SE SOFTWARE ENGINEERING

RWTH AACHEN UNIVERSITY

Generating Systems from UML

MODELS 2011 Tutorial

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Presenters

Bernhard Rumpe
rumpe @ se-rwth.de

Martin Schindler
schindler @ se-rwth.de

Ingo Weisemöller
weisemoeller @ se-rwth.de

Software Engineering
RWTH Aachen University
Ahornstr. 55
52074 Aachen
Germany

SE SOFTWARE ENGINEERING

RWTH AACHEN UNIVERSITY

Generating Systems from UML

1. Introduction

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Concept of a Model?

A model is essentially a reduced or abstracted representation of the original system in terms of measure, precision and functionality. (Stachowiak 1973)

- Mapping feature: A model is based on an original.
- Reduction feature: A model only reflects a (relevant) selection of the original's properties.
- Pragmatic feature: A model needs to (be) usable in place of the original with respect to some purpose. (Kühne 2004)

Use of Models

- Example from the Internet:
- Note for the use of mathematical models:
  - Do not apply any model calculation, as long as you did not ensure that the simplifications on which they are based on do prevent their applicability
- Be sure to follow the instruction before using a model!
- Do not confuse the model with reality.
  - Note: Do not try to eat the menu!

Modeling in Software Engineering

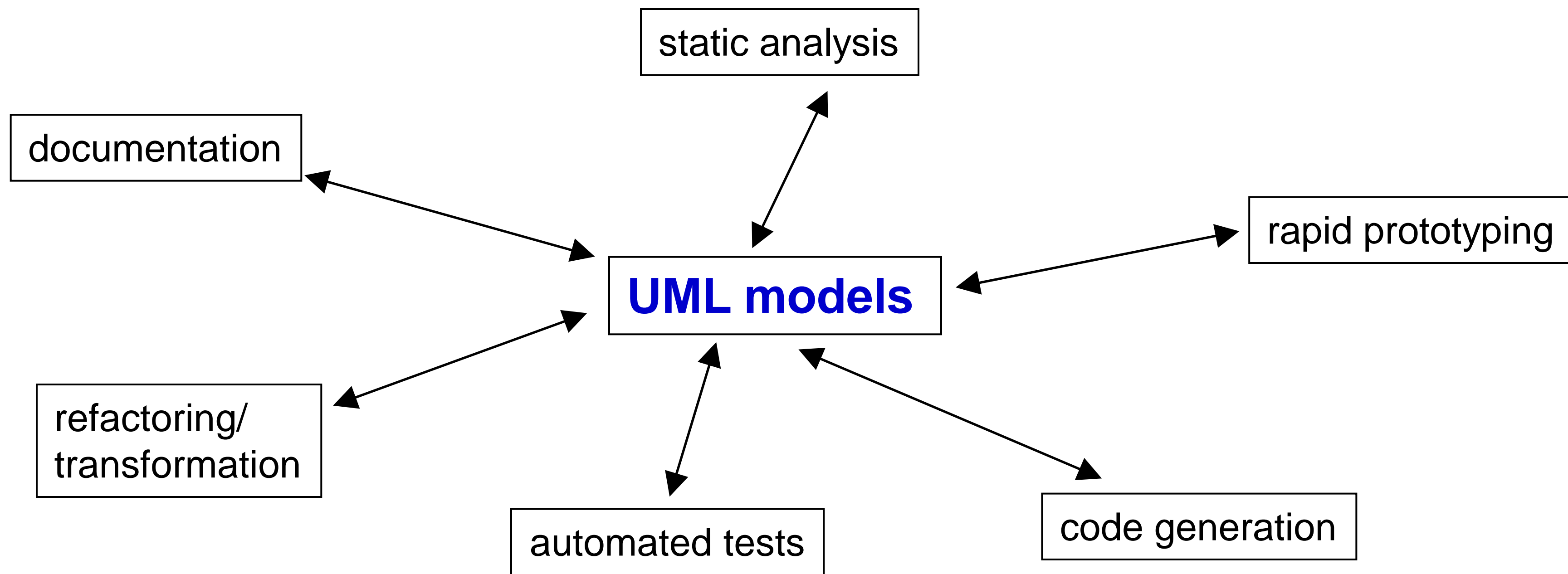
- Industry standard: Unified Modeling Language
  - 13 kinds of diagrams (Class diagrams, Statecharts etc.)

- But also:
  - Petri nets
  - logic
  - relations
  - data flow diagram
  - Nassi-Shneidermann diagrams
  - SDL
  - finite automata
  - etc.

  algebraic specification
  entity-relationship-model
  Jackson structured diagrams
  control flow diagram
  grammars
  regular expressions

Model Based Development with the UML

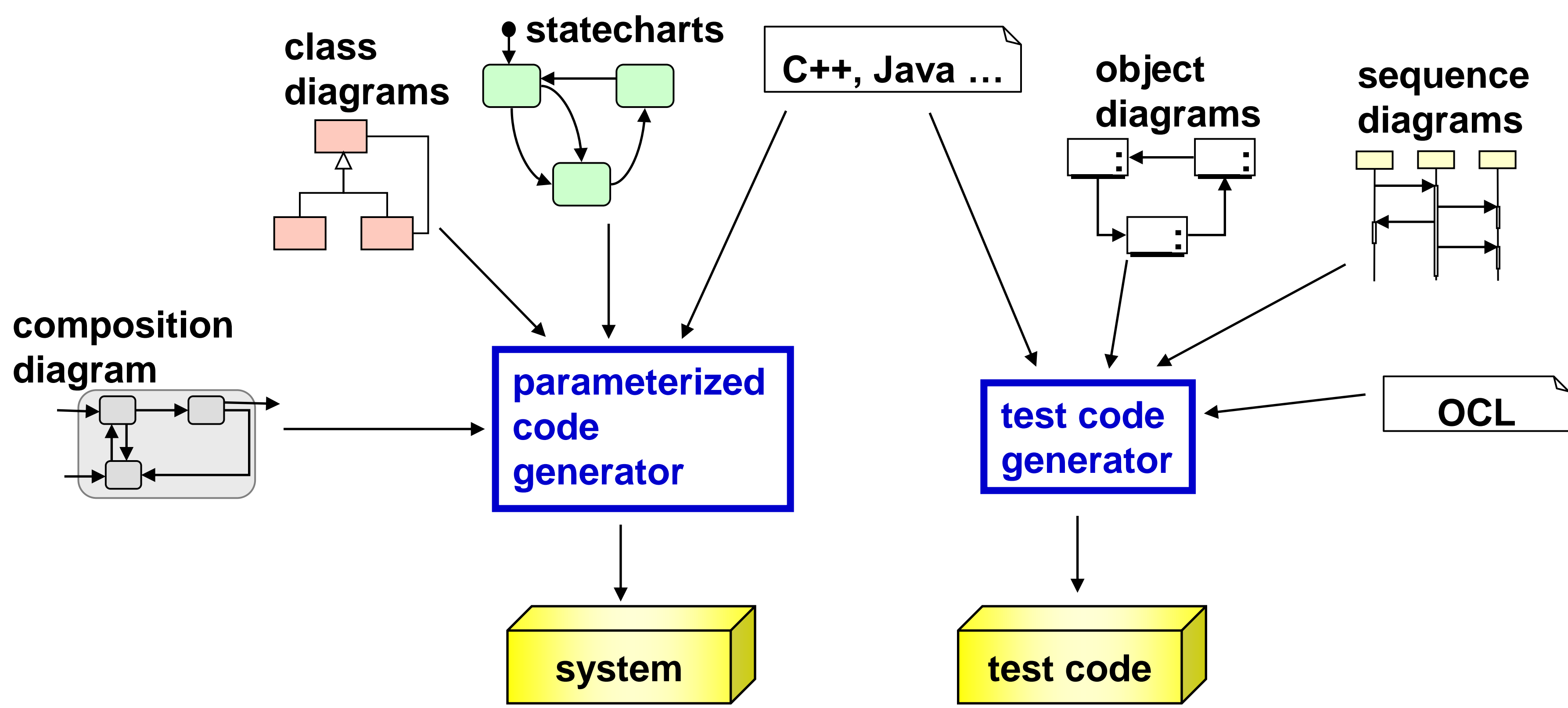
- Models as central notation



- UML serves as central notation for development of software
- UML is programming, test, and modeling language at the same time

UML-based Modeling

- UML + source code snippets allow for code & test generation



⇨ Code and test models mutually assert correctness.

SE SOFTWARE ENGINEERING

RWTH AACHEN UNIVERSITY

Farbe!

Generating Systems from UML

2. MontiCore

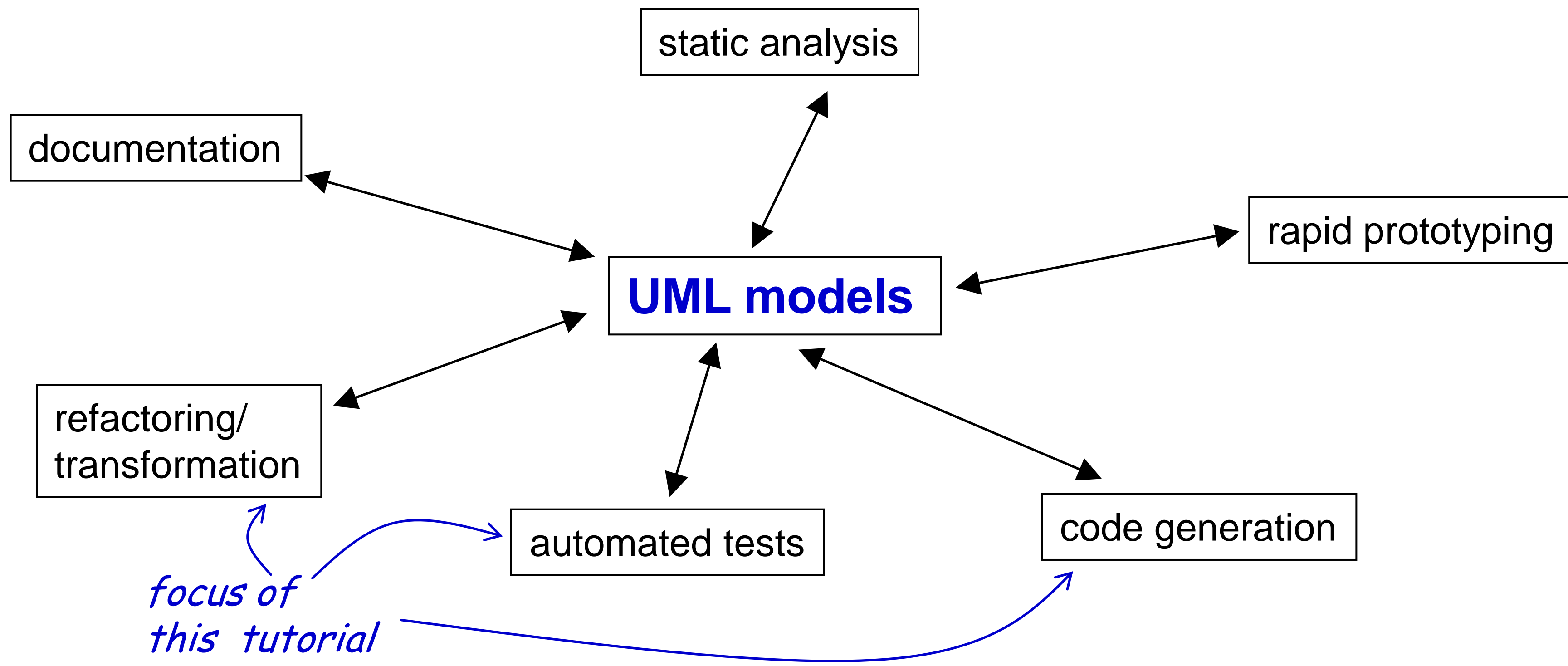
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Rep.

Model Based Development with the UML

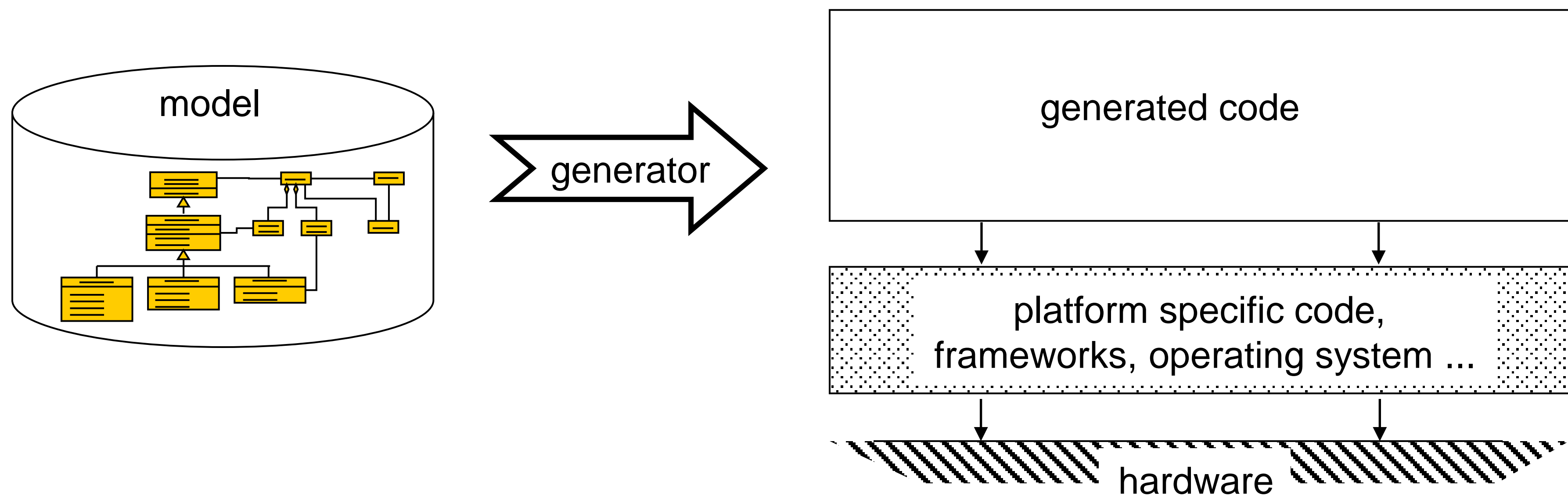
- Models as central notation



- UML serves as central notation for development of software
- UML is programming, test, and modeling language at the same time

Principle of Code Generation

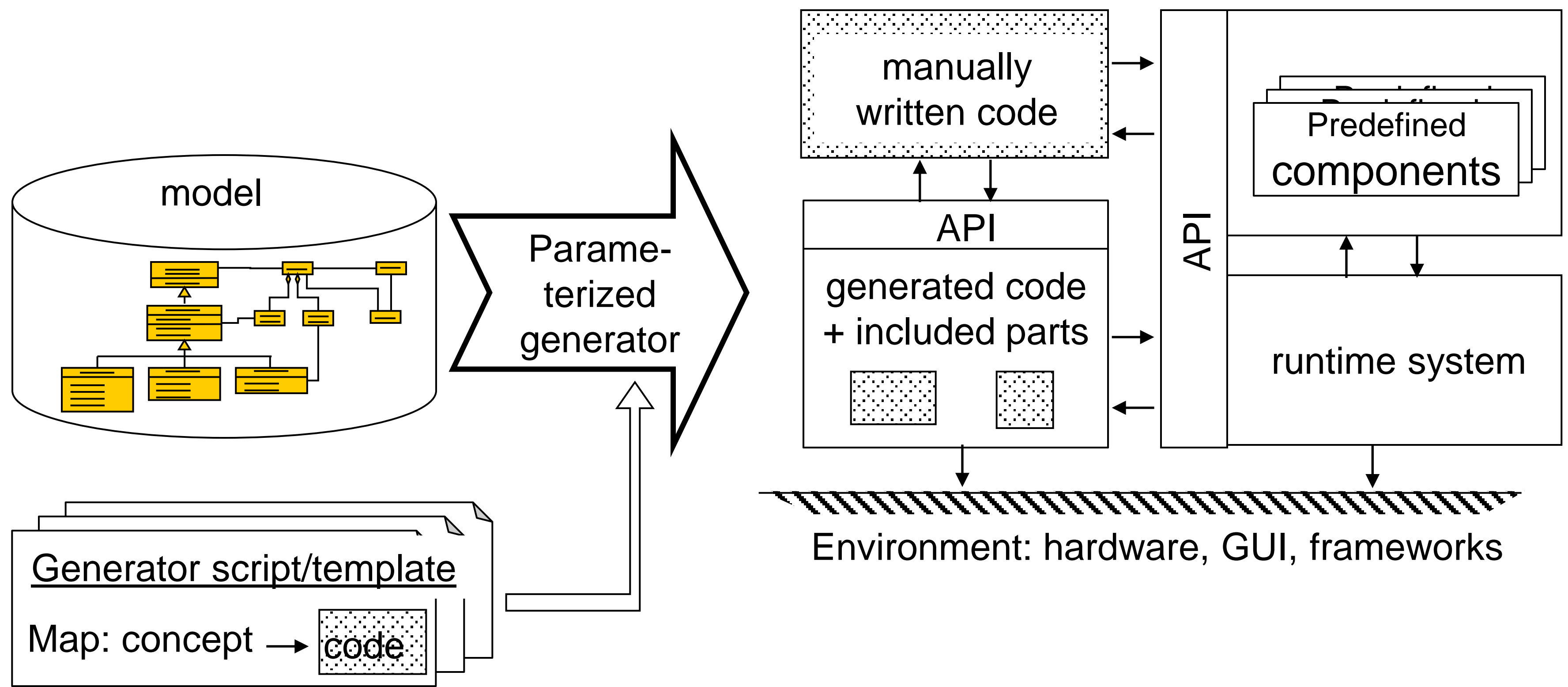
- Basically: mapping the model into code



⇨ Generator maps models to code on top of predefined frameworks.
Problem 1: not all needed code can/should be generated.
Problem 2: generated code is platform specific

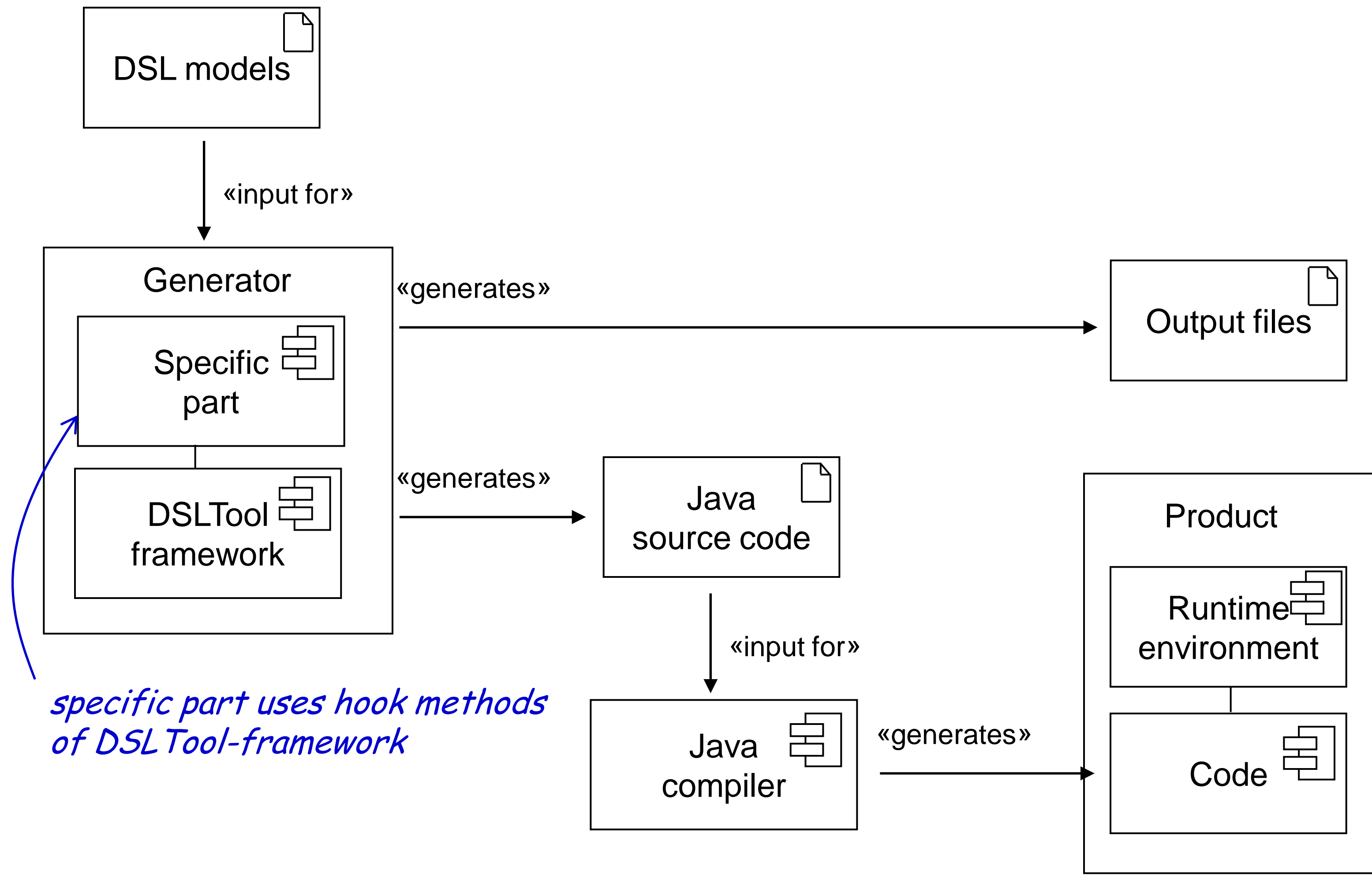
Code Generator: Parameterization + Runtime System

This is the principal structure of the generated code allowing parameterization, manual coding, and using a runtime system



DSLTool Framework

- DSLTool framework structures the code generation process



MontiCore - Overview

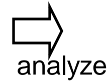
- Compact grammar-based definition of DSLs

- Framework-based solution
- Easy integration of additional components
- Assists generative development

www.monticore.de

- Assists:
  - parsing, AST, transformation
  - symbol-tables (name + type analysis)
  - pretty-printing, editor-generation
  - …
- Independency
  - available as command line tool, Eclipse-plugin, or online-service

Running Example: Finite Automata



Automaton

```
automaton PingPong {
  state NoGame <<initial>>;
  state Ping;
  state Pong;

  NoGame - startGame > Ping;

  Ping - stopGame > NoGame;
  Pong - stopGame > NoGame;

  Ping - returnBall > Pong;
  Pong - returnBall > Ping;
}
```

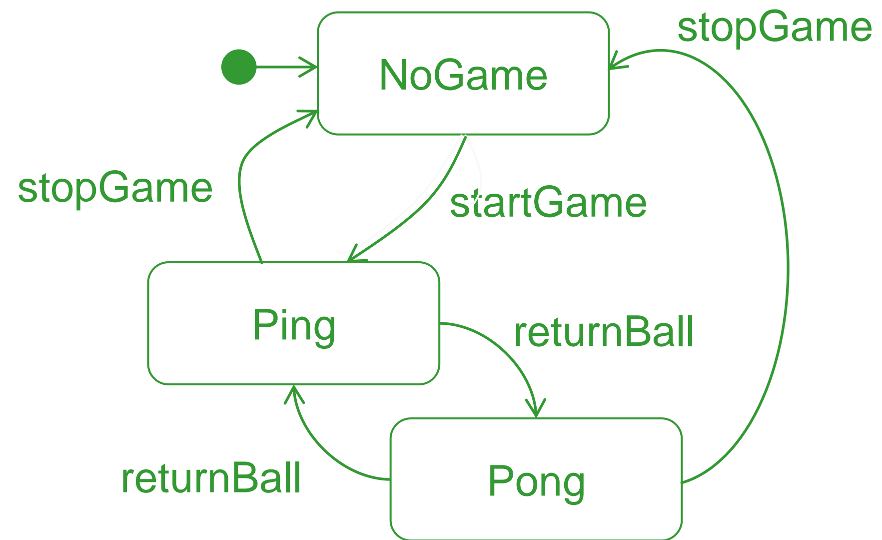
state

transition of form
source - input > target

Running Example: Finite Automata

```
automaton PingPong {
  state NoGame <<initial>>;
  state Ping;
  state Pong;

  NoGame - startGame > Ping;

  Ping - stopGame > NoGame;
  Pong - stopGame > NoGame;

  Ping - returnBall > Pong;
  Pong - returnBall > Ping;
}
```

Automaton



Example: Finite Automata Definition as MontiCore Grammar (1/2)

MG

```
grammar Automaton {

  Automaton =                     Name
          ( State | Transition )*

  State =           Name
   (       "initial"        |       "final"       )*


  Transition =
   From              Input             To
```

- Abstract syntax
  - nonterminals (e.g. Transition)
  - semantically relevant constants (e.g. “initial“)
  - identifiers (e.g. Name, From)

Example: Finite Automata Definition as MontiCore Grammar (2/2)

MG

```
grammar Automaton {

  Automaton =                     Name
        ( State | Transition )*

  State =             Name
   (      ["initial"]         |        ["final"]        )*


  Transition =
   from:Name       input:Name       to:Name
```

- Abstract syntax extended with type information:
  - constants (e.g. “initial“) are ensured to have an type
  - identifiers (e.g. Name, from) get a type (e.g. Name)

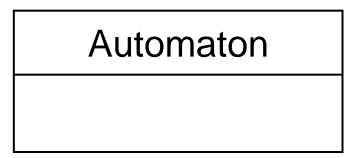
© Lehrstuhl für Software Engineering, RWTH Aachen

Automatic Derivation of Abstract Syntax (1/3)

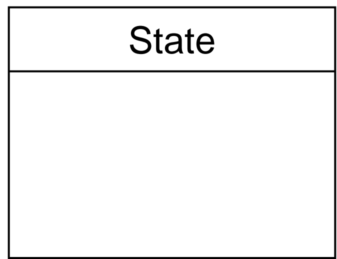
MG

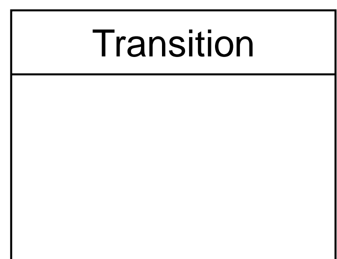
```
grammar Automaton {

  Automaton =              Name
       ( State | Transition )*

  State =          Name
   (     ["initial"]        |       ["final"]       )*


  Transition =
   from:Name      input:Name      to:Name
```

Meta-CD



a class for each nonterminal

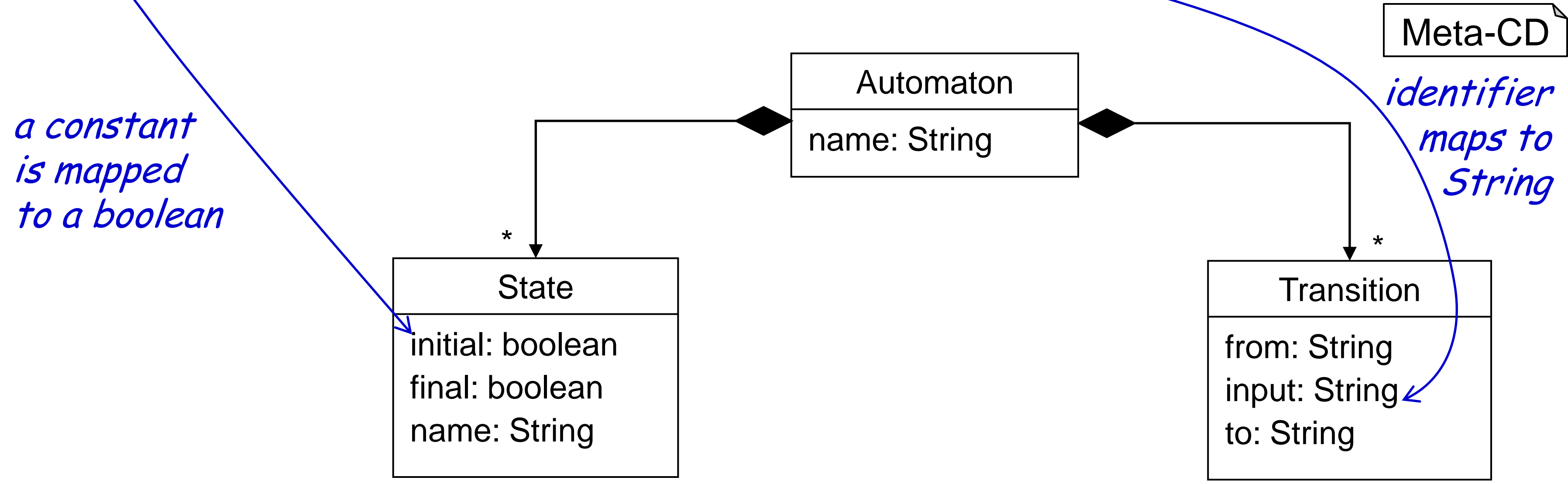




Automatic Derivation of Abstract Syntax (2/3)

MG

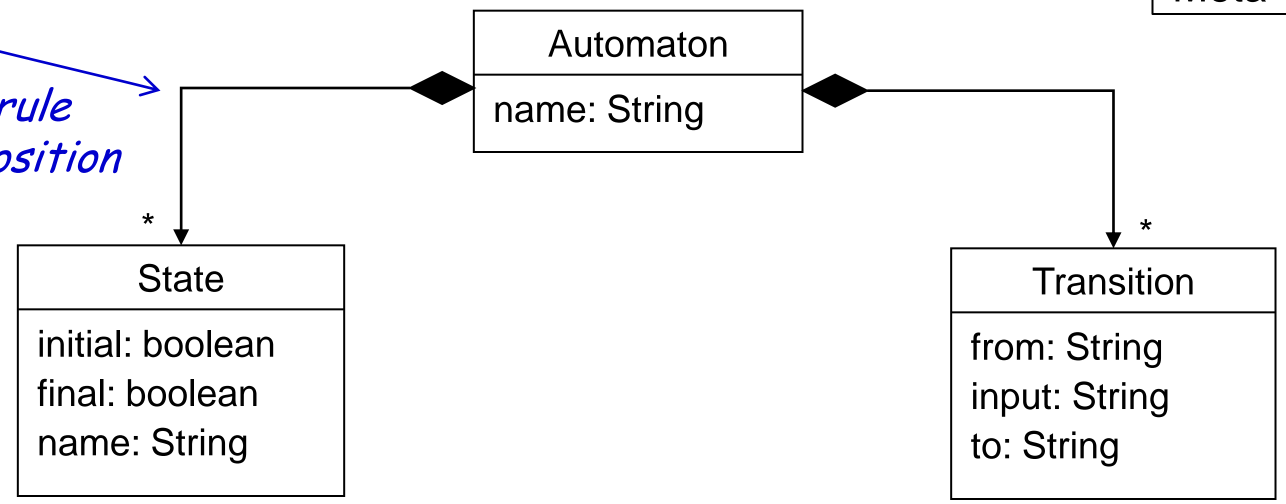
```
grammar Automaton {

  Automaton =                 Name
       ( State | Transition )*

  State =          Name
   (      ["initial"]        |        ["final"]       )*

  Transition =
   from:Name      input:Name      to:Name
```



Automatic Derivation of Abstract Syntax (3/3)

MG

```
grammar Automaton {

  Automaton =                  Name
        ( State | Transition )*

  State =          Name
   (      ["initial"]        |        ["final"]        )*


  Transition =
   from:Name       input:Name       to:Name
```



nonterminal in the rule is handled as composition

Adding Concrete Syntax (1/2)

MG

```
grammar Automaton {

  Automaton = "automaton" Name
   "{" ( State | Transition )* "}" ;

  State = "state" Name
   ("<<" ["initial"] ">>" | "<<" ["final"] ">>" )*
   ";" ;

  Transition =
   from:Name "-" input:Name ">" to:Name ";" ; }
```

- Concrete syntax:
  - syntactic sugar through
    - unnamed constants (e.g. “`automaton`“)
    - separators (e.g. “`;`“ , “`<<`“)

- Result: Definition of abstract and concrete syntax for the DSL

SE SOFTWARE ENGINEERING

RWTH AACHEN UNIVERSITY

Generating Systems from UML

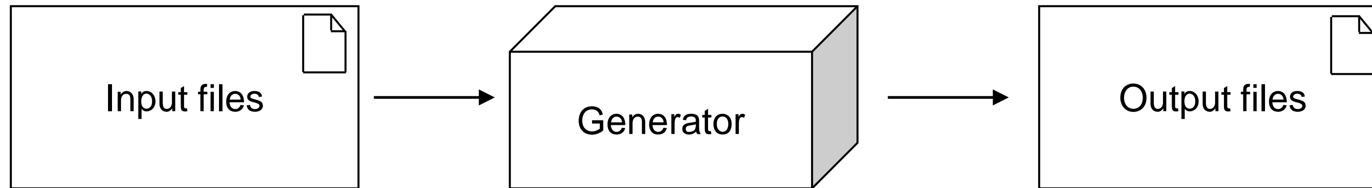
3. Generators

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

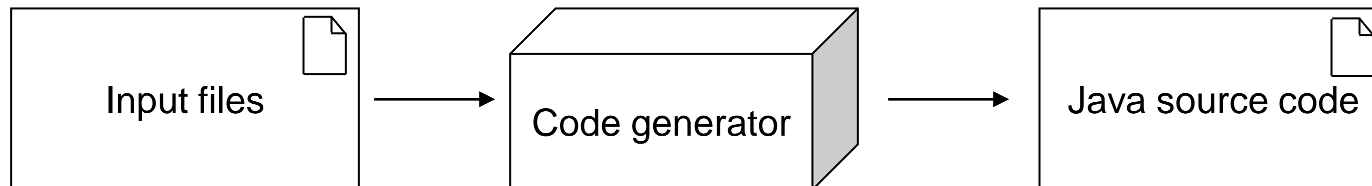
http://www.se-rwth.de/

Definition: Generator and Code Generator

- A generator transforms a set of input files to a set of output files



- A code generator transforms a set of input files to a set of source files in a programming language



- Transformations are automatic

Essential Terms

- Generic
  - *"relating to or characteristic of a whole group or class"* [MWO09]
  - solution space technique for developing components reusable in varying contexts
- Generative
  - *"having the power or function of generating, originating, producing, or reproducing"* [MWO09]
  - system for producing other systems; it comprises problem space, configuration knowledge, and solution space

[MWO09] Merriam-Webster Online Dictionary. 2009.

Steps in Developing a Generator

1. Select a modeling language (e.g. Class Diagrams)
   - … or design your own DSL
2. Understand the target language (e.g. Java)
   - and the frameworks to rely upon (EJB's, …)
3. Hand-code a concrete model to understand the mapping
   - Identify where each concept maps to
   - What can/should be optimized?
   - Is the result stable?
   - Does the generated code scale up to large projects?
4. How to integrate hand written code?
   - Please do not allow modifications of generated code
   - Define and publish explicit APIs instead

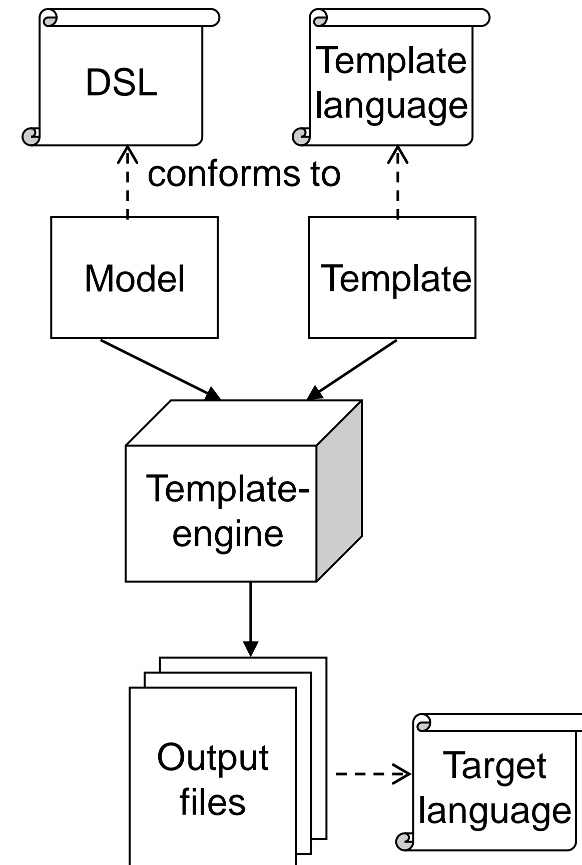
Types of Generators

- Template-based generation
  - template refers to elements of the abstract syntax
  - simple pieces of code embedded in templates
  - target language with embedded holes
    - holes are pieces of tool code to be executed and inserted when generating
  - control structures in the template language allow e.g. to handle lists or call sub-templates
  - readable; relatively easy to change
  - beneficial if generated code structure is similar to model

- API-based
  - API allows to build AST of target language
  - last step: pretty print the AST

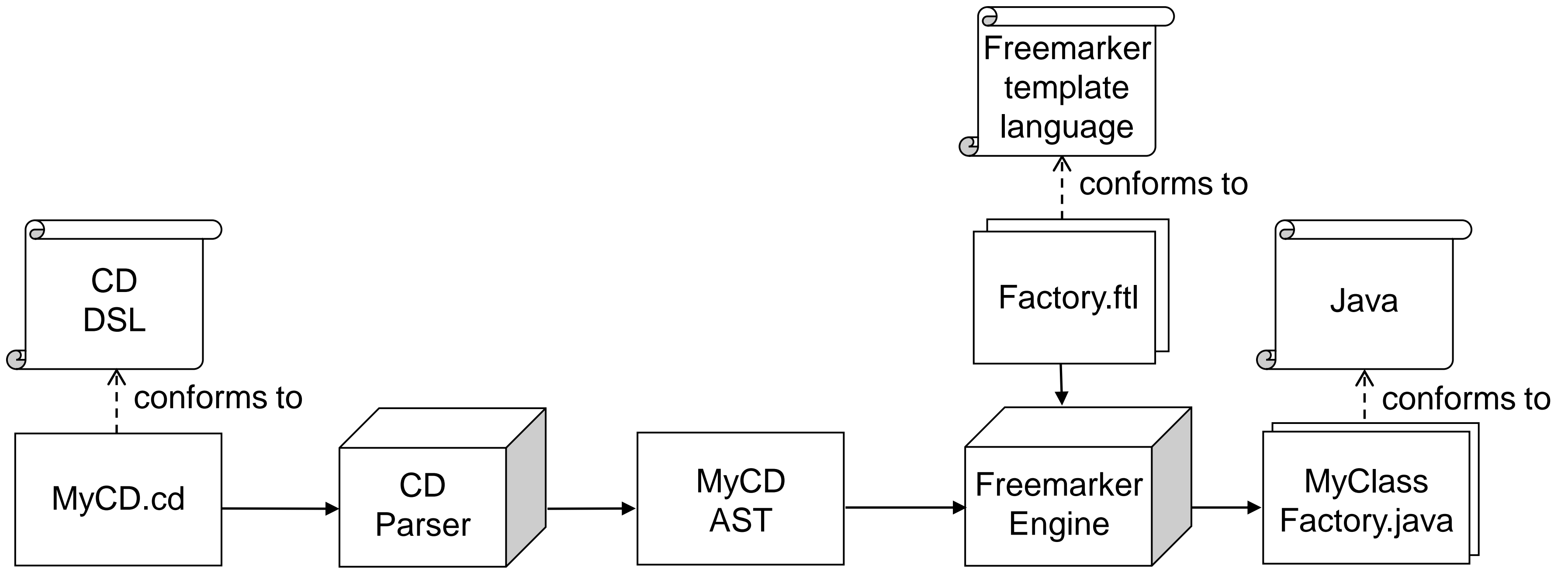
Template-based Model-to-Text-Transformations

- Template engines
  - transform a DSL model into output files using templates
- Properties of template-based Model-to-Text-transformation:
  + relatively easy to adapt
  - complex code generations pollute templates with GPL code
- Tool support: various engines exist:
  - Velocity/Freemarker
  - MOFScript (JET)
  - ....



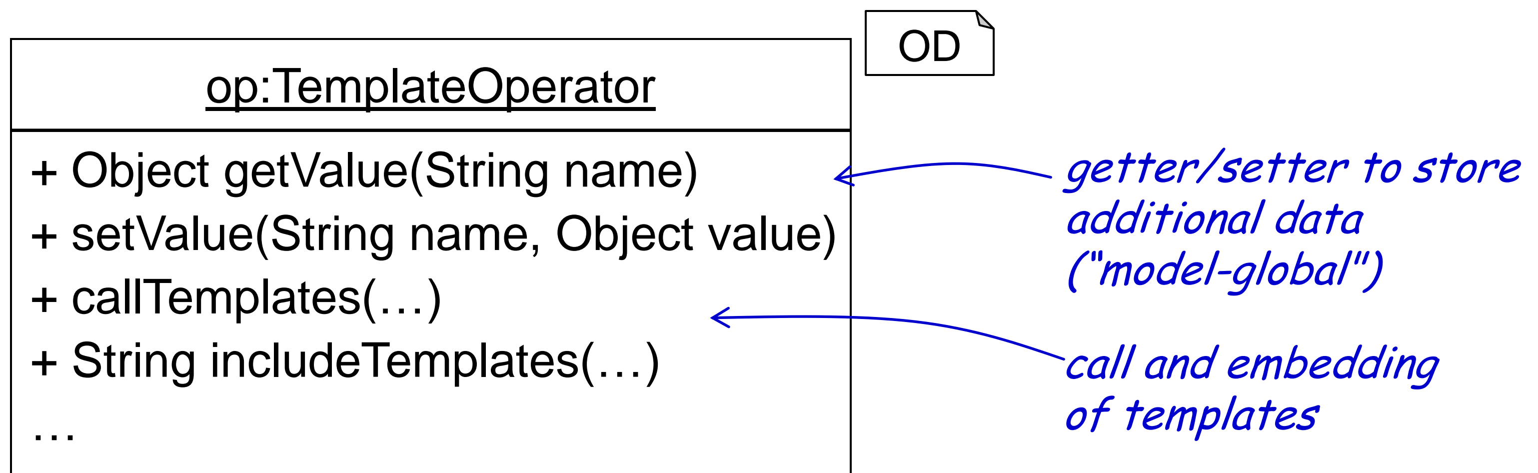
Example: Generation of Java Code from UML/P Classdiagrams with Freemarker

- Automatic transformation of specification into executable Java code in two steps
  1. create representation of specification in main memory (AST)
  2. generate code in target language (Java) from AST



Two Objects providing Information

- MontiCore provides two objects with various methods:
- ast: current AST-node (varies)
  - to navigate through the AST (e.g. to print children etc.).
- op: predefined Operations from the Template-Operator
  - provides a multitude of additional methods for handling templates



Sample Class Diagram and Generated Code

Model/DSL

```
package a.b;

classdiagram MyCD {

  class Person;

}
```

class name is mapped to factory name

Generated/Java

```
package a.b;


public class PersonFactory {

  protected static
    PersonFactory f = null;

  protected PersonFactory() {}

  public static Person create() {
    if (f == null) {
      f = new PersonFactory();
    }
    return f.doCreate();
  }

  protected Person doCreate() {
    return new Person();
  }

}
```

AST-Access using $ast

FM Factory.ftl

```
package ${ast.printPackage()};

<#assign cname = ${ast.printName()}>
public class ${cname}Factory {

  protected static
    ${cname}Factory f = null;

  protected ${cname}Factory() {}

  public static ${cname} create() {
    if (f == null) {
    ...
```

Generated/Java

```
package a.b;


public class PersonFactory {

  protected static
    PersonFactory f = null;

  protected PersonFactory() {}

  public static Person create() {
    if (f == null) {
    ...
```

- `${ast.printPackage()}`:
  - use of print-methods defined by the language designer for easier templates
- `<#assign cname = ${ast.printName()}>`:
  - use of local variable to compactify code

Class-Template with Extension Points

FM Class.ftl

```
package ${ast.printPackage()};

<#list ast.printImportList() as i>
  import ${i};
</#list>

${ast.printModifier()} class ${ast.printName()} {

  ${op.includeTemplates(t_class, ast)}
  ${op.includeTemplates(t_attr, ast.getCDAttributes())}
  ${op.includeTemplates(t_meth, ast.getCDMethods())}

}
```

- `${op.includeTemplates(TemplateList, AST-Node)}`
  - calls Templates on a given node and includes the results here
  - Templates are defined using variables (t_class, t_attr, t_meth)

-> next slide

Definition of Extension Point Values

- Main template of a template-structure used to define variables:

FM ClassMain.ftl

```
${op.setValue("t_attr", ["umlp.templates.Attribute",
                         "umlp.templates.GetterSetter"])}
${op.setValue("t_meth",  "umlp.templates.Method")}

${op.callTemplates(
    "umlp.templates.Class",
    ast.printPackage() + "." + ast.printName(),
    ast)}
```

defining the extension-points

template call with target file name as 2nd argument

templates focusing on generation from one type of AST-node

- `${op.setValue(Varname, Value)}`
  - assigns a value to a variable
  - values are globally available (also in sub-templates)
  - Remark: <#assign … > defines a template-local variable only

SE SOFTWARE ENGINEERING

RWTH AACHEN UNIVERSITY

Generating Systems from UML

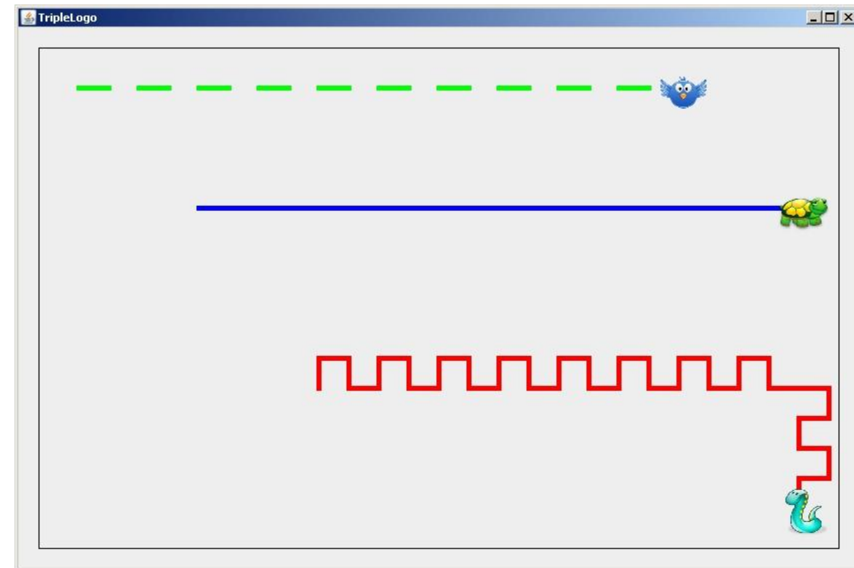
4. Examples

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

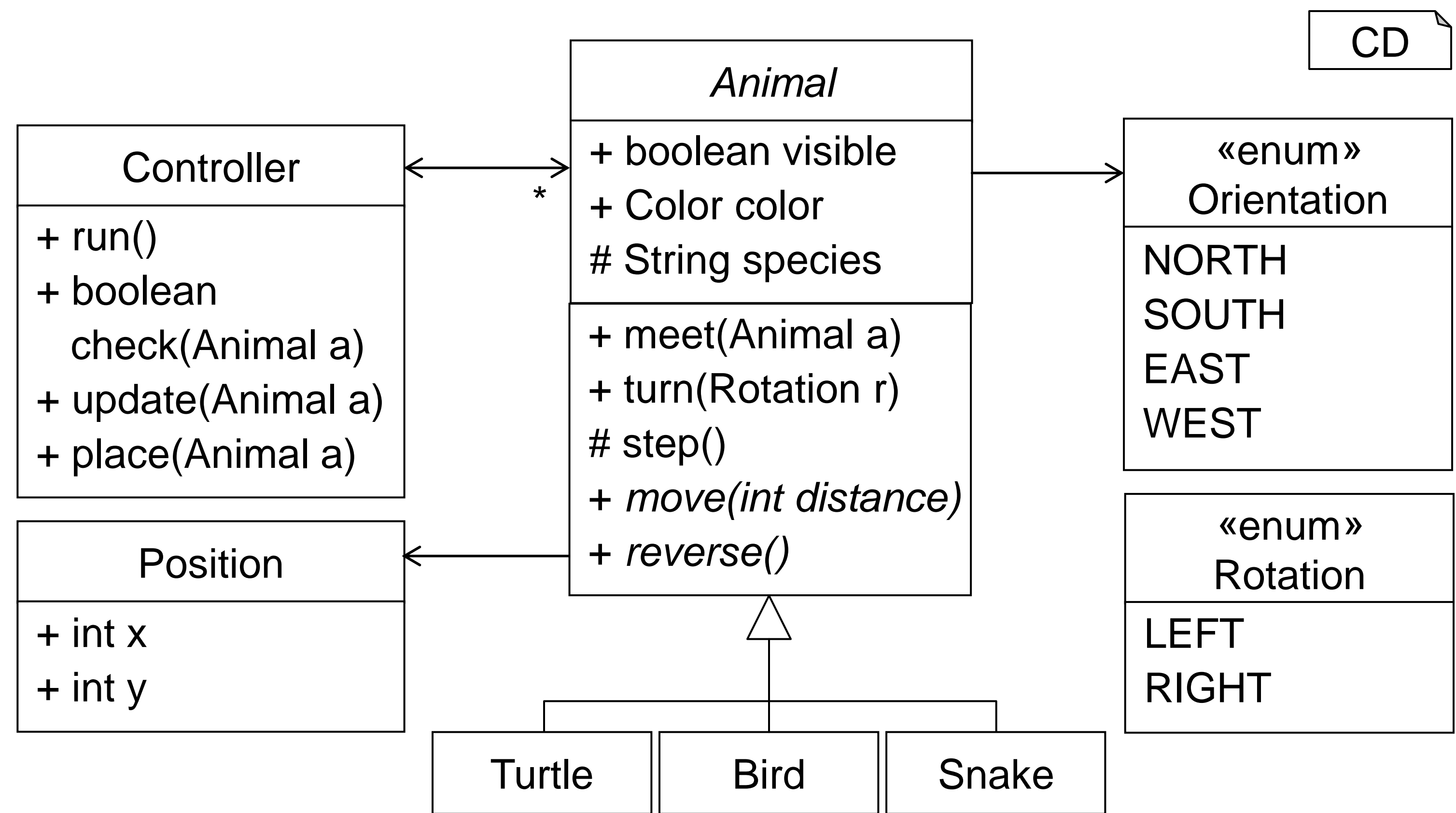
http://www.se-rwth.de/

Running Example: TripleLogo-Application

- Drawing application based on three animals
- Animals can receive commands for turning and moving
- Each animal has specific properties and behaviors, e.g., line color, movement type, or reaction when reaching the border of the drawing area

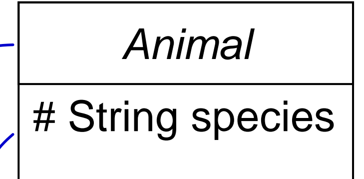


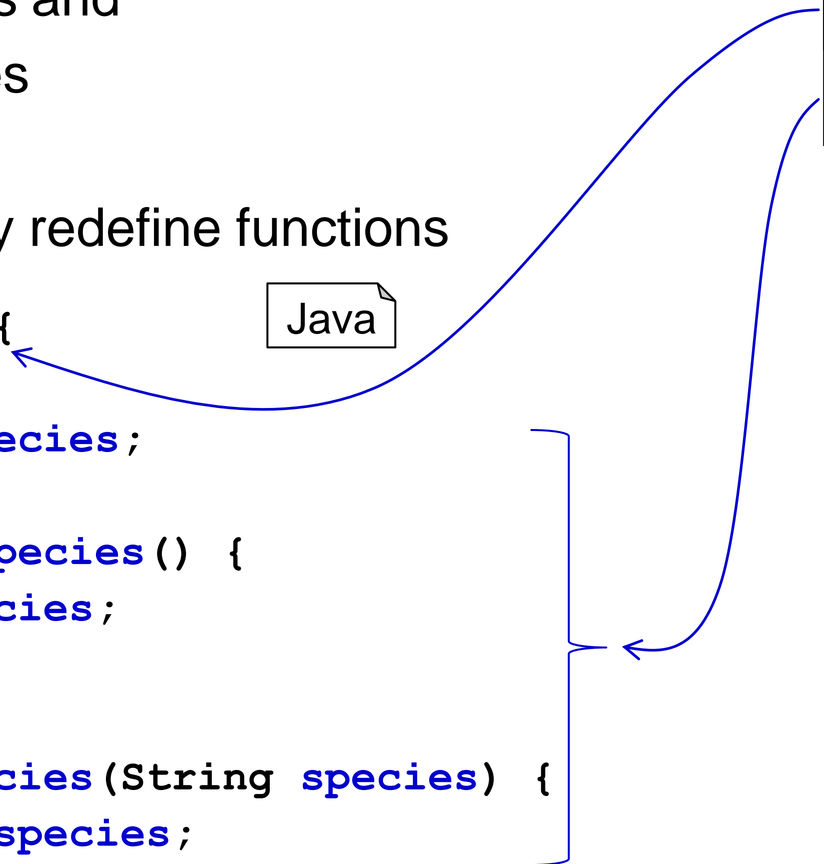
TripleLogo Architecture



Codegeneration: Classes and Attributes

- Here a version with
  - get/set-methods and
  - hidden attributes
- Advantage:
  - subclasses may redefine functions





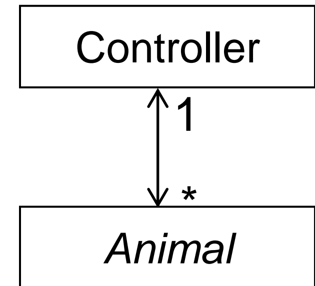

```
public class Animal {

  private String _species;

  public String getSpecies() {
    return this._species;
  }

  public void setSpecies(String species) {
    this._species = species;
  }
}
```

Signature of a *-Association (adapted from Set)

Association stored as set of links

Java

```
private Set _animals = new HashSet();

boolean      addAnimal(Object Animal o)
void         clearAnimals()
boolean      containsAnimal(Object Animal o)
boolean      isEmptyAnimals()
Iterator     iteratorAnimals()
boolean      removeAnimal(Object Animal o)
int          sizeAnimals()
// ... more
```



Association name extends signature (necessary when several associations occur)
Observe the added "s" and caption adaptations!

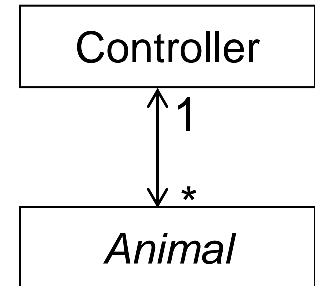
- Type-specific adaptations of Set-functionality in class Controller

*-Associations with Redundant Storage

Java

```
// from class Controller
  boolean addAnimal(Animal o) {
    o.xGENXsetController(this);
    return _animals.add(o);
  }
  boolean xGENXaddAnimal(Animal o) {
    return _animals.add(o);
  }
```

Internal helper



- Consistency ensured through
  - link management within the functions
- Generated helpers like "**xGENXaddAnimal**"
  - need to be public, but not be used by developers
  - Solution: "**xGENX**" is not communicated to developers and may change
- Further topics: Thread safety

Associations Summary

- Associations map systematically to an API of the attached classes
  - API provides Set-functionality
- API uses Association or Role-name to name its functions
- Navigation in both directions →
  - Redundancy and thus extra effort to maintain consistency
- Many additional concepts extend the API
  - Qualified access through key
  - Ordering with array-like or List-like access

SOFTWARE ENGINEERING

RWTH AACHEN UNIVERSITY

Generating Systems from UML

5. Statecharts

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Statecharts

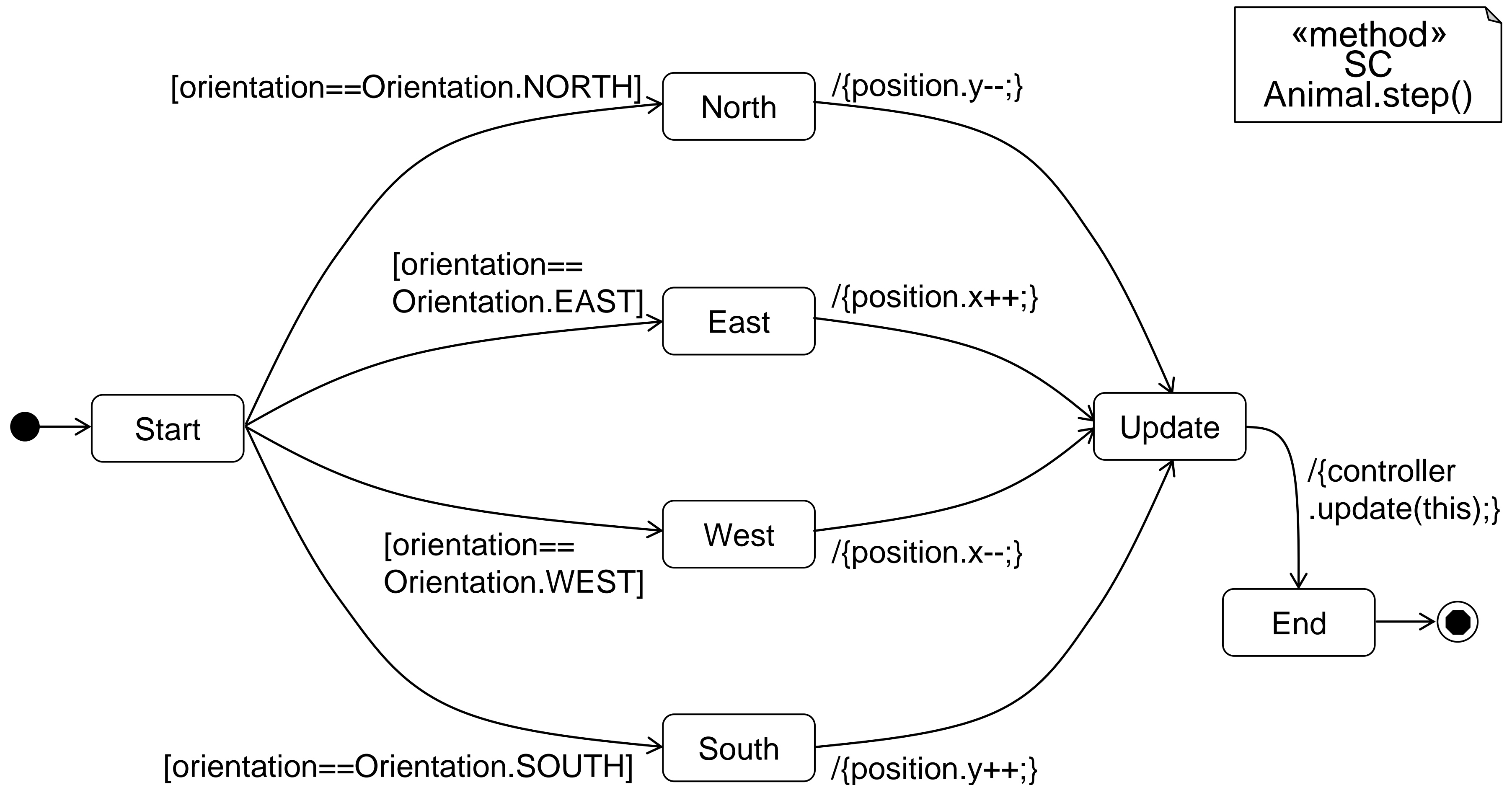
- Goal is the description of object behavior
- Assumptions:
  - objects have a data part and a control part in their state
  - control part is described by finite state space
  - object changes are defined by transitions

- Statecharts extend automata theory:
  - hierarchical states,
  - actions in transitions and states, ...

- History of Statecharts:
  - Statecharts introduced by David Harel in 1987
  - incorporated in many modeling languages
  - many variants developed
  - part of the UML from the beginning

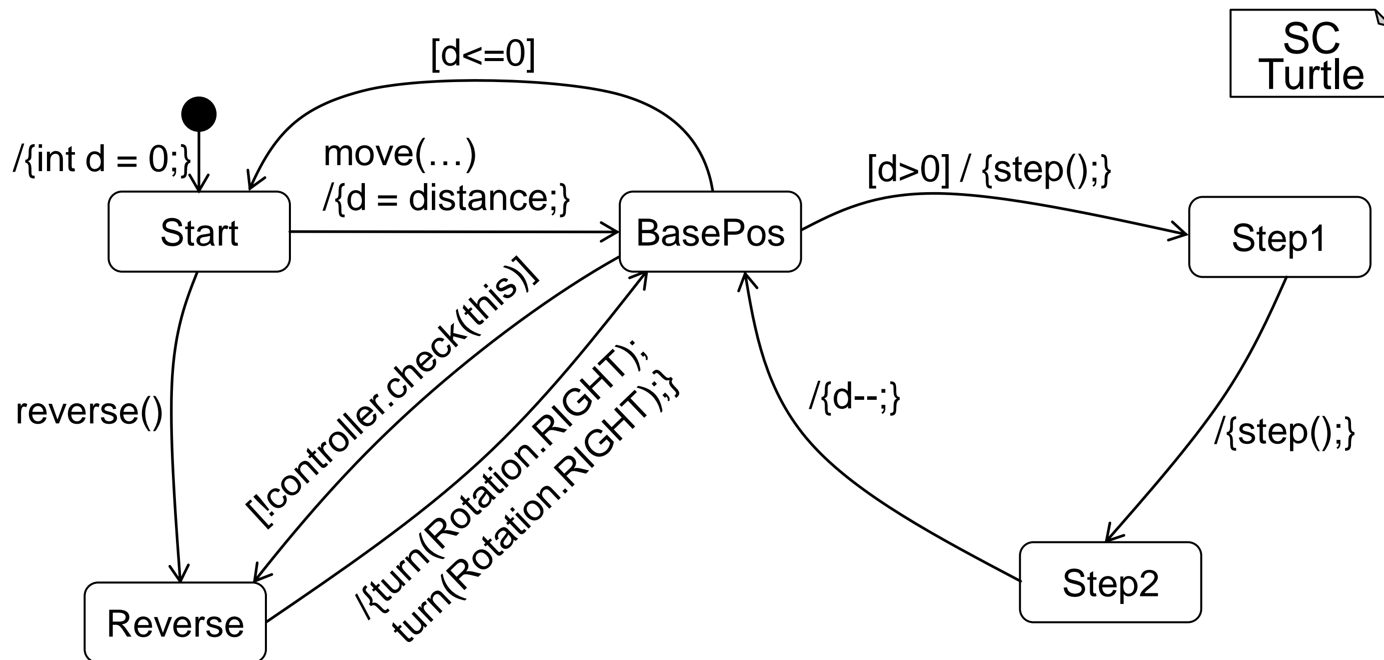
Method-Behavior: Animal.step()

- Simple Statechart for one step (same for each Animal):



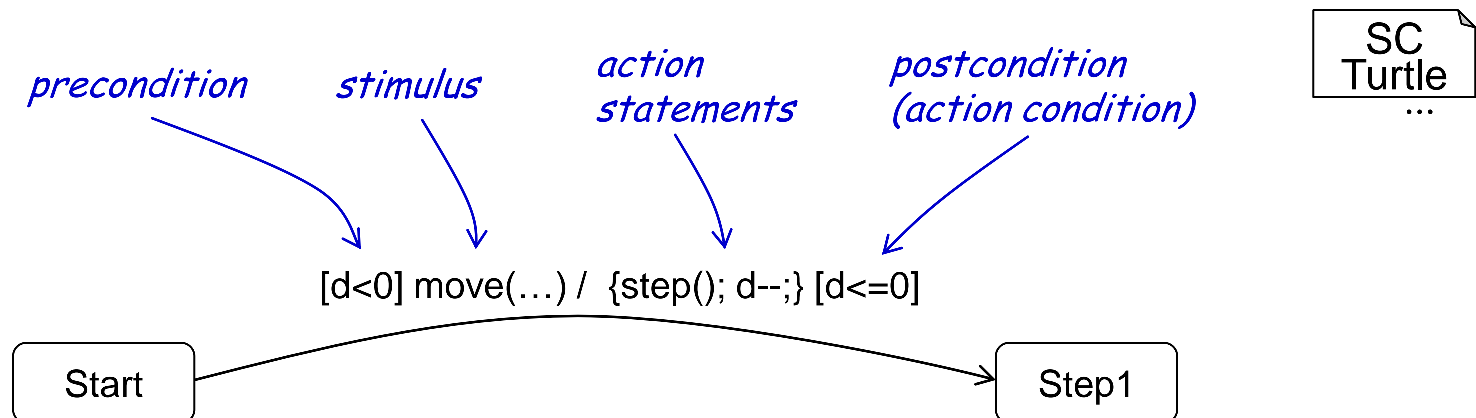
Object-Behavior: Turtle

- Turtle-specific behavior (as specialization of Animal):



Transitions

- A transition describes a portion of the object behavior
- A transition has
  - source state
  - precondition
  - stimulus
  - action
  - postcondition
  - target state



Code Generation

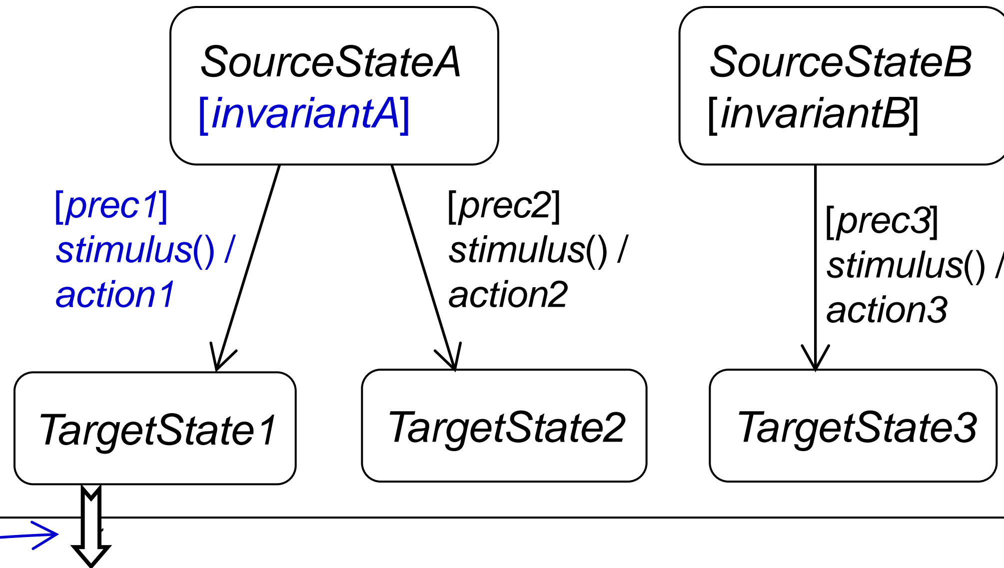
- Starting point:
  - simplified Statecharts (no hierarchy, state invariants not yet expanded)

- Several variants for representation of states:
  - explicit state attribute describes state, or
  - invariants of disjoint states as predicates, or
  - state pattern: each state is associated with an own object

- Remark: Method Statecharts are treated in a different way.

Disjoint Invariants for States

SC



Java

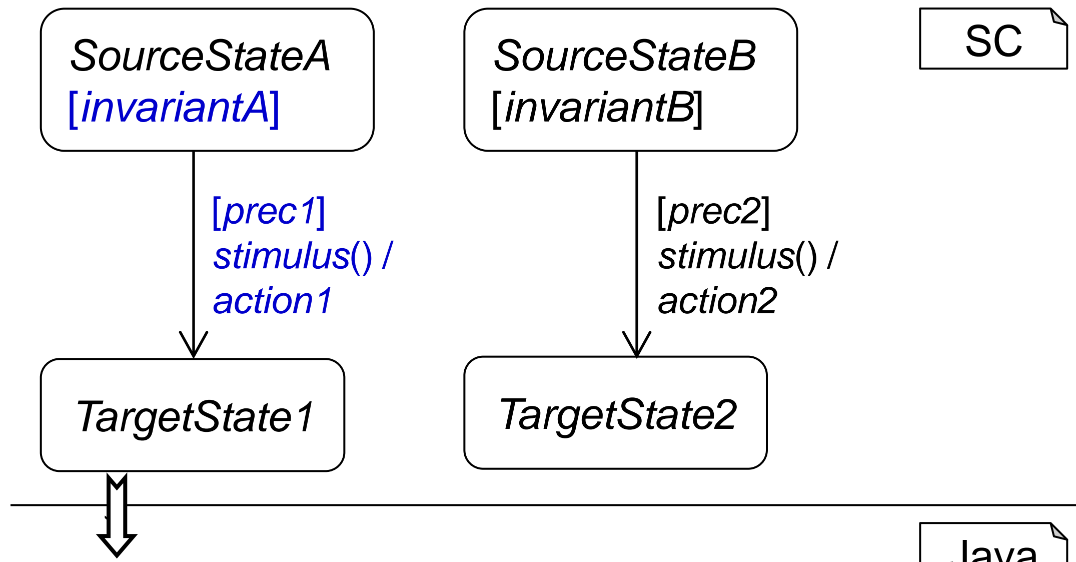
transformation rule: from top to bottom

```
public ... stimulus() {
  if(invariantA) {
    if(prec1) {
      action1;
    } else if (prec2) {
      action2;
    } else {
      // error handling
    }
  } else if(invariantB) {
    ...
}}
```

state invariant and preconditions are used to distinguish the transitions

disadvantage: code "invariantA" is repeatedly present in the code

Outsourcing State Invariants into own Predicates



Java

```
public boolean invSourceStateA() {
    return invariantA;
}
public boolean invSourceStateB() {
    return invariantB;
}
public ... stimulus() {
  if(invSourceStateA()) {
    ...
}
```

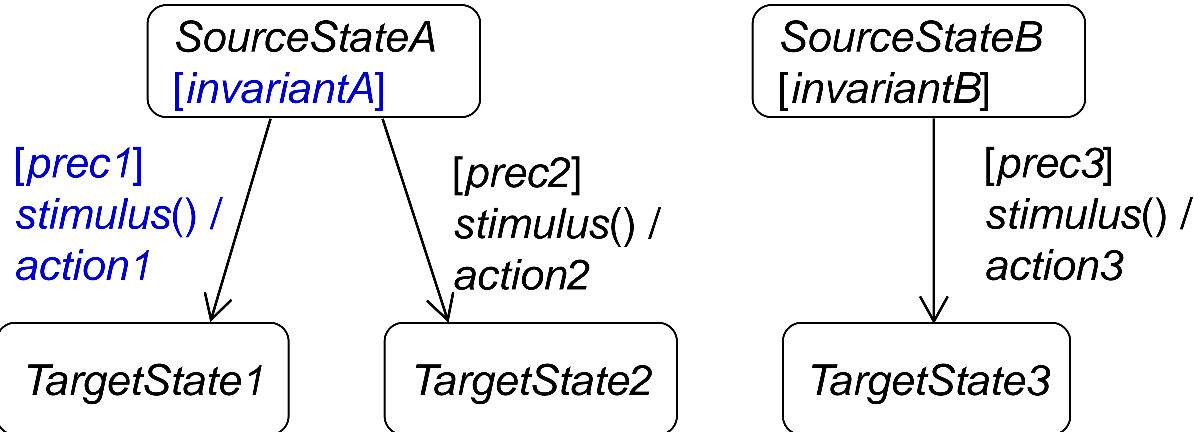
each state is mapped to a predicate that evaluates the state invariant

Advantage:
"invariantA" generated only once.

Disadvantage:
- "invariantA" can be complex and time-consuming to be calculated

Better:
- states attribute remembers current state

Introduction of a State Attribute




```
private int status;
final static int SOURCE_STATE_A = 1;
final static int SOURCE_STATE_B = 2;
final static int TARGET_STATE_1 = 3; ...
```

state diagram is stored as enumeration

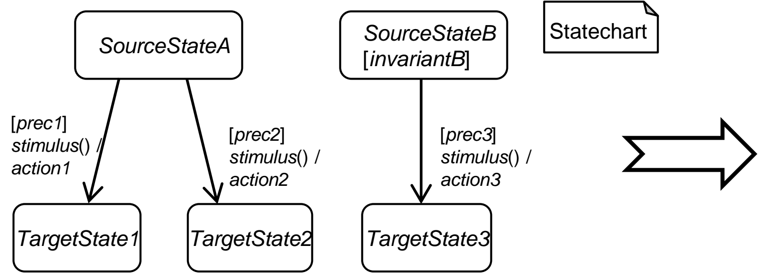
Advantage: efficient
Disadvantage: redundant storage,
Consistency not assured:
(status==SOURCE_STATE_A) implies invariantA

Java

```
public ... stimulus() {
  switch(status) {
    case SOURCE_STATE_A :
      if(prec1) {
        action1;
        status = TARGET_STATE_1;
      } else if (prec2) {
        action2;
        status = TARGET_STATE_2;
      } ...
    break;
    case SOURCE_STATE_B :
      ...
}}
```

Using Invariants for Tests

Statechart as on previous slide



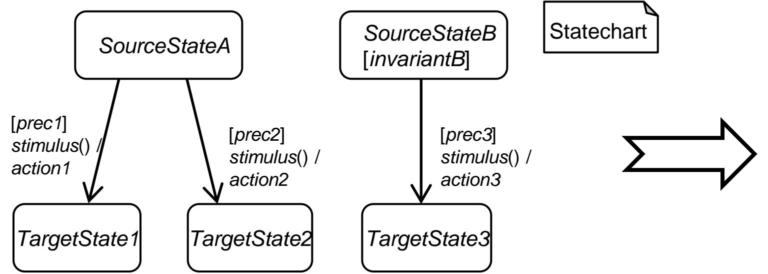
Java

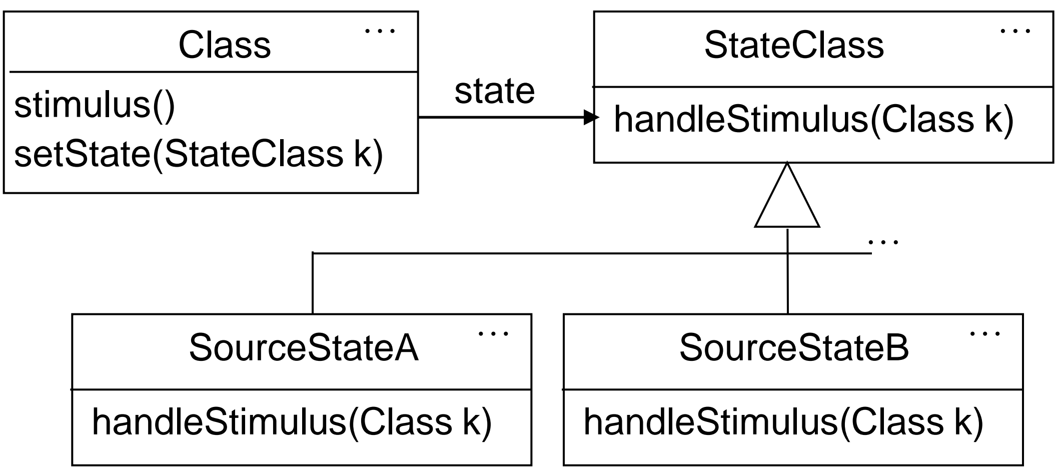
```
private int status;
final static int SOURCE_STATE_A = 1;
final static int SOURCE_STATE_B = 2;
final static int TARGET_STATE1 = 3;
...

public ... stimulus() {
  switch(status) {
    case SOURCE_STATE_A:
      ocl invariantA;
      if(prec1) {
        action1;
        status = TARGET_STATE_1;
      } else {
        ocl prec2;
        action2;
        status = TARGET_STATE_2;
      } ...
    break;
    case SOURCE_STATE_B:
      ...
}}
```

state invariants and some preconditions can be used in OCL-instructions as assertions for testing purposes, if it is assumed that the diagram is complete

Design Patterns: State (Gamma and others1994)

Statechart as on previous slide





Java

```
class Class {
  SourceStateA sourceStateA = ...
  TargetState1 targetState1 = ...

  public ... stimulus() {
    state.handleStimulus(this);
  }
}
```

```
class SourceStateA {
  public .. handleStimulus(Class k)
{
  ocl invariantA‘;
  if(prec1) {
   action1‘;
   k.setState(k.targetState1);
  } else
   ...
}}
```

Advantage: the state design pattern can be used for additional flexibility
Disadvantage: overhead due to additional objects: one for each state.

Summary Statecharts

- Statecharts are an extension of the Mealy machines.
- Statecharts build a powerful form to define behavior based on states.
- The combination with pieces of code for actions, or with OCL for conditions makes Statecharts fully descriptive and comfortable.

- A number of variations for Statecharts allows different areas of application:
  - method descriptions
  - life cycles
  - test sequences
- in various phases of software development: analysis, design, implementation.

SE SOFTWARE ENGINEERING

RWTH AACHEN UNIVERSITY

Generating Systems from UML

6. Generating Test Cases

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

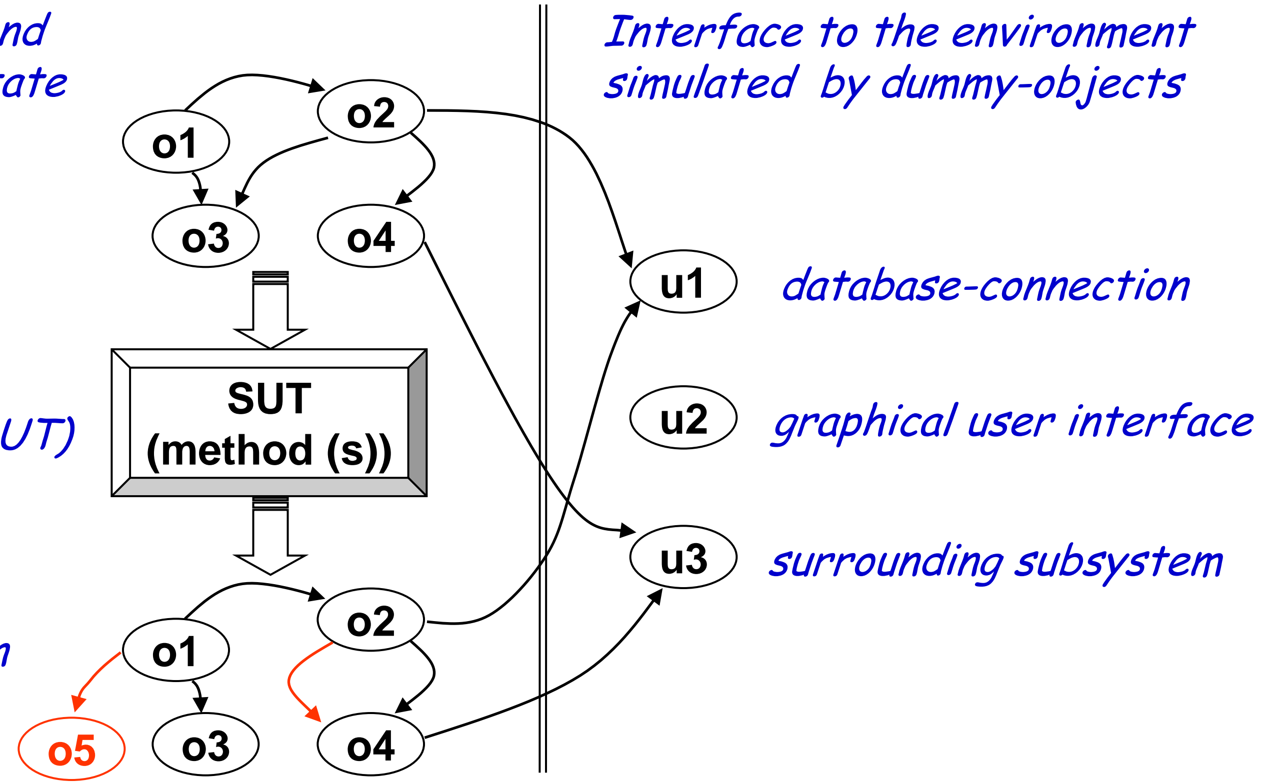
http://www.se-rwth.de/

Software Tests: Targets

- Focus on
  - automatic tests to ensure easy repetition
    - test set-up, execution, and evaluation of the test result
  - deterministic tests ensure unambiguous, reproducible result
  - no side effects
  - efficient execution to enable developers to often run tests
  - easy test suite management

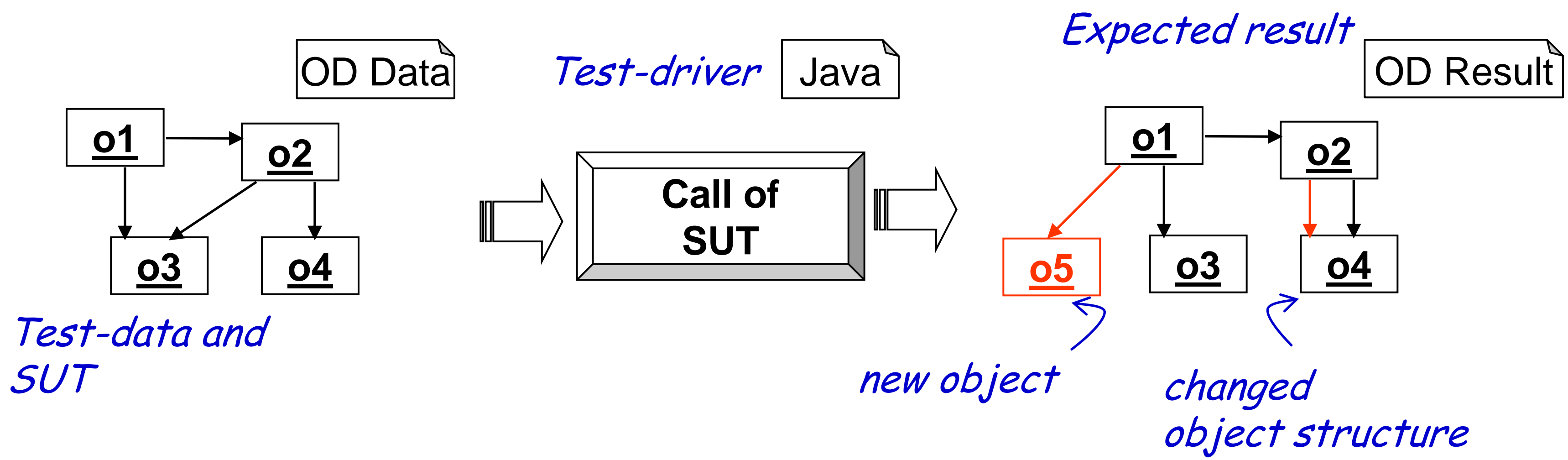
- Targets:
  - demonstration of quality
  - "definition" of requirements still to be implemented
  - confidence in correctness of the code
  - efficient further development of the system

Typical Test Structure



Modeling of a Test Case

- Using two object diagrams and Java
- Object diagrams model test data and expected result
- Java provides the test driver



Language for Test Case Structure

TC

- Represents test-structure

```
testcase MyTestCase {
  MyTest {
    setup MyInitialOD;
    {
      // Code
    }
    result MyFinalOD;
  }
}
```

Object diagrams preparing the test data

Java code as process description

Object diagrams checking the actual result

- Possible extensions:
  - additional Java code for adaptation of the test data
  - OCL for additional checks

Test Definition: Moving of a Turtle

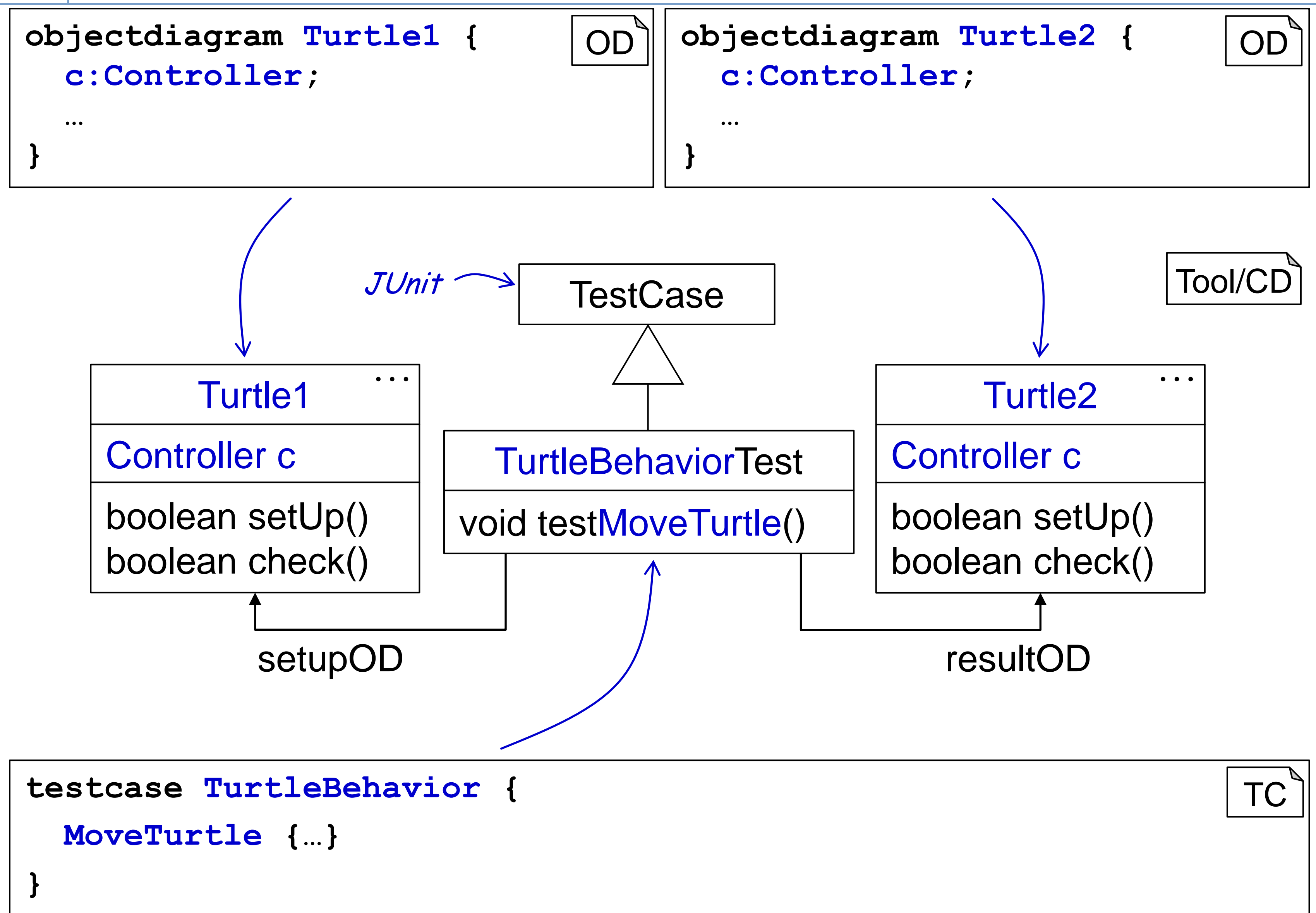
OD
```
objectdiagram Turtle1 {
  c:Controller;
  p:Position {x = 5; y = 7;}
  t:Turtle {
    orientation =
      Orientation.NORTH;
  }
  link c <-> t;
  link t -> p;
}
```

OD
```
objectdiagram Turtle2 {
  c:Controller;
  p:Position {x = 3; y = 7;}
  t:Turtle {
    orientation =
      Orientation.EAST;
  }
  link c <-> t;
  link t -> p;
}
```

TC

test case
```
testcase TurtleBehavior {
  MoveTurtle {
    setup Turtle1;
    {
      t.turn(Rotation.LEFT);
      t.move(1);
    }
    result Turtle2;
  }
}
```

Generated Classes from Test Definition



Principles of Generation

- Each OD is coded in its own class
  - OD-classes provide methods setUp() and check()
- Testcase becomes JUnit-testcase
- OD’s setUp() creates all objects and sets links and attributes
  - underlying assumptions:
    - standard get/set-methods exist
    - standard link handling is defined
    - standard factory methods exist

© Lehrstuhl für Software Engineering, RWTH Aachen

Codegen: Object Instantiation from OD

OD

```
objectdiagram Turtle1 {
  c:Controller;
  t:Turtle {
    orientation =
      Orientation.NORTH;
  }
  link c <-> t;
  …
}
```

Code generator

Generated/Java

```
public boolean setUp() {
  c = ControllerFactory.create();

  t = TurtleFactory.create();
  t.setOrientation(Orientation.NORTH);

  t.setController(c);

  …

  return true;
}
```

excerpt from
generated
class Turtle1

Codegen: Comparison of Objects

OD

```
objectdiagram Turtle2 {
  p:Position {x = 3; y = 7;}
  ...
}
```

excerpt from generated class Turtle2

Generated/Java

```
public boolean check() {
  boolean ret = true;
  ...
  if (p == null) {
    addMessage("p not present");
    ret = false;
  }
  if (p.getX() != 3) {
    addMessage("Attr x of p: "
       + "Expected 3 but was "
       + p.getX());
    ret = false;
  }
  ...
  return ret;
}
```

Code generator

Variable ret allows to check multiple conditions and raise several messages

Codegen: Comparison of Links

OD

```
objectdiagram Turtle2 {
  c:Controller;
  t:Turtle {…}
  link c <-> t;
  …
}
```

bidirectional check

Generated/Java

```
public boolean check() {
  …
  if (!c.containsAnimal(t)) {
    addMessage("c: Expected link to t");
    ret = false;
  }
  if (!t.getController != c) {
    addMessage("t: Expected link to c");
    ret = false;
  }
  …
}
```

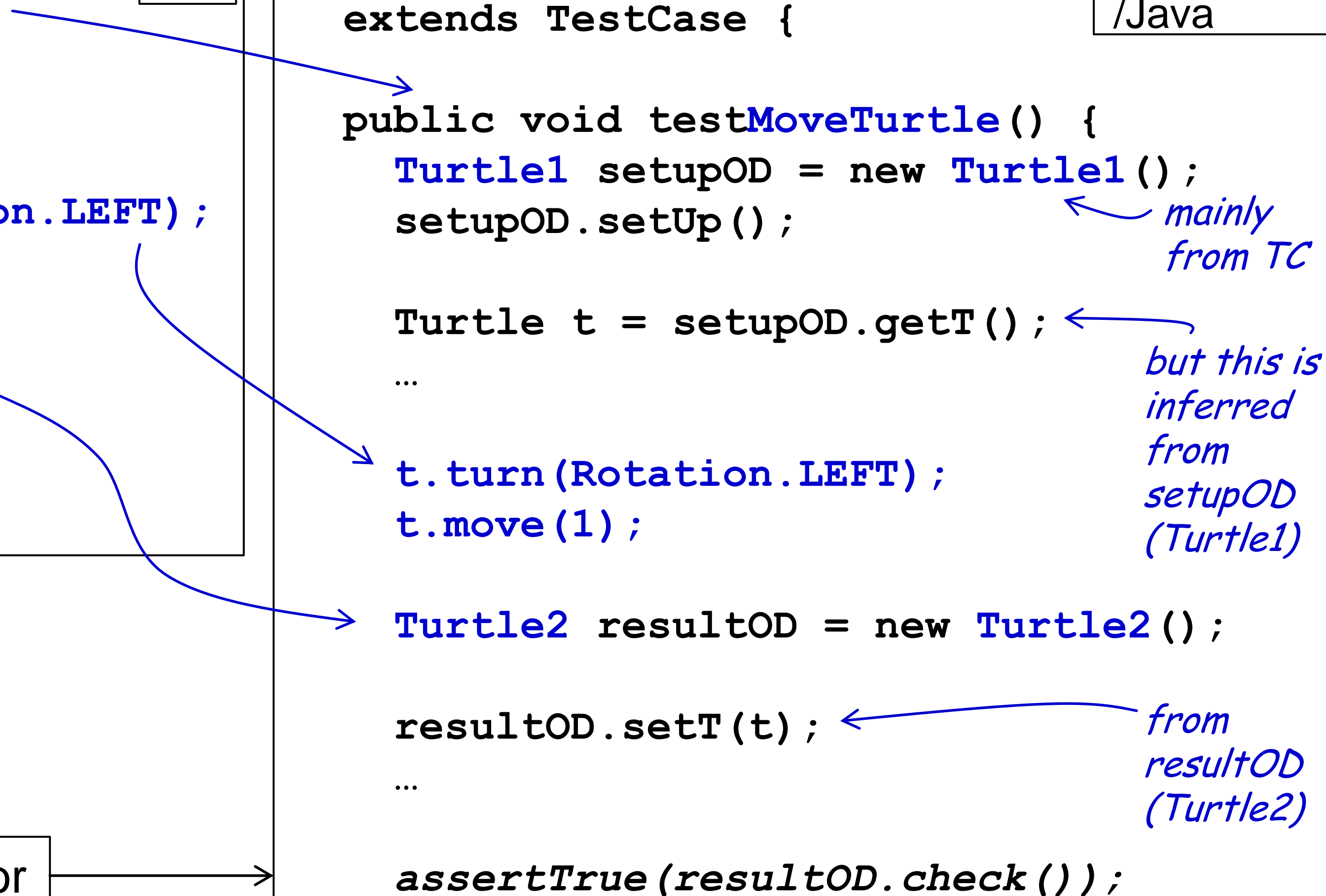
Code generator

excerpt from generated class Turtle2

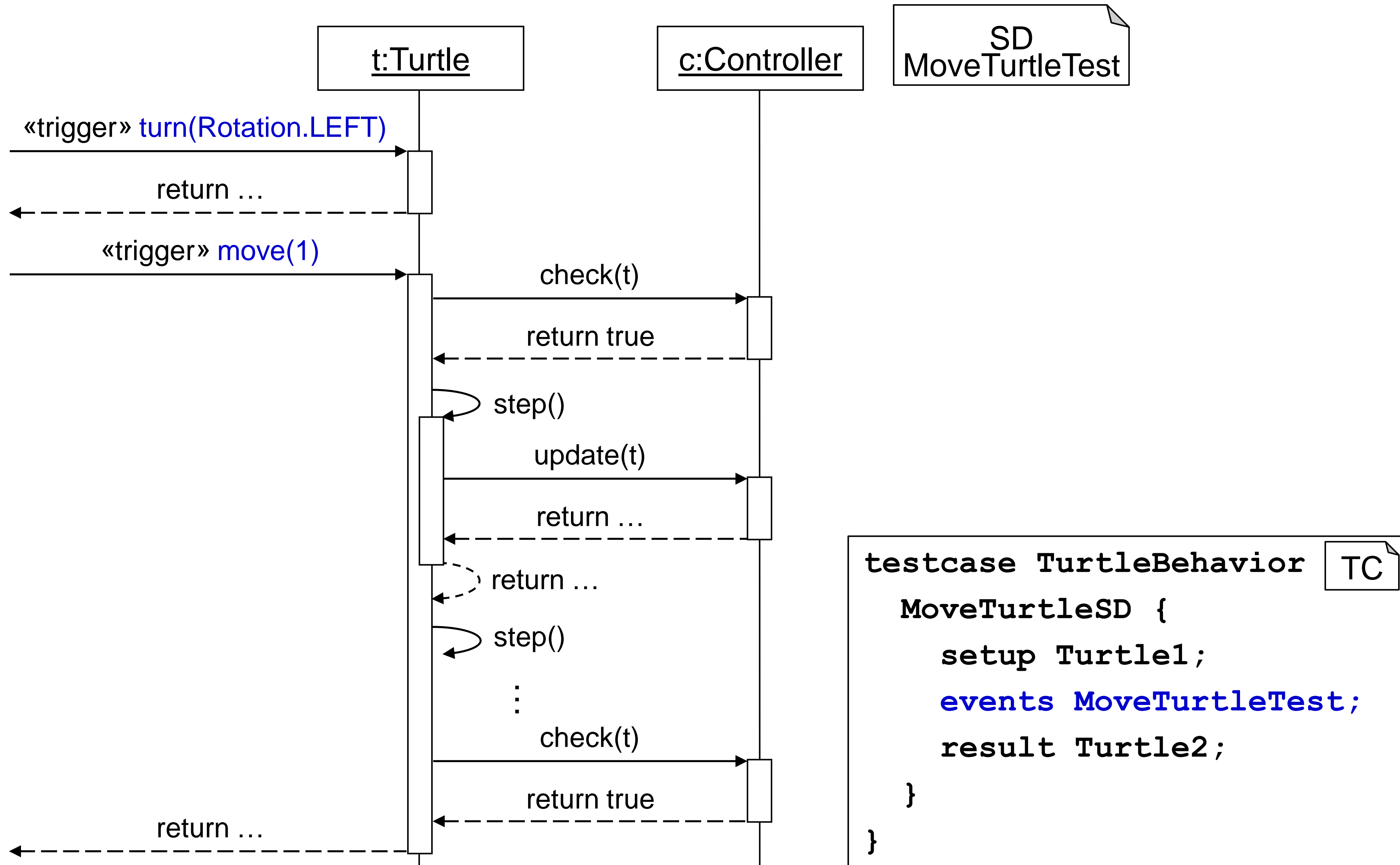
Codegen: Test Case

```
testcase TurtleBehavior {
  MoveTurtle {
    setup Turtle1;
    {
      t.turn(Rotation.LEFT);
      t.move(1);
    }
    result Turtle2;
  }
}
```

TC



Detailed Test with a Sequencediagram



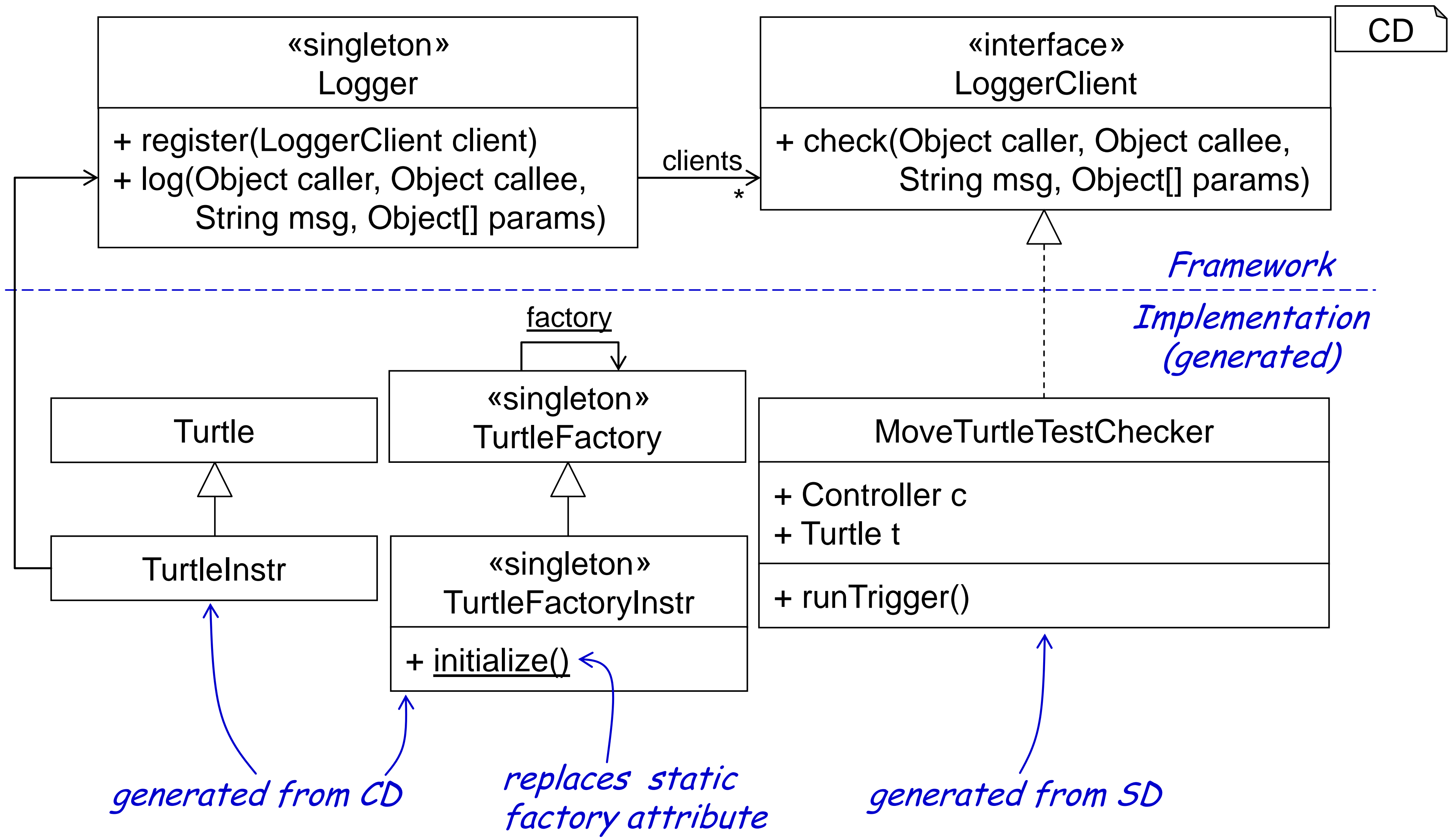
Code-Instrumentation

Generated/Java

```
public class TurtleInstr
extends Turtle {
  …
  public void move(int distance) {
    Object caller = Logger.getCurrentCaller();
    Object callee = this;
    Logger.log(caller, callee, "move", distance);
    Logger.addCurrentCaller(this);
    super.move(distance);
    Logger.log(caller, callee, "move", distance);
    Logger.removeCurrentCaller(this);
  }
  …
}
```

- Method- and attribute-calls are logged using code-instrumentation
- Subclass encapsulates instrumentation
- Factory enables to start instrumentation dynamically (next slide)

SD-Codegen and Code-Instrumentation



SOFTWARE ENGINEERING

RWTH AACHEN UNIVERSITY

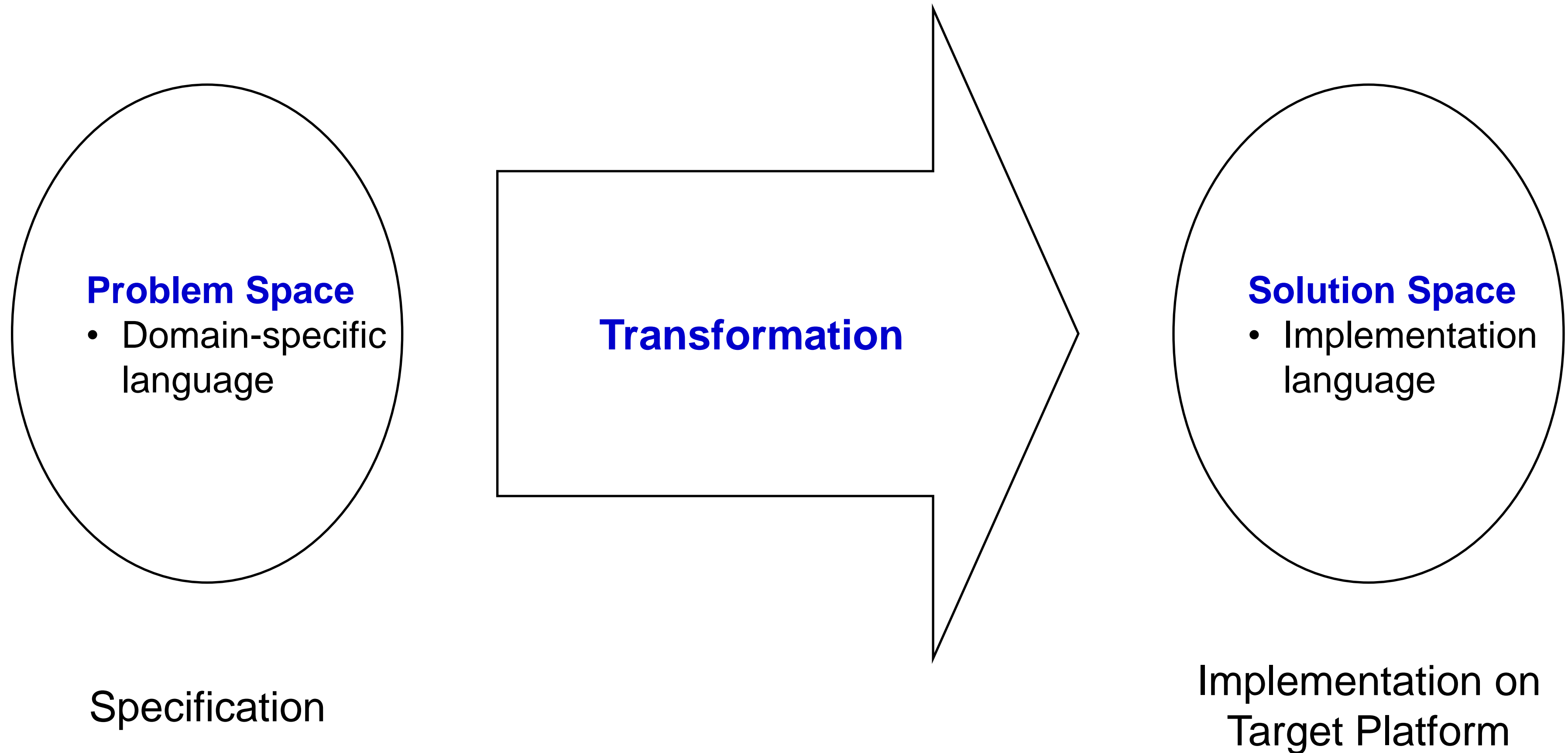
Generating Systems from UML

7. Customizing the Generator

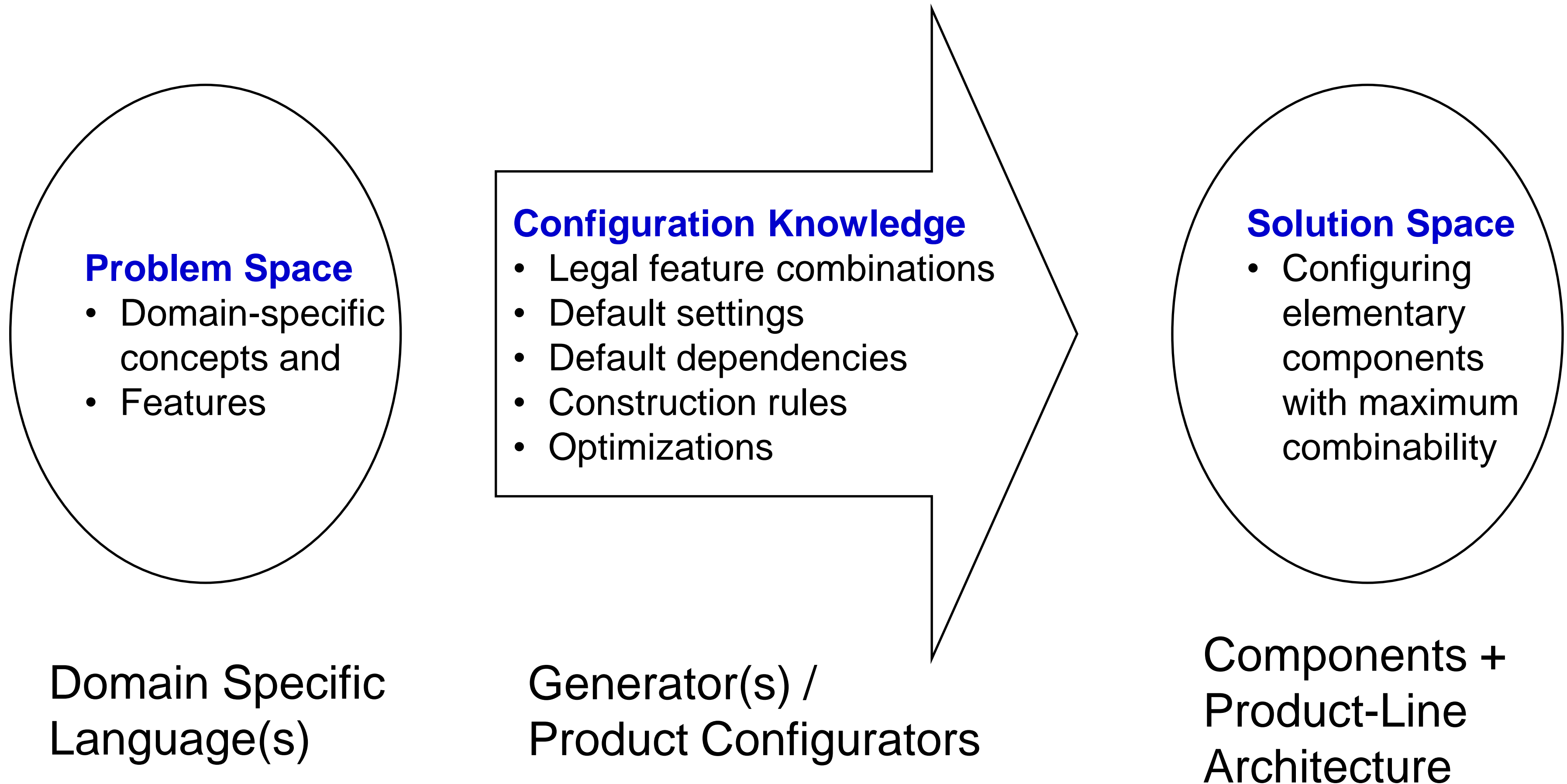
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

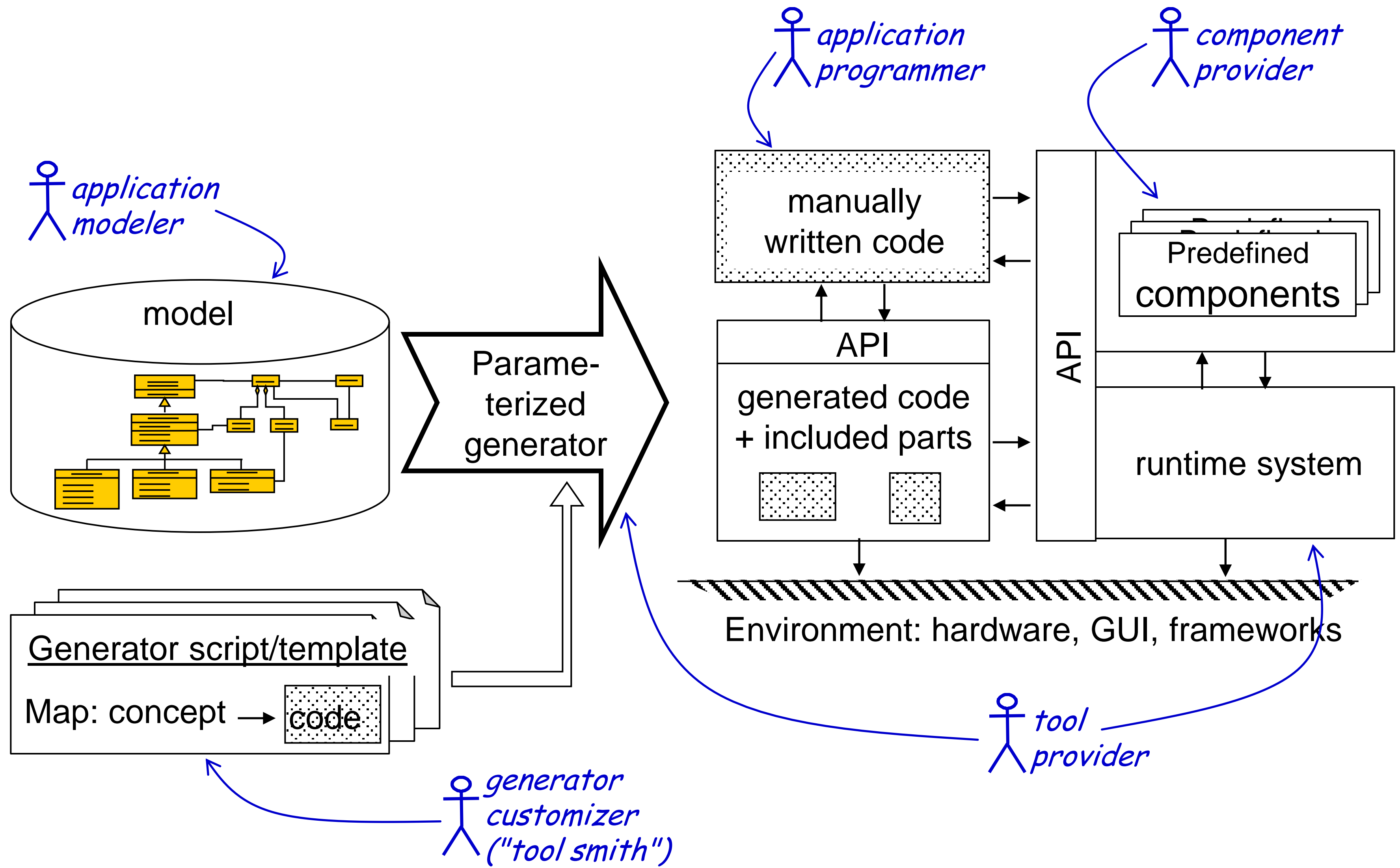
Transformational View



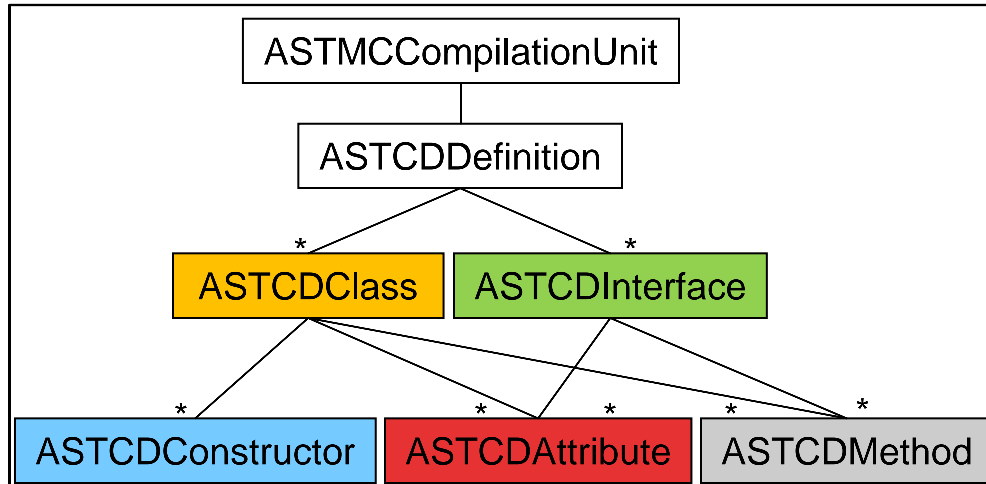
Configuration View

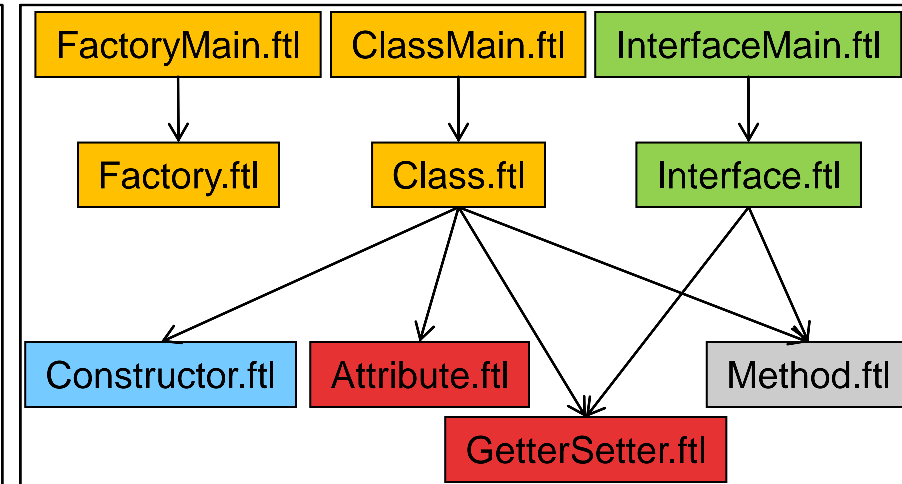


Code Generator: Parameterization + Runtime System



AST- vs. Template-Structure

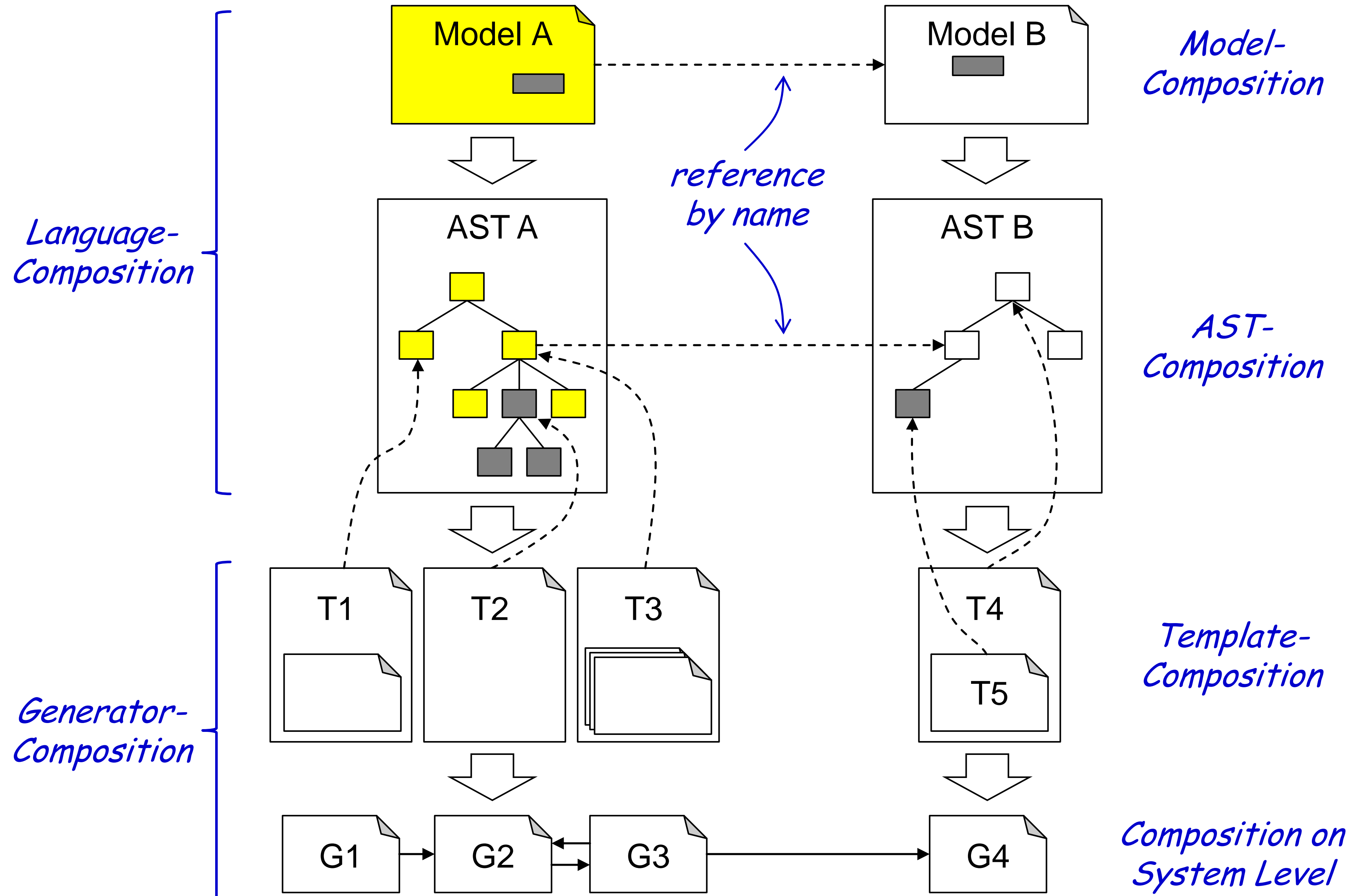




Observations:

- more than one Template for the same type of ASTNode (colors)
- hierarchy of AST corresponds to Template-Hierarchy

Composition



Example for an Extension of the Generator

- Each class with Stereotype <<printable>> shall have a print-method returning information of the class.

CD

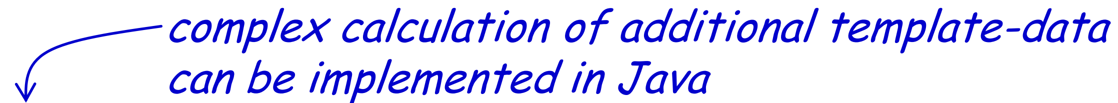
```
<<printable>> class Foo {
  void bar();
}
```

Generated/Java

```
public class Foo {
  public void bar(){…}
  public String print(){
    return "Classname: Foo\n" +
           "Methods: bar()";
  }
}
```

Example for an Extension: Calculation of additional Template Data in Java

Tool/Java




```
public class CDPrintableCalculator extends TemplateCalculator<ASTCDClass> {

 public boolean calc(ASTCDClass ast, TemplateOperator op) {
    for (ASTStereoValue v : ast.getModifier().getStereotype().getValues()) {
      if (v.getName() == "printable") {
        String m = "Classname: " + ast.getName() + "\\n";
        for (ASTCDMethod meth : ast.getCDMethods()) {
          m += ...;
        }
        op.setValue("message", m);
        return true;
      }
    }
    return false;
  }
}
```

Example for an Extension: Template and Integration

- Template (this is complete!):

FM Printable.ftl

condition (freemarker standard syntax)

```
<#if op.callCalculator(c_printable) >
  public String print(){
    return "${message}";
  }
</#if>
```

call of Java-calculation

access of calculated data as variables

- Adding to class-template:

FM ClassMain.ftl

```
${op.addValue("t_class",      "umlp.templates.Printable")}
${op.addValue("t_attr",       ["umlp.templates.Attribute",
                               "umlp.templates.GetterSetter"])}
${op.addValue("c_printable", "umlp.calculators.CDPrintableCalculator")}
…
${op.callTemplates(
    "umlp.templates.Class",
    ast.printPackage() + "." + ast.printName(),
    ast)}
```

SE SOFTWARE ENGINEERING

RWTH AACHEN UNIVERSITY

Generating Systems from UML

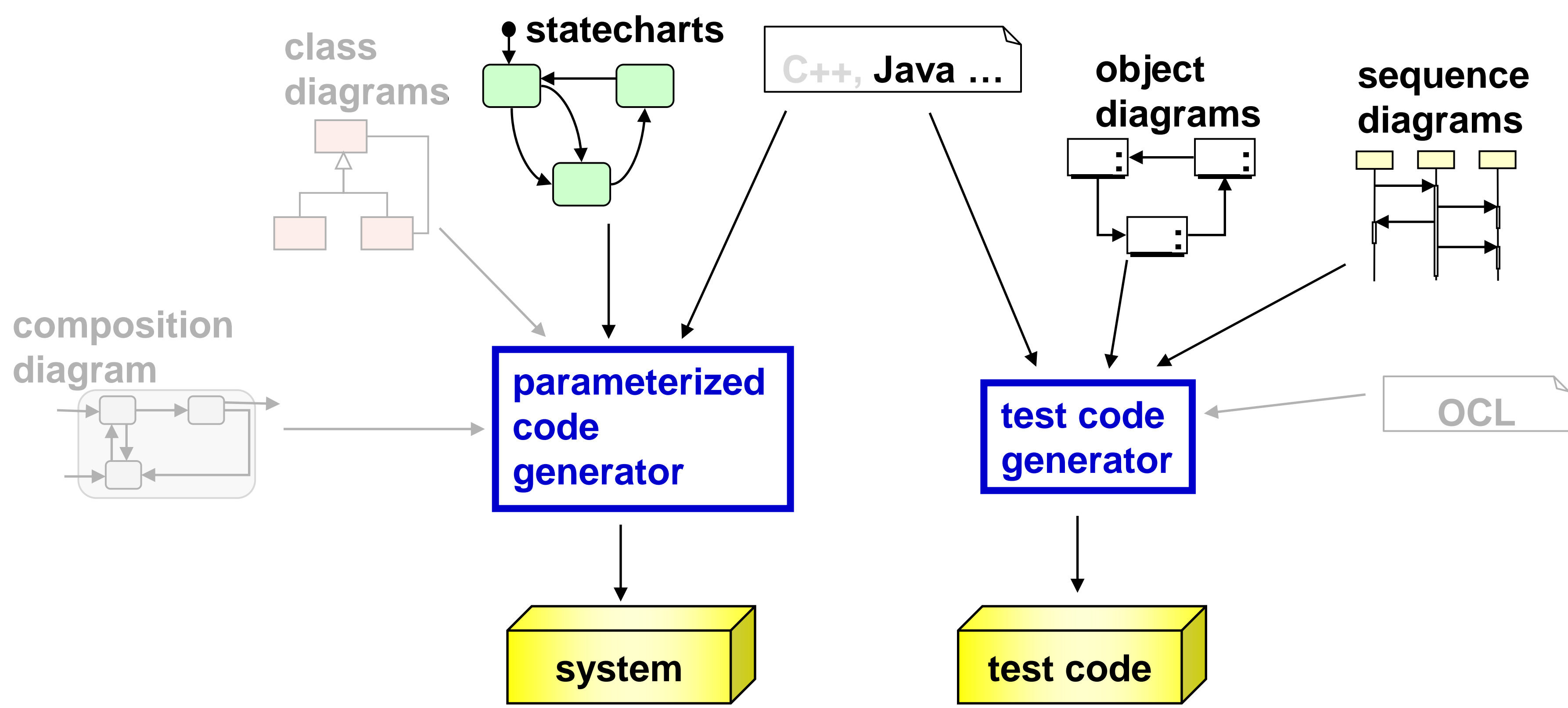
8. Summary

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Summary: UML-based Modeling

- We have considered a subset of the available UML elements:



The level of abstraction can be raised by generating substantial parts of the system and tests from UML models.

Thank you very much for your attention!

Generating Systems from UML

MoDELS 2011 Tutorial

Prof. Dr. Bernhard Rumpe,
Martin Schindler, Ingo Weisemöller
Software Engineering
RWTH Aachen

http://www.se-rwth.de/