MULTI-LEVEL MODELLING WITH METADEPTH

Tutorial

Juan de Lara, Esther Guerra

miso research group

http://miso.es @miso_uam

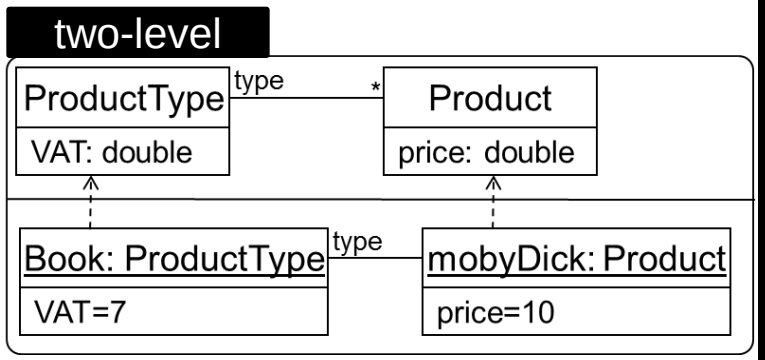
UNIVERSIDAD AUTONOMA DE MADRID

MoDELS'2016, Saint Malo (France)

WHAT IS THE TUTORIAL ABOUT?

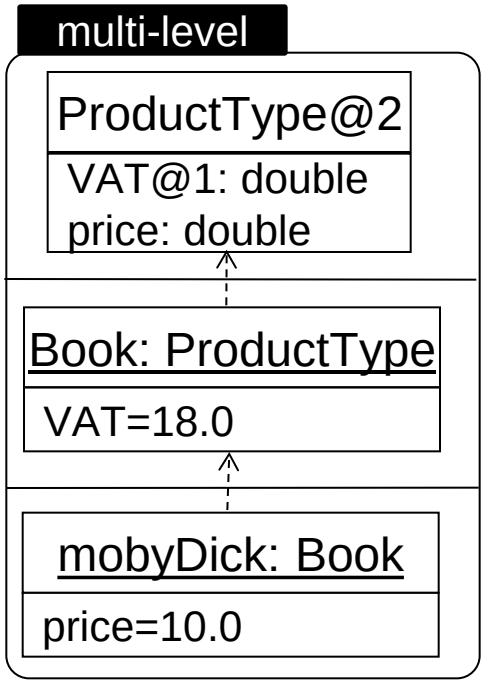
Model-Driven Engineering

- Models are core assets of development
- Automatically processable, refined into code, etc.
- Mainstream modelling approach is two-level (meta-models/models)



Multi-level modelling

- More than two meta-levels at a time
- Meta-levels can influence other levels beyond the immediate one
- Simplifies modelling in some scenarios



WHAT WILL BE COVERED

Concepts

- Basic concepts of potency-based multi-level modelling
- Use of model management languages in a multi-level setting

When to use multi-level modelling

- Examples: realistic, beyond toy examples
- “Smells” signalling scenarios where a multi-level solution has benefits
- Results of analysis over more than 400 meta-models

Use in practice

- Support by the metaDepth meta-modelling tool (http://metaDepth.org)
- Integrated with the Epsilon model management languages

MATERIAL

Download the last version of metaDepth (0.2c) at

- http://metaDepth.org
- Java 8 is needed
- PlantUML is optional (http://plantuml.com/)

All files used in this tutorial are available at

- http://metaDepth.org/tutorial

Downloading this material is not needed to follow the tutorial

…but you will get a better understanding of the tool

…and you'll be able to play with our running example

AGENDA

Multi-level modelling: the basics

- motivation
- clabjects, potency
- orthogonal classification, linguistic extensions
- meta-modelling features
- examples

MetaDepth

Multi-level model management

Advanced techniques

When and where to use multi-level modelling

Summary and conclusion

MOTIVATION

Sometimes, one needs to explicitly model meta-modelling facilities, like:

- Instantiation (modelling both types and objects)
- Declaration and instantiation of attributes and references
- Inheritance

For example:

- Product types and instances [**e-commerce** applications]
- Task types and instances [**process modelling** languages]
- Component types and instances [**architectural** languages]
- Player types and instances [**gaming** applications]
- …

EXAMPLE

Model product types (books, CDs) and their instances, to be added on demand. Product types have a VAT, while instances have a price.

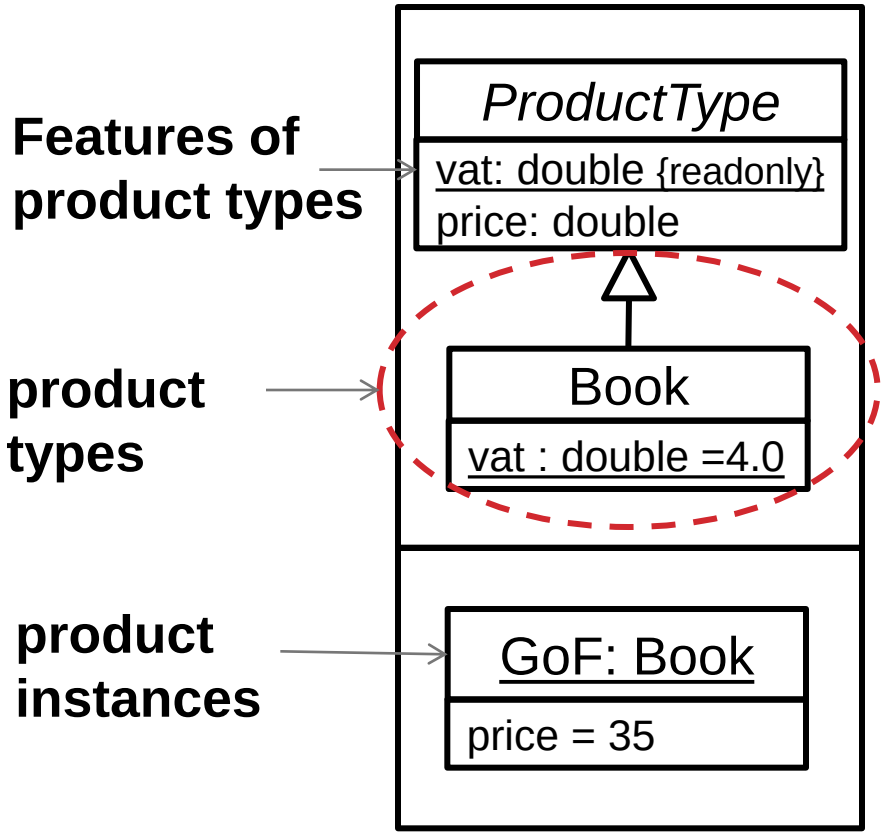
Explicit modelling of types and instances

- Types can be added dynamically
- Features of types are fixed and known a priori

Common modelling pattern, also known as: Type object [Martin et. al 97]**, item descriptor** [Coad 1992]**, metaobject** [Kiczales and Rivieres 1991]

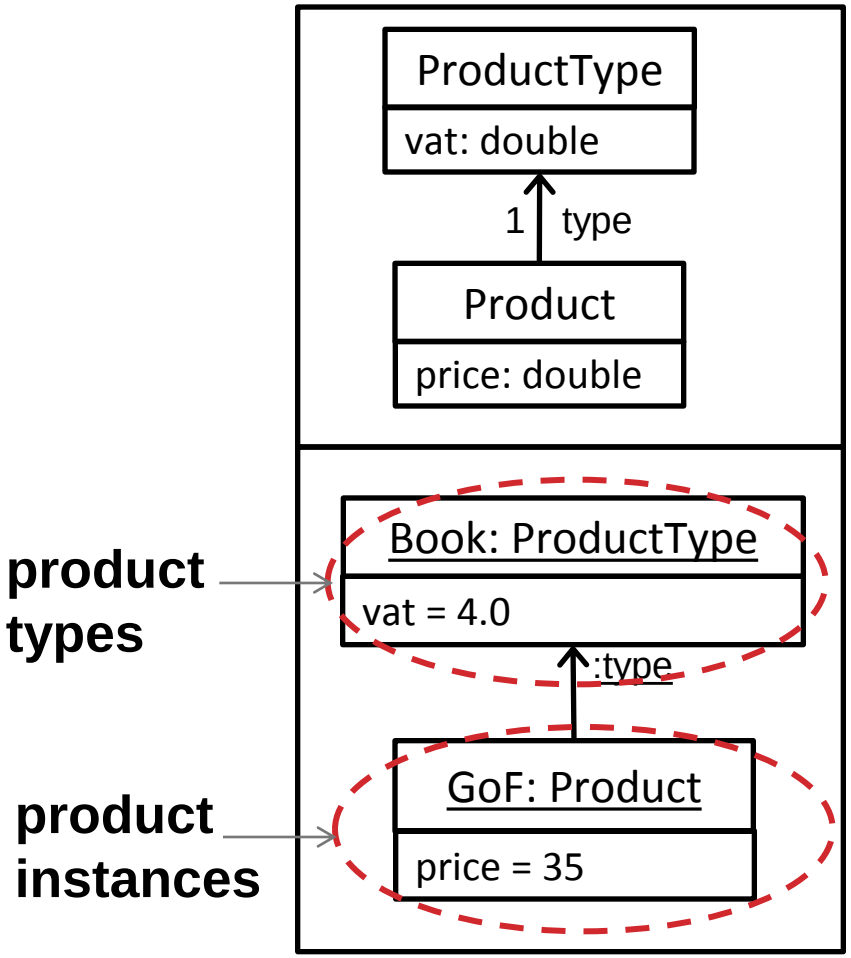
SOLUTION 1: STATIC TYPES

- Does not meet the requirements (product types are not dynamic)
- Emulates values at the meta-level via redefinition
- Uses a native meta-modelling facility (instantiation)



SOLUTION 2: EXPLICIT DYNAMIC TYPES

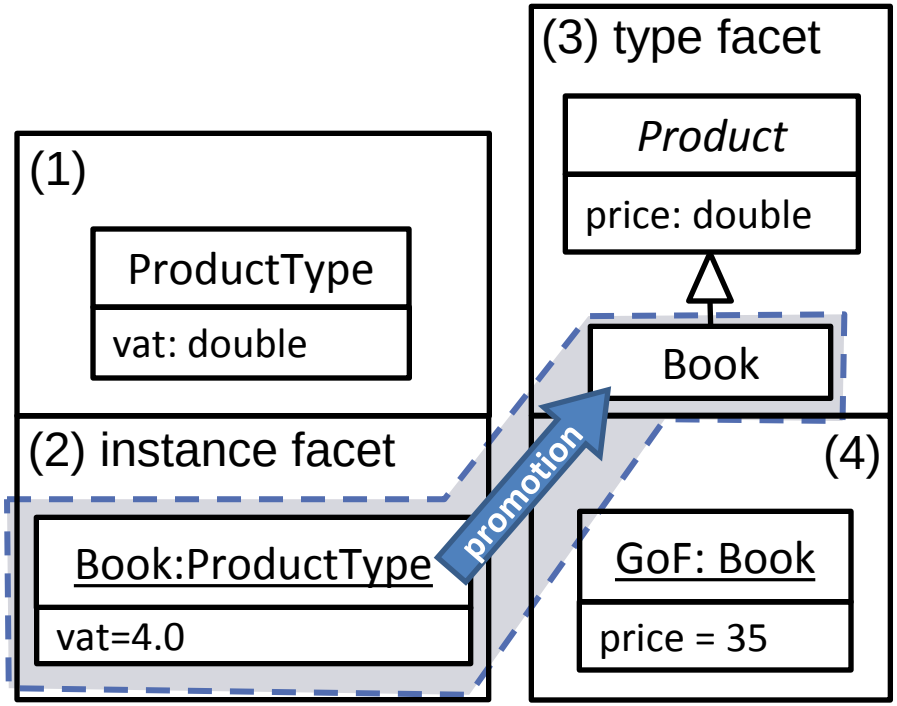
- Requires adding an extra class (more if dynamic features are needed)
- Requires emulating the instantiation relation
- Explicit modelling of instantiation yields flexibility



SOLUTION 3: PROMOTION TRANSFORMATION

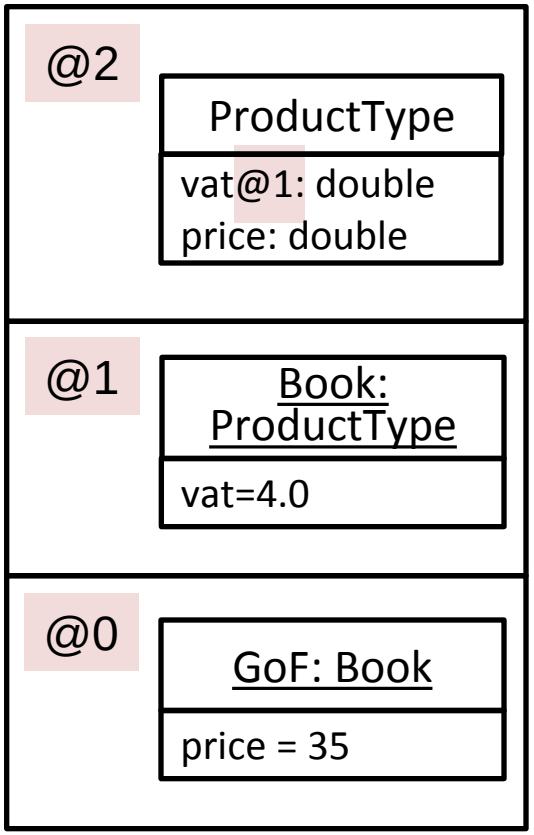
- Needs a model transformation
- Hides information: "*every ProductType is a Product*" hardcoded in the transformation
- Disconnection between type and instance facets of ProductTypes
- Flexibility of the transformation

Similar to the concept of ***powertype***



SOLUTION 4: MULTI-LEVEL

- ✗ Instantiation semantics is fixed
- ✓ Elements with instance and type facets at the same time
- ✓ Simplicity (less objects)



Multi-level modelling

the basics

CLABJECT = CLASS + OBJECT

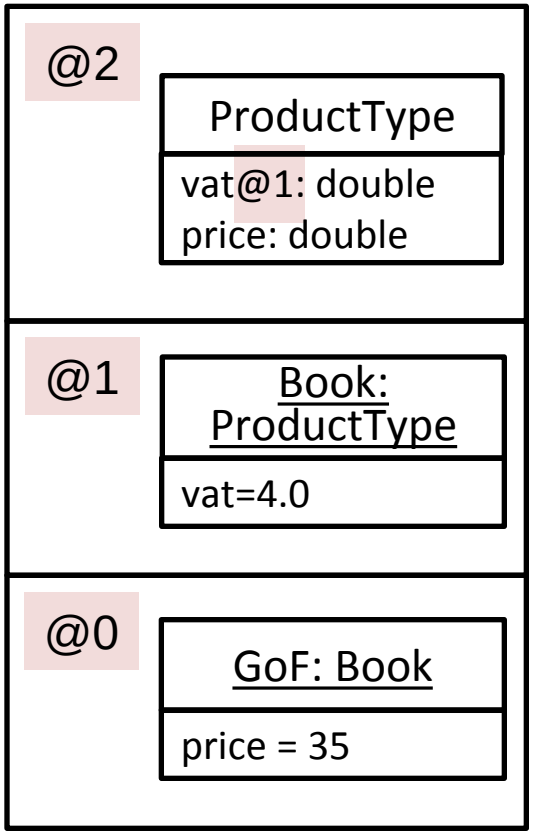
Elements have a combined type and instance facet

Book

- Instance of `ProductType`
  - Can provide a value for vat
- Type for `GoF`
  - Can declare new features
  - (We'll see how, using the OCA)

ProductType has type facet only

GoF has instance facet only



C. Atkinson and T. Kühne. 2001. The essence of multilevel metamodeling. *UML'01 (LNCS)*, Vol. 2185. Springer, 19–33.

POTENCY

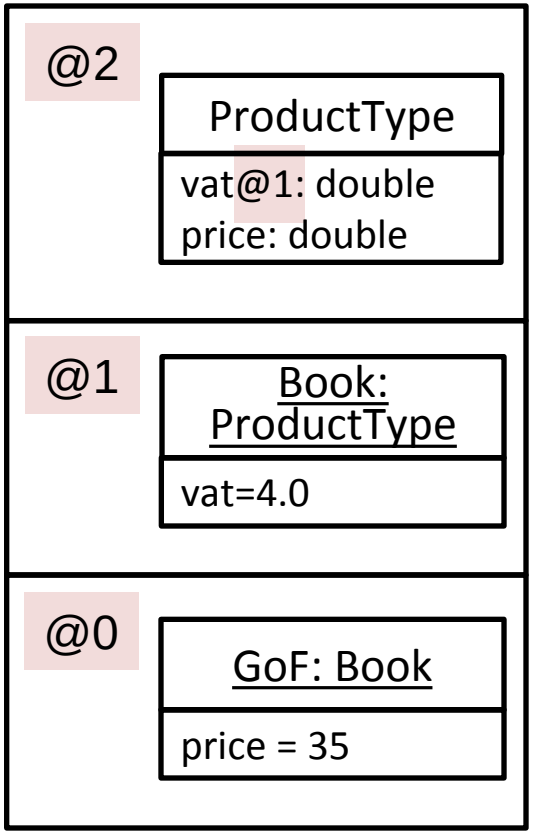
Used to characterize instances beyond the next meta-level

Models, Clabjects and their features have a potency

- Natural number (or zero)
- Decreased at each lower meta-level
- Indicates at how many meta-levels the element can be instantiated

We use the "@potency" notation

By default elements take the potency of their containers

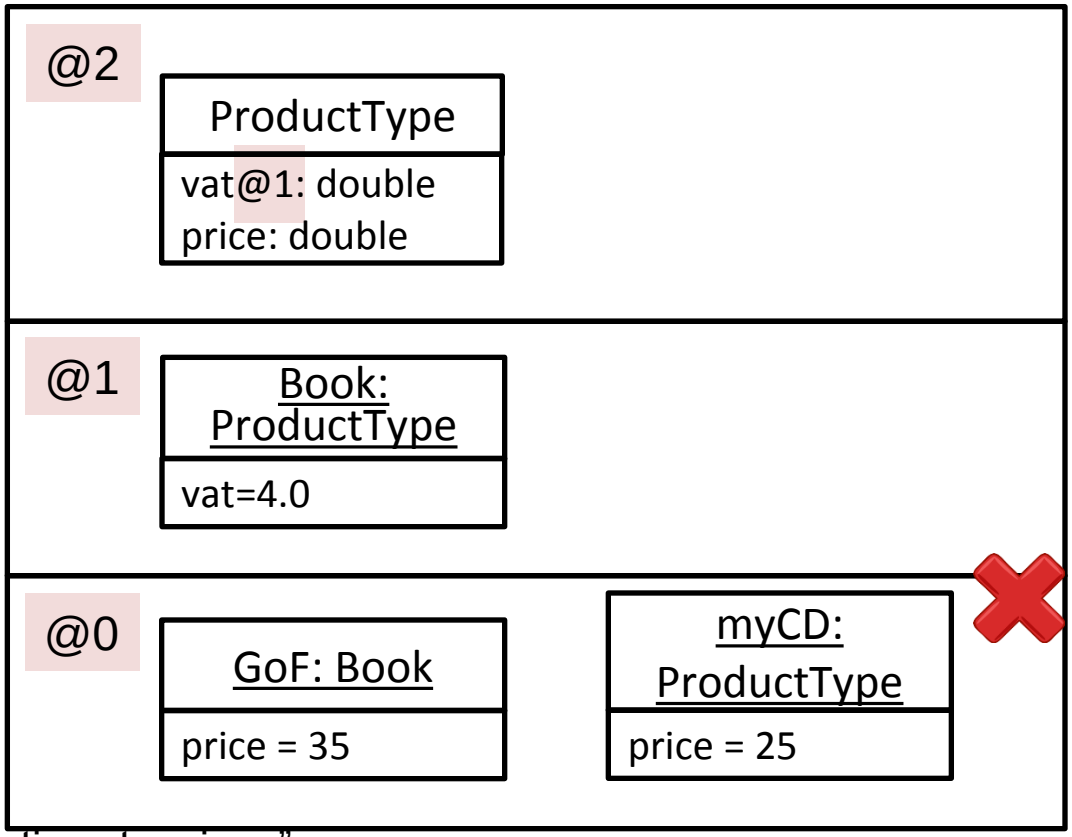


C. Atkinson and T. Kühne. 2001. The essence of multilevel metamodeling. *UML'01 (LNCS)*, Vol. 2185. Springer, 19–33.

POTENCY

Strict meta-modelling approach (*)

Instantiation is *mediated*

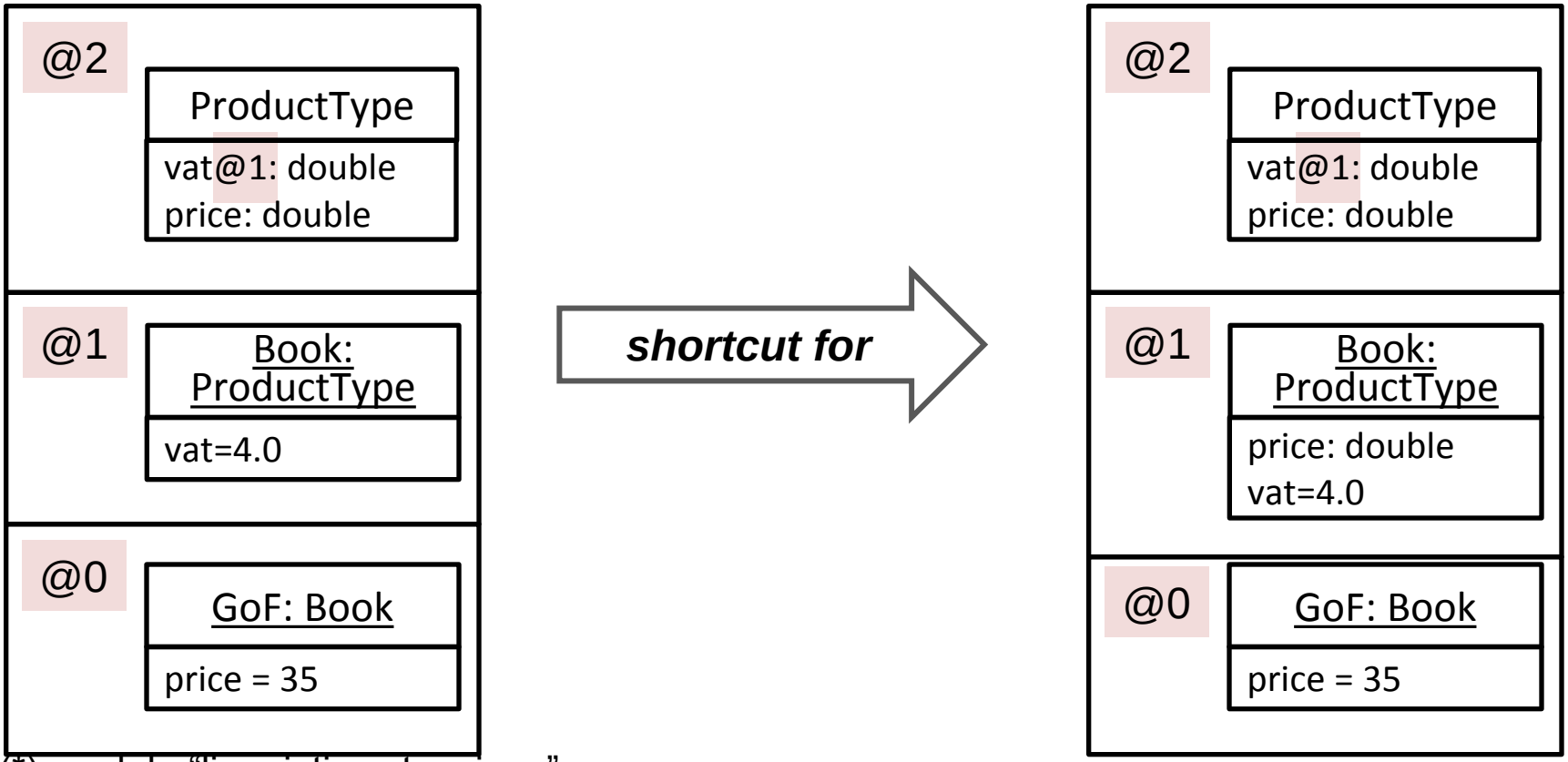


(*) modulo “linguistic extensions”

POTENCY

Strict meta-modelling approach (*)

Instantiation is *mediated… also for fields*

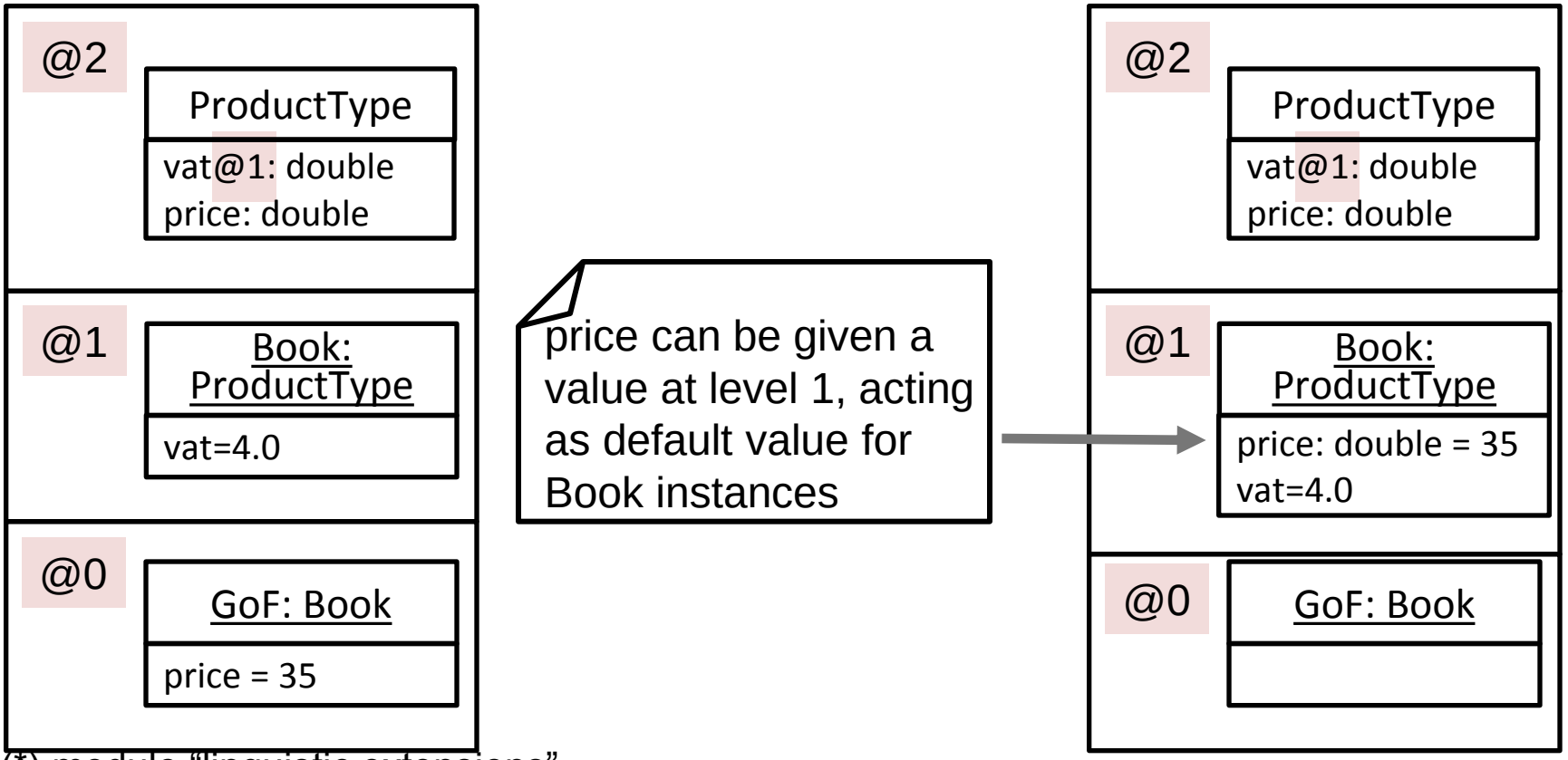


(*) modulo "linguistic extensions"

POTENCY

Strict meta-modelling approach (*)

Instantiation is *mediated… also for fields*

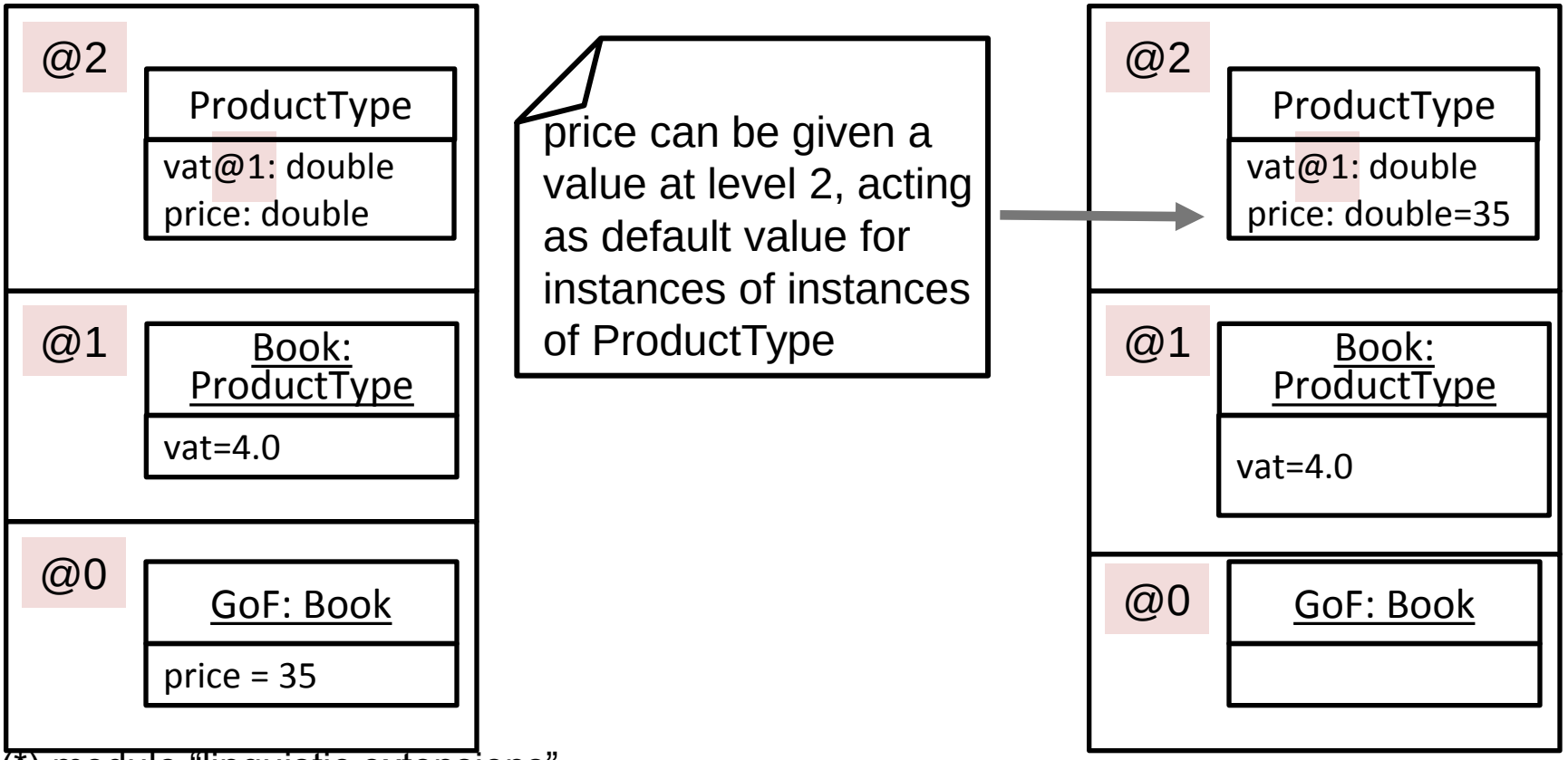


(*) modulo "linguistic extensions"

POTENCY

Strict meta-modelling approach (*)

Instantiation is *mediated… also for fields*

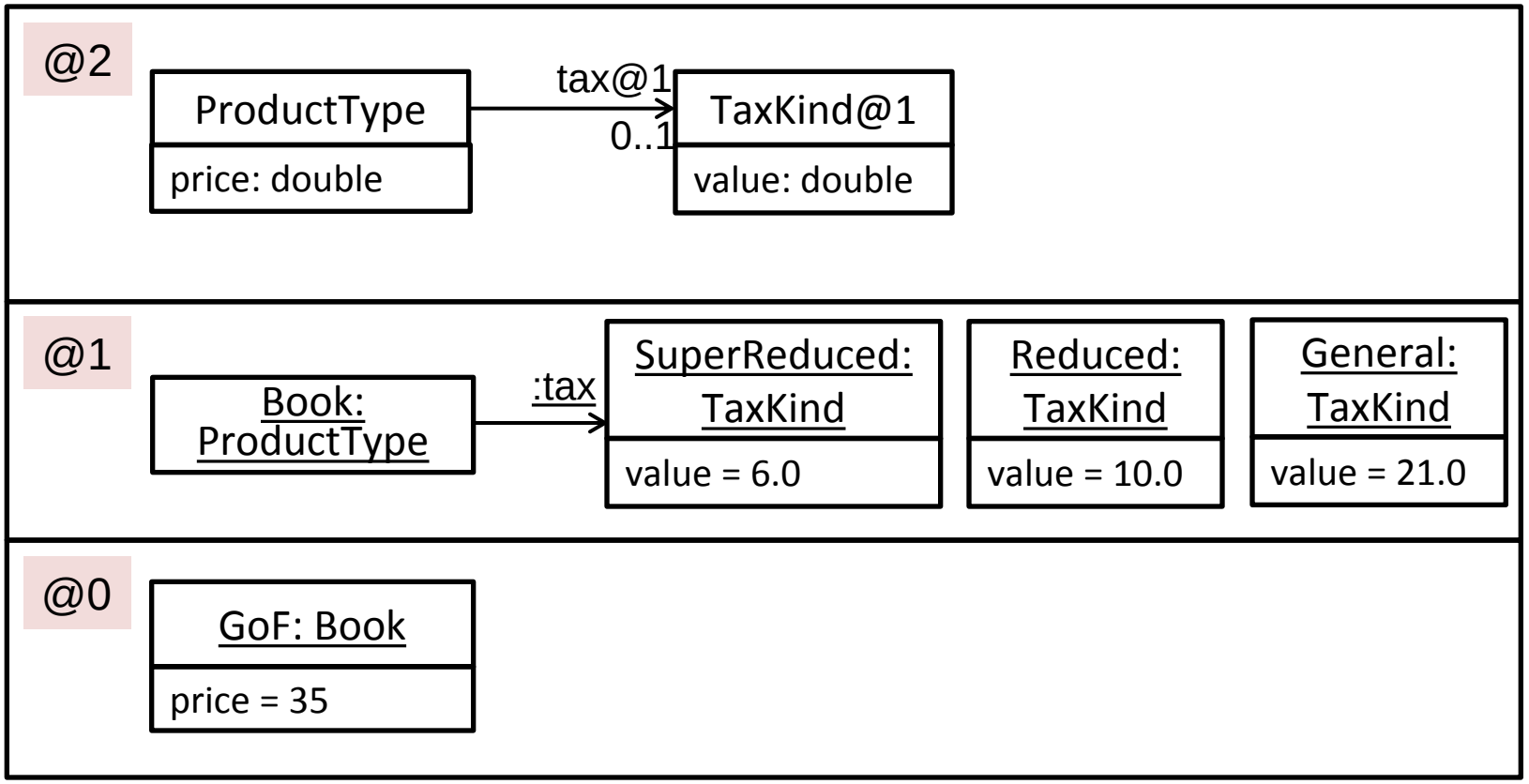


(*) modulo "linguistic extensions"

LEVEL

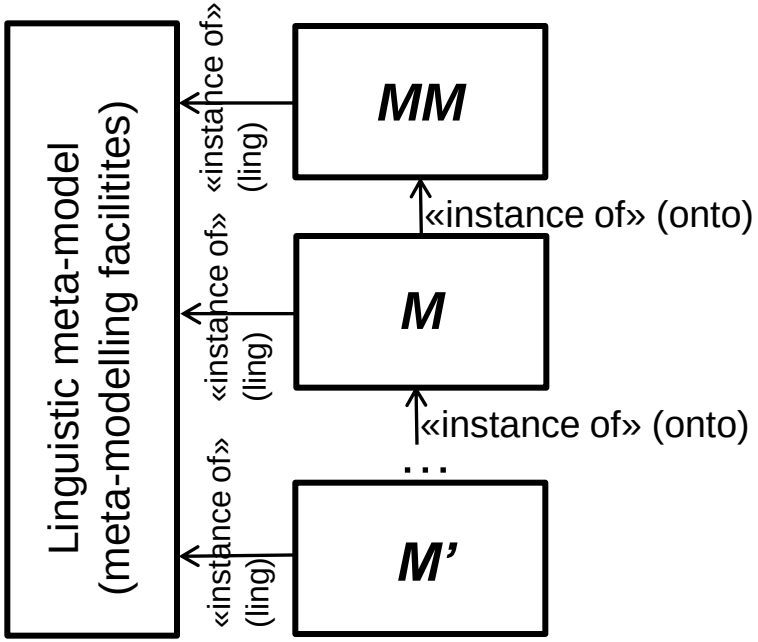
Refers to the potency of the model

A model with level L can hold elements with potency ≤ L



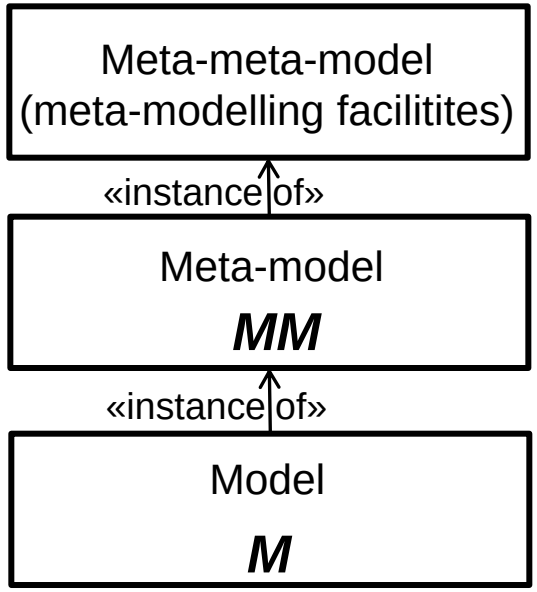
ORTHOGONAL CLASSIFICATION (OCA)

Multi-level



- Dual typing (ontological, linguistic)
- Make meta-modelling facilities available at every meta-level

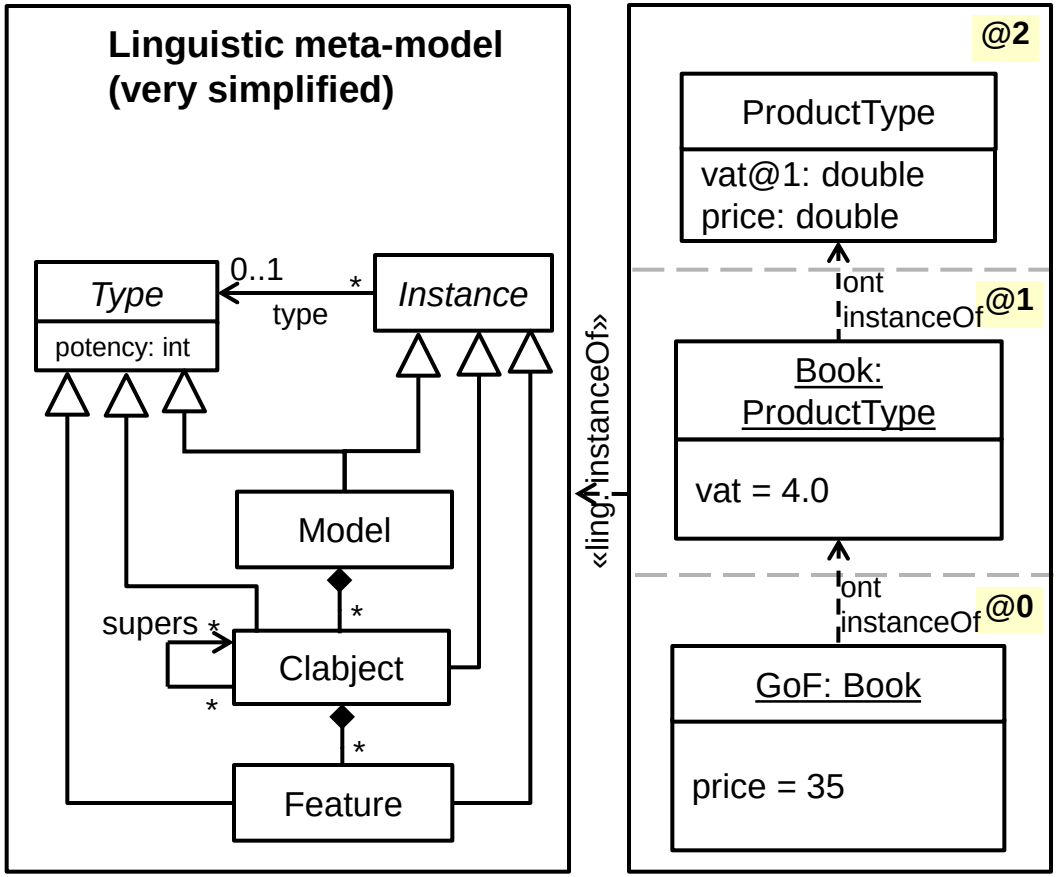
Two-Level



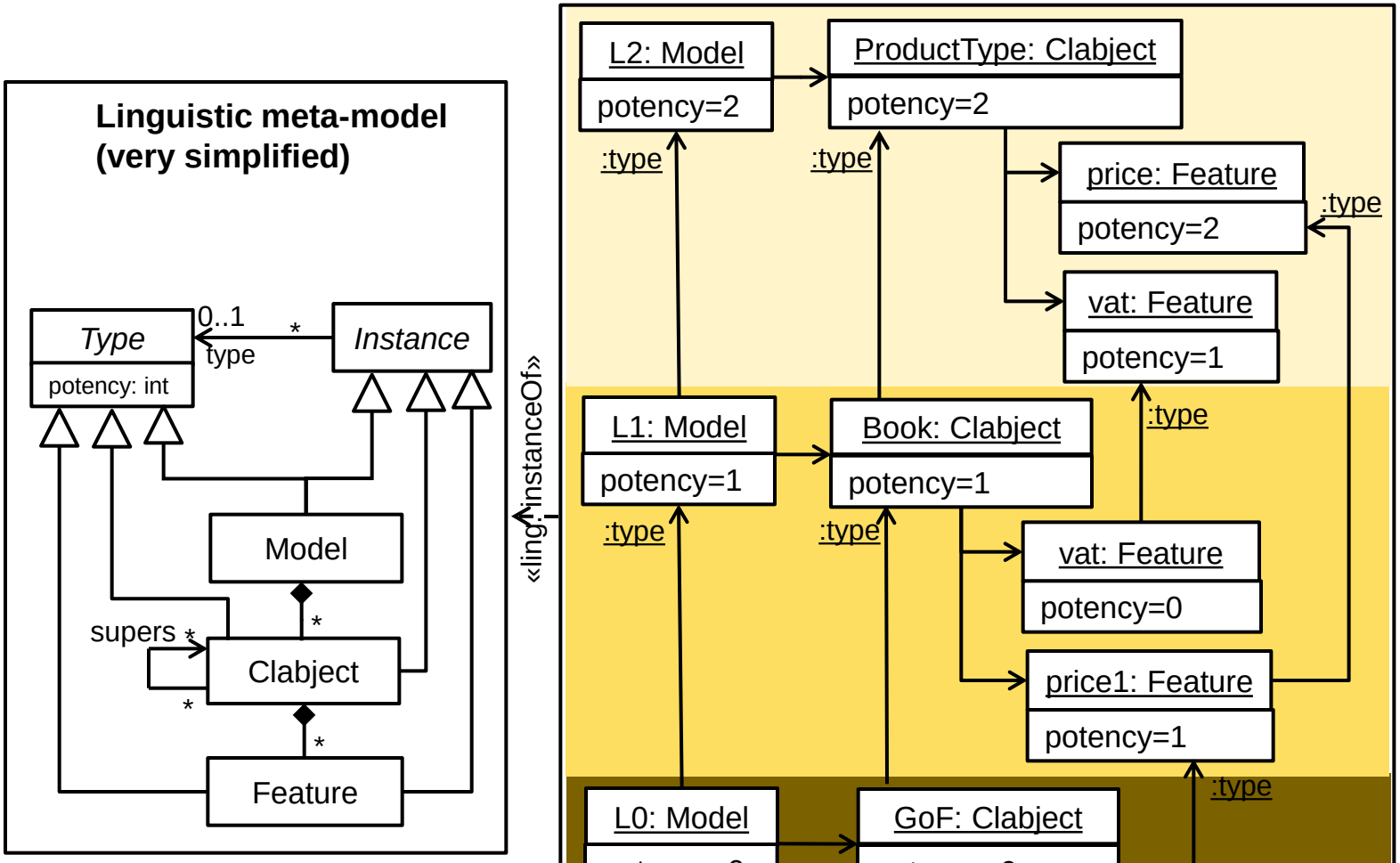
- Types and meta-modelling facilities only at the meta-model level

C. Atkinson, T. Kühne. 2002. Rearchitecting the UML infrastructure. ACM Trans. Model. Comput. Simul. 12, 4 (2002), 290–321.

ORTHOGONAL CLASSIFICATION (OCA)



LINGUISTIC VIEW



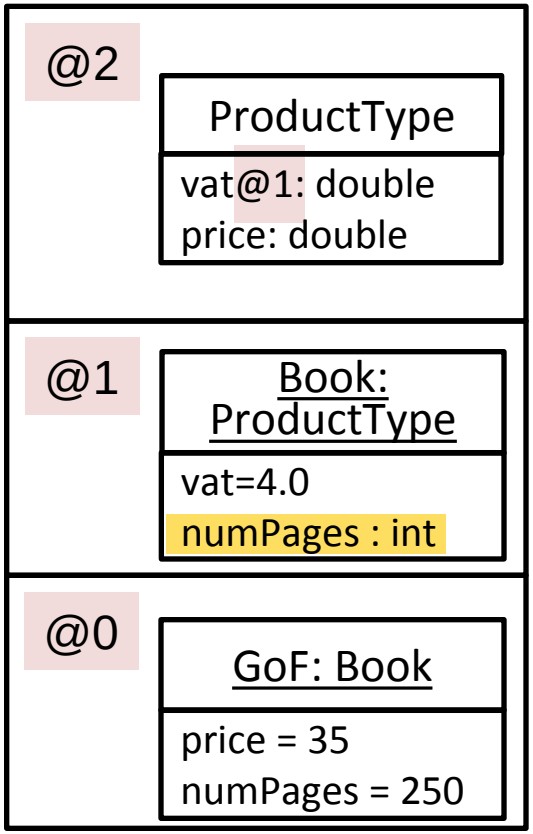
LINGUISTIC EXTENSIONS

Elements with no ontological type

- Ontological typing is optional
- Linguistic typing is mandatory

New clabjects or features

Not everything can be anticipated at the top-most level



J. de Lara, E. Guerra. 2010. Deep meta-modelling with metaDepth. In TOOLS'10 (LNCS), Vol. 6141. Springer, 1–20.

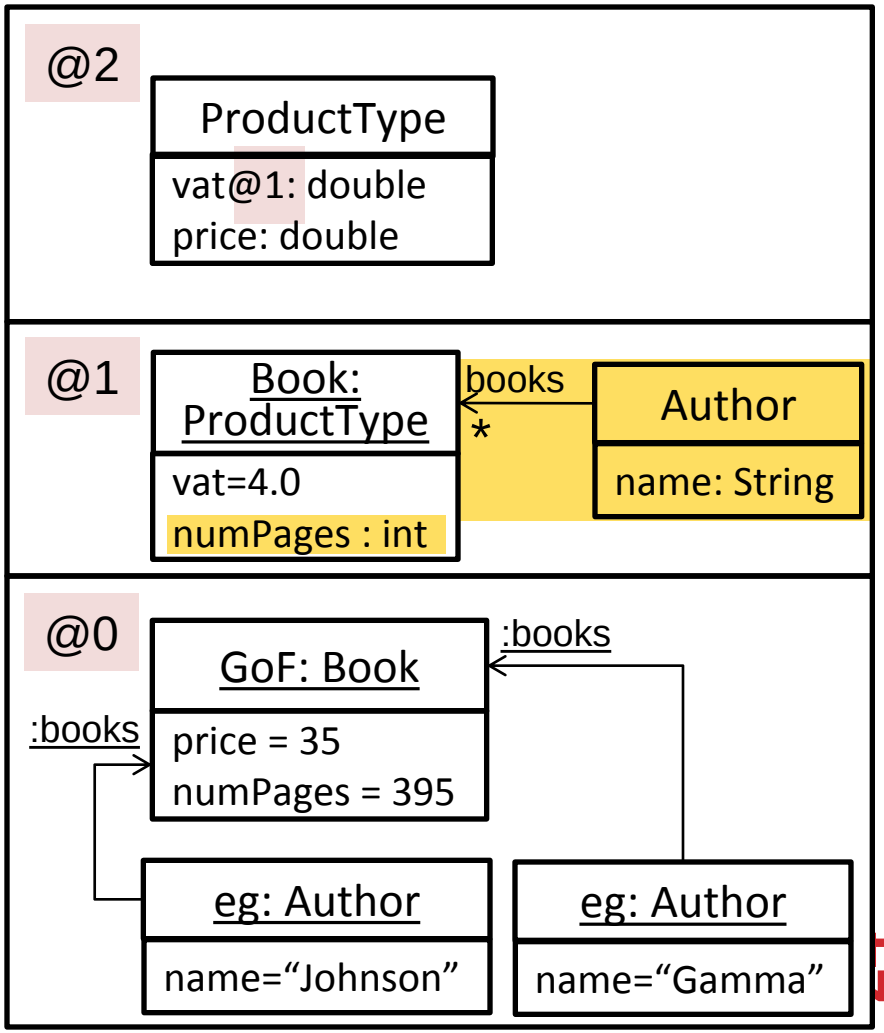
LINGUISTIC EXTENSIONS

Elements with no ontological type

- Ontological typing is optional
- Linguistic typing is mandatory

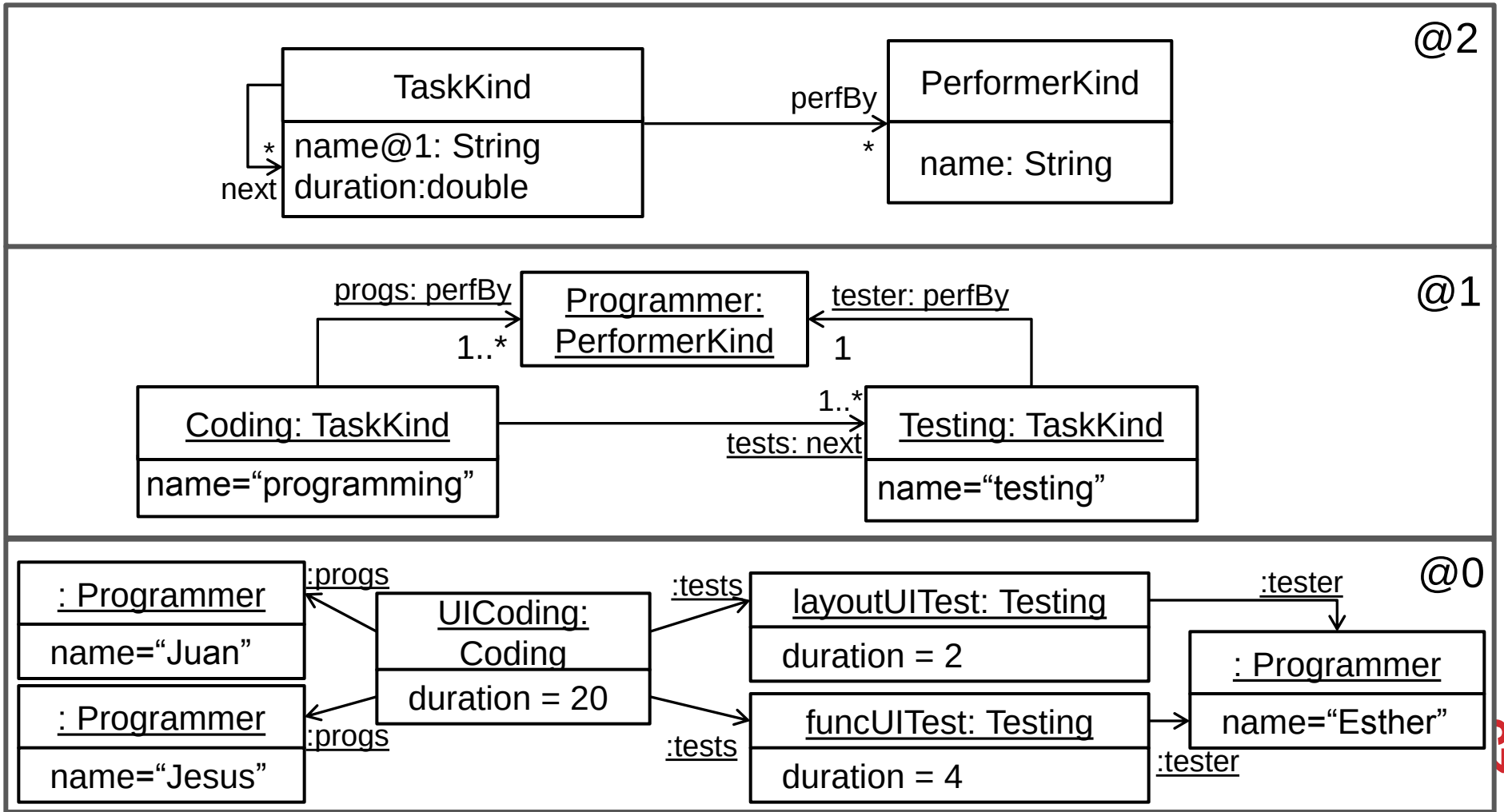
New clabjects or features

Not everything can be anticipated at the top-most level



META-MODELLING FEATURES: REFERENCES AND CARDINALITIES

Reference cardinality: only applies to next meta-level

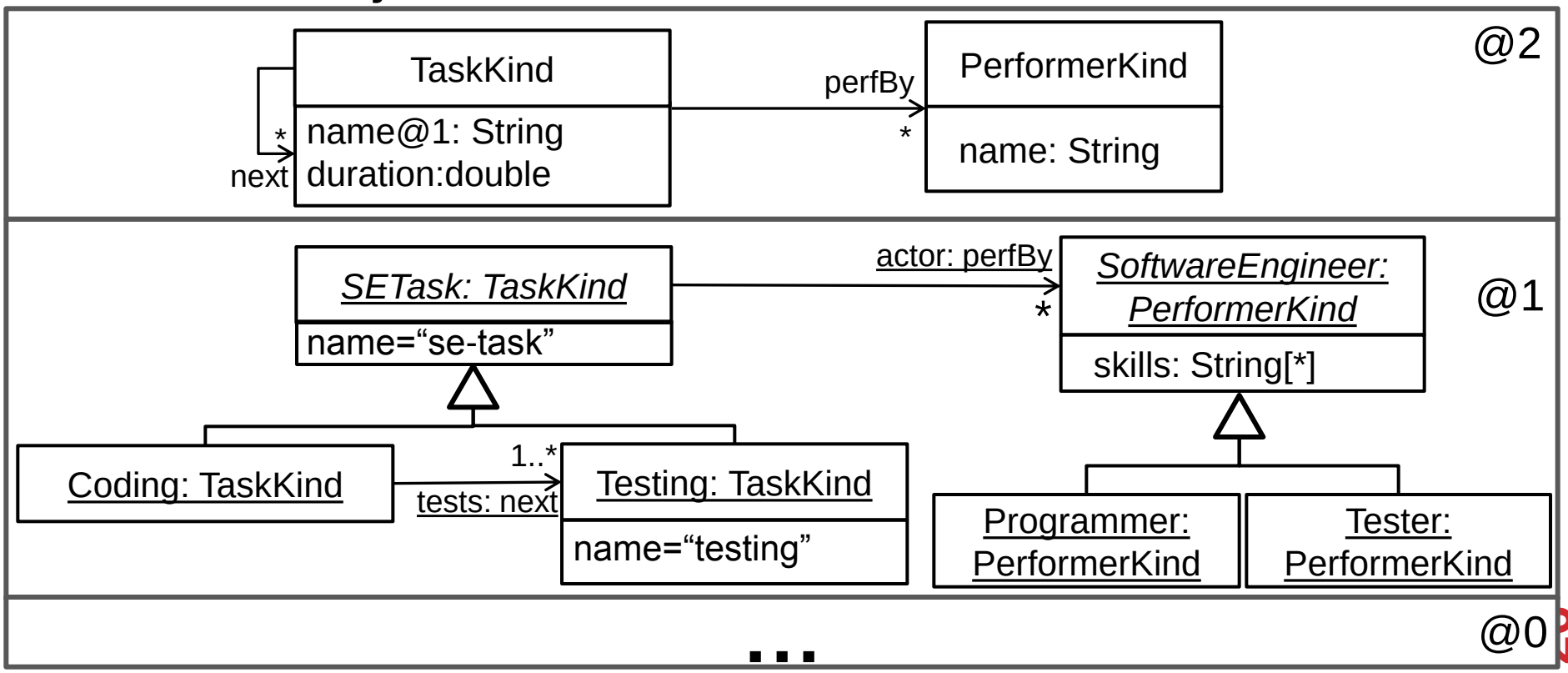


META-MODELLING FEATURES: INHERITANCE

Can be used at any meta-level

Inheritance of type and instance facets

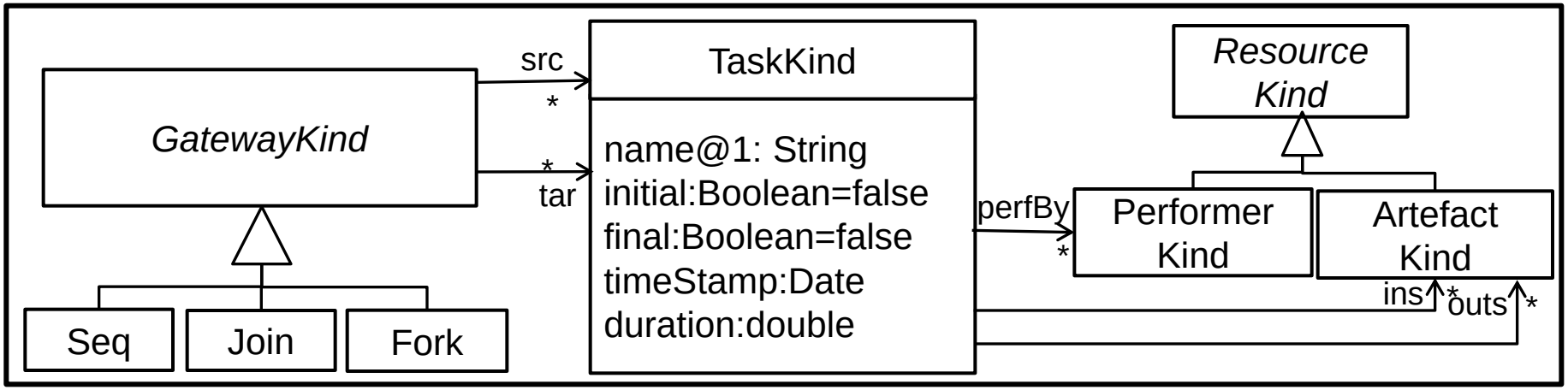
Abstract clabjects



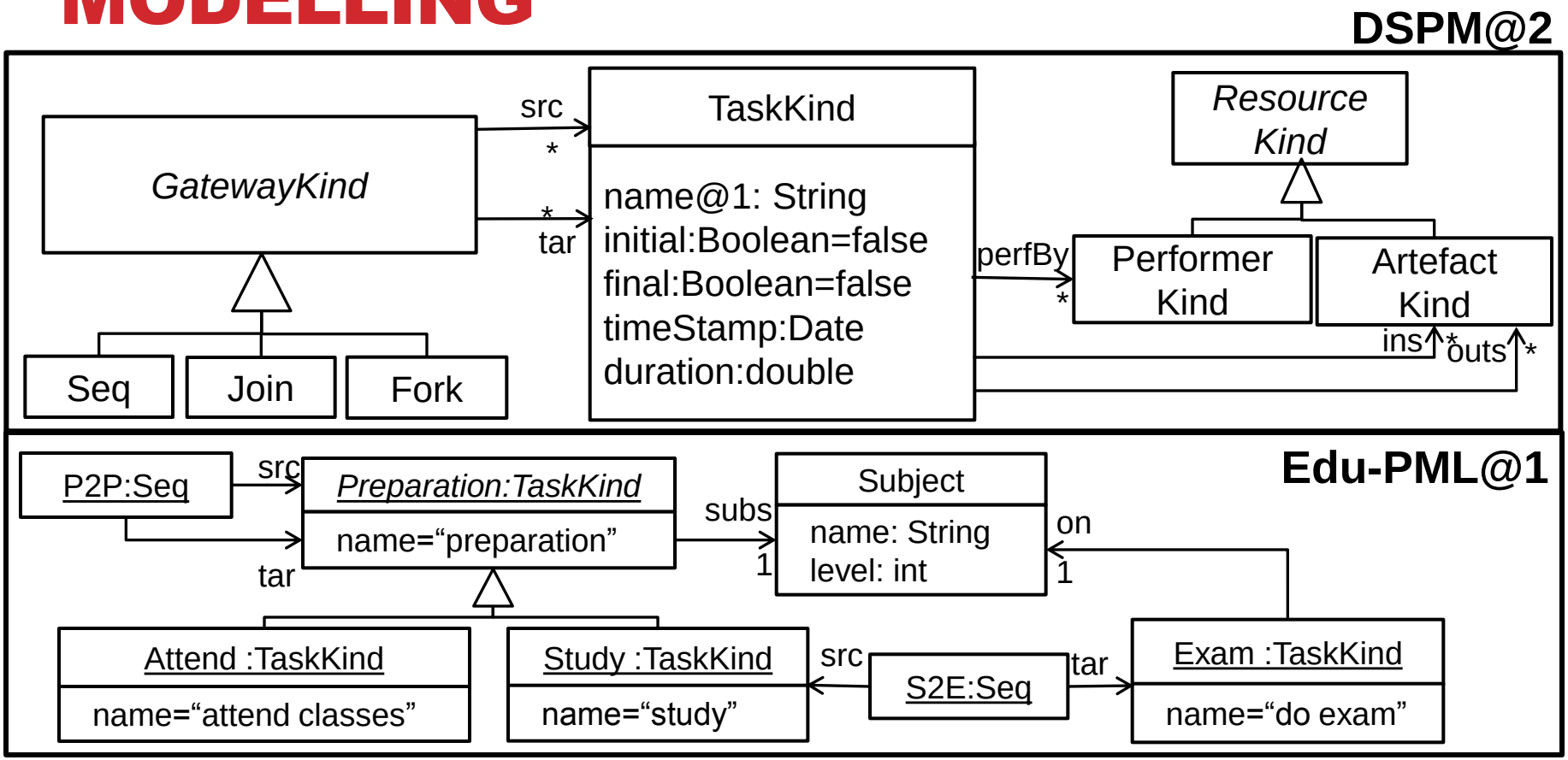
Examples

DOMAIN SPECIFIC PROCESS MODELLING

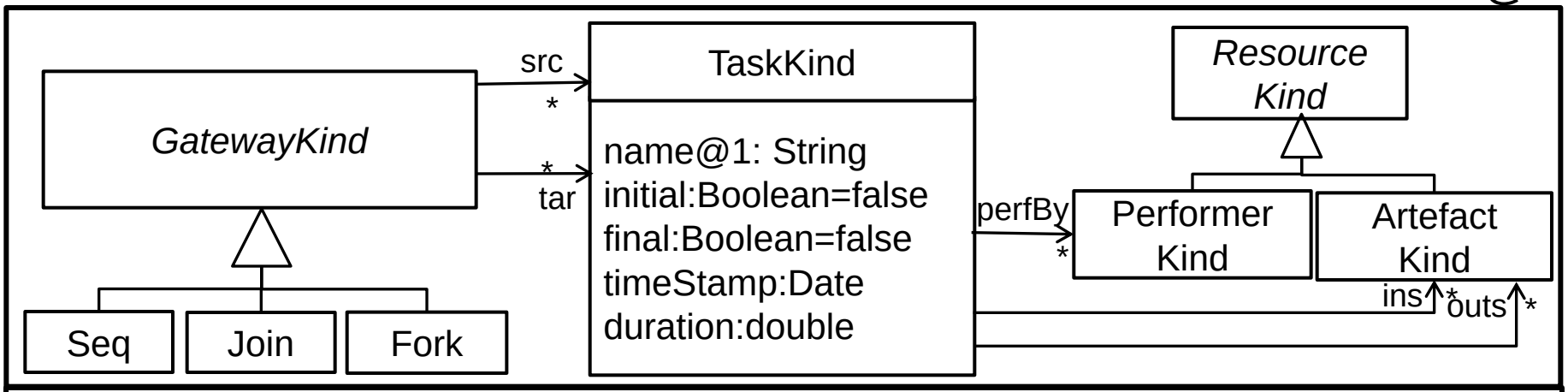
DSPM@2

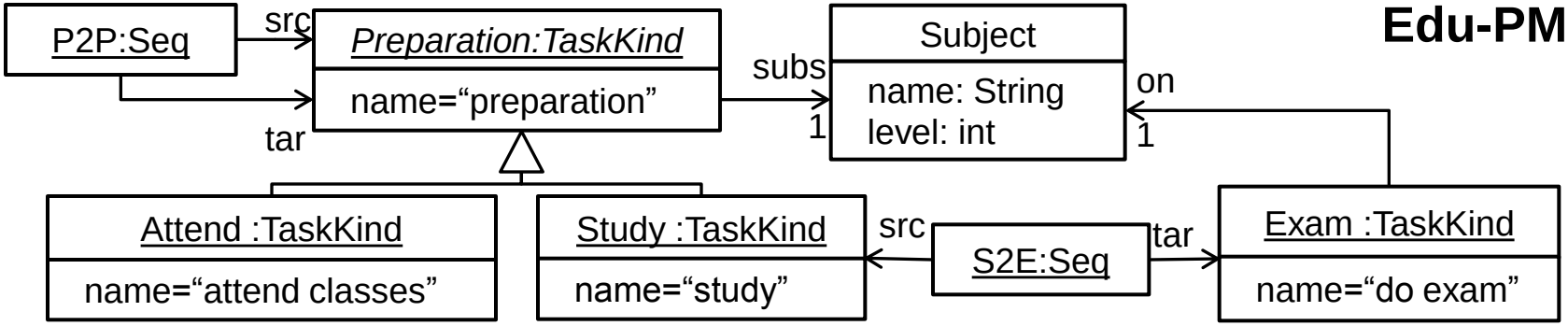


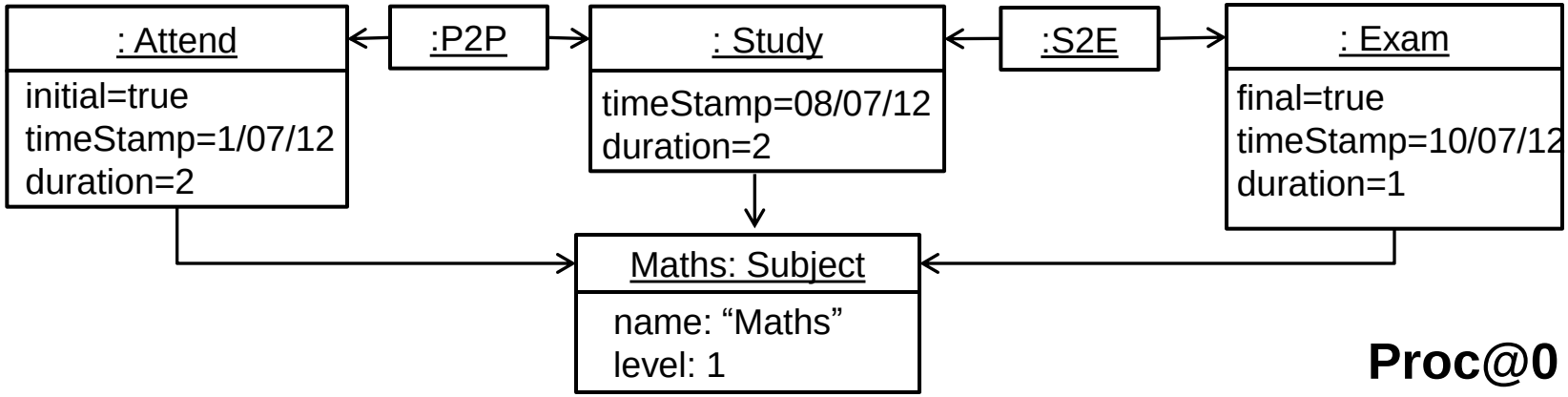
DOMAIN SPECIFIC PROCESS MODELLING



J. de Lara, E. Guerra, J. Sánchez Cuadrado. *Model-driven engineering with domain-specific meta-modelling languages*. Software and System Modeling 14(1): 429-459 (2015)

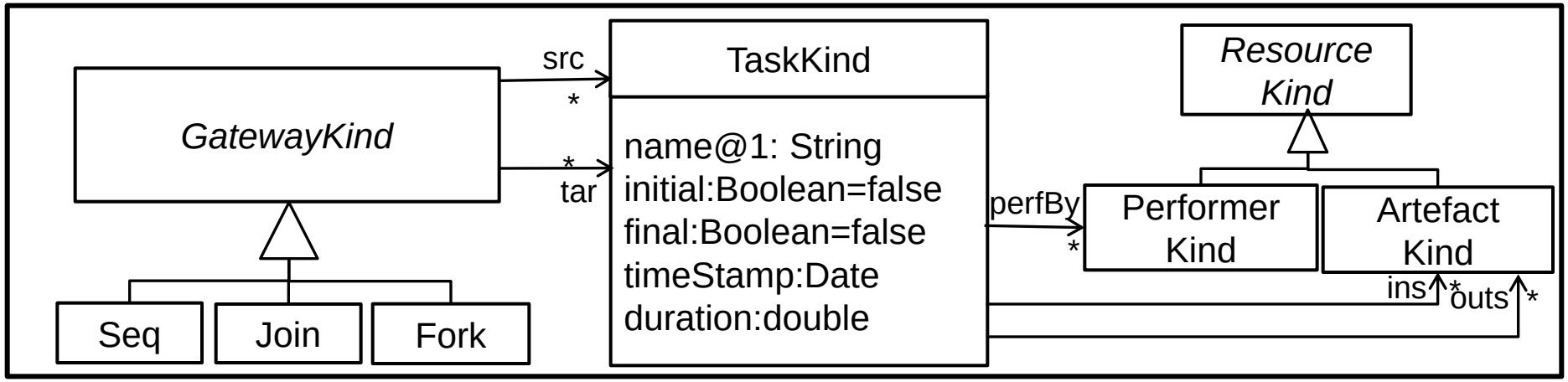

GatewayKind
src
*
*
tar
TaskKind
name@1: String
initial:Boolean=false
final:Boolean=false
timeStamp:Date
duration:double
Resource Kind
perfBy
*
Performer Kind
Artefact Kind
ins
*
outs
*
Seq
Join
Fork


P2P:Seq
src
Preparation:TaskKind
name="preparation"
tar
subs
1
Subject
name: String
level: int
on
1
Attend :TaskKind
name="attend classes"
Study :TaskKind
name="study"
src
S2E:Seq
tar
Exam :TaskKind
name="do exam"

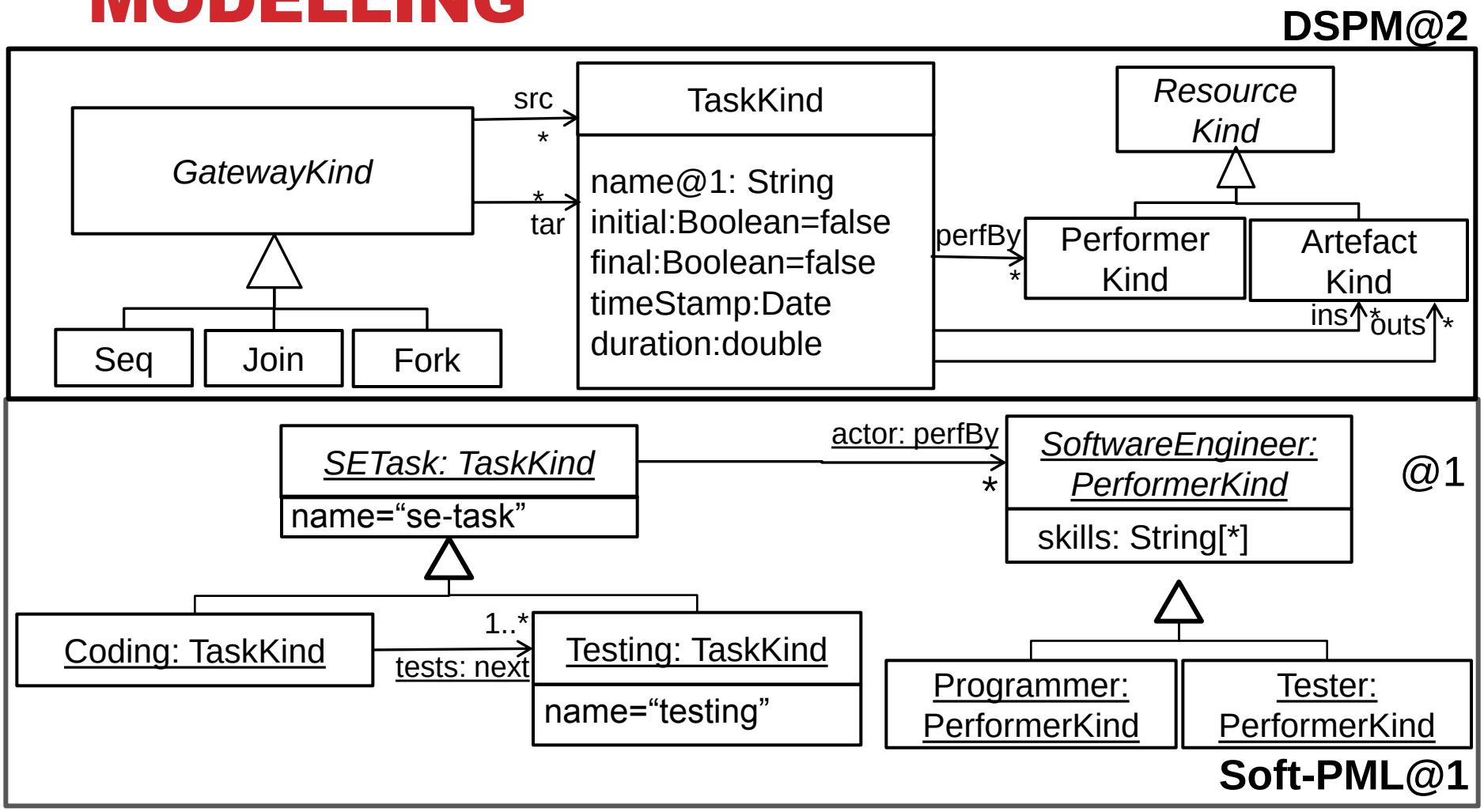

: Attend
initial=true
timeStamp=1/07/12
duration=2
:P2P
: Study
timeStamp=08/07/12
duration=2
:S2E
: Exam
final=true
timeStamp=10/07/12
duration=1
Maths: Subject
name: "Maths"
level: 1

DOMAIN SPECIFIC PROCESS MODELLING

DSPM@2



DOMAIN SPECIFIC PROCESS MODELLING



ADVANTAGES

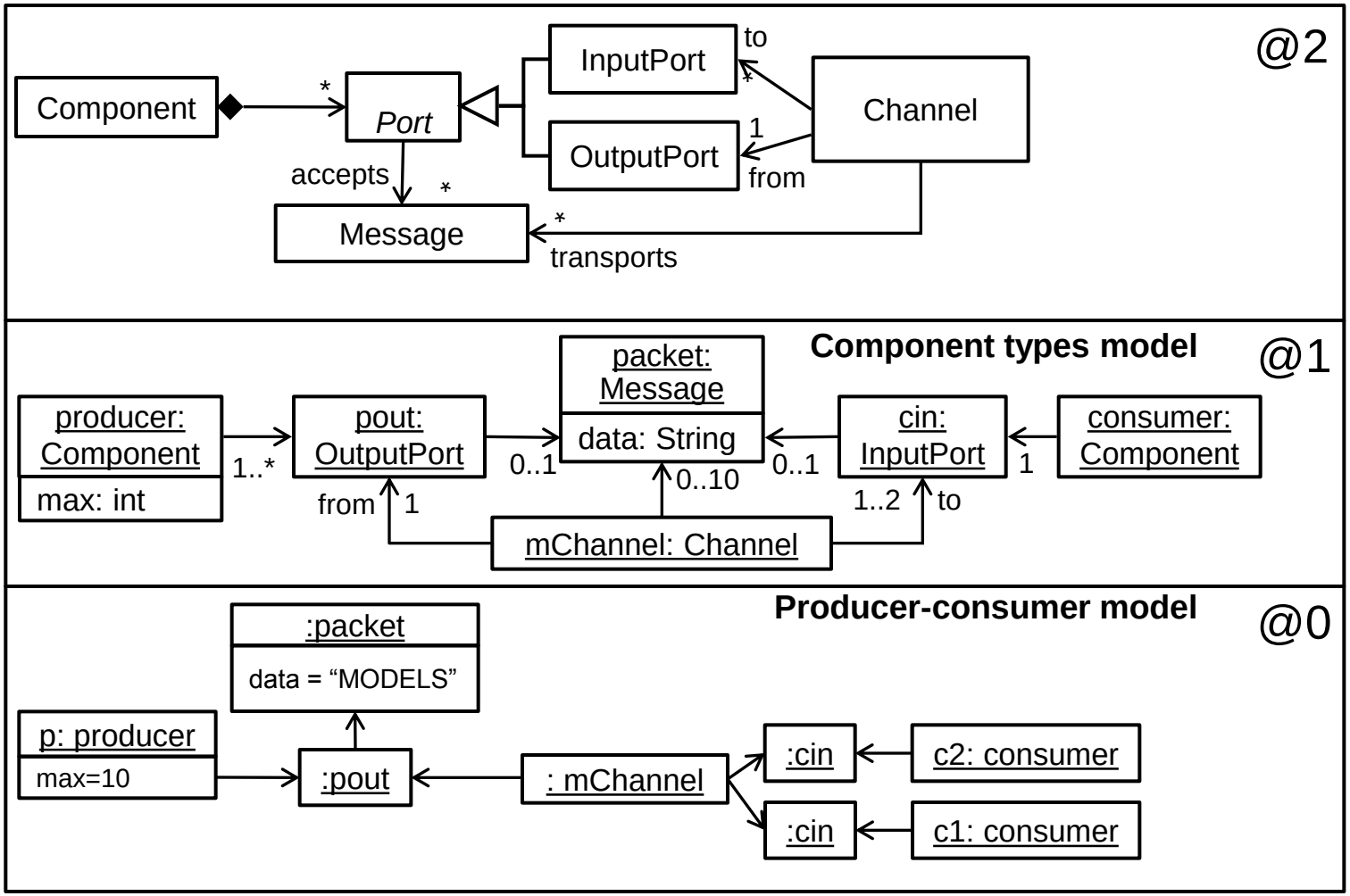
The top level can be customized for the process domain

- Family of DSLs for process modelling

Transformations can be defined over the top level and reused across the whole family

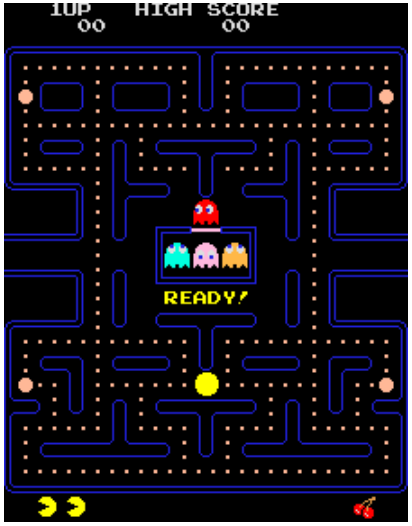
- Code generators
- Model-to-model transformations
- In-place transformations
- Queries

COMPONENT-BASED META-MODELLING



EXERCISE/RUNNING EXAMPLE: ARCADE/ADVENTURE GAMES

Sabre Wulf



Pacman

Commando

Atic atac

Gauntlet

EXERCISE/RUNNING EXAMPLE: A META-MODEL FOR ARCADE GAMES

Create a language to describe adventure games (like Pacman, Sabre Wulf, Gauntlet, Atic-atac, etc)

Games based on connected tiles, able to hold treasures and playing characters

- “heroes” (like Pacman)
- “evils” (like Ghosts)

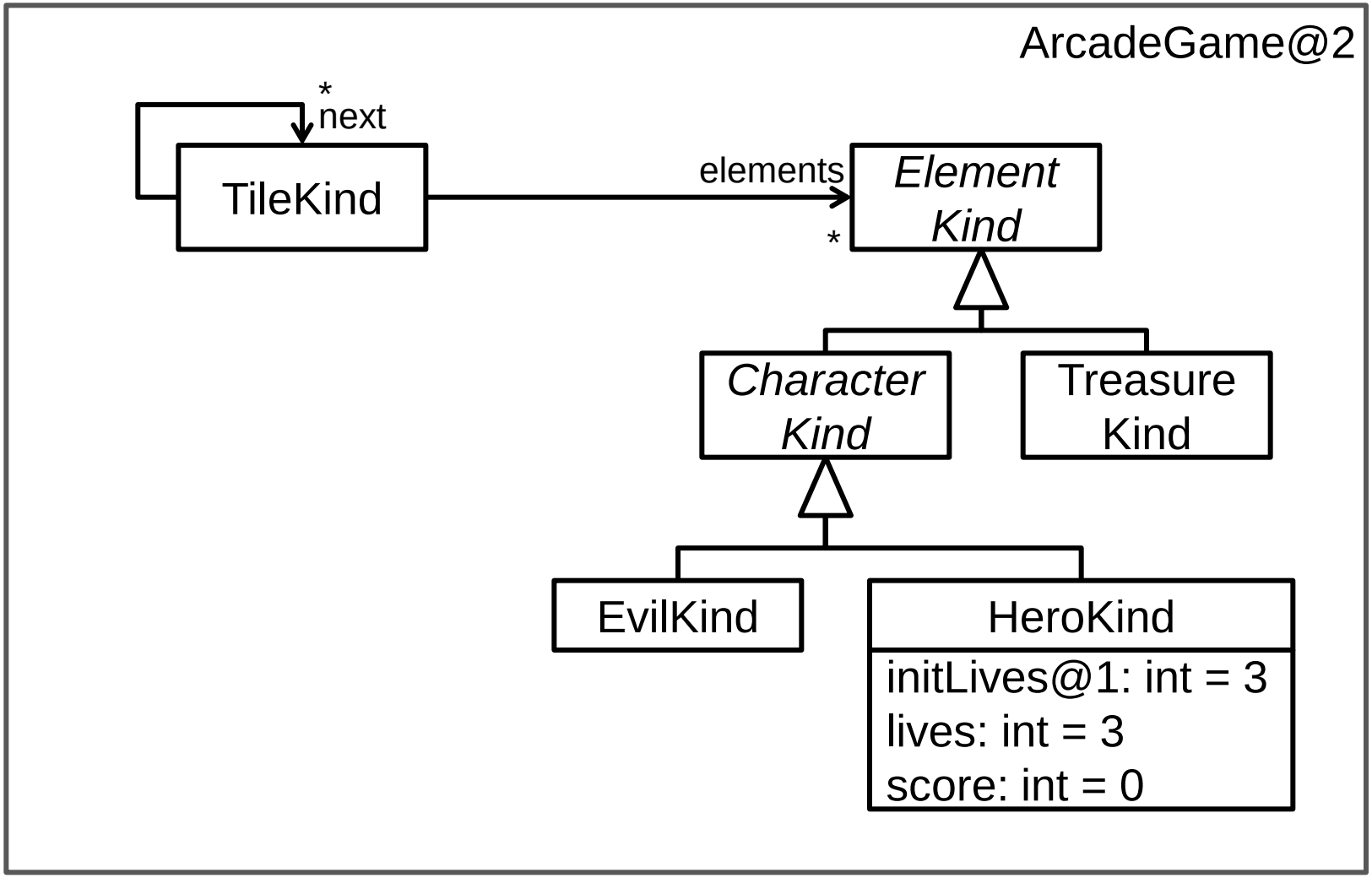
Heros will have an initial number of lives, while at run-time, they will hold an actual number of lives and a score

The language should permit creating types of tiles, heroes, evils and treasures (with custom properties)

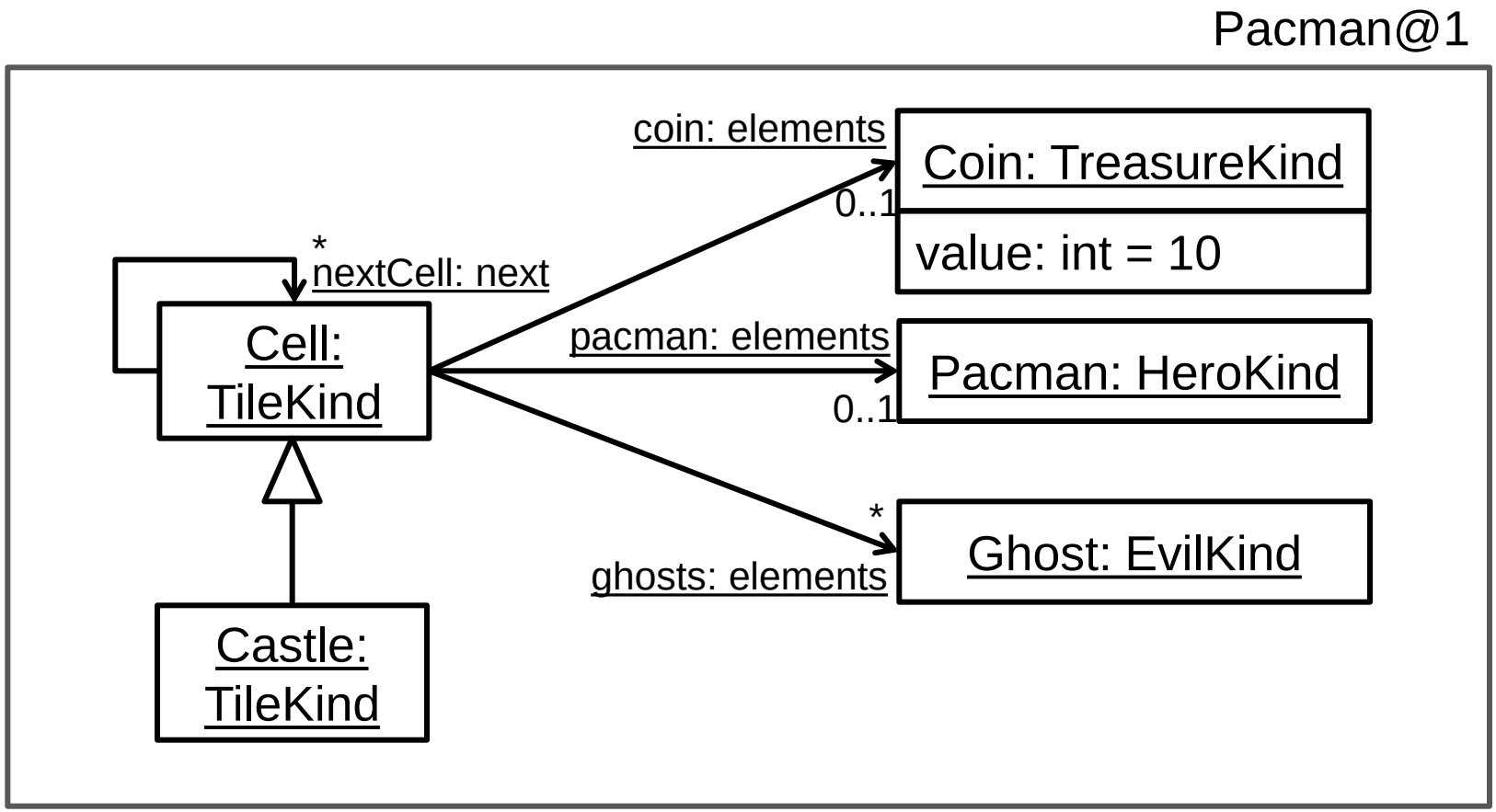
The language should permit describing the initial game configuration

Later on, we will define the generic rule games…

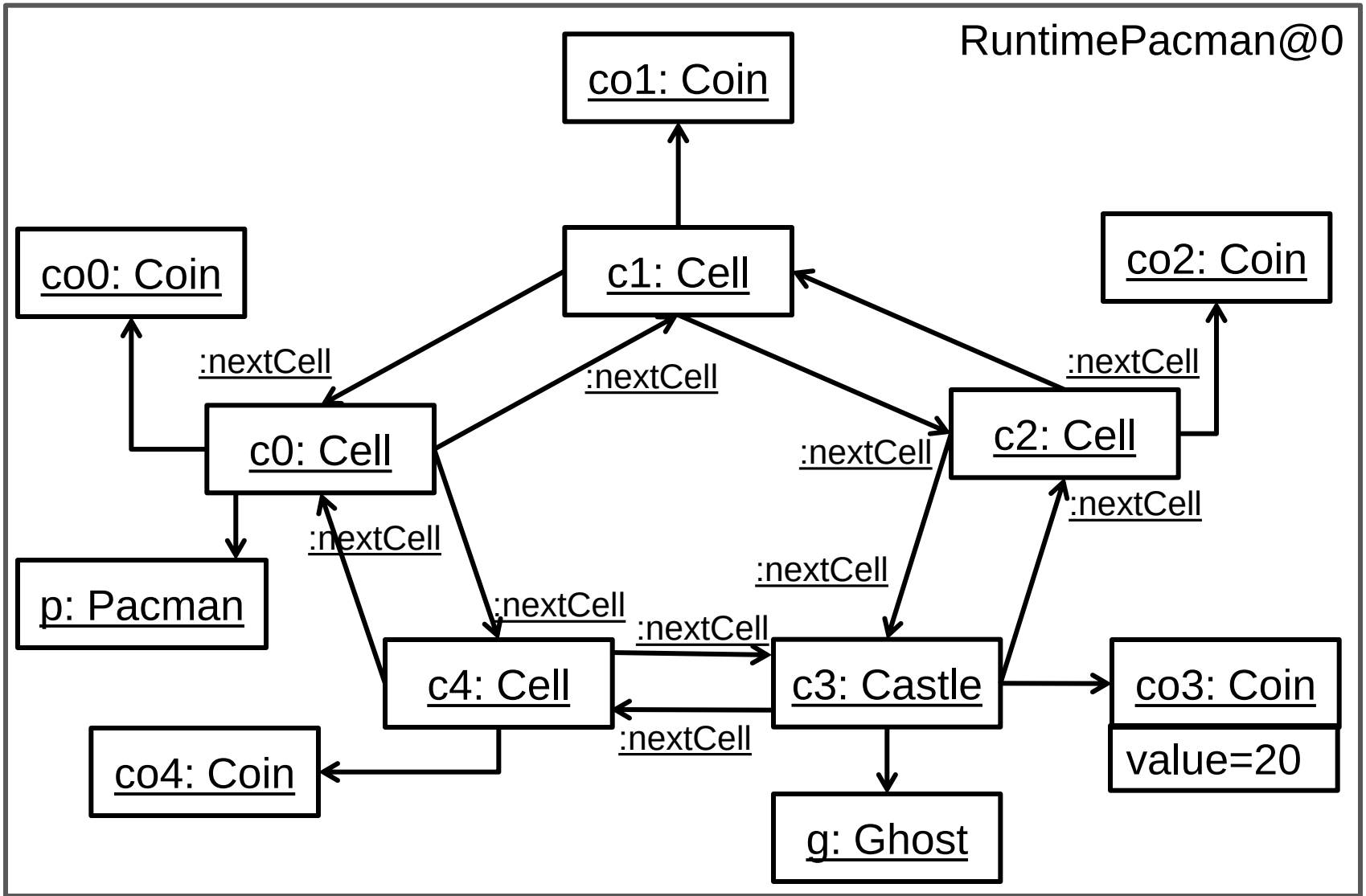
A POSSIBLE SOLUTION



MODELLING THE PACMAN GAME



MODELLING AN INITIAL CONFIGURATION FOR PACMAN



AGENDA

Multi-level modelling: the basics

MetaDepth

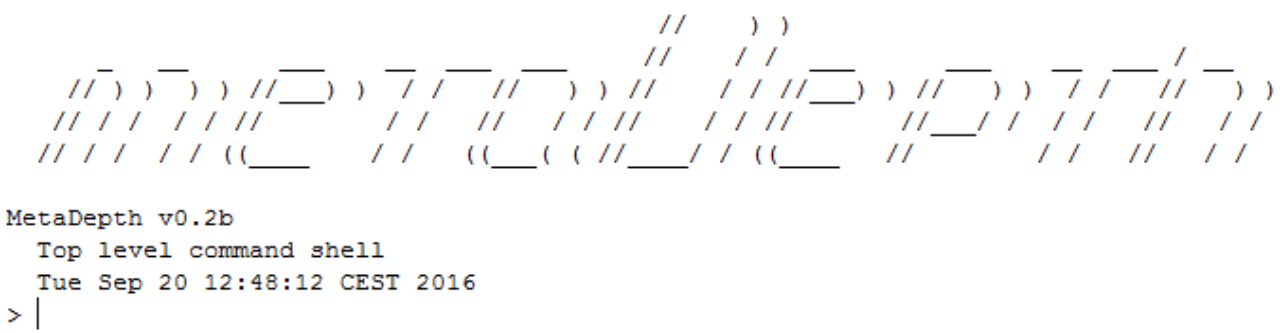
hands-on examples

Multi-level model management

Advanced techniques

When and where to use multi-level modelling

Summary and conclusion



Textual multi-level modelling tool

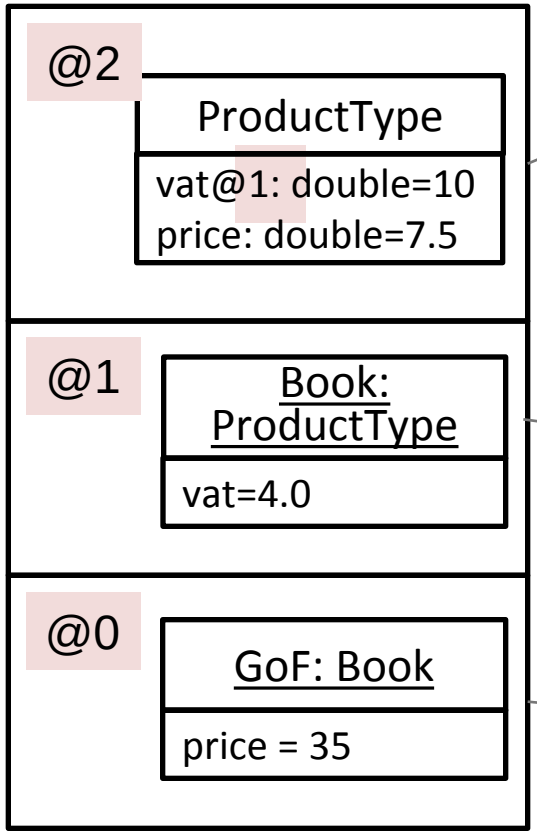
- Started in 2009
- Deep characterization based on clabjects/potency
- Orthogonal Classification Architecture
- http://metaDepth.org

Integrated with the Epsilon Languages for model management

- Constraints in EOL/EVL
- Derived attributes in EOL
- In-place transformations in EOL
- Model-to-Model transformations in ETL
- Code generation in EGL

Juan de Lara, Esther Guerra: Deep Meta-modelling with MetaDepth. TOOLS (48) 2010: 1-20

METADEPTH



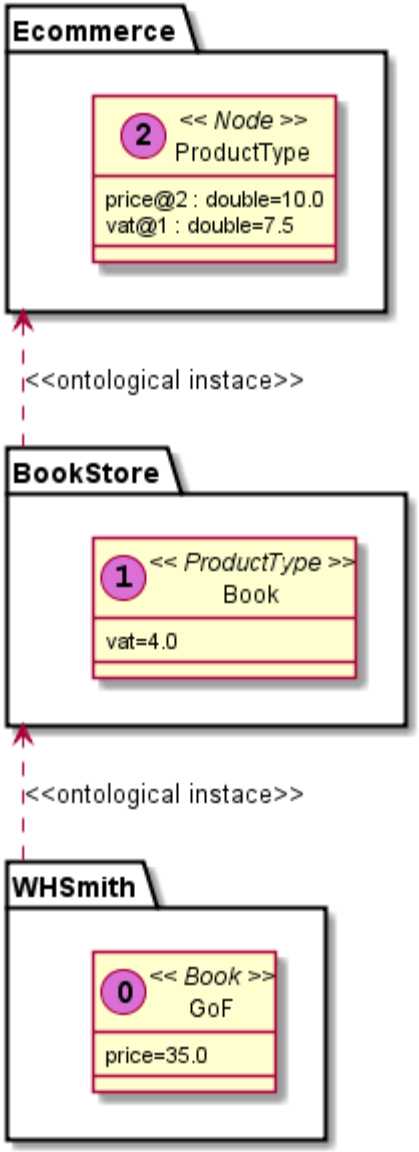

```
Model Ecommerce@2 {
  Node ProductType {
    vat@1 : double=10;
    price : double=7.5;
   }
}
```

```
Ecommerce BookStore{
  ProductType Book { vat = 4.0; }
}
```

```
BookStore WHSmith {
   Book GoF { price = 35; }
}
```

SAMPLE SESSION

```
> load “ProductSimple”

:: loading ../metadepth.samples/src/tutorial/ProductSimp
clabjects created in 0.22 s).

> dump

:: dumping all

… (shows the models in memory)

> compile PlantUML

Executing...

:: compiling all models to PlantUML format,

generated file ./Ecommerce.txt
```



(generated png by PlantUML)

COMMANDS & TIPS

Use set to show all environment variables

You'll need to set DIR to your working folder

```
> set DIR "../metadepth.samples/src/tutorial"
:: setting environment variable DIR
```

Use context to enter in the context of a specific model

```
> context WHSmith
:: entering context WHSmith
```

You can use verify to check all (current model's) constraints

```
> verify
:: Constraints in WHSmith evaluated, (0) violations
```

COMMANDS & TIPS

You can add new elements to a model using the # command

```
> #
:: entering metadepth execution mode
> Book another { price = 45; }
> #
> dump
:: dumping WHSmith
ext BookStore WHSmith{
  ext Book GoF
  {
    price=35.0;
  }
  ext Book another
  {
    price=45.0;
  }
}
```

COMMANDS&TIPS

The # command permits also using any embedded language

Currently EOL, ETL and EGL are embedded

```
> # EOL
:: entering eol execution mode
> new Book;
> ProductType.allInstances().println();
> #
Set {another, MD_3825653c837c4d12b1f21e09c83f2fa9, GoF}
```

(more about “transitive typing” later)

COMMANDS & TIPS

Retyping shell commands is time consuming

You can save your commands in a ".mdc" file

Command files can be executed with the run command

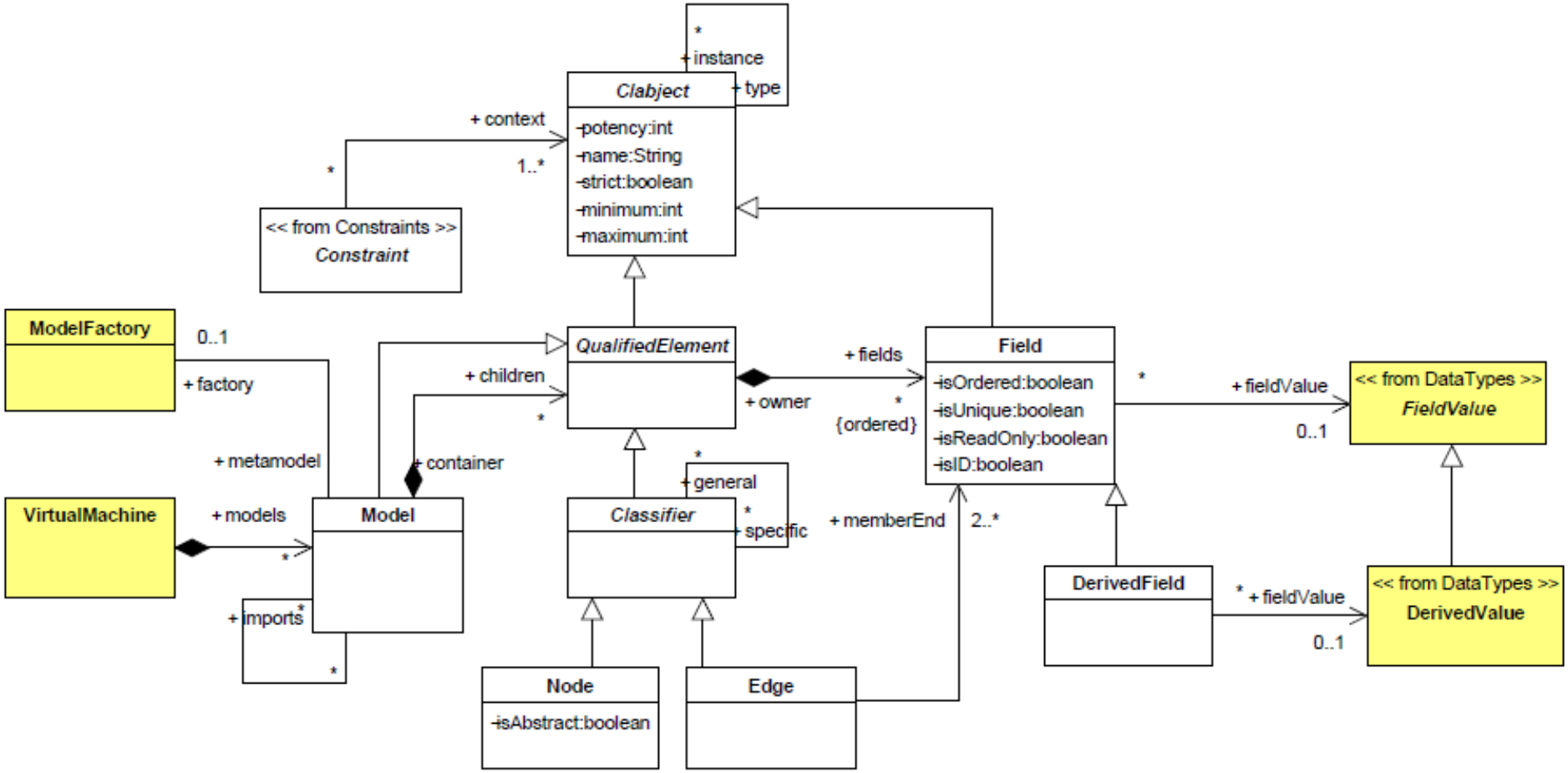
```
run "Product"
```

Files are sought in folder of DIR variable

Product.mdc

```
load "ProductSimple"
dump
context WHSmith
Verify

#
Book JavaBible { price = 89; }
#

# EOL
var t := new Book;
t.price := JavaBible.price;
ProductType.all.println();
#

dump
compile PlantUML
```

LINGUISTIC META-MODEL



USING LINGUISTIC TYPES

It is possible to use either ontological or linguistic types

```
> # EOL
:: entering eol execution mode
> Node.all.println();
> #
Sequence {GoF, another}
```

The linguistic meta-model is transparently available through reflection

```
> # EOL
:: entering eol execution mode
> GoF.type.println();
> #
Book
```

The "^" notation can be used if needed to disambiguate (GoF.^type)

LINGUISTIC EXTENSIONS

```
Model Ecommerce@2 {
  Node ProductType@2 {
    vat@1        : double = 7.5;
    price@2      : double = 10;
   }
}
Ecommerce BookStore{
  ProductType Book {
    vat  = 4.0;
    pages : int;
    authors : Author[*] {ordered, unique};
  }
  Node Author {
    name : String;
  }
}
BookStore WHSmith {
  Book GoF {
    price = 35;
    pages = 395;
    authors = [Gamma, Helm, Johnson, Vlissides];
  }
  Author Gamma     { name = "Eric Gamma"; }
  Author Helm      { name = "Richard Helm"; }
  Author Johnson   { name = "Ralph Johnson"; }
  Author Vlissides { name = "John Vlissides"; }
}
```

FEATURES

Derived fields

- Expressions using the Epsilon Object Language

Constraints and features in models

- Avoids creating an artificial class just to set a global constraint

Nested models, model import

- Models are first-class citizens, unlike in EMF

Customizable textual syntax

Model extension

A-posteriori typing

Concepts (genericity mechanism)

DERIVED FIELDS

```
Model Store@2 {
  Node ProductType{
    vat@1 : double = 7.5;
    price : double = 10;
    /finalPrice@2: double = $self.vat*self.price/100 +self.price$;
  }
}
```

Computation expressions for derived fields using the Epsilon Object Language (EOL)

Derived fields with primitive type, or references

RUNNING EXAMPLE (1)

```
Model ArcadeGame@2 {
  Node TileKind {
    next      : TileKind[*];
    elements  : ElementKind[*];
  }

  abstract Node ElementKind{}

  abstract Node CharacterKind : ElementKind {  }

  Node HeroKind : CharacterKind {
    initLives@1 : int = 3;
    lives : int = 3;
    score : int = 0;
  }
  Node EvilKind : CharacterKind {}
  Node TreasureKind : ElementKind {}
}
```

RUNNING EXAMPLE (2)

```
ArcadeGame PacMan {
  TileKind Cell {
    nextCell : Cell[*] {next};
    coin     : Coin[0..1] {elements};
    pacman   : Pacman[0..1] {elements};
    ghosts   : Ghost[*] {elements};
  }
  TileKind Castle : Cell {}

  TreasureKind Coin {
    value : int=10;
  }

  HeroKind Pacman {
    initLives = 3;
  }

  EvilKind Ghost {}
}
```

RUNNING EXAMPLE (3)

```
PacMan RuntimePacman {
  Cell c0  { nextCell = [c1, c4]; coin = co0; pacman = p; }
  Cell c1  { nextCell = [c2, c0]; coin = co1; }
  Cell c2  { nextCell = [c3, c1]; coin = co2; }
  Castle c3{ nextCell = [c4, c2]; coin = co3; ghosts = g; }
  Cell c4  { nextCell = [c0, c3]; coin = co4;}

  Pacman p {}
  Ghost g {}
  Coin co0 {}
  Coin co1 {}
  Coin co2 {}
  Coin co3 { value = 20; }
  Coin co4 {}
}
```

AGENDA

- Multi-level modelling: the basics
- MetaDepth
- **Multi-level model management**
  - constraints
  - transformations
  - code generation
- Advanced techniques
- When and where to use multi-level modelling
- Summary and conclusion

MULTI-LEVEL MODEL MANAGEMENT

For a multi-level modelling framework to be useful, we need:

- Constraint languages
- Transformation languages supporting
    - Code generation
    - Model-to-model transformation
    - In-place model transformation

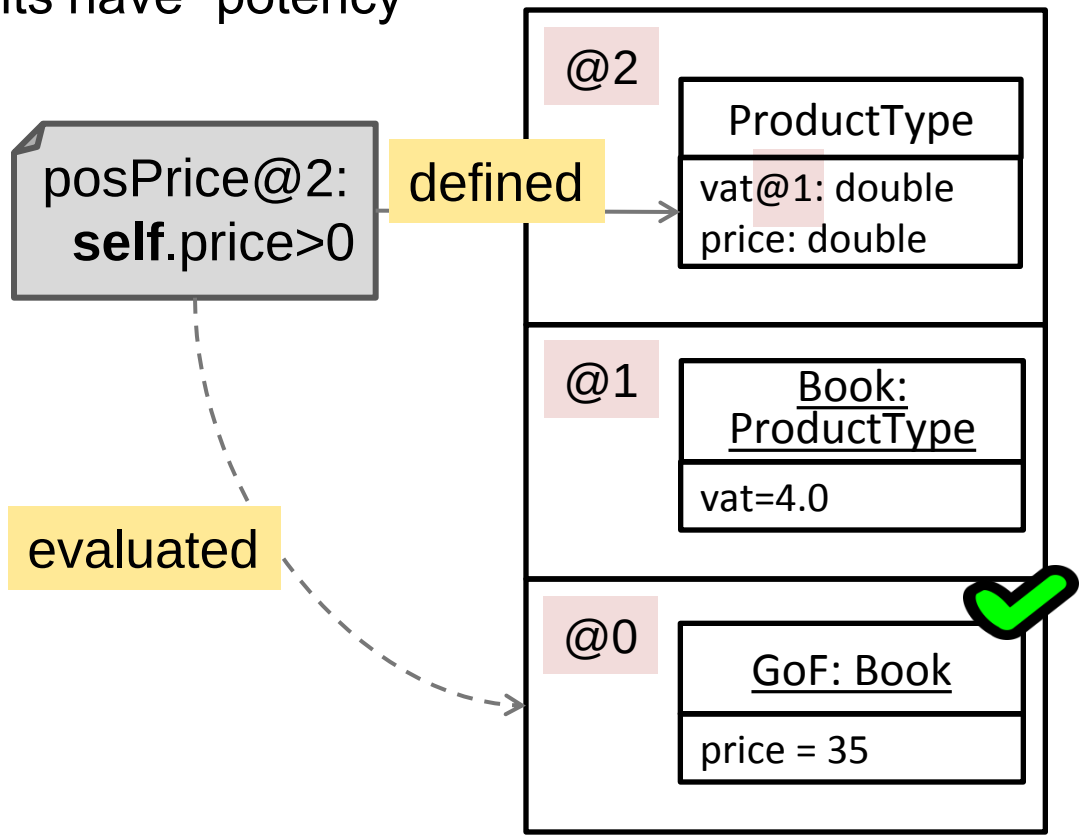
These need to be adapted to a ML setting

In practice, metaDepth has been integrated with the Epsilon languages

DEEP CONSTRAINT LANGUAGE

Level at which the constraint is defined vs level at which we want to evaluate it

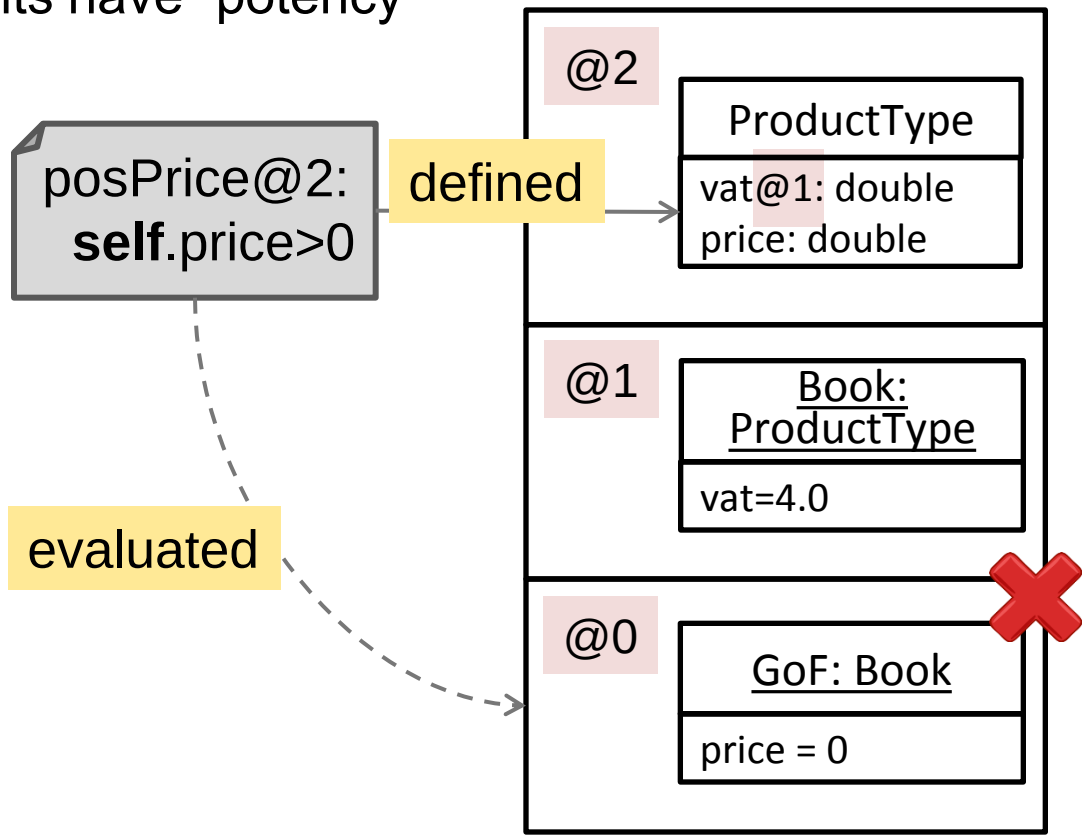
- Constraints have “potency”



DEEP CONSTRAINT LANGUAGE

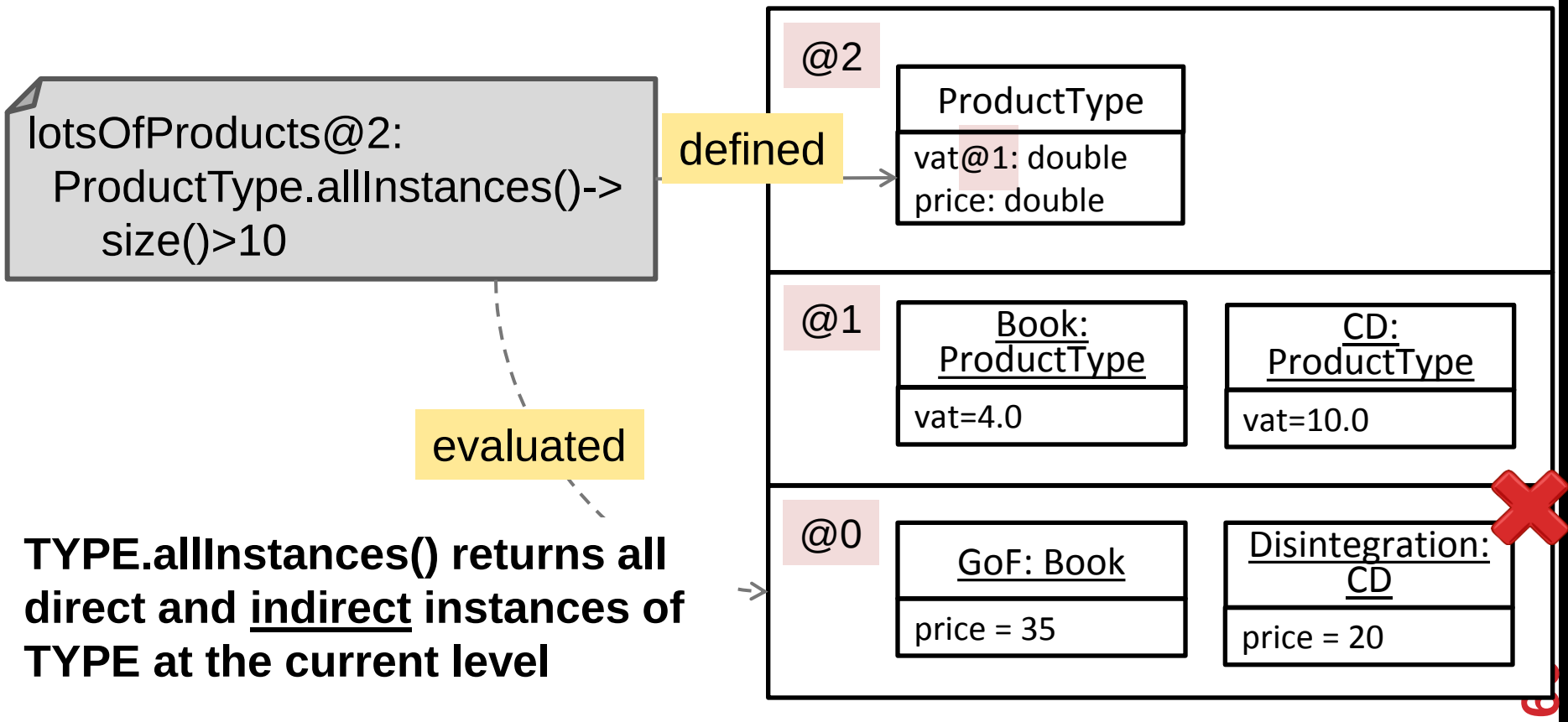
Level at which the constraint is defined vs level at which we want to evaluate it

- Constraints have “potency”



TRANSITIVE TYPING

We might not know type names at intermediate levels



TYPE.allInstances() returns all direct and indirect instances of TYPE at the current level

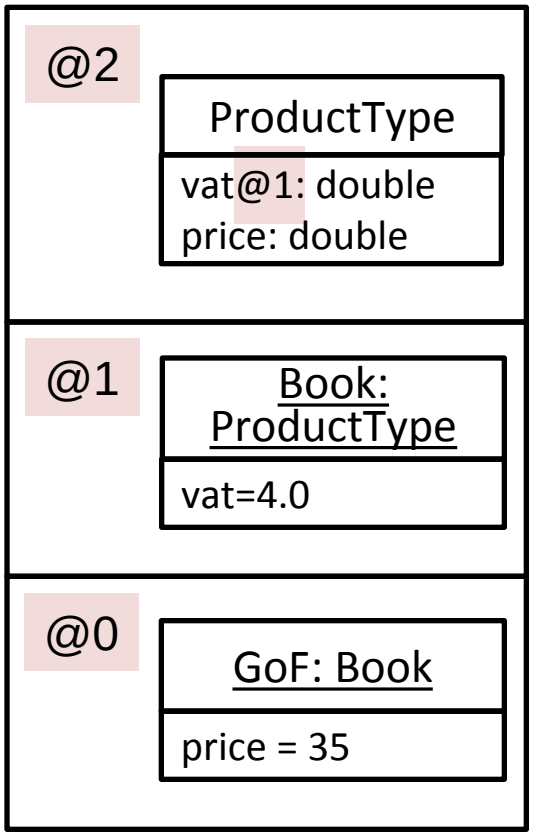
LEVEL NAVIGATION

Make (upwards) meta-level navigation implicit

- At level 0:

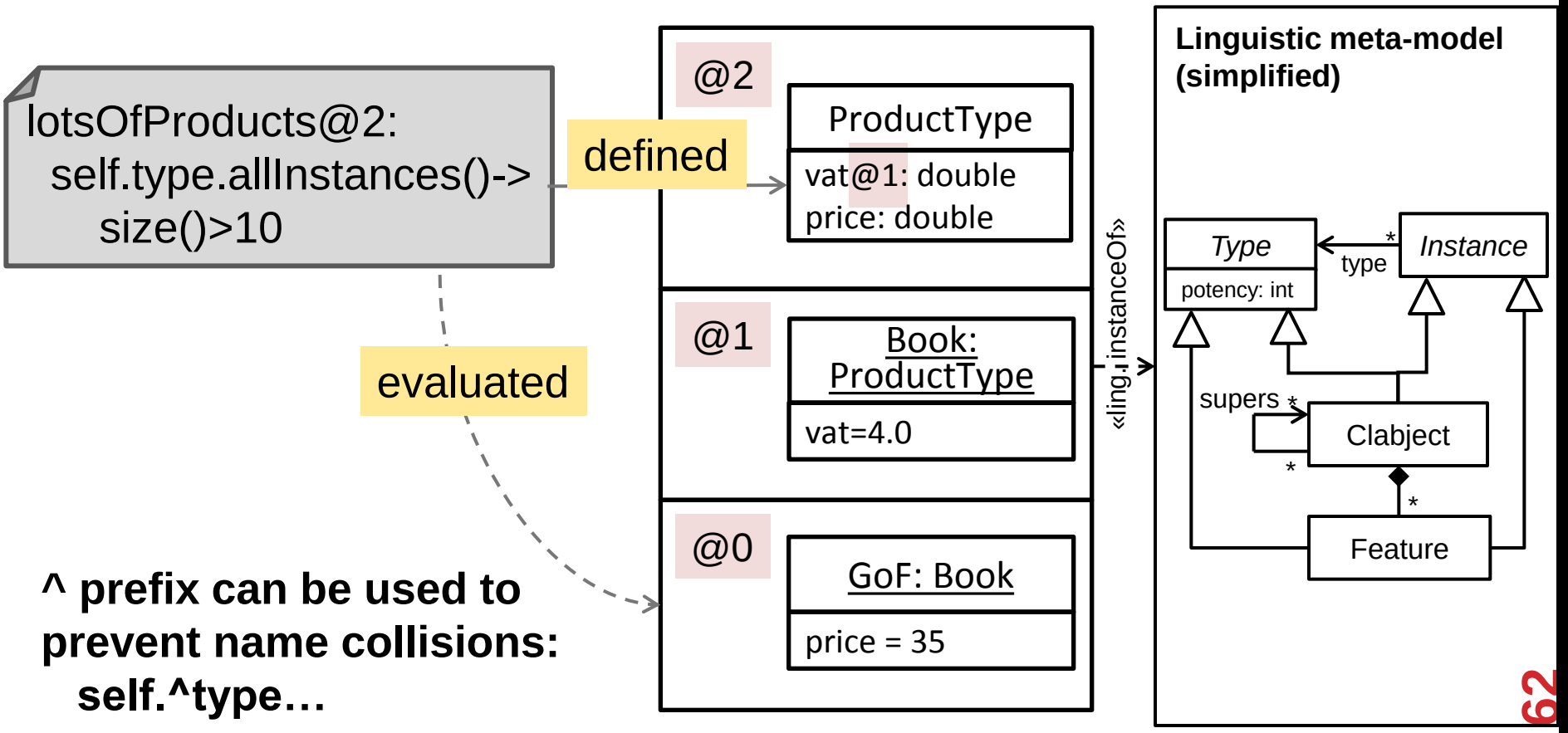
> GoF.vat.println();
> 4.0

Fields with instance facet in clabjects can be accessed in instance clabjects.



LINGUISTIC META-MODEL

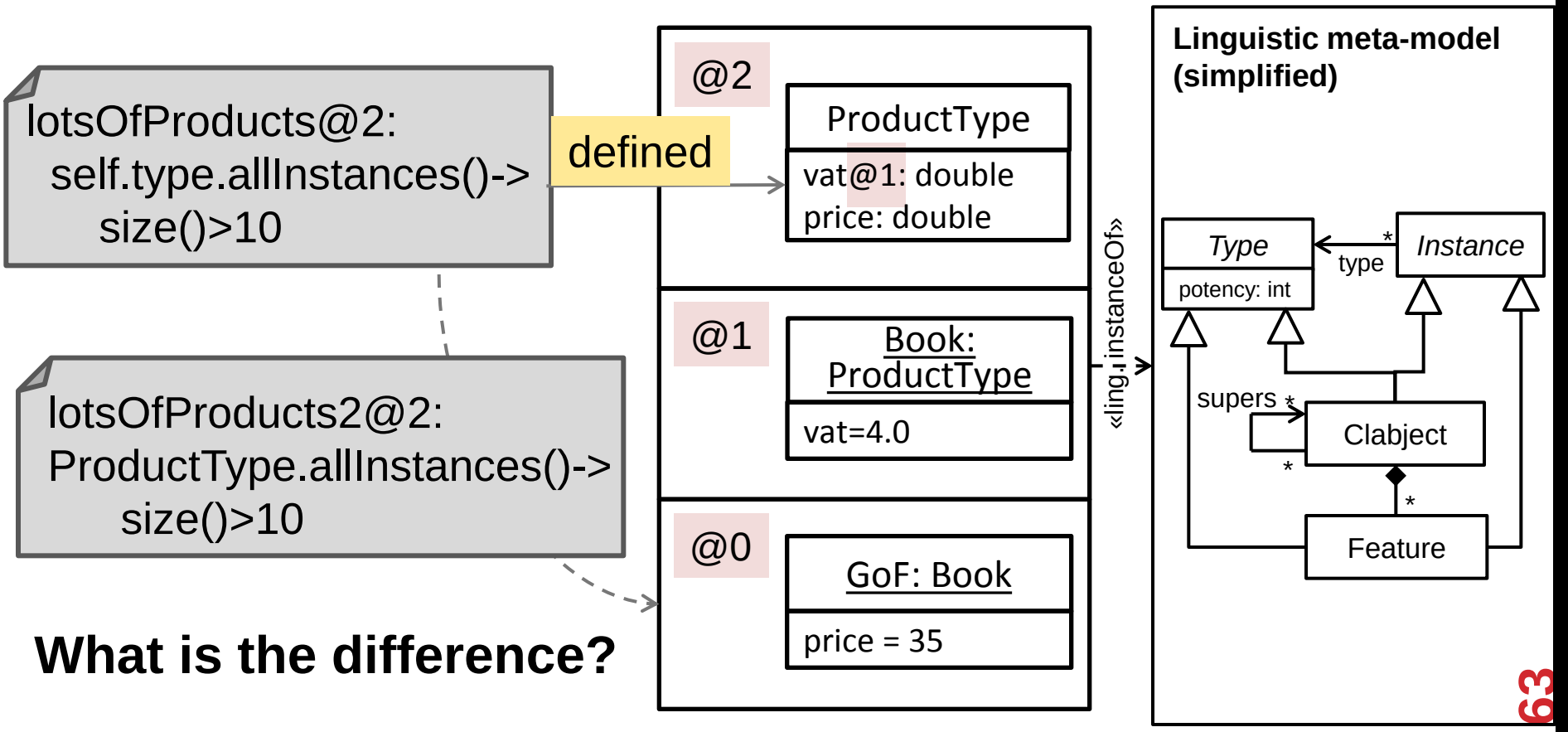
Transparent access to the linguistic meta-model in constraint expressions



^ prefix can be used to prevent name collisions:
self.^type…

LINGUISTIC META-MODEL

Transparent access to the linguistic meta-model in constraint expressions



What is the difference?

METADEPTH

MetaDepth uses the Epsilon Object Language (EOL) to express constraints

- Also for operations and expressions for derived fields

EOL is a variant of OCL

- Permits side effects (assignments, object creation)
- Has imperative constructs (e.g., loops)

The basis of other Epsilon languages

- ETL for model-to-model transformation
- EGL for code generation

EXERCISE

Add constraints to the running example to ensure that:

- There are no isolated tiles at level 0
- Treasures, Heroes and Evils are connected to some tile at level 0
- There is at least one hero at level 0
- There are no less evils than heroes at level 0

- There can be no Pacman in the castle

Add a derived attribute to calculate the number of (direct) reachable tiles at level 0.

SOLUTION (1)

```
Model ArcadeGame@2 {
  Node TileKind {
    next      : TileKind[*];
    elements  : ElementKind[*];

    /numReachable : int = $self.next.size()$;
    noIsolated : $self.numReachable>0 or
                  TileKind.all.exists( t | t.next.includes(self))$
  }

  abstract Node ElementKind{
     connected : $TileKind.all.one( t | t.elements.includes(self))$
  }

  abstract Node CharacterKind : ElementKind {  }

  Node HeroKind : CharacterKind {
    initLives@1 : int = 3;
    lives : int = 3;
    score : int = 0;
  }
  Node EvilKind : CharacterKind {}
  Node TreasureKind : ElementKind {}

  someHero@2  : $HeroKind.all.size()>=1$
  moreEvils@2 : $EvilKind.all.size() >= HeroKind.all.size()$
}
```

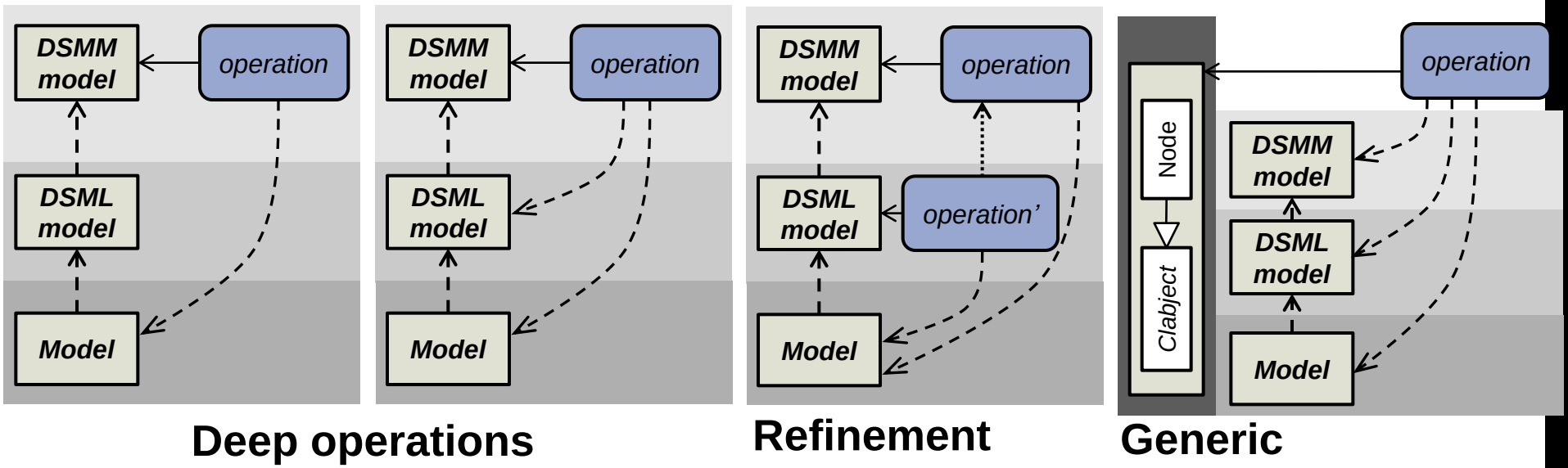
SOLUTION (2)

```
ArcadeGame PacMan {
  TileKind Cell {
    nextCell : Cell[*] {next};
    coin     : Coin[0..1] {elements};
    pacman   : Pacman[0..1] {elements};
    ghosts   : Ghost[*] {elements};
  }

  TileKind Castle : Cell {
    noPacmans : $not self.pacman.isDefined()$
  }

  TreasureKind Coin {
    value : int=10;
  }

  HeroKind Pacman {
    initLives = 3;
  }

  EvilKind Ghost {
  }
}
```

Conflicts and consistency of (ML) constraints can be analyzed:

Towards automating the analysis of integrity constraints in multi-level models. MULTI 2014@MODELS. Esther Guerra, Juan de Lara.

MULTI-LEVEL MODEL MANAGEMENT



Deep operations

Refinement

Generic

Legend

defined on — applicable to - - - - - - redefines ·····

IN-PLACE MODEL TRANSFORMATION

Infrastructure for a family of DSLs

- Operations can be attached to meta-clases
- Operations with potency

Design similar to an object oriented framework

- Common operations
- Operations expected to be overriden at next meta-level

For the running example:

- A game engine for the DSL (autonomous play)
- Core of the engine will be common (basic movement rules, game over and winning condition)
- Some operations can be overriden at level (ie., particular to Pacman)

RUNNING EXAMPLE

```
@metamodel(name=ArcadeGame,file=Arcade.mdepth)
@potency(value=2)
operation main() {
      'Simulating the arcade game'.println();
      var maxStep : Integer := 30;-- to limit infinite behaviours
      var numStep : Integer := 0;

      var hero : HeroKind := HeroKind.all.first();
      hero.~initCell := hero.getCurrentTile();
      for (e in EvilKind.all) {
          e.~initCell := e.getCurrentTile();
      }

      while (numStep<maxStep) {
        ('-----------------------------------------------------').println();
        ('Time : '+numStep).println();
        hero.move();

        if (hero.isWin()) {
           (' ******* Our hero '+hero+' has won! *******').println();
           (' ******* score '+p.score).println();
           (' ******* lives '+p.lives).println();
           return;
        }

        if (checkDamaged(hero)) return;// game over

        for (e in EvilKind.all) {
          e.move();
        }

        if (checkDamaged(hero)) return;// game over

        numStep := numStep+1;
      }
}
```

```
operation checkDamaged(hero : HeroKind) : Boolean { // returns true if game over
   if (hero.isDamaged()) {
      ('  Our hero '+hero+' has been damaged').println();
      hero.die();
      if (hero.hasDied()) {
          ' ******* Game over! *******'.println();
          (' ******* score '+hero.score).println();
          return true;
      } else {
           hero.restart();
           for (e in EvilKind.all) {
              e.restart();
           }
       }
     }
     return false;
}

operation ElementKind getCurrentTile() : TileKind {
   return TileKind.all.select( t | t.elements.includes(self) ).first();
}

operation HeroKind move() {
   var currentTile : TileKind := self.getCurrentTile();
   var toTile : TileKind := currentTile.next.random();
   ('  Moving hero from '+currentTile+' to '+toTile).println();
   currentTile.removeHero(self);
   toTile.addHero(self);
}
…
```

Operation in global context

Operations for instances of ElementKind and HeroKind at level 0

```
operation TileKind removeHero(h : HeroKind) {
   ('    Removing hero '+h+' from '+self).println();
}

operation TileKind addHero(h : HeroKind) {
   ('    Adding hero '+h+' to '+self).println();
}

operation TileKind removeEvil(g : EvilKind) {
   ('    Removing evil '+g+' from '+self).println();
}

operation TileKind addEvil(g : EvilKind) {
   ('    Adding evil '+g+' to '+self).println();
}
```

"Hot spots": typically to be overriden at the next meta-level

Cannot provide "true" code for adding objects, since we do not know the name of the reference

These operations should be overriden at the next meta-level

```
// -------------------------------------------------------
// Specific to Pacman....
// -------------------------------------------------------
operation Cell removeHero(h : HeroKind) {
   self.eatCoin(h);
   ('    Removing pacman '+h+' from cell '+self).println();
   self.pacman:=null;
}

operation Cell eatCoin(h : HeroKind) {
  if (self.coin.isDefined()) {
      h.score := h.score + self.coin.value;
      delete self.coin;
      self.coin := null;
      ('    Pacman score = '+h.score).println();
  }
}

operation Cell addHero(h : HeroKind) {
   ('    Adding pacman '+h+' to cell '+self).println();
   self.pacman:= h;
   self.eatCoin(h);
}
operation Cell removeEvil(g : EvilKind) {
   ('    Removing ghost '+g+' from cell '+self).println();
   self.ghosts.remove(g);
}
operation Cell addEvil(g : EvilKind) {
   ('    Adding ghost '+g+' to cell '+self).println();
   self.ghosts.add(g);
}
```

MODEL-TO-MODEL TRANSFORMATION

Many scenarios:

- Deep transformations (eg. defined at @2, applied at @0)
- Co-transformations (eg. defined at @2, applied at @1 and @0)
- Refining transformations (overriding rules at @1)
- Reflective & linguistic transformations (generic, using linguistic types)
- Multi-level target (eg creating models at @1 and @0)

For the running example:

- A (deep) transformation into Petri nets
- Reusable for any concrete adventure game

J. de Lara, E. Guerra, J. Sánchez Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. SoSyM 14(1): 429-459 (2015)

MODEL-TO-MODEL TRANSFORMATION

```
@metamodel(name=ArcadeGame,file=Arcade.mdepth,domain=source)
@metamodel(name=PetriNet,file=PetriNet.mdepth,domain=target)
rule CharacterToToken
  transform h : SLevel0!CharacterKind
  to        p : Target!Token {
     p.^name := h.^type+'_'+h.^name;
  }

rule TileToPlace
  transform t : SLevel0!TileKind
  to        p : Target!Place  {
    p.name := t.type.name+'_'+t.^name;
    p.^name := t.type.name+'_'+t.^name;
    p.tokens ::= t.elements;// implicit call to equivalents()
    for ( n in t.next) {
      var tr := new Target!Transition;
      tr.name := t.^name+' to '+n.^name;
      tr.^name := t.^name+'_to_'+n.^name;
      p.outTr := tr;
      n.equivalents().first().inTr := tr;
    }
}
```

CODE GENERATION

Visualization of the game

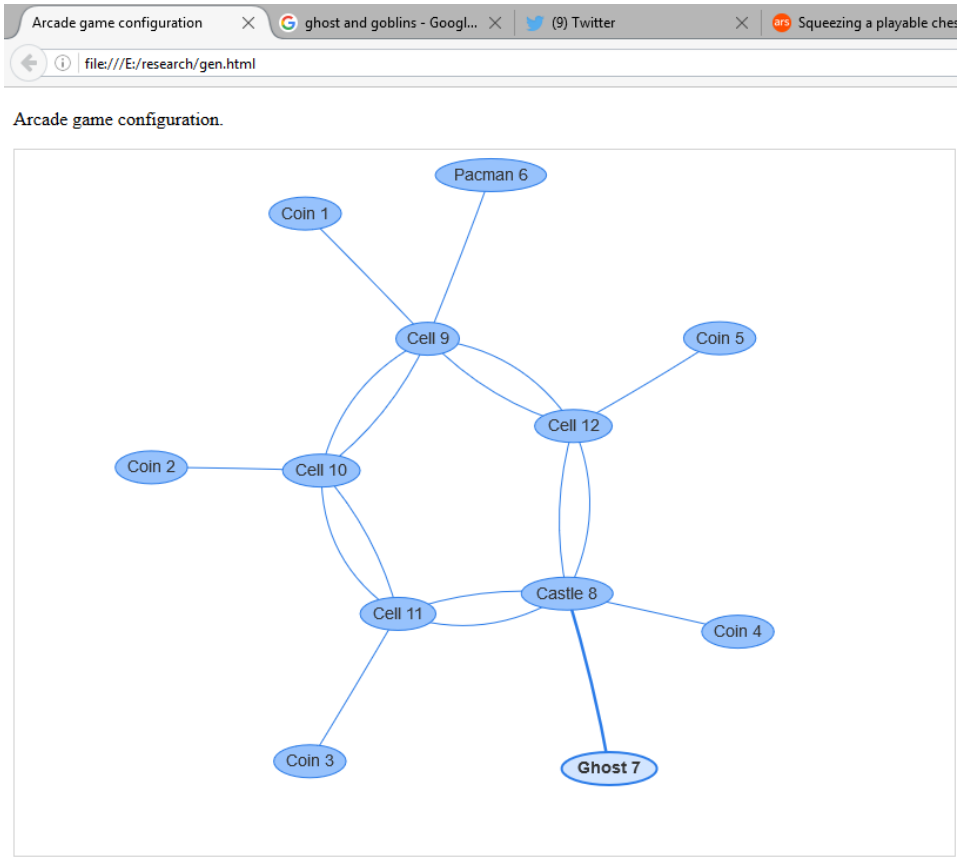
- vis.js (http://visjs.org)

Code generator

- Deep (reusable for the DSL)
- Generic (linguistic types)

```
<!doctype html>
<html>
<head>
…
<script type="text/javascript">
  // create an array with nodes
  var nodes = new vis.DataSet([
  [%  var counter : Integer := 0;
      // Generic
      for(t in Level0!Node.all) {
        counter := counter+1;
  %]
  [%if (counter > 1) {%], [%}%]
    {id: [%=counter%],
     label: '[%=t.type%] [%=counter%]'}
    [% t.~identif := counter; %]
  [%}%]
  ]);
…
</html>
```

GENERATED CODE




```
<script type="text/javascript">
  // create an array with nodes
  var nodes = new vis.DataSet([
    {id: 1, label: 'Coin 1'}
,     {id: 2, label: 'Coin 2'}
,     {id: 3, label: 'Coin 3'}
,     {id: 4, label: 'Coin 4'}
,     {id: 5, label: 'Coin 5'}
,     {id: 6, label: 'Pacman 6'}
,     {id: 7, label: 'Ghost 7'}
,     {id: 8, label: 'Castle 8'}
,     {id: 9, label: 'Cell 9'}
,     {id: 10, label: 'Cell 10'}
,     {id: 11, label: 'Cell 11'}
,     {id: 12, label: 'Cell 12'}
  ]);
```

AGENDA

Multi-level modelling: the basics

MetaDepth

Multi-level model management

Advanced techniques

- splitting models
- leap potency
- deep references

When and where to use multi-level modelling

Summary and conclusion

SPLITTING MODELS

Having all elements in one model does not scale

- Models created to “annotate” other ones

Facility for model import

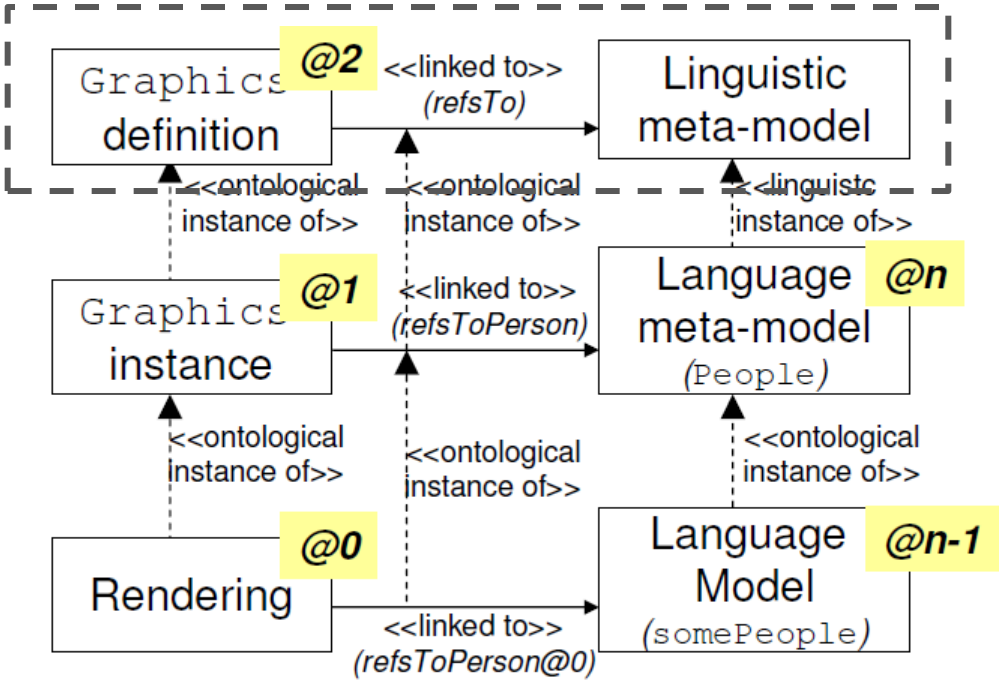
- Makes accesible the model elements of other models

Level mismatches

- Importing and imported models do not need to have the same level
    - Often reconciled through deep references

EXAMPLE: DEFINING A CONCRETE SYNTAX

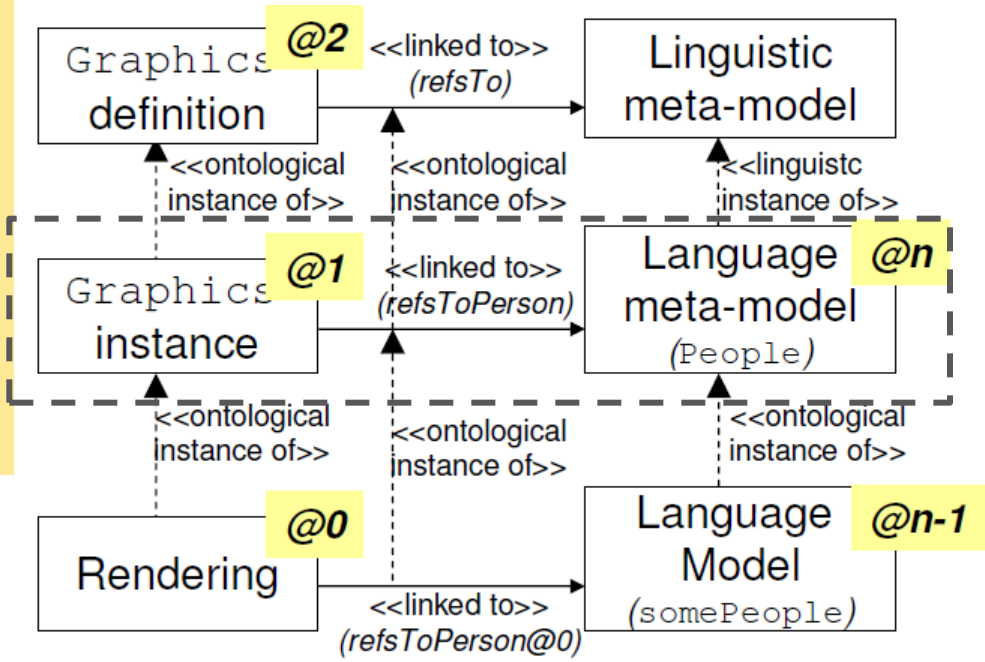
```
Model Graphics@2 {
  abstract Node Figure {
     x: int;
     y: int;
     rotation : double = 0;
     scale : double = 1;
     refsTo : Node;
   }

   Node Rectangle : Figure {
      width@1 : int;
      height@1 : int;
   }
}
```



EXAMPLE: DEFINING A CONCRETE SYNTAX

```
Graphics PeopDiag
  imports People{
    Rectangle PersonRep {
      width = 6;
      height = 6;
      refsToPerson : Person {refsTo};
    }
}
```

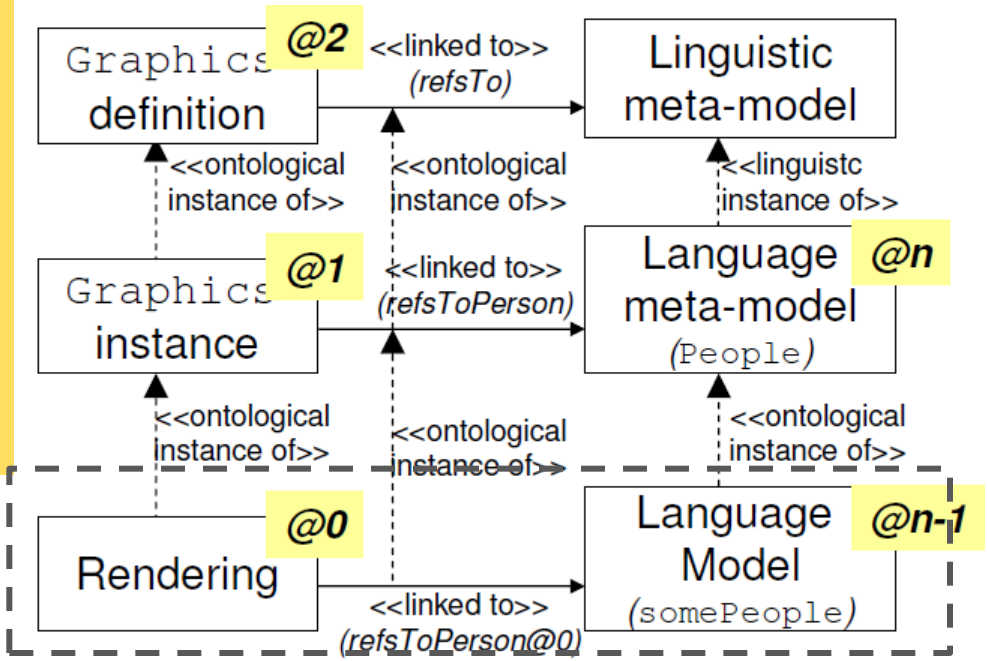
```
Model People{
  Node Person {
      name : String;
      age : int;
   }
}
```



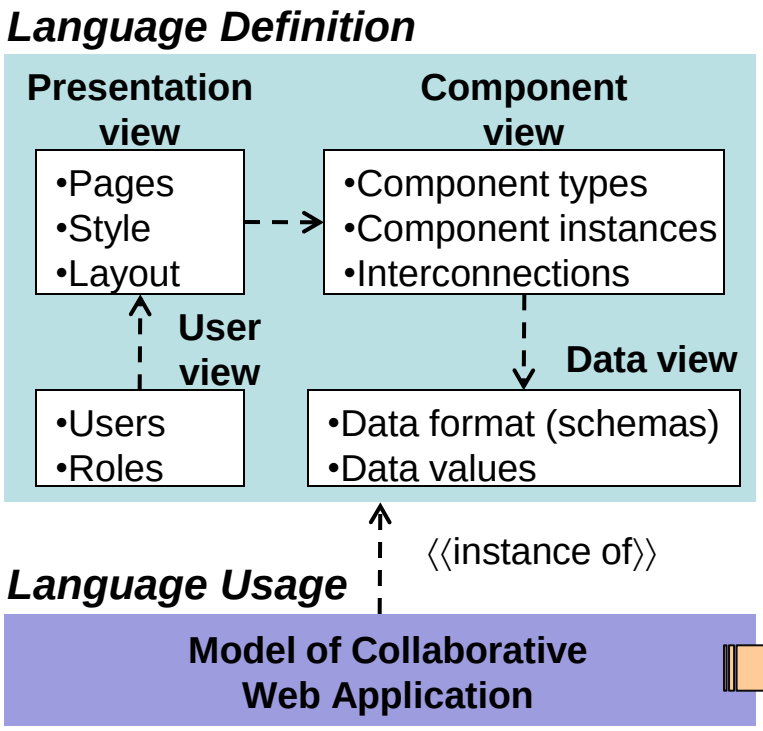
EXAMPLE: DEFINING A CONCRETE SYNTAX

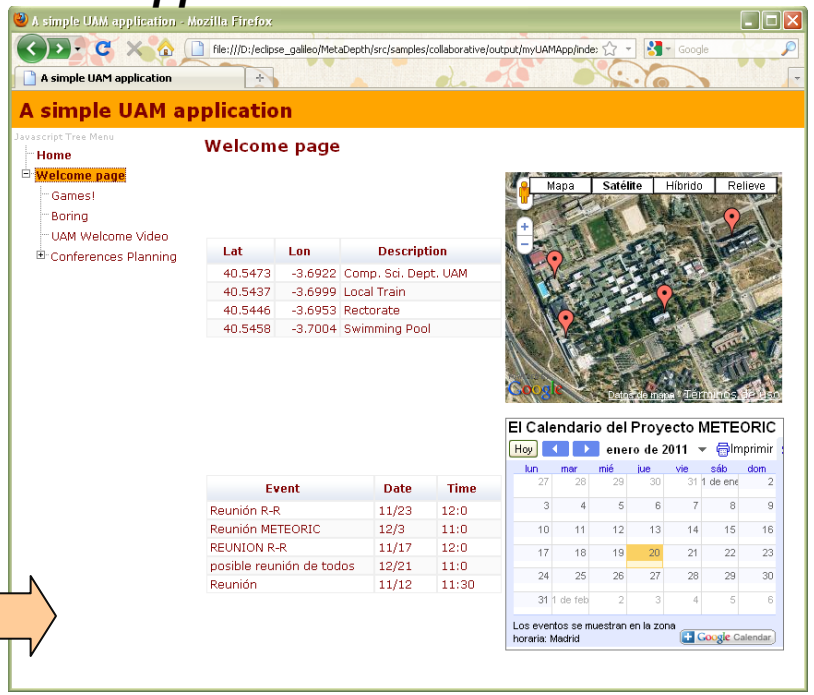
```
PeopDiag somePeopleDiag
   imports somePeople
{
  PersonRep p1r {
    x = 10; y = 10;
    refsToPerson = p1;
   }
}
```

```
People somePeople{
   Person p1 {
      name : String;
      age : int;
   }
}
```



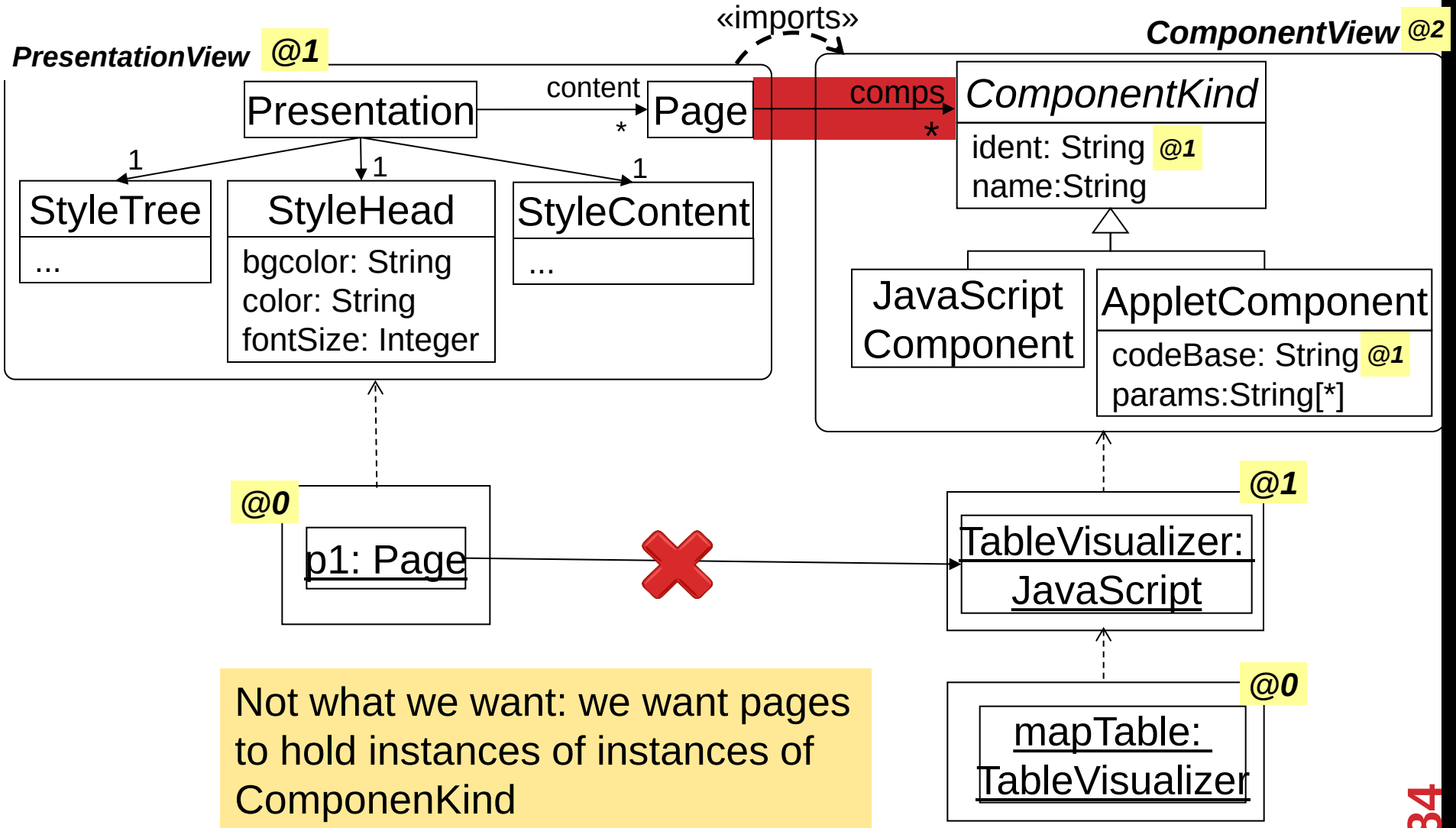
EXAMPLE: COMPONENT-BASED COLLABORATIVE WEB APPLICATIONS





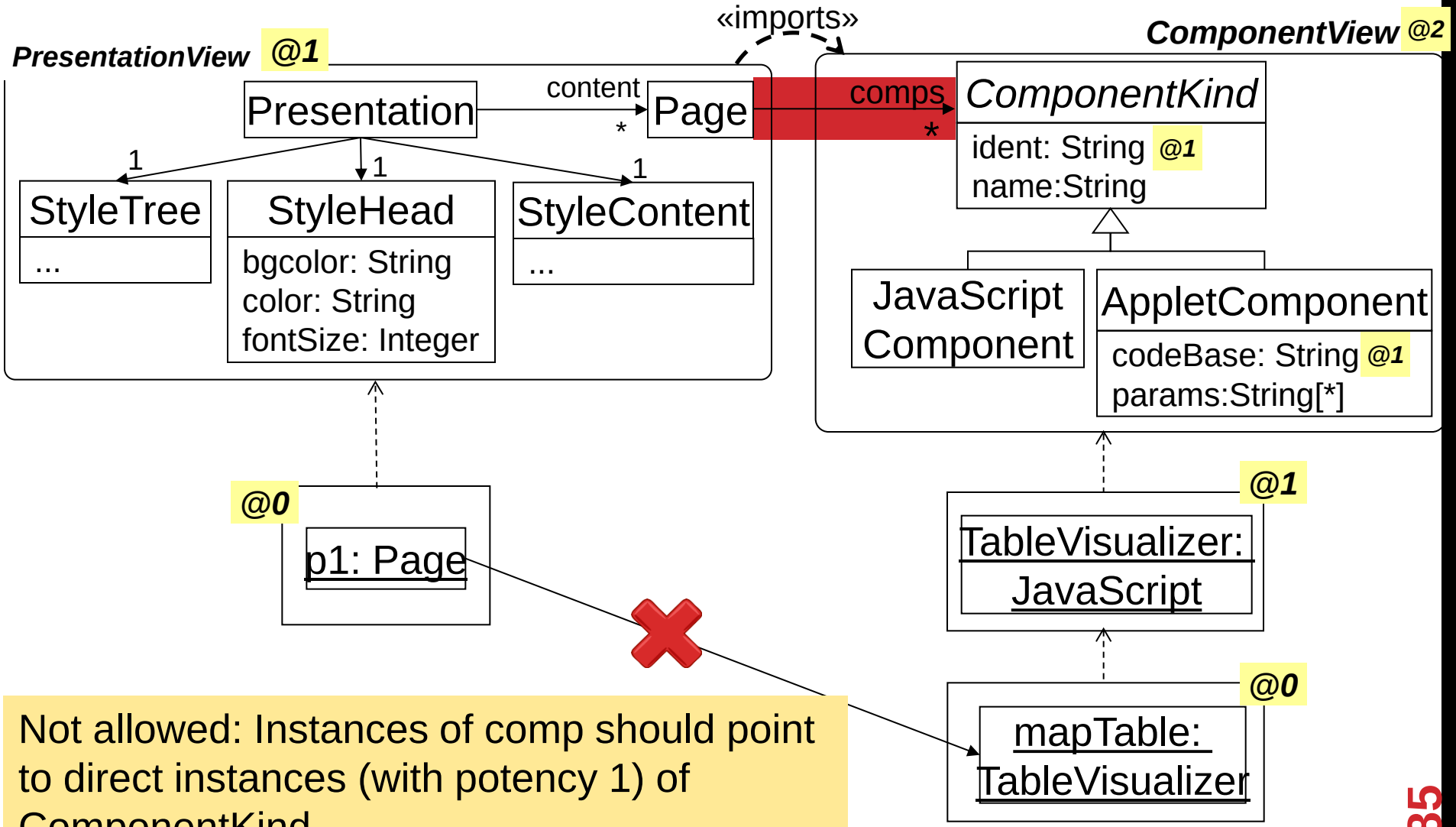
Juan de Lara, Esther Guerra, Ruth Cobos, Jaime Moreno-Llorena: Extending Deep Meta-Modelling for Practical Model-Driven Engineering. Comput. J. 57(1): 36-58 (2014)

SPLITTING MODELS AND DEEP REFERENCES



Not what we want: we want pages to hold instances of instances of ComponenKind

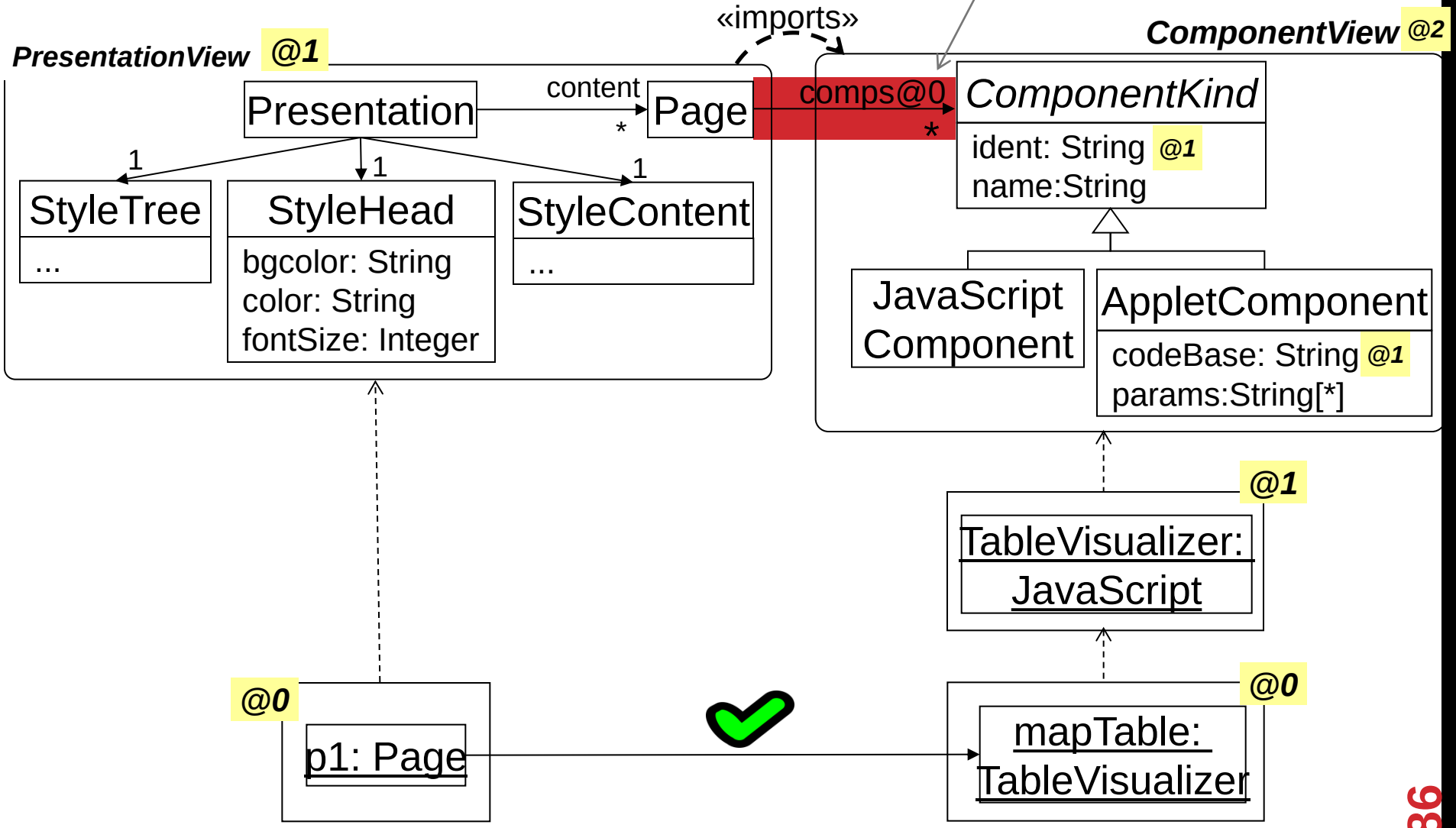
SPLITTING MODELS AND DEEP REFERENCES



Not allowed: Instances of comp should point to direct instances (with potency 1) of ComponentKind

SPLITTING MODELS AND DEEP REFERENCES

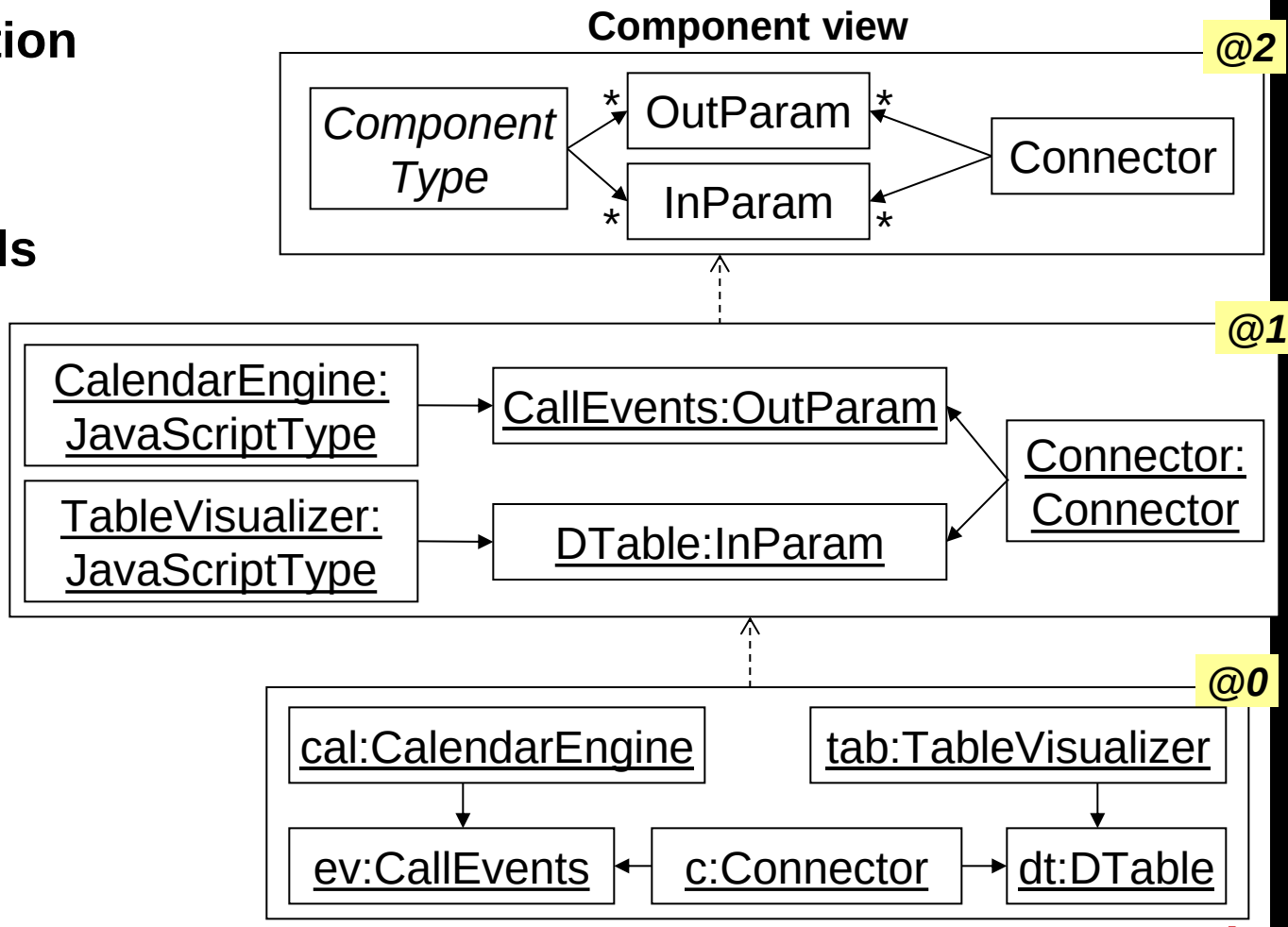
Deep reference: a reference to a clabject with certain potency



LEAP POTENCY

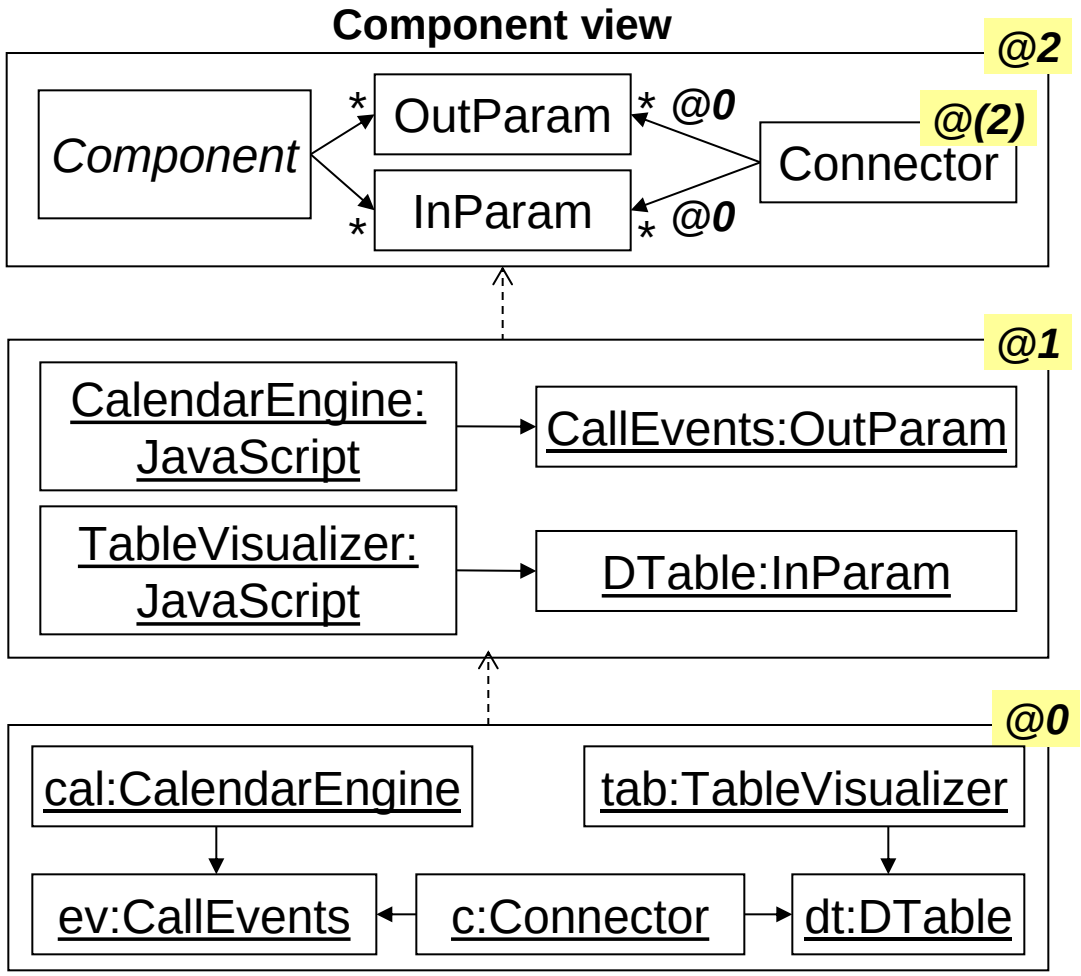
Identity instantiation

Elements instantiated at intermediate levels with no extra information

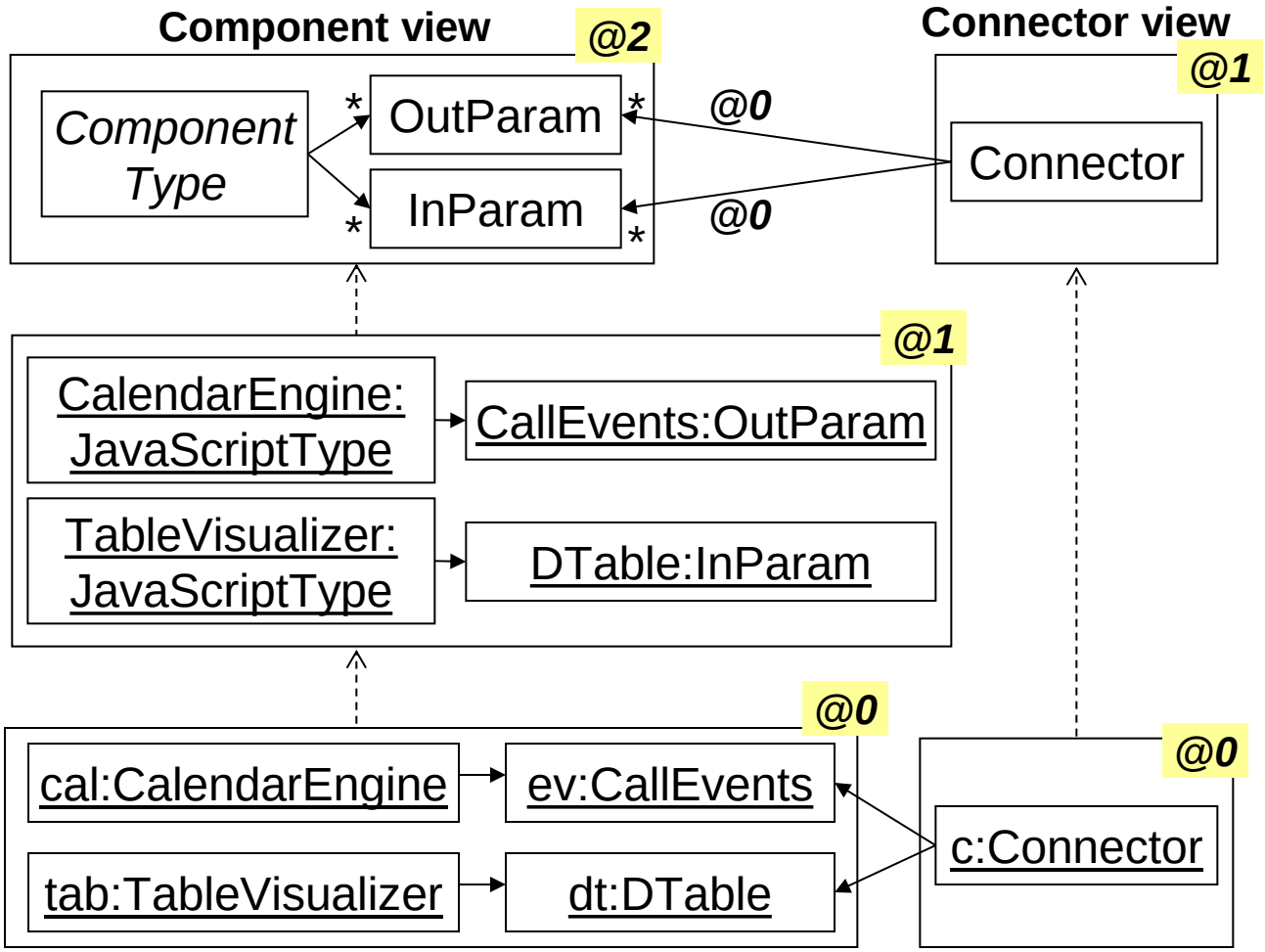


LEAP POTENCY

“Syntactic sugar” to avoid identity instantiation



LEAP POTENCY AS MODEL SPLITTING



RUNNING EXAMPLE

Assume we do not need to define types of tiles

- All tiles are the same: same behaviour and connectivity

We have a "potency mismatch"

- We only want tile instances, not types (ie., potency 1)
- We still want to define character types (ie., potency 2)

SOLUTION 1: LEAP POTENCY

```
Model ArcadeSimple@2 { // we are not interested in defining types of tiles
  Node Tile@(2) {      // leap potency (otherwise identity instantiation)
    next      : Tile[*];
    elements  : ElementKind@0[*]; // deep reference
  }

  abstract Node ElementKind{
  }

  abstract Node CharacterKind : ElementKind {
  }

  Node HeroKind : CharacterKind {
    initLives@1 : int = 3;
    lives : int = 3;
    score : int = 0;
  }
  Node EvilKind : CharacterKind {}
  Node TreasureKind : ElementKind {}
}
```

SOLUTION 1: LEAP POTENCY

```
ArcadeSimple PacMan {
  TreasureKind Coin {
    value : int=10;
  }

  HeroKind Pacman {
    initLives = 5;
  }

  EvilKind Ghost {
  }
}
```

```
PacMan RuntimePacman {
    Tile c0  { next = [c1, c4]; elements = [co0, p]; }
    Tile c1  { next = [c2, c0]; elements = [co1]; }
    Tile c2  { next = [c3, c1]; elements = [co2]; }
    Tile c3  { next = [c4, c2]; elements = [co3, g]; }
    Tile c4  { next = [c0, c3]; elements = [co4];}

    Pacman p {}
    Ghost g {}
    Coin co0 {}
    Coin co1 {}
    Coin co2 {}
    Coin co3 { value = 20; }
    Coin co4 {}
}
```

SOLUTION 2: MODEL SPLIT

Make explicit the two concerns of the DSL:

- Board layout
- Character definition

```
Model ArcadeCharacters@2 {
  abstract Node ElementKind{}

  abstract Node CharacterKind : ElementKind {}

  Node HeroKind : CharacterKind {
    initLives@1 : int = 3;
    lives : int = 3;
    score : int = 0;
  }
  Node EvilKind : CharacterKind {}
  Node TreasureKind : ElementKind {}
}
```

```
ArcadeCharacters PacmanCharacters {
  TreasureKind Coin { value : int=10;}
  HeroKind Pacman { initLives = 5; }
  EvilKind Ghost {}
}
```

```
PacmanCharacters PacmanRuntime {
  Pacman p {}
  Ghost g {}
  Coin co0 {}
  …
}
```

SOLUTION 2: MODEL SPLIT

Make explicit the two concerns of the DSL:

- Board layout
- Character definition

```
Model ArcadeBoard imports ArcadeCharacters {
  // Not interested in defining types of tiles
  Node Tile {
    next      : Tile[*];
    elements  : ElementKind@0[*];
  }
}

ArcadeBoard PacmanBoard imports PacmanRuntime {
  Tile c0  { next = [c1, c4]; elements = [co0, p]; }
  Tile c1  { next = [c2, c0]; elements = [co1]; }
  Tile c2  { next = [c3, c1]; elements = [co2]; }
  Tile c3  { next = [c4, c2]; elements = [co3, g]; }
  Tile c4  { next = [c0, c3]; elements = [co4];}
}
```

AGENDA

- Multi-level modelling: the basics
- MetaDepth
- Multi-level model management
- Advanced techniques
- When and where to use multi-level modelling
  - Multi-level patterns
  - How frequent are these ML “smells”?
- Summary and conclusion

MULTI-LEVEL PATTERNS

Patterns signalling that a multi-level solution could be beneficial

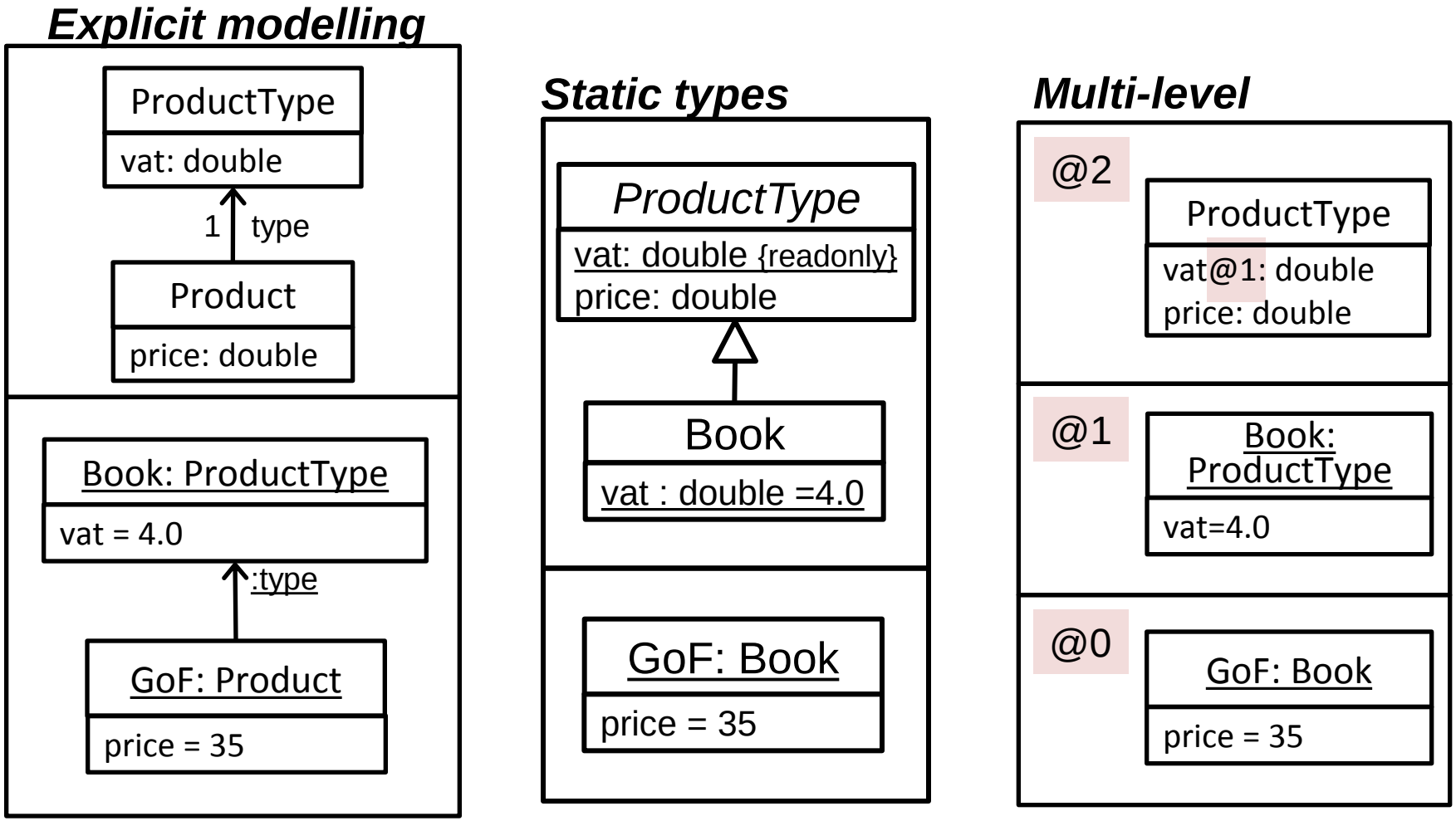
Emulate a meta-modelling facility within a meta-model:

- **Type-object**: typing and instantiation
- **Dynamic features**: feature definition and instantiation
- **Dynamic auxiliary domain concepts**: class definition and instantiation
- **Relation configurator**: Reference cardinality and its semantics
- **Element classification**: Inheritance

In multi-level these facilities are native at every meta-level

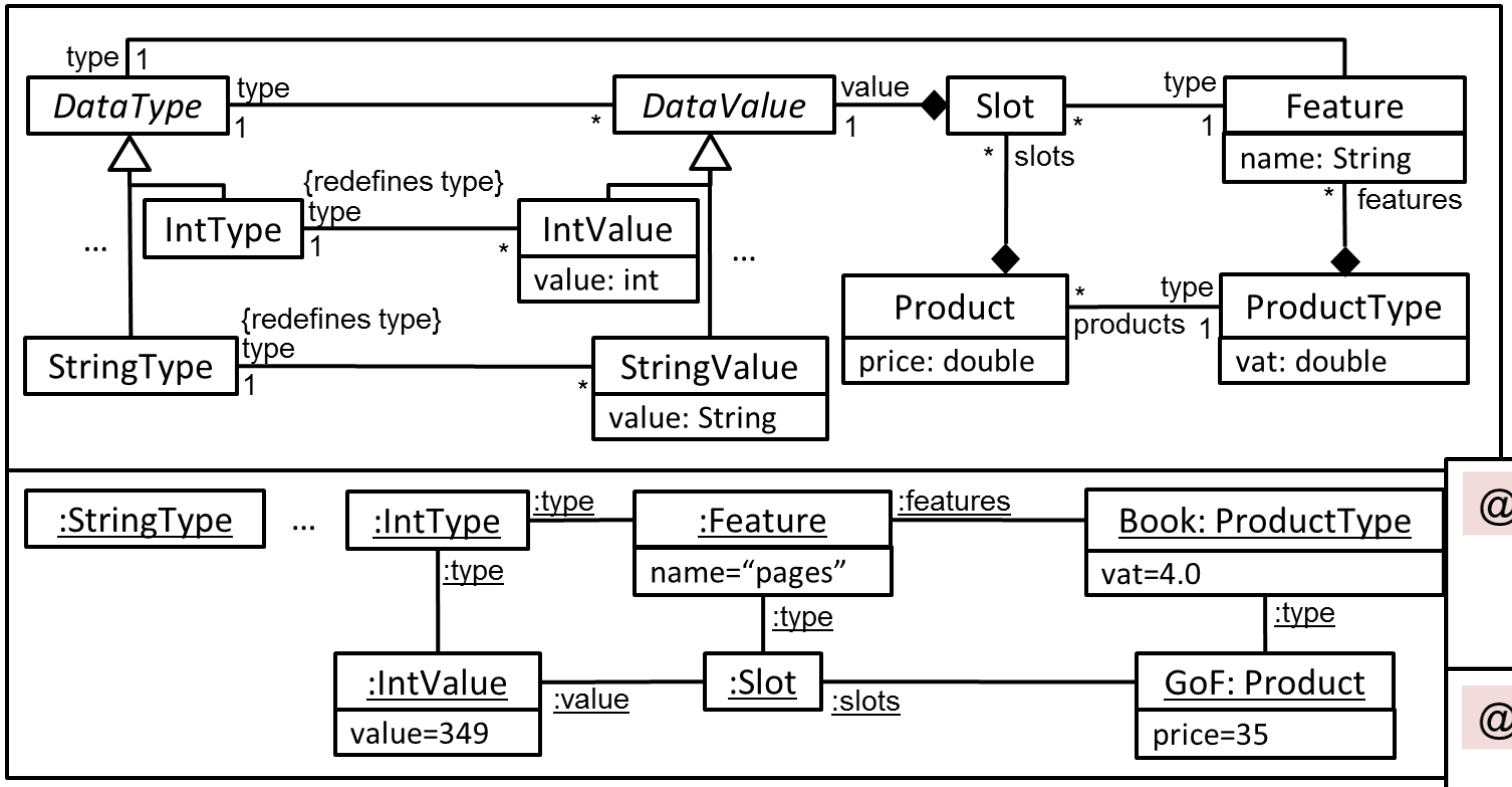
J. de Lara, E. Guerra, J. Sánchez Cuadrado: When and How to Use Multilevel Modelling. ACM Trans. Softw. Eng. Methodol. 24(2): 12:1-12:46 (2014)

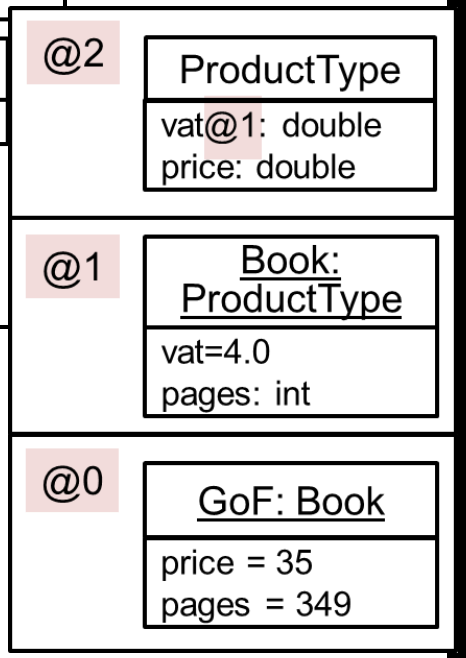
TYPE-OBJECT



Dynamic addition of new types, and instantiation of these

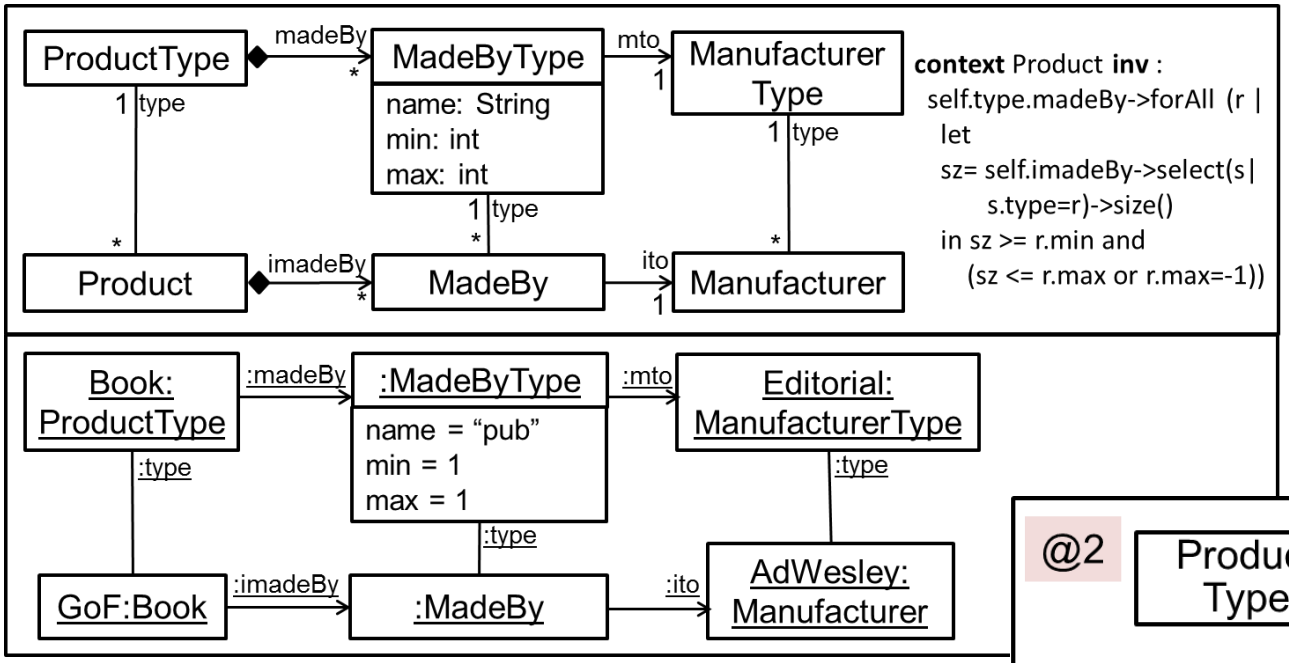
DYNAMIC FEATURES

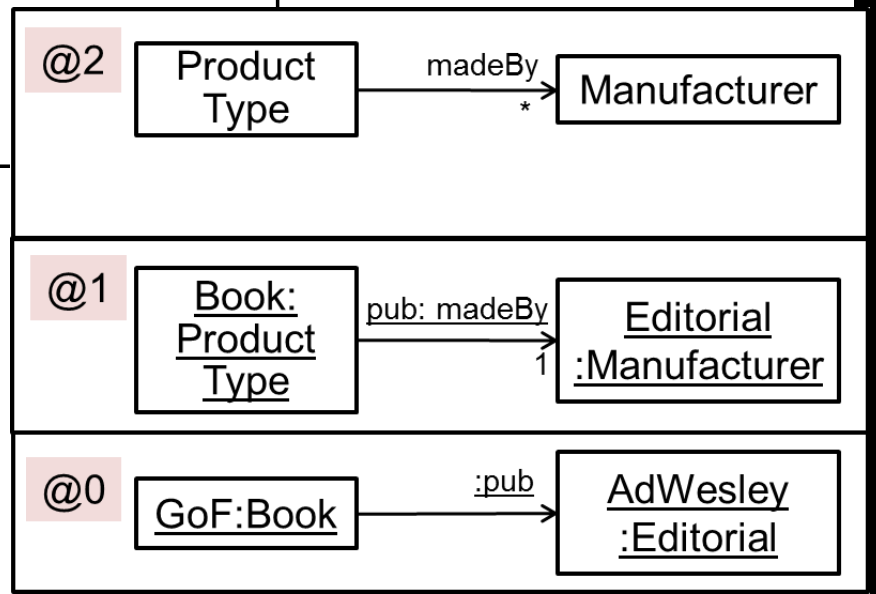




Dynamic addition of new features to a type, and corresponding slots to their instances

RELATION CONFIGURATOR

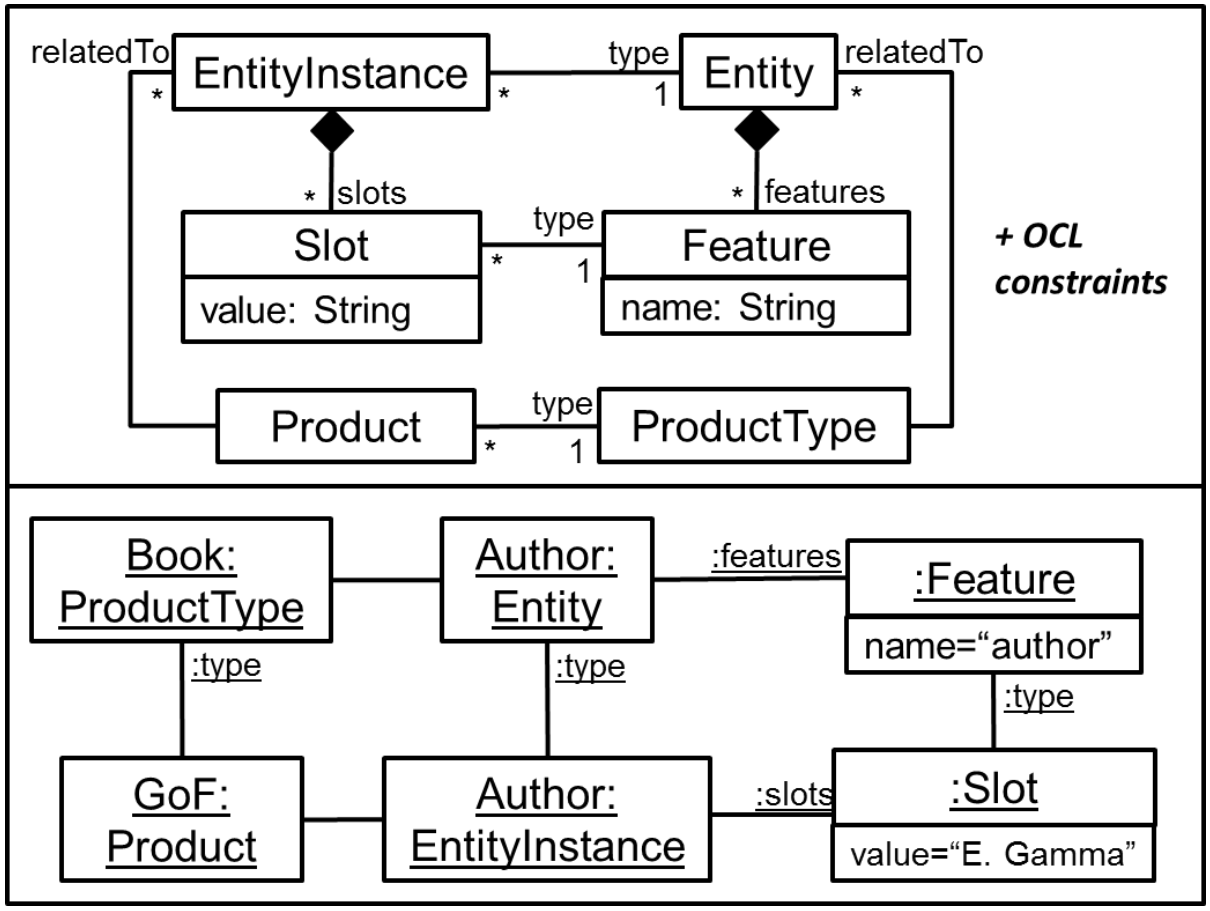


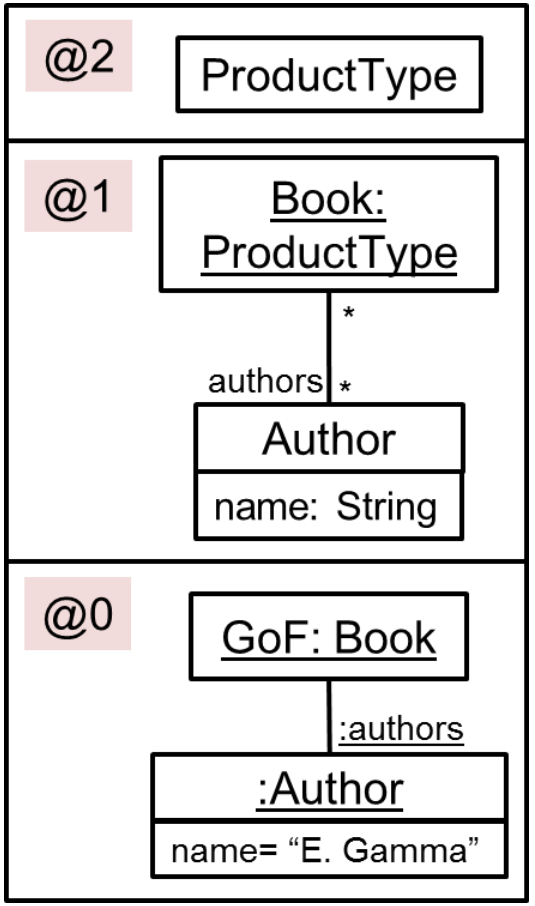


Configuration of a reference type dynamically created

Instantiation of the reference type according to that configuration.

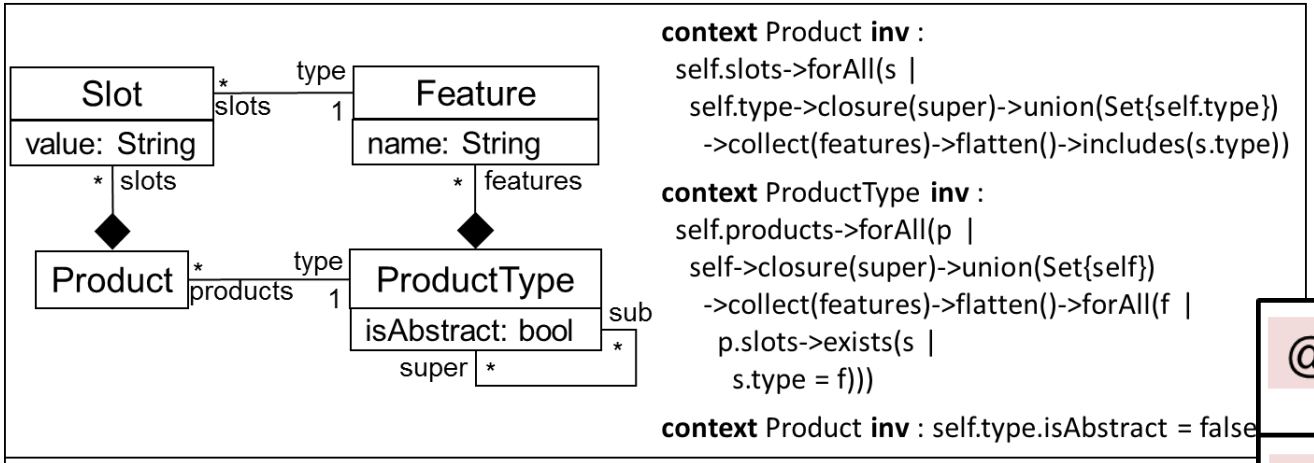
DYNAMIC AUXILIARY DOMAIN CONCEPTS

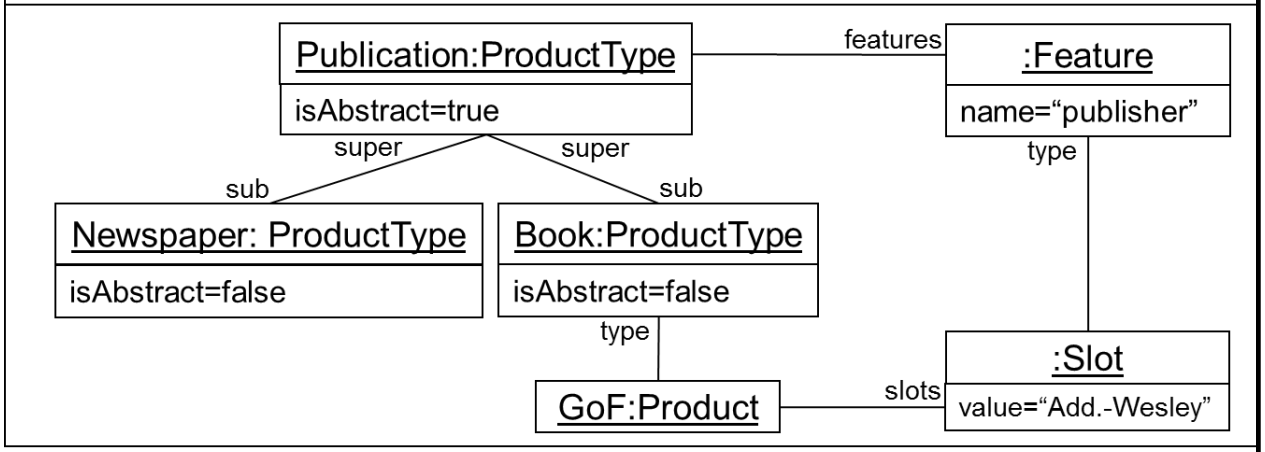


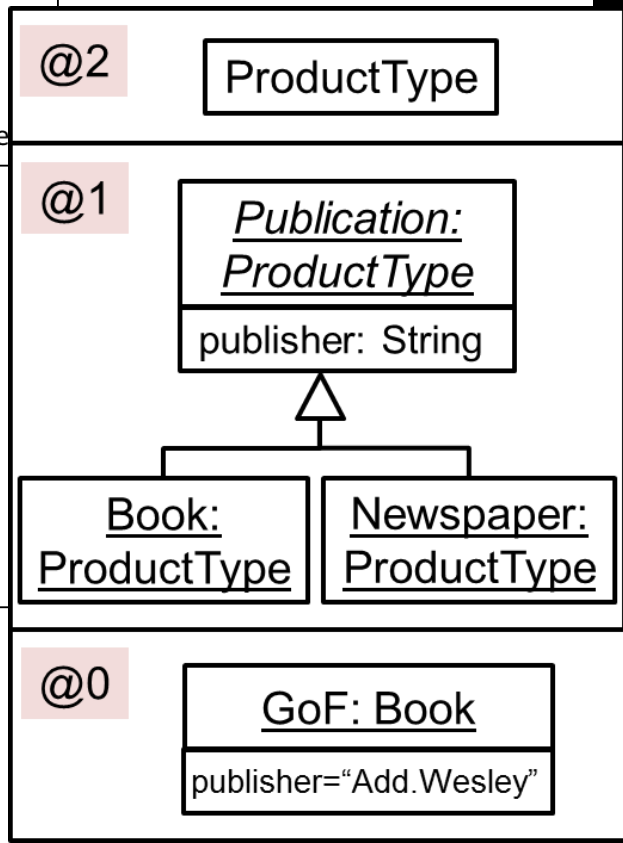


Dynamic addition of new entities related to a type, as well as the instantiation of those entities

ELEMENT CLASSIFICATION







Organization of dynamically created elements along subtyping and inheritance hierarchies.

PRACTICAL APPLICABILITY

How often do these patterns occur?

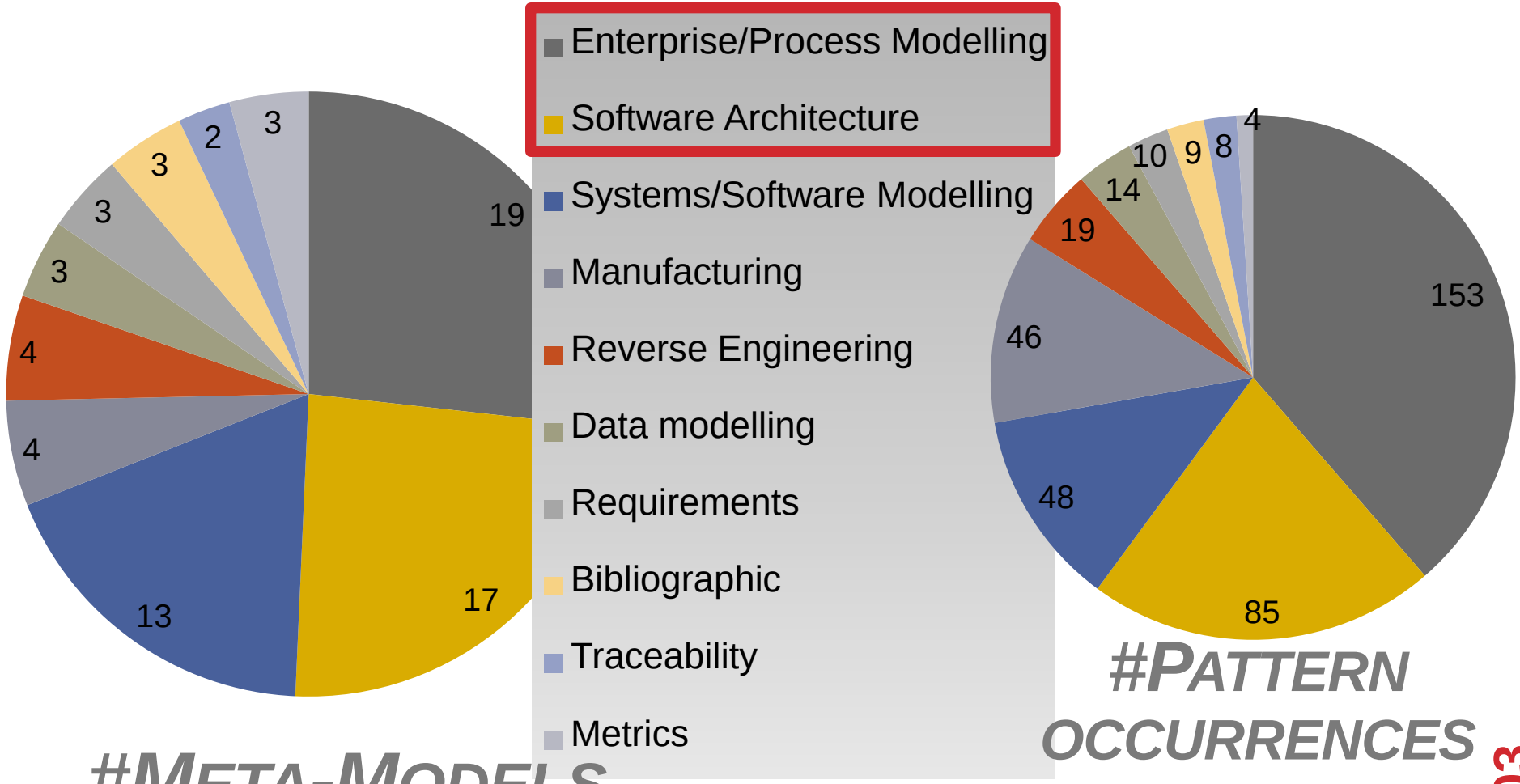
Analysis of meta-model repositories:

- ATL meta-model zoo (305 meta-models) [8.5% occur.]
- OMG specifications (116 specs) [35%~38% occur.]
- ReMoDD (15 meta-models) [20% occur.]
- Articles in journals/conferences

84 meta-models with 459 occurrences (avg 5.5)

- Most common pattern is the ***type-object***
- Most used solution is ***explicit types*** (~70%)

WHERE?



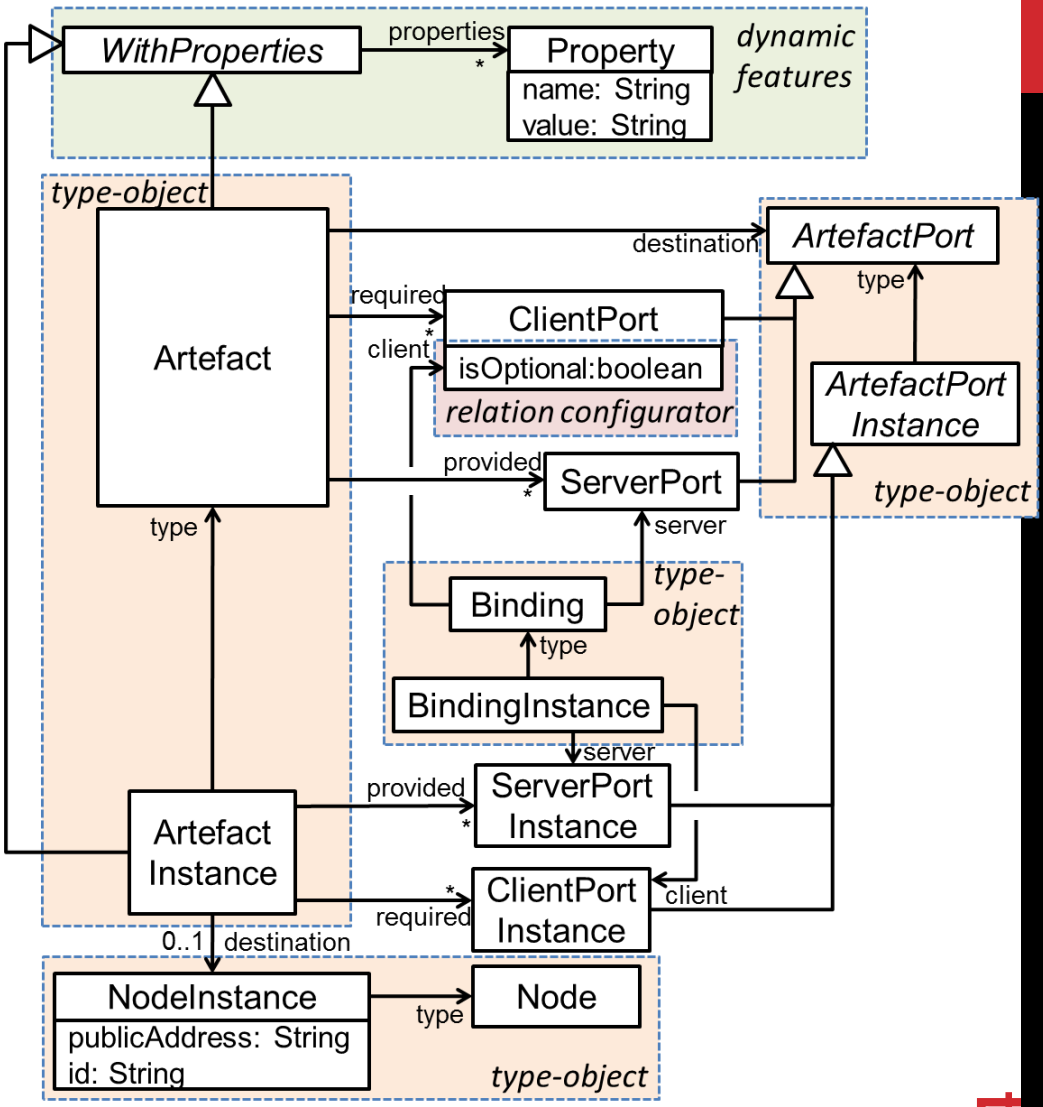
#META-MODELS

#PATTERN OCCURRENCES

WHERE?

Software architecture, components and services

- Component types and instances, port types and instances
- Mostly explicit types solution



Example: CloudML

WHERE?

Business process/enterprise modelling

- Base meta-model customized for different enterprises or project types (OrganizationType/Organization, ProjectType/Project, etc)
- Process modelling languages (TaskType/Task)
- Powertypes, explicit modelling for implementation
- Many occurrences per meta-model

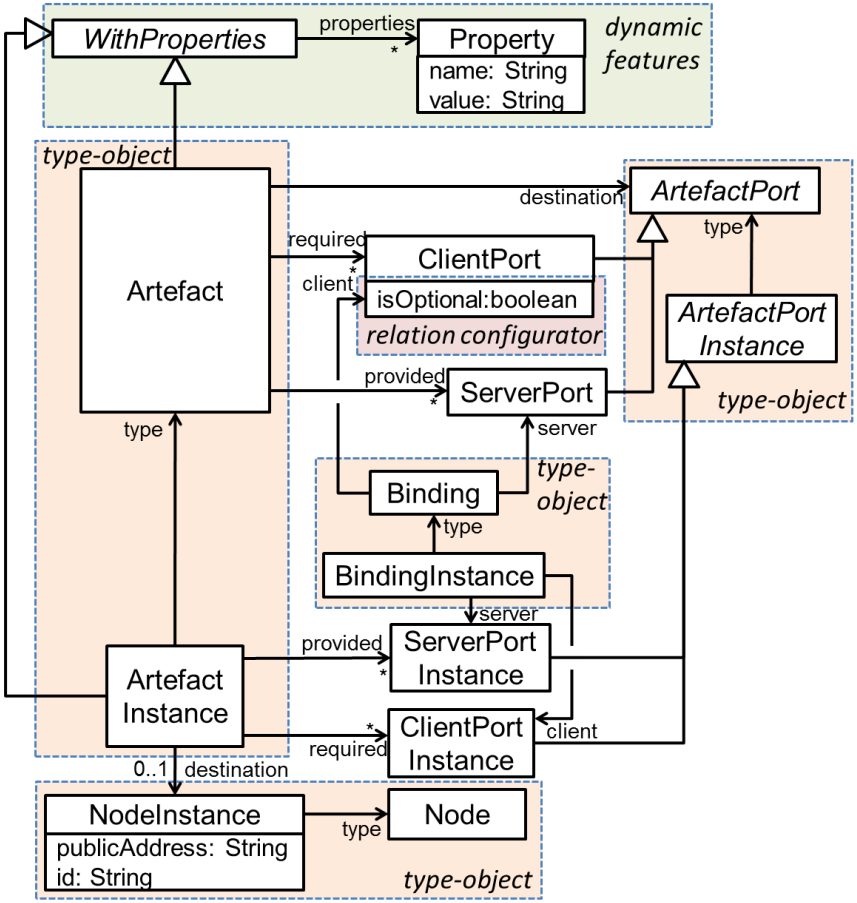
BUSINESS PROCESS/ENTERPRISE

Name	App.	Type Object	Dyn. Feats.	Domain Concepts	Relation Config.	Element Classif.	Meta-Model size
Agate [AtlanEcore 2014]	E	3	–	–	–	–	69c, 123r
Ant [AtlanEcore 2014]	E	1	2	–	–	–	48c, 28r
BPDM [OMG 2008b]	E	4	1	–	1	1	152c, 139a
BPMN 2.0.1 [OMG 2013f]	E	12	3	1	–	–	137c, 193a
Intalio BPMN 1.1 [AtlanEcore 2014]	EN	1	–	–	–	–	18c, 31r
BPMNProfile [OMG 2013b]	ST	12	1	1	1	–	130s, 179a
CMMN [OMG 2013g]	E	2	1	–	–	–	30c, 65a
DeclarativeWorkflow [ReMoDD 2014]	E	4	–	–	–	1	39c, 31r
DoDaF 2.02 [DoDAF 2010]	PT	35	–	–	–	–	130c, 135a (m)
DT4BP [ReMoDD 2014]	E	2	1	–	–	–	86c, 73r
ISO/IEC 24744 [González-Pérez and Henderson-Sellers 2007]	PT	5	–	–	–	–	19c, 5r (m)
ITPMF [OMG 2007]	E	13	–	–	–	–	49c, 61a
Promenade [AtlanEcore 2014]	E	1	–	–	–	–	18c, 22r
REA (Resource-Event-Agent) [Google code]	E	6	–	–	–	–	23c, 57r
SPEM [OMG 2008a; AtlanEcore 2014]	E	5	–	–	–	–	302c, 380a
UEML [AtlanEcore 2014; Bergholtz et al. 2005]	E,PT	3	–	–	–	–	23c, 27r
UML2 Activities [OMG 2011g]	E	1	–	–	–	–	228c, 274a
UPDM 2.1 [OMG 2013j]	ST	25	3	–	–	–	226s
XPDL [AtlanEcore 2014]	EN	1	–	–	–	–	52c, 66r

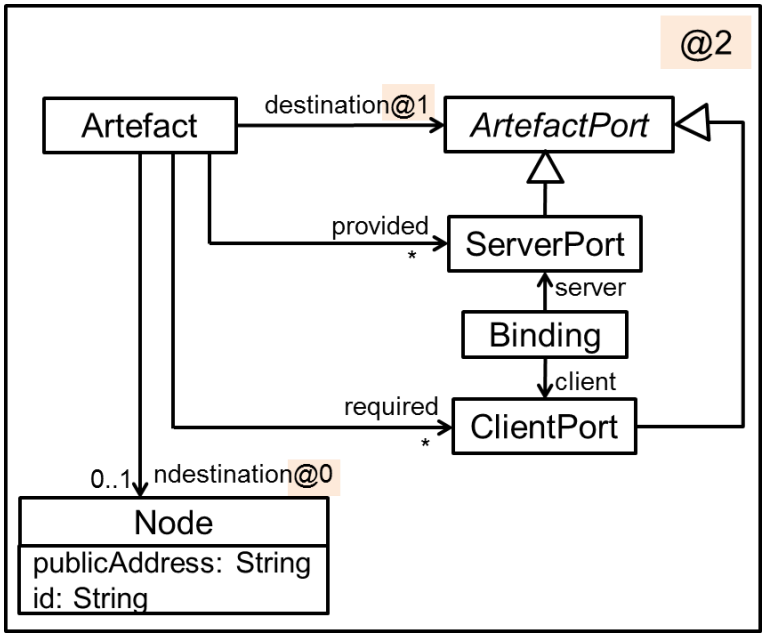
E=Explicit, S=Static, ST=Stereotypes, EN=Enumerated Types, PT=Powertypes

EXPLICIT TYPES TO MULTI-LEVEL

- CloudML (6 T-O, 1 DF)
- 14 classes

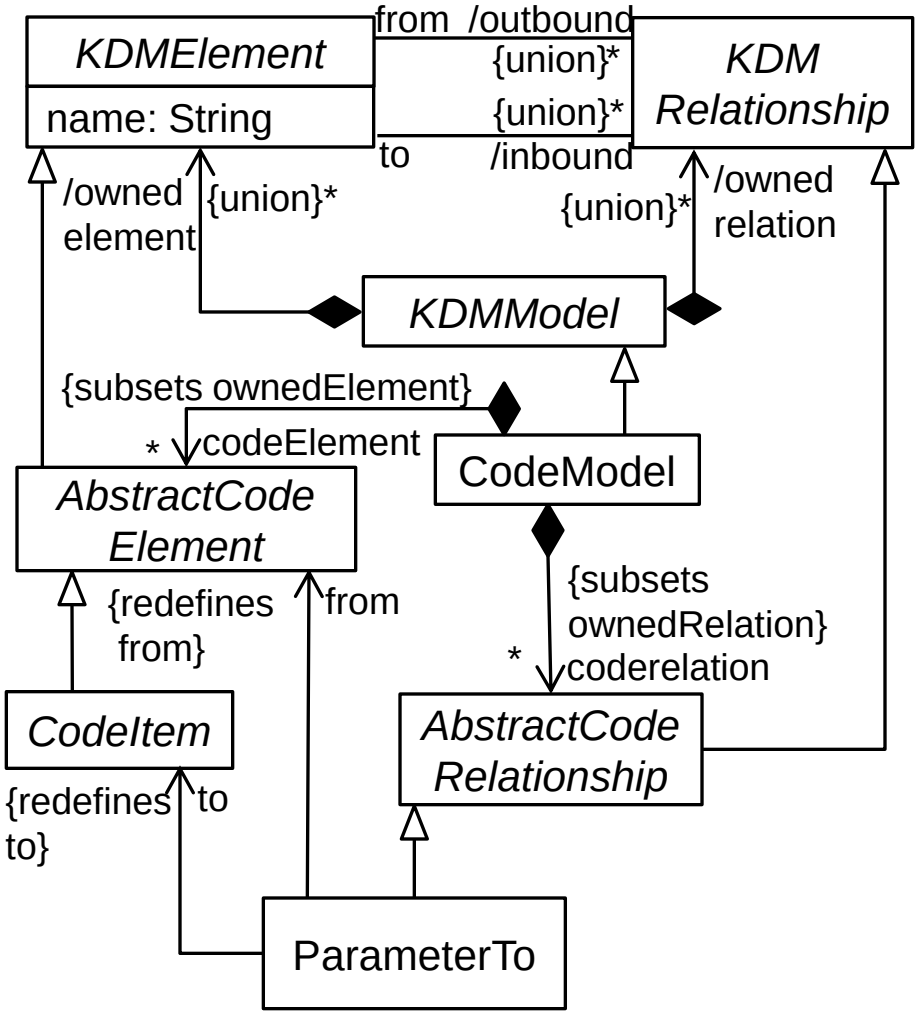


- Multi-level solution
- 6 classes

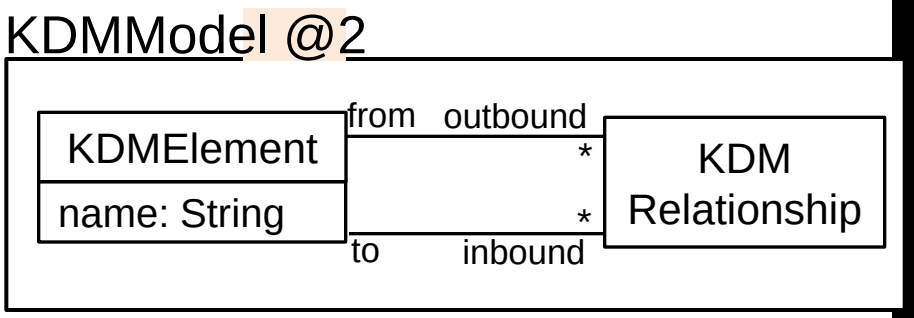


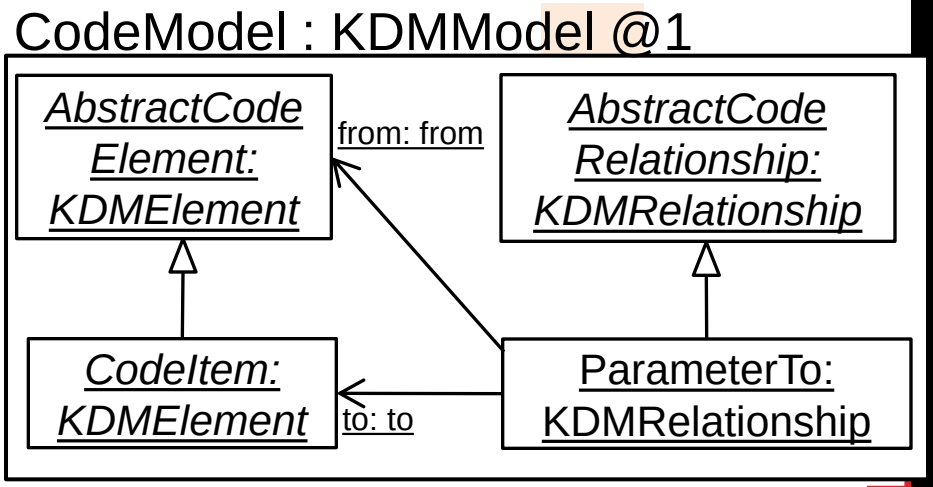
STATIC TYPES TO MULTI-LEVEL

- KDM (OMG)



- Simpler base meta-model





AGENDA

- Multi-level modelling: the basics
- MetaDepth
- Multi-level model management
- Advanced techniques
- When and where to use multi-level modelling
- **Summary, conclusions and outlook**

SUMMARY

The limits of my language mean the limits of my world.
Ludwig Wittgenstein

Multi-level modelling

- Generalizes two-level modelling overcoming some of its limitations
- Useful when the type-object pattern arises
- Leads to simpler models

MetaDepth

- A tool for Multi-level modelling
- Integrated with the Epsilon languages for model management

OUTLOOK

Use it in real MDE scenarios

- Let us know if you use metaDepth

Improve tooling

- Eclipse plugin

Refactoring of multi-level models

- flattenings and un-flattenings

A-posteriori typing

MORE…

Other related tools

- melanee (http://www.melanee.org/)
- DPF Workbench
- …

MULTI series of workshops (at MODELS)

- MULTI 2014 (Valencia)
- MULTI 2015 (Ottawa)
- MULTI 2016 (Saint Malo)

MLM wiki

- http://homepages.ecs.vuw.ac.nz/Groups/MultiLevelModeling/

MAIN METADEPTH REFERENCES

1. Original publication about the tool:

   J. de Lara and E. Guerra. "*Deep meta-modelling with MetaDepth*". 2010. Proc. TOOLS 2010. LNCS 6141, Springer, pp.: 1-20.

2. Integration with ML management languages. Concrete syntax:

   J. de Lara, E. Guerra, J. Sánchez-Cuadrado. "*Model-driven engineering with domain-specific meta-modelling languages*". 2015. SoSyM (Springer), Vol 14(1). pp.:429-459. Best papers of ECMFA'12.

3. Leap potency, deep reference, deep code generation, application in practice:

   J. de Lara, E. Guerra, R. Cobos, J. Moreno Llorena**.** "*Extending deep meta-modelling for practical model-driven engineering*". 2014. The Computer Journal, Vol 57(1). pp.:36-58

4. Analysis of ML constraints:

   E. Guerra, J. de Lara. "*Towards automating the analysis of integrity constraints in multi-level models*". 2014. MULTI'2014, (Valencia) (CEUR). pp.:1-10.

5. ML patterns. Applicability study for MLM:

   J. de Lara, E. Guerra, J. Sánchez Cuadrado. "*When and How to Use Multi-Level Modelling*". 2014. ACM TOSEM, Vol 24(2). pp.:1-46.

THANKS!

Juan.deLara@uam.es , Esther.Guerra@uam.es

@miso_uam http://www.miso.es