuse frameworks, well defined)

- Sprache wird erweitert und kann überladen sein
- UML Profiles for UML only

DMS:

+ komplett neue sprache für ein spezifisches Problem

- Es muss alles von Grund auf erstellt werden

XPAND Allgemeine Informationen:

- An exception is thrown, if the FOREACH statement type does not correspond to the called DEFINE block
- XPand supports polymorphism for templates (If there exist two or more templates with the same name, then the most specific template is executed)
- Protected Regions mark an area in the generated code, which must not be overwritten by subsequent generator runs

XTEND2

- Is a statically typed language like Java
- Modern tool support
- Better expressions
- faster (XPAND interpreter is slow)
- Better features (lambda expression, new methods, build-in methods, etc.)

Template is given by ‘’’ like

```
def someHTML(List<Paragraph> paragraphs) '''
  <html>
    <body>
      «FOR p : paragraphs BEFORE '<div>' SEPARATOR '</div><div' AFTER '</div>'»
        «IF p.headline != null»
          <h1>«p.headline»</h1>
        «ENDIF»
        <p>
          «p.text»
        </p>
      «ENDFOR»
    </body>
  </html>
'''
```

UML Profiles

Wichtig zur Syntax: <<metamodel>> muss vorhanden sein, von diesem erweitern die <<stereotypes>>. Erweiterung hat schwarzen, ausgefüllten Pfeil.

Von <<stereotypes>> können andere <<stereotypes>> erben.

Unterschied Inheritance vs Extension - Relationships MOF und UML

Fall1:
Bei UML, kann man für eine Erweiterung einen Stereotypen auf eine Metaklasse erweitern. Dieser Stereotyp, erweitert diese Metaklasse sowie alle Teilnehmer die von dieser Klasse erben.

Bei MOF, gibt es diese Erweiterungsmöglichkeit nicht. Hier muss man, um eine Erweiterung einer Klasse zu erstellen, eine eigene Spezialisierung (Vererbung) dieser Klasse vornehmen.

Fazit: Bei UML kann man eine komplette Gruppe (Parent + ChildNodes) erweitern, während man bei MOF einen eigenen Typen (durch Generalisierung) erstellen muss, und die Erweiterung auch nur für diesen Typen gilt.

Fall2:
Bei UML Profiles, kann man mehrere Metaklassen durch einen Stereotypen erweitern. Hat man nun eine Instanz dieses Stereotypen, kann dieser entweder zur Metaklasse A oder Metaklasse B gehören, wobei dies aber keine Zusammenlegung (Merge) dieser Metaklassen bedeutet. Kurz gesagt: Der Stereotyp kann entweder zu A, oder zu B gehören, aber nicht zwangshalber zu A und B.

Bei MOF, wenn es eine Erweiterung zu mehreren Klassen gibt, erbt diese alle Elemente der Parent-Klassen. Somit gehört die Erweiterung immer zu allen Parents dazu (Merge-Aspekt).

Fazit: Bei UML Profiles ist es möglich, eine Erweiterung zu modellieren, die für mehrere Klassen instanzierbar ist, ohne dass diese Erweiterung von allen Parentklassen erbt und eine Zusammenlegung darstellt.
.
Fall3:
Bei UML Profiles ist es möglich, eine Metaklasse durch mehrere Stereotypes zu erweitern. Dadurch kann eine Instanz der Metaklasse alle Stereotypen anführen (e.g. <<stereotyp1>><<stereotyp2>>...<<stereotypN>>)

Bei MOF hat man ein XOR. D.h. eine Instanz kann nur einen Typen der Erweiterungen annehmen, aber nicht alle wie bei UML.

Wofür Template-Engine bzw. was ist ihre Aufgabe?

Program replaces meta-markers with data at runtime and produces output documents

Profile vs MOF:

http://www.uml-diagrams.org/profile-diagrams.html

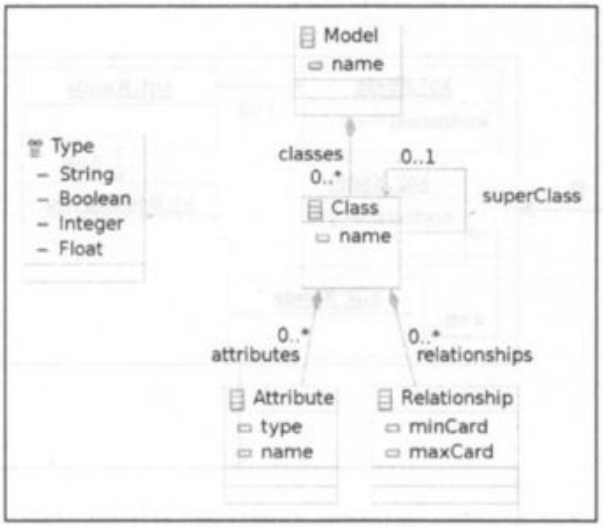
MOF:

- Creation of new meta models
- No extension of existing models
- Supported by metamodeling tools

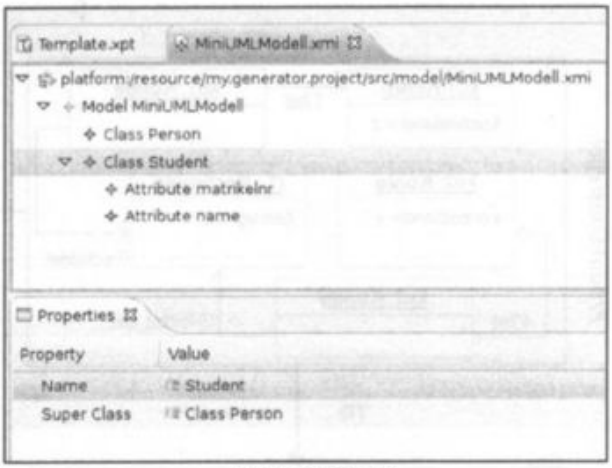
Profile:

- Extension of UML
- Supported by UML tools
- Guarantee UML compliant models

XPand Example:



MiniUMLMetamodell



MiniUMLModell

```
public class Person {
      public Person() { }
}

public class Student extends Person {
      private String matrikelnr;
      private String name;

      public Student() { }
}
```

Generierter Java Code

```
<< IMPORT MiniUMLMetamodell >>
<<DEFINE model FOR Model>>
        <<EXPAND createClass FOREACH Model.classes>>
<<ENDDEFINE>>

<<DEFINE createClass FOR Class>
        public class <<Class.name>> <<EXPAND MySuperClass FOR Class.superClass>> {
                <<EXPAND MyAttributes FOREACH Class.attributes>>
        }
<<ENDDEFINE>>
```

```
<<DEFINE MySuperClass FOR Class>>
	extends <<Class.name>>
<<ENDDEFINE>>

<<DEFINE MyAttributes FOR Attribute>>
	private <<Attribute.type>> <<Attribute.name>>;
<<ENDDEFINE>>
```