

# III. The Technical Space Graphware

## 30. Flat Analysis in Graphware: Graph Querying, Metrics, Reachability Analysis and Megamodel Dependency Analysis

Prof. Dr. U. Aßmann  
Technische Universität Dresden  
Institut für Software- und  
Multimediatechnik  
<http://st.inf.tu-dresden.de>  
Version 19-1.1, 13.01.20

- 1) Graph-Based DDL and CDL
  - 1) Relational Schema
  - 2) Entity-Relationship Diagrams
  - 3) MOF and ERD
- 2) Graph Query Languages
  - 1) .QL
  - 2) Metrics with .QL
  - 3) TGreQL
- 3) Macromodel Dependency Analysis
- 4) Model Invariant Specification
  - 1) OCL
  - 2) Spider Diagrams

# Obligatory Literature

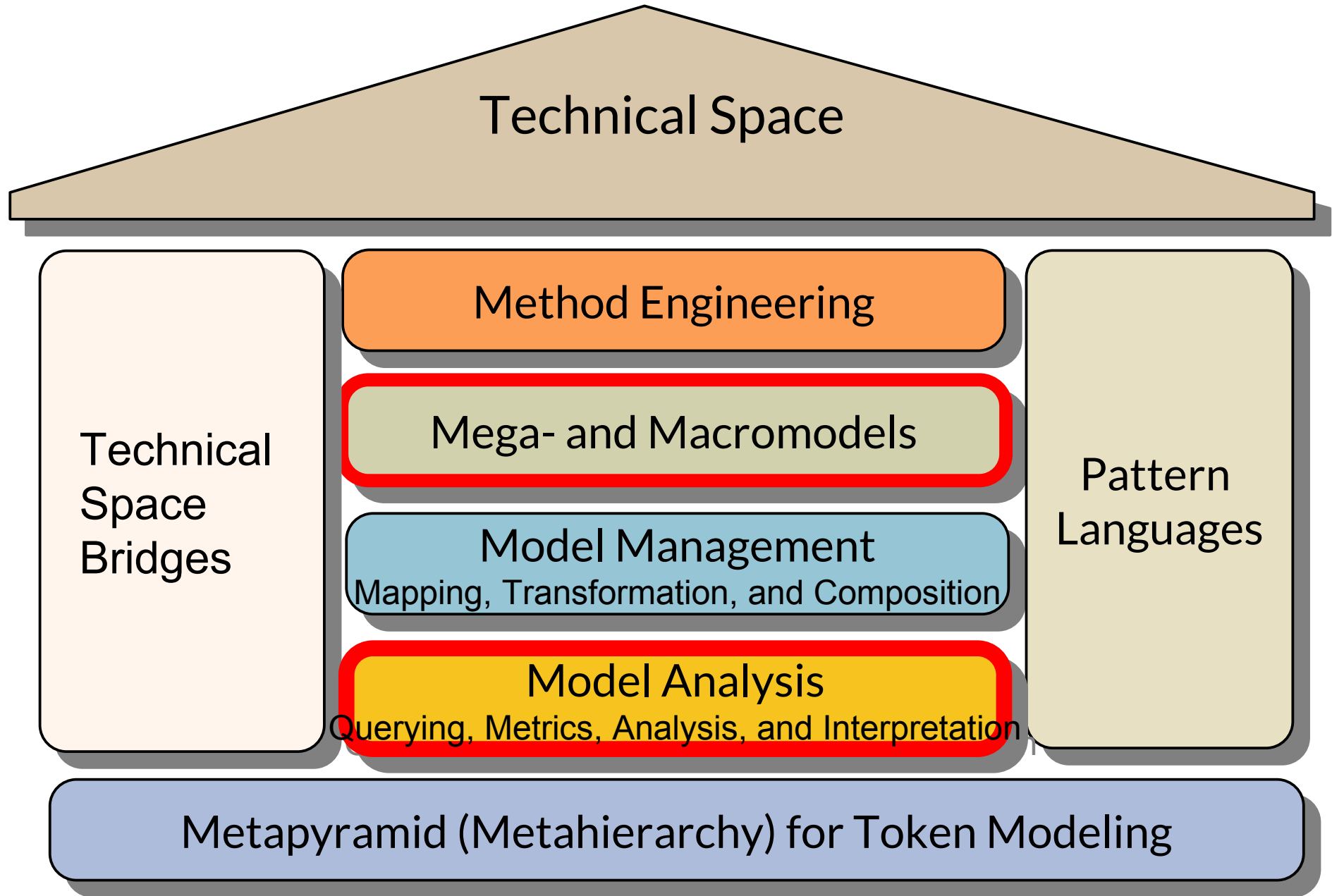
- ▶ [http://en.wikipedia.org/wiki/List\\_of\\_UML\\_tools](http://en.wikipedia.org/wiki/List_of_UML_tools)
- ▶ [http://en.wikipedia.org/wiki/Entity-relationship\\_model](http://en.wikipedia.org/wiki/Entity-relationship_model)
- ▶ <http://www.utexas.edu/its/archive/windows/database/datamodeling/index.html>
- ▶ [deMoor] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, Julian Tibble, "Keynote Address: .QL for Source Code Analysis", SCAM, 2007, 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 3-16, doi:10.1109/SCAM.2007.31
- ▶ <https://semml.com/codeql>, <https://help.semml.com/QL/learn-ql/>
- ▶ <https://help.semml.com/QL/learn-ql/java/ql-for-java.html>
- ▶ Language handbook <https://help.semml.com/QL/ql-handbook/index.html>
  - Specification <https://help.semml.com/QL/ql-spec/language.html>
- ▶ Thief detective game: <https://help.semml.com/QL/learn-ql/beginner/find-thief-1.html>
- ▶ Industrial case studies: <https://semml.com/case-studies>
- ▶ Community-driven security analysis:
  - Github repo of LGTM examples <https://github.com/Semml/ql>
  - <https://lgtm.com/help/lgtm/about-lgtm>
  - Query console <https://lgtm.com/query>
  - <https://lgtm.com/help/lgtm/console/ql-java-basic-example>

# References

- ▶ [Chen] P. P.-S. Chen. The entity-relationship model - towards a unified view of data. Transactions on Database Systems, 1(1):9-36, 1976
- ▶ A Comparison of ATL and Story-Driven Modeling (Fujaba-style GRS)

[http://www.es.tu-darmstadt.de/fileadmin/download/publications/spatzina/PP\\_AGTIVE\\_2011.pdf](http://www.es.tu-darmstadt.de/fileadmin/download/publications/spatzina/PP_AGTIVE_2011.pdf)

# Q10: The House of a Technical Space



# Flat and Deep Model and Code Analysis

- ▶ DQL answer questions about the materials in a repository or in a stream
  - Analytics for one document alone (metrics, “Business Intelligence”)
  - Filtering of a stream
  - Combining input streams

CQL do the same for programs and models:

- ▶ **Flat model analysis** asks questions on
  - the direct context of a model element (context-free queries, pattern matching)
  - the global knowledge about a model element
- ▶ **Deep model analysis** (value flow analysis, data-flow analysis, inter-procedural analysis, inter-component analysis) respects the main structure of a model and asks the question
  - whether certain parts of a model are reachable from each other (connected)
  - what is the context of a model element in a structured environment (abstract syntax tree, control flow graph, value flow graph, dependency graph)
  - where do attributes flow (in an attribution)
- ▶ **Software metrics:** counting objects, relationships, dependencies
- ▶ **Inter-model dependencies** between models in a megamodel

## 30.1 DDL in the Graph-Based Technical Spaces

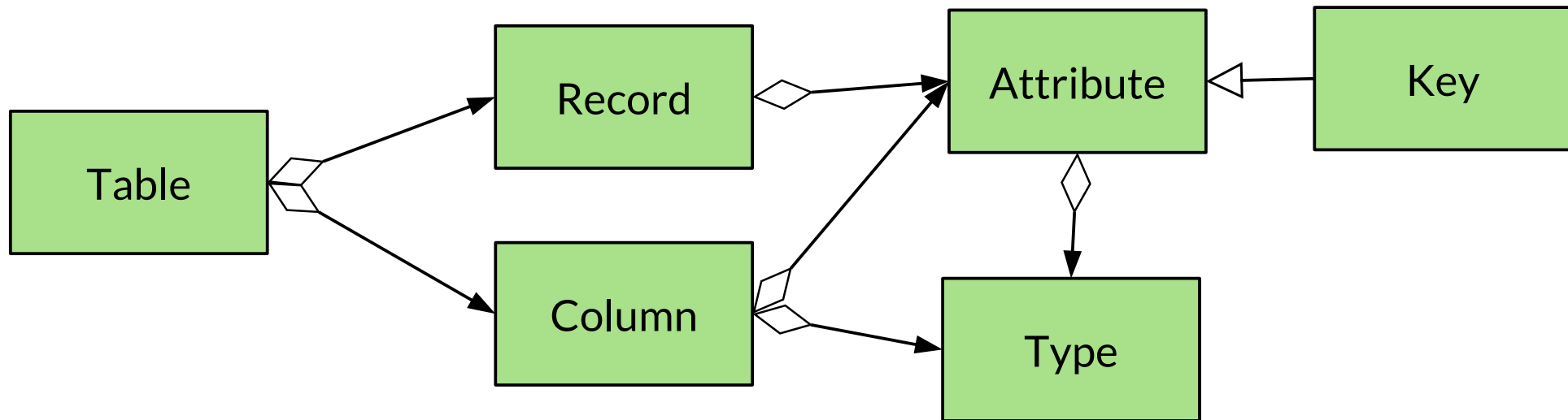
## 30.1.1 Technical Space RelationWare with DDL Relational Schema

Relational Algebra works with *typed relations*



# Technical Space Relational Algebra mit Metalanguage Relational Schema

- ▶ Relational Algebra (Codd) works on tables of tuples with attributes
  - See courses on databases



Relational Schema  
Metamodel

Key	FirstName	Surname	Street	Town
@1	Uwe	Aßmann	Bakerstreet 5	New York
@2	Frank	Miller	Northstreet 9	Pittsburgh
@3	Mary	Baker	Magdalenstr eet	Oxford



## 30.1.2 Technical Space ER-Ware with DDL Entity-Relationship-Diagrams (ERD)

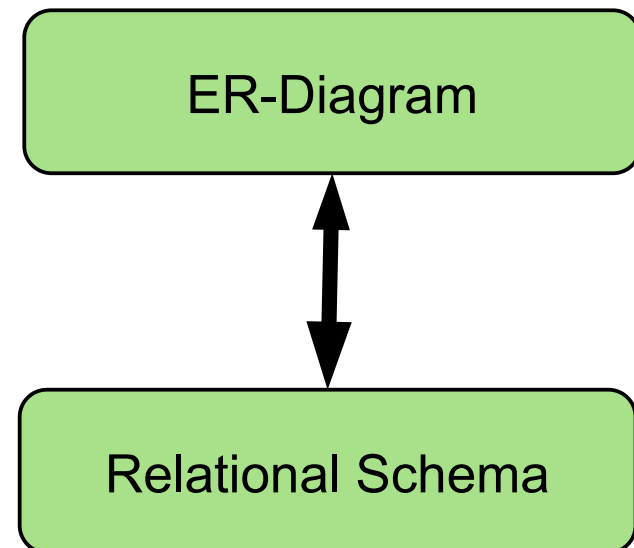
A Simple DDL/CDL with Mapping to the  
Relational Algebra

Relations and Entities (without inheritance)



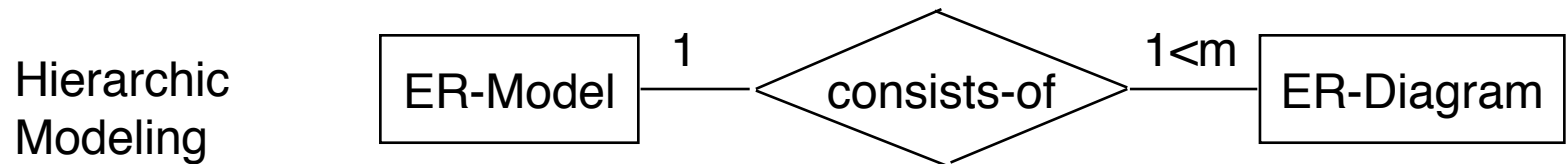
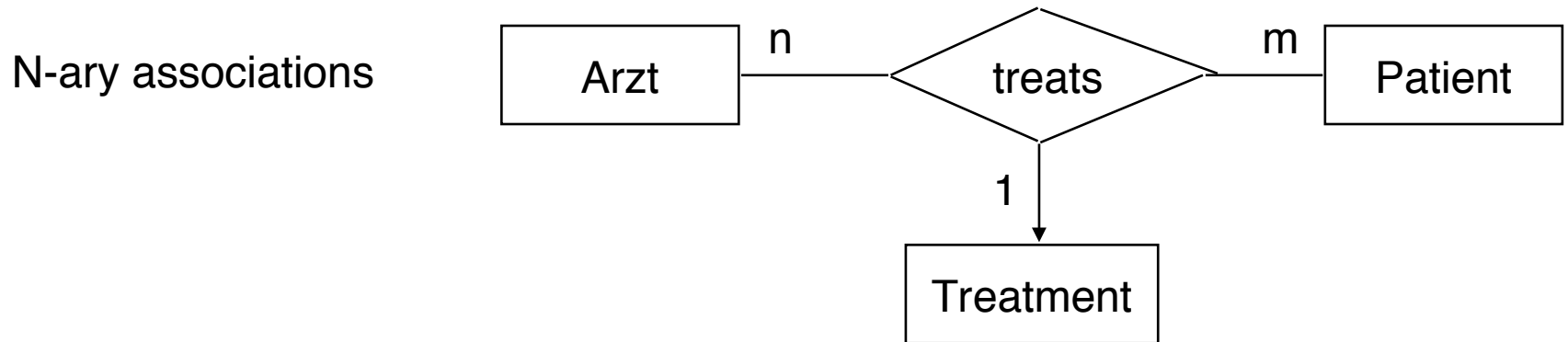
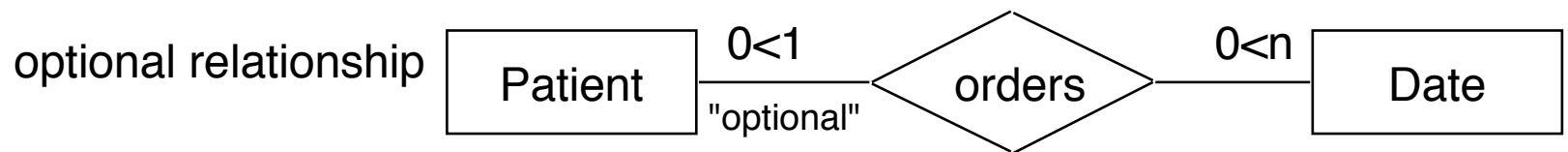
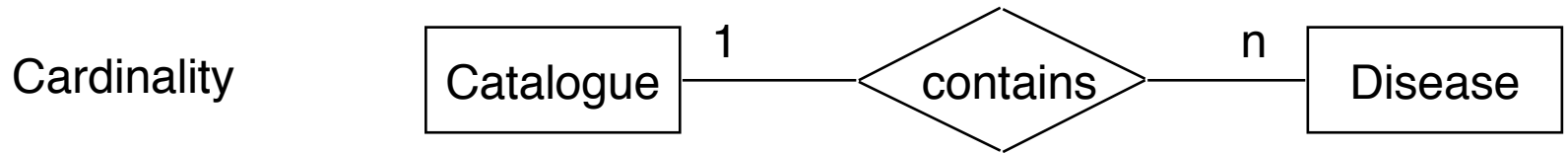
# Modeling with Entity-Relationship-Diagrams (ERD)

- ▶ ERD can be mapped easily to relational schema (with an invertible 1:n-mapping, **ER-RS-mapping**)
  - Entities form special relations with “identifer” (key, surrogate)
  - ER-diagrams can be stored easily in databases (simple persistence)
- ▶ ERD is often used as CDL in larger integrated development environments (simple persistence of code and models)

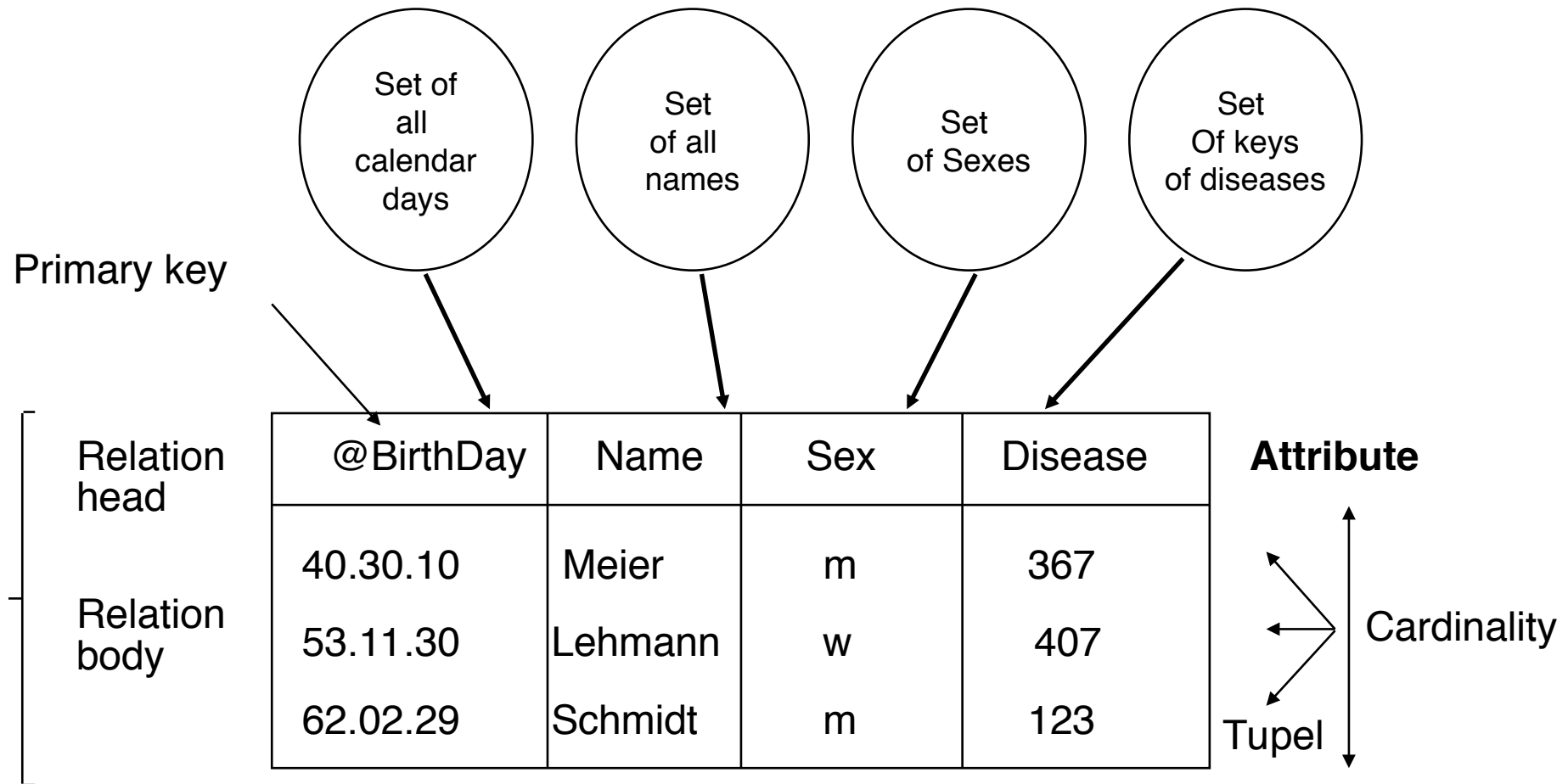


# ERD-Relationships in Chen-Notation (unlike UML)

- ▶ All “entities” (classes) are represented as “entity-”tables

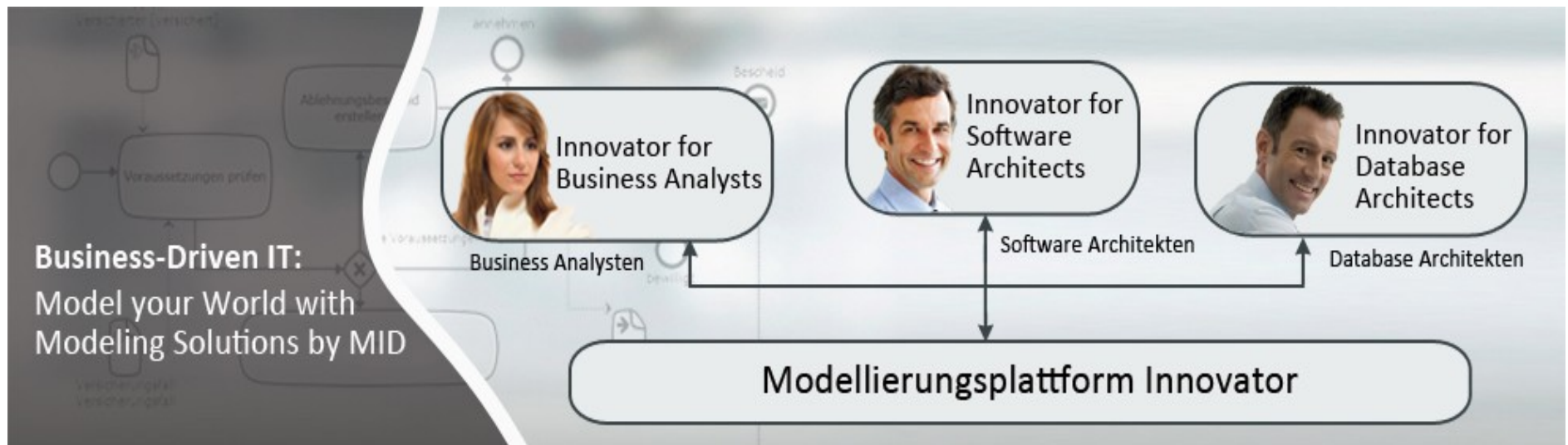


# Mapping of Entity Type "Patient" to the Relational Schema



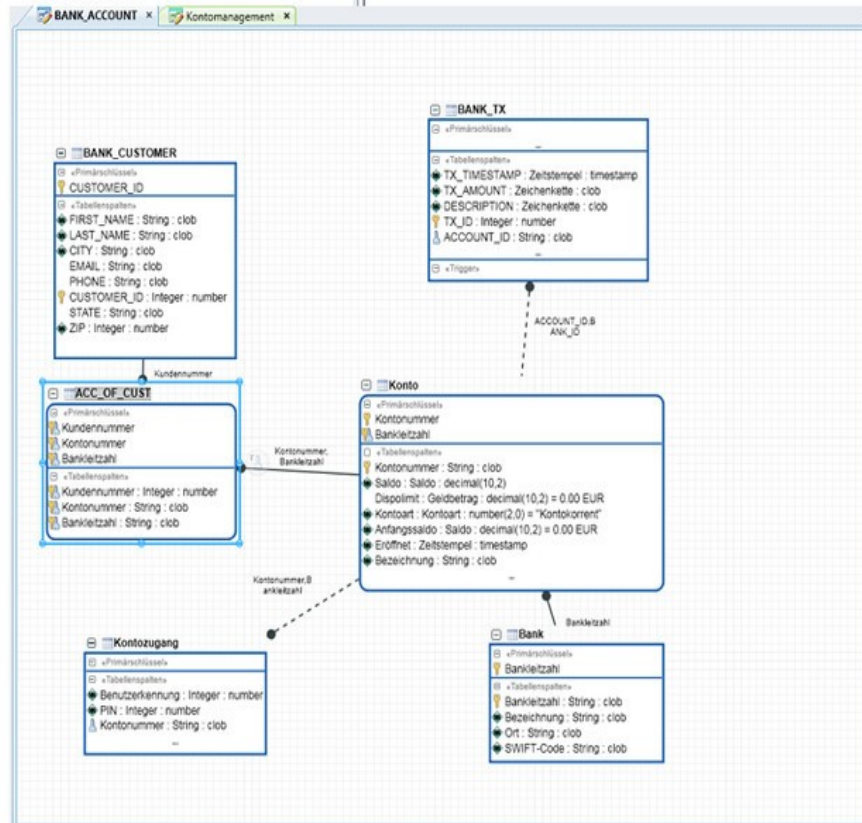
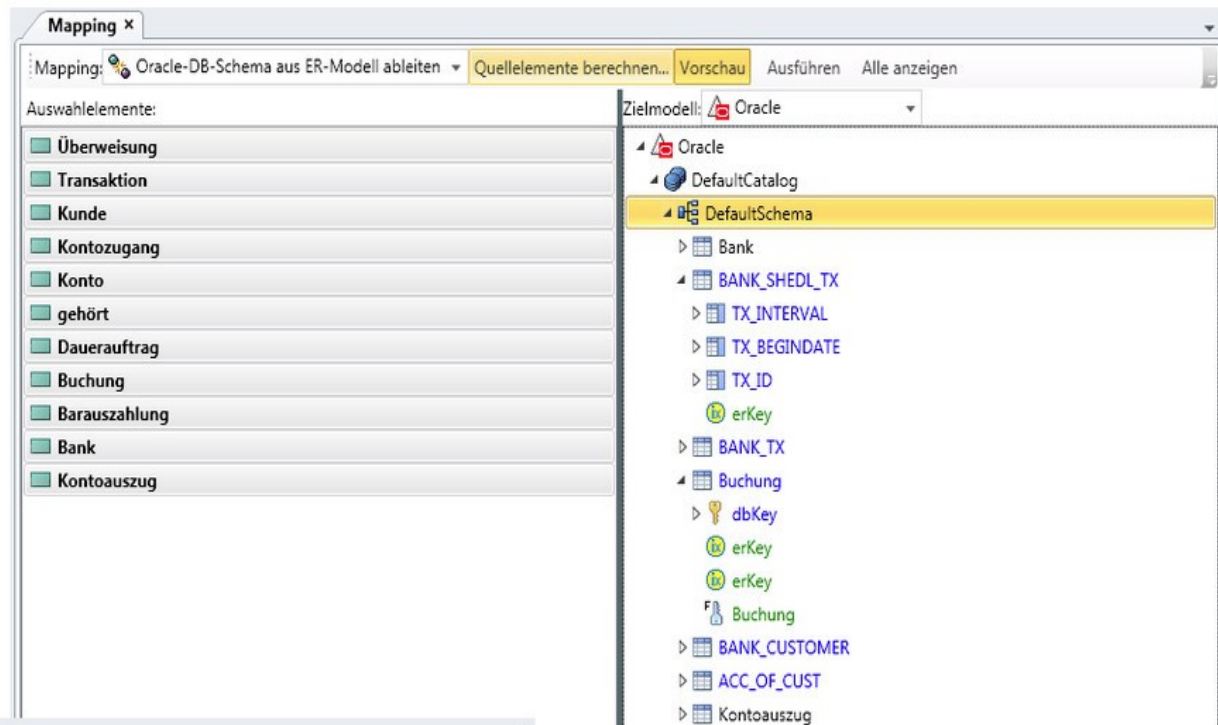
# Importance of ERD

- ▶ ERD is the “better” relational schema, because it treats objects (entities)
  - Often used for data dictionaries in information systems
- ▶ ERD, however, does not support inheritance
  - Applications can easier be verified, e.g., for embedded or safety-critical systems
- ▶ Typical Tool: MID Innovator for database architects:



# Mapping ERD to RS in MID

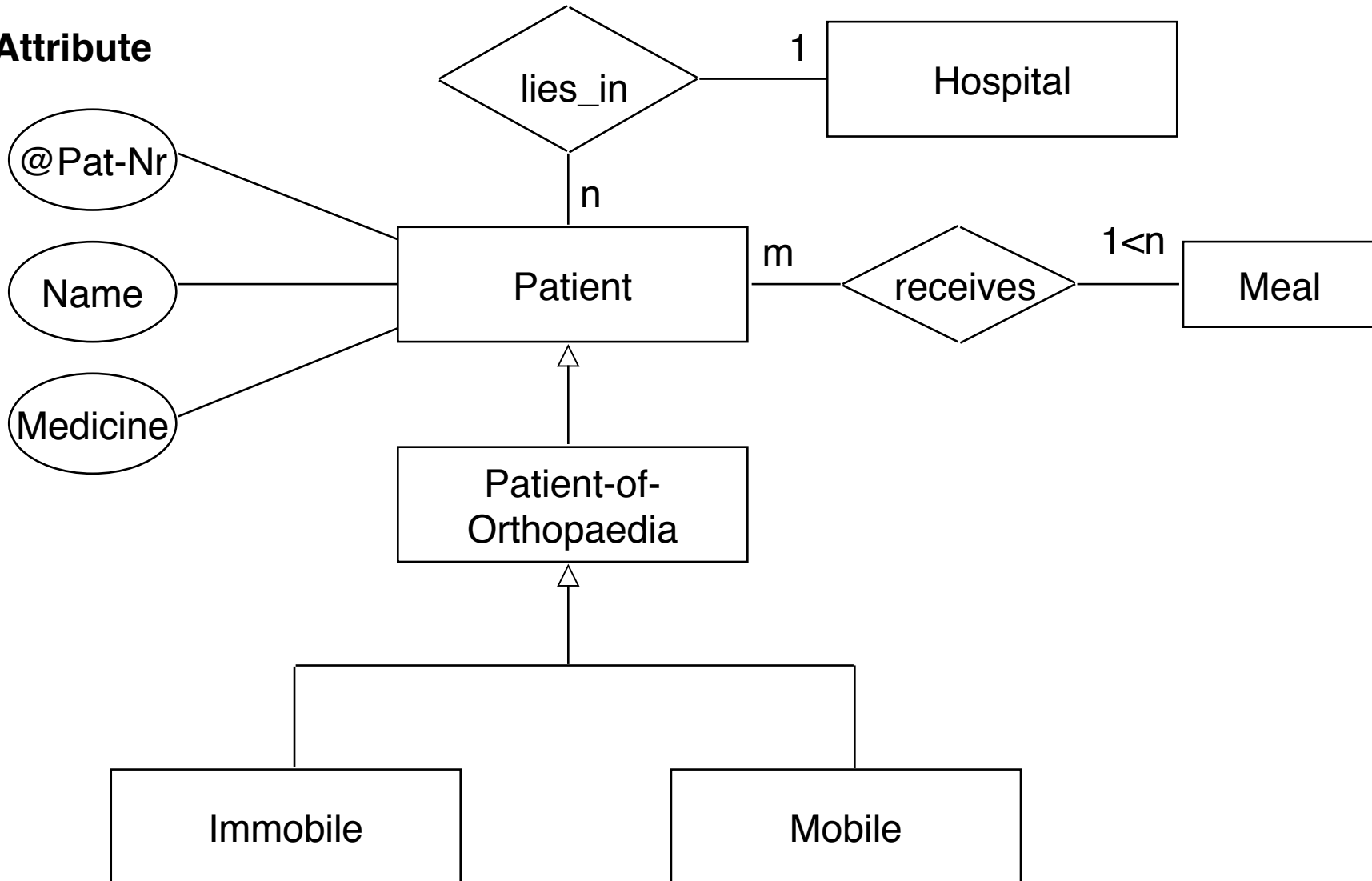
<http://www.mid.de/typo3temp/pics/f0df65b8a2.jpg>



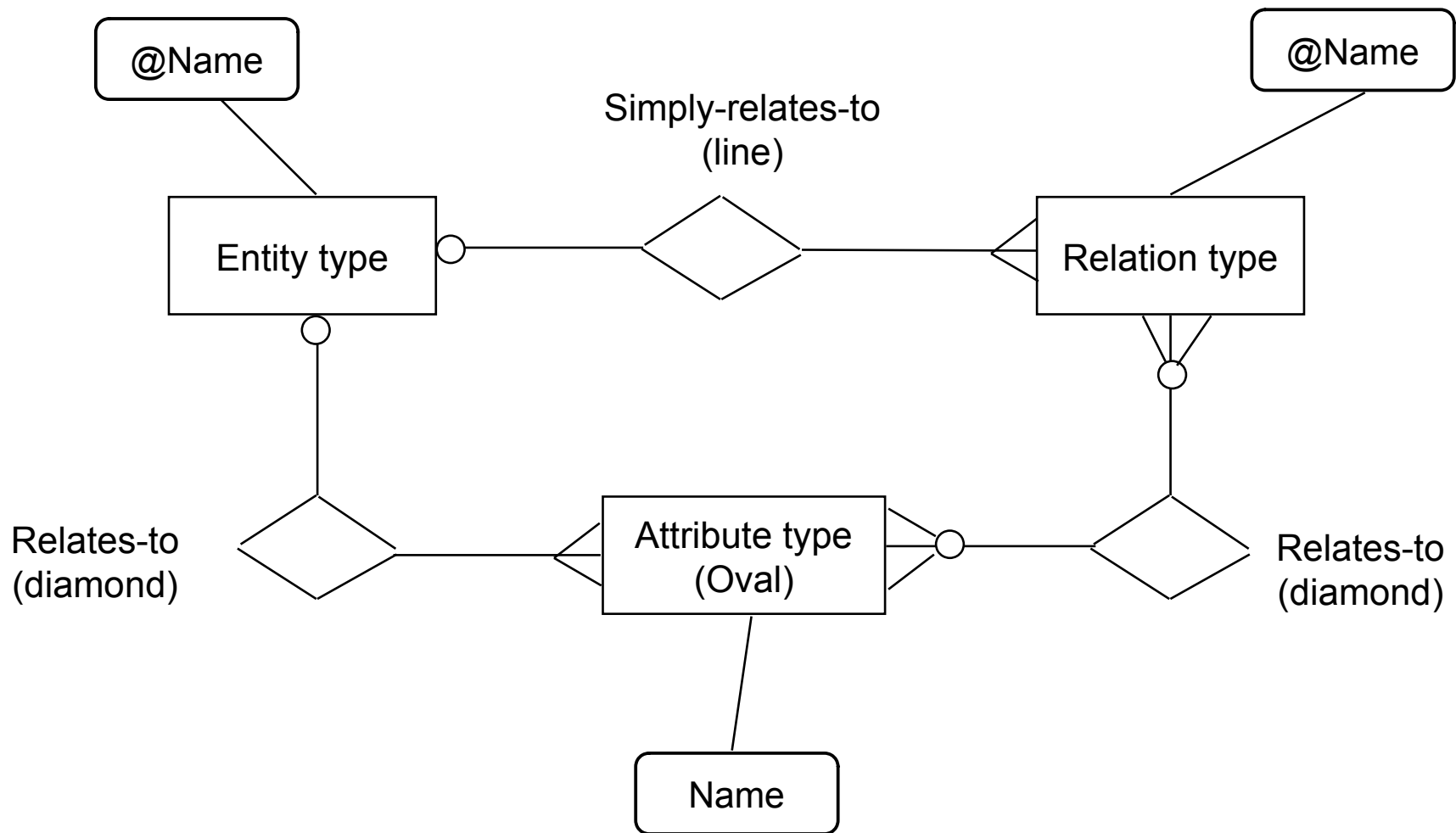
# Extended ERD (EERD) Uses Inheritance

## Example: Patient Record

### Attribute



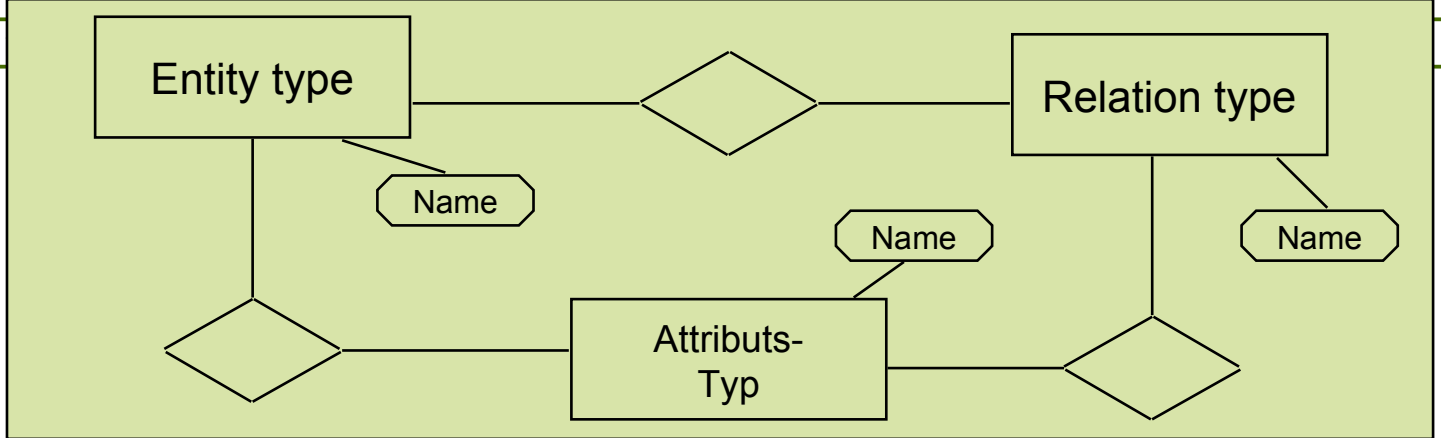
# The Metamodel of ERD in ERD (lifted ERD Metamodel)





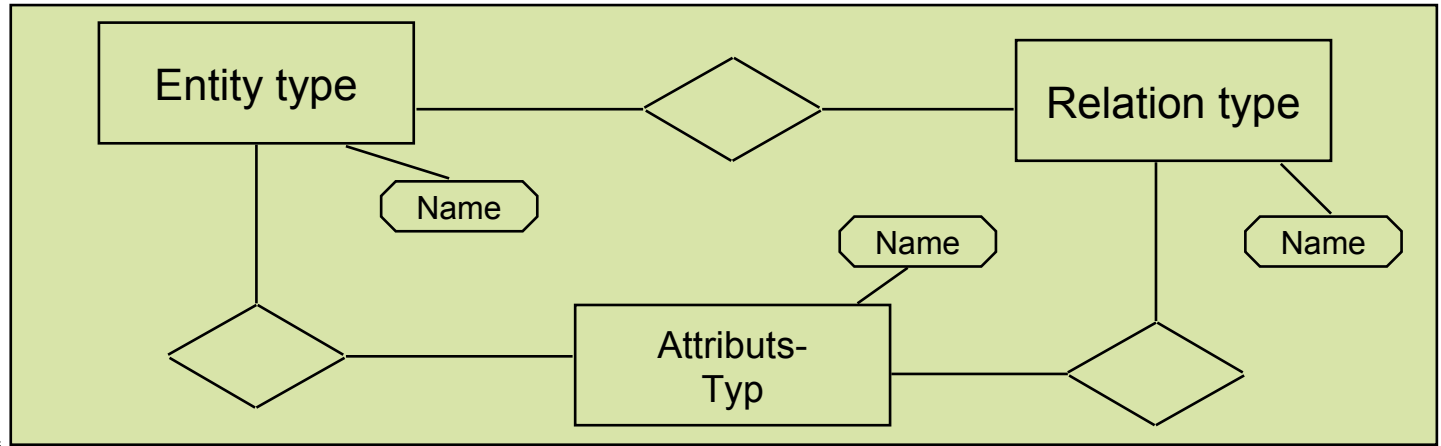
# Metahierarchy with ERD as Metalanguage (lifted metamodel)

**Metametamodel**



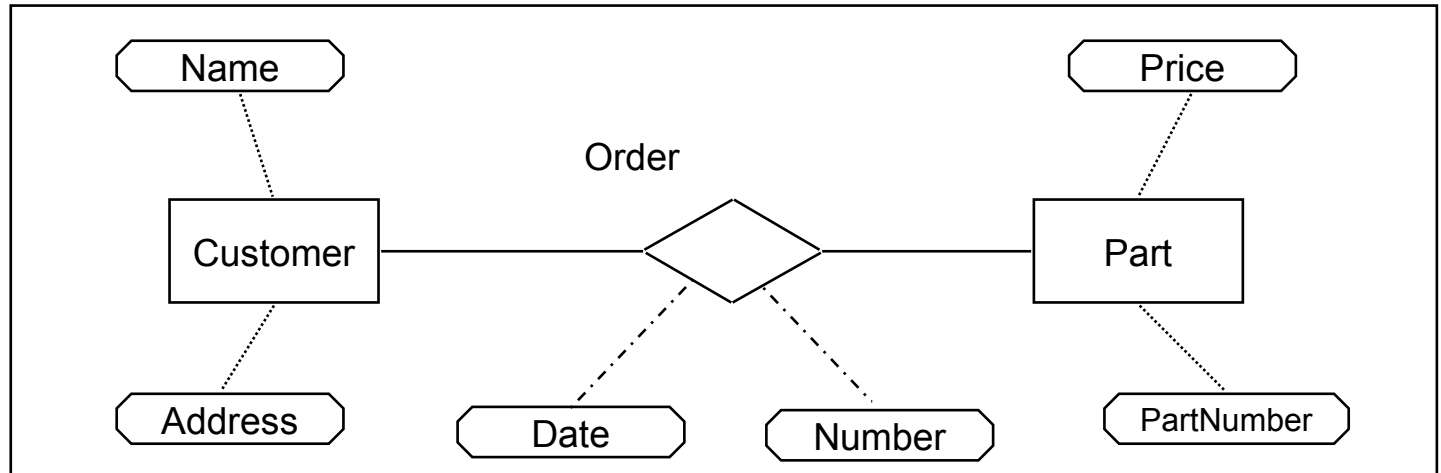
**M3**

**Metamodels**



**M2**

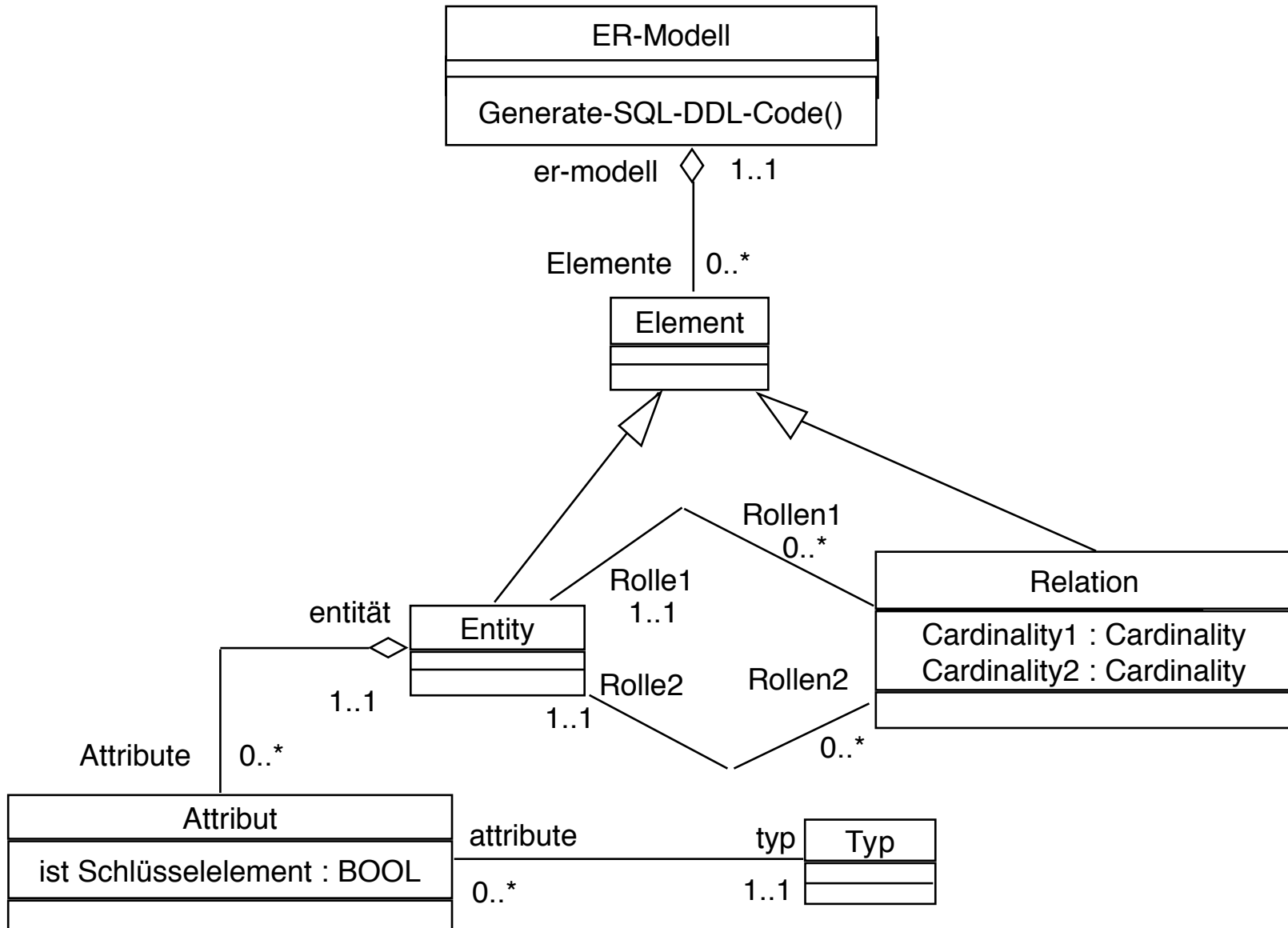
**Models**



**M1**

# MOF is ERD with Inheritance

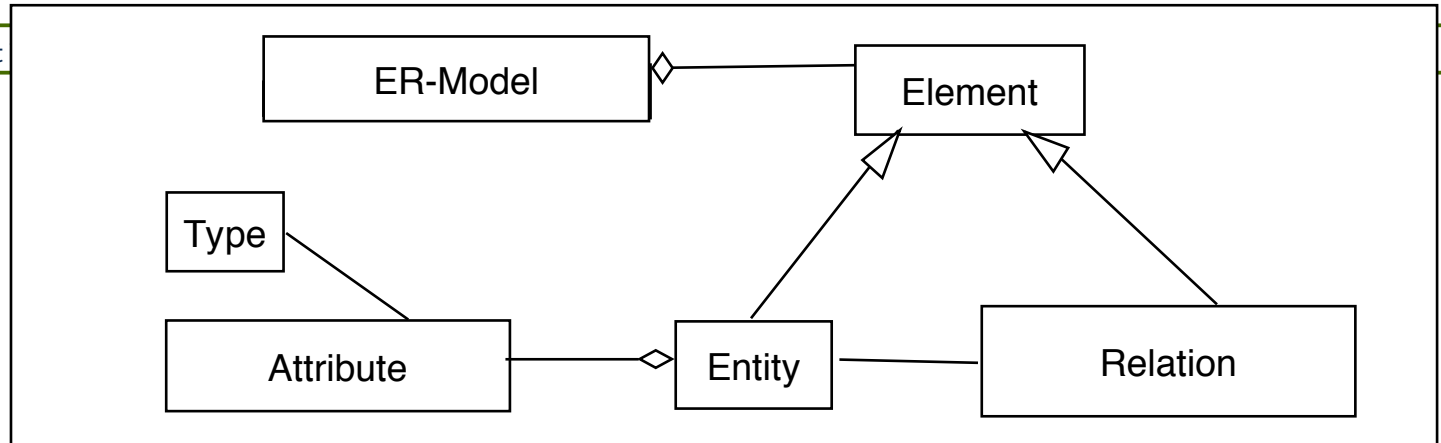
## Meta-Modell of Entity-Relationship-Diagramms (in MOF)



# Metahierarchy with MOF as Metalanguage (non-lifted)

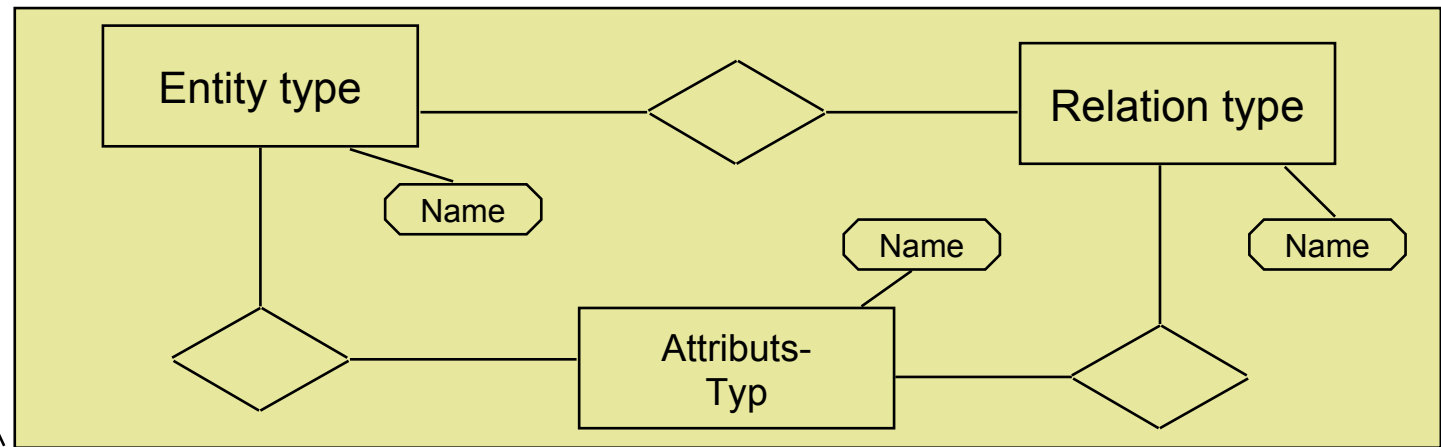
M3

Metametamodel



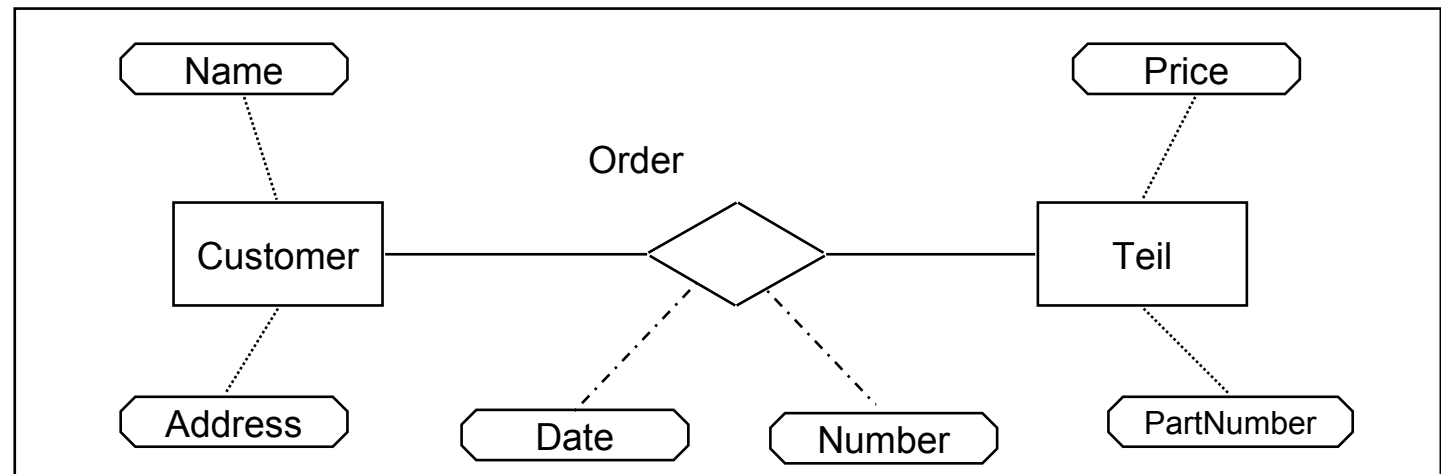
M2

Metamodels



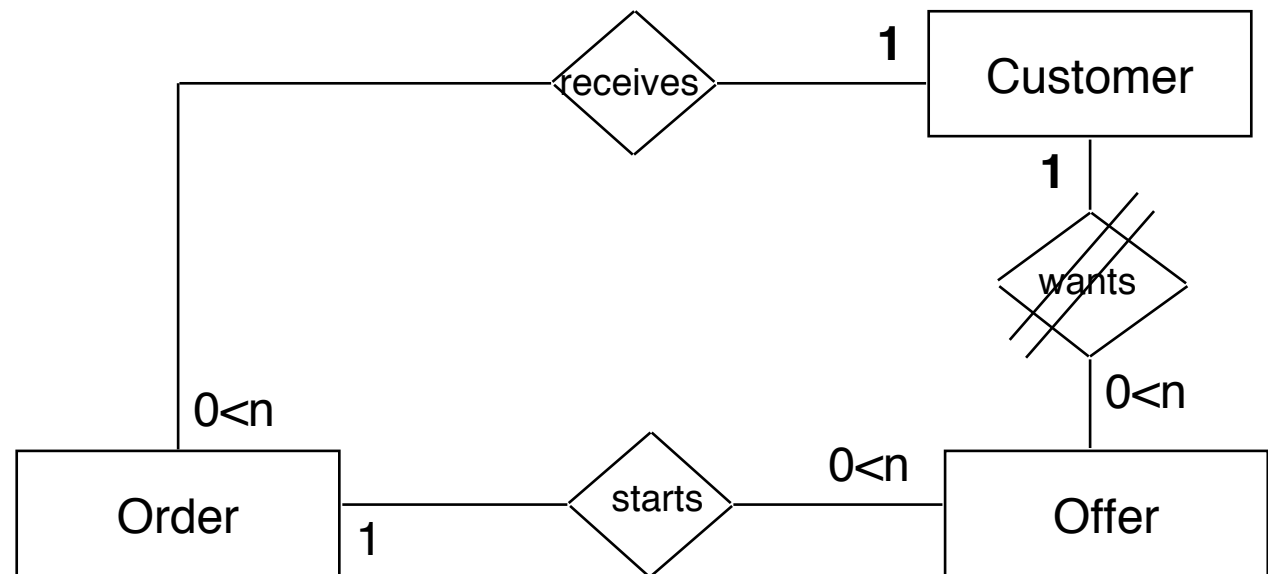
M1

Models



# Consistency Constraints in ERD Models

- ▶ An ERD can contain integrity constraints (consistency constraints)
- ▶ Ex.: **Cycle-freedom constraint:** Check: find cycles in the graph of a ER diagram
- ▶ Correct by
  - cutting a cycle at the least important position (human intervention)
  - Finding a spanning tree and cutting all other edges
- ▶ Instead of cutting, edges can be made secondary links (then we have link trees)



after: [Raasch]

# Other Consistency Constraints of ER-Models

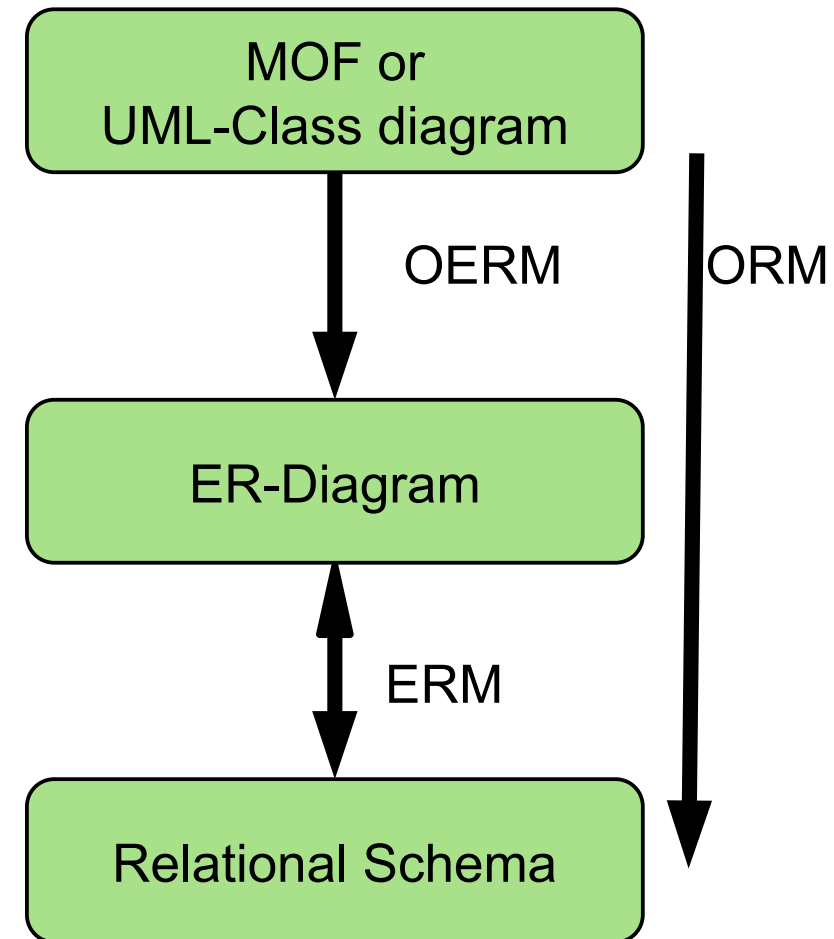
- ▶ **Range checks** for attributes
- ▶ **Key dependencies (functional dependencies):**
  - Uniqueness of attribute values: An attribute K of a relation R is a key candidate, if only one tuple has the same value of K
  - Key minimality: Is the attribute K compound, no component can be removed to loose the key condition.
  - Primary key serves for identification of a tuple (“entity check”)
  - Secondary keys: other keys
  - Foreign key reference (primary key reference): A foreign key (link) is referencing a tuple in another relation by its primary key
- ▶ **Referential Integrity**
  - The model does not contain undefined foreign keys (links)
  - i.e., all names (links) can be resolved by name analysis

## 30.1.3 MOF as Extended ERD



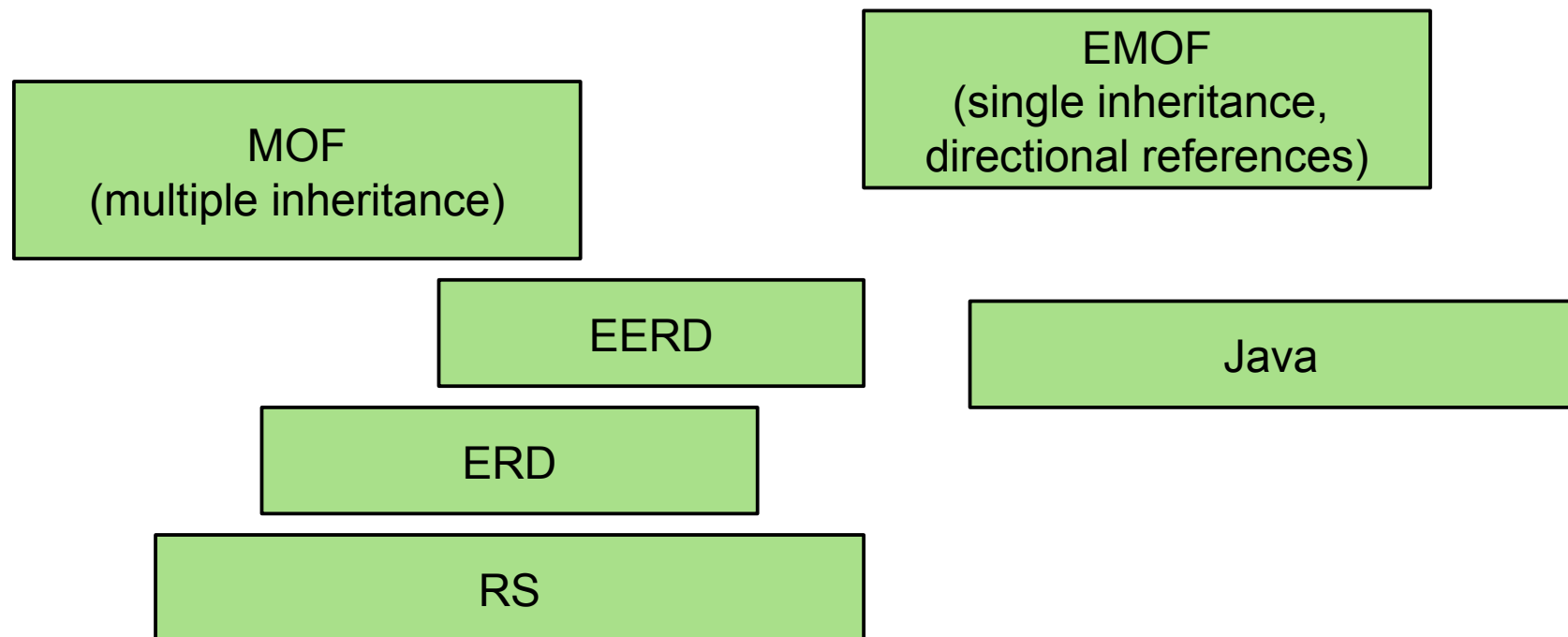
# Data Modeling for Information Systems (Object-Relational Mapping, ORM) with UML-CD, ERD and RS

- ▶ For persistence, objects should be stored with an object-relational mapping to a database (OR-Mapping)
- ▶ OERM-Mapping of class diagrams to ERD is (unfortunately) indeterministic
  - Inheritance mapping
  - Identification of keys (primary, secondary, foreign)
  - Resolution of multiple inheritance by copying
  - Cannot be inverted automatically
- ▶ Between ERD und RS exists a *deterministic, bidirectional* mapping (ER-Mapping) by which the data models can be synchronized (restored without information loss)



# The Difference of ERD, MOF and EMOF

- ▶ MOF extends ERD with multiple inheritance and method signatures
- ▶ However, MOF must be mapped down to Java
  - Inheritance
  - Bidirectional associations
- ▶ EMOF has only directed references, no bidirectional associations
  - Only simple inheritance
- ▶ EMOF can directly be mapped down to Java, C++, or C#





## 30.2 Flat Model Analysis with Graph Query Languages (Graph QL)

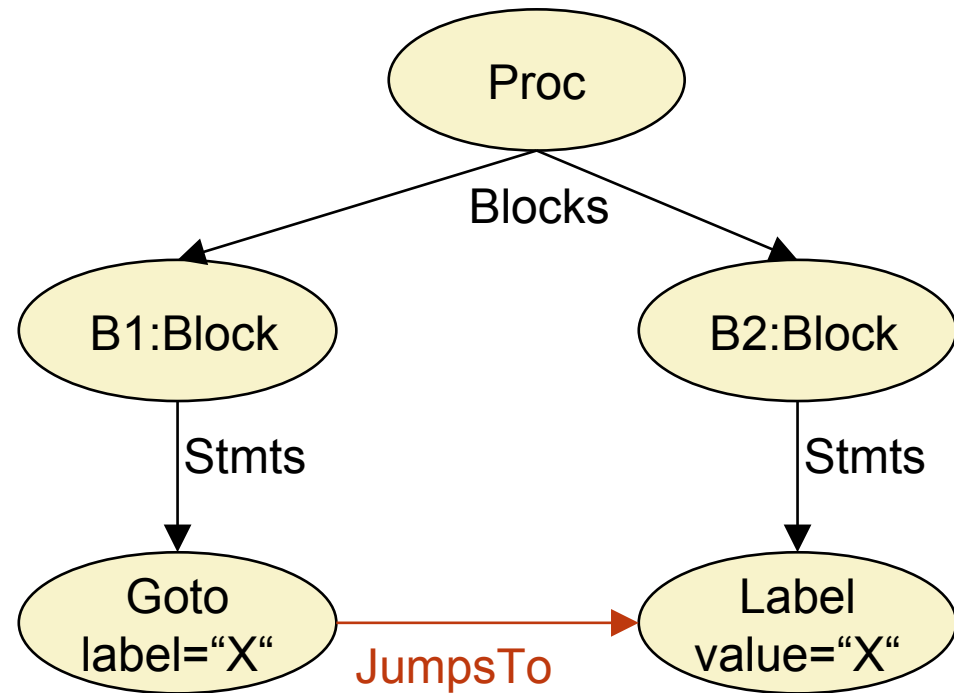
DQL – Data Query Languages

CQL – Code Query Languages

# Graph Pattern Matching of Non-Tree Patterns

- ▶ **Graph pattern matching** works by mapping a graph pattern (graphlet) to the manipulated graph.
- ▶ Ex.: Linking gotos and Block-entry statements to build up the control-flow graph

```
-- Datalog notation:  
JumpsTo(Goto, Label) :-  
    Blocks(Proc, B1:Block),  
    Blocks(Proc, B2:Block),  
    Stmts(B1, Goto), Stmts(B2, Label),  
    Goto.label==X, Label.value==X.  
  
-- if-then rules:  
If  Blocks(Proc, B1:Block),  
    Blocks(Proc, B2:Block),  
    Stmts(B1, Goto), Stmts(B2, Label),  
    Goto.label==X, Label.value==X  
Then  
    JumpsTo(Goto, Label).  
  
- regular expression notation (TGreQL):  
JumpsTo := Proc.Blocks.Stmts.Goto.label(X)  
AND Prod.Blocks.Stmts.Label.value(X)
```

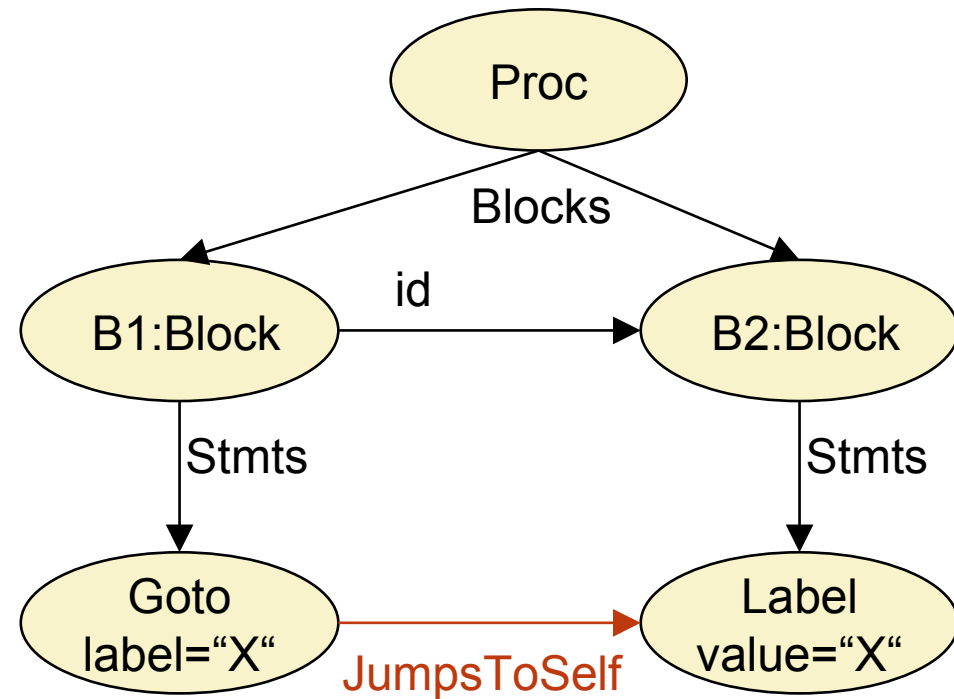


# Pattern Matching of Non-Tree Patterns

- ▶ Block self edges point back on their own block

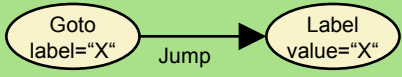
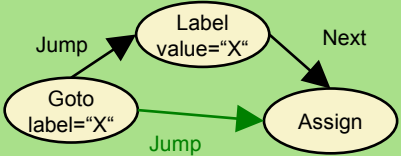
**-- Datalog notation:**

```
JumpsToSelf(Goto, Label) :-  
  Blocks(Proc, B1:Block),  
  Blocks(Proc, B2:Block), id(B1, B2)  
  Stmts(B1, Goto), Stmts(B2, Label),  
  Goto.label==X, Label.value==X.
```



# Different Notations for Node-Edge Patterns

- ▶ For notation of edges (and predicates), textual as well as graphical notations exist

	Datalog Prolog	Graphic (Optimix, EARS)	Textual graphics (TgreQL, GrGen)	Juxtaposition	Object- oriented (.QL)
edges	$e(N,M)$		$-N-e-M->$ $N -e-> M$	$N e M$	$N.e(M)$
recursion	$r(N,M) :-$ $e(N,Z),$ $r(Z,M)$		$N -e^* \rightarrow M$	$N e^* M$	$N.e^*(M)$

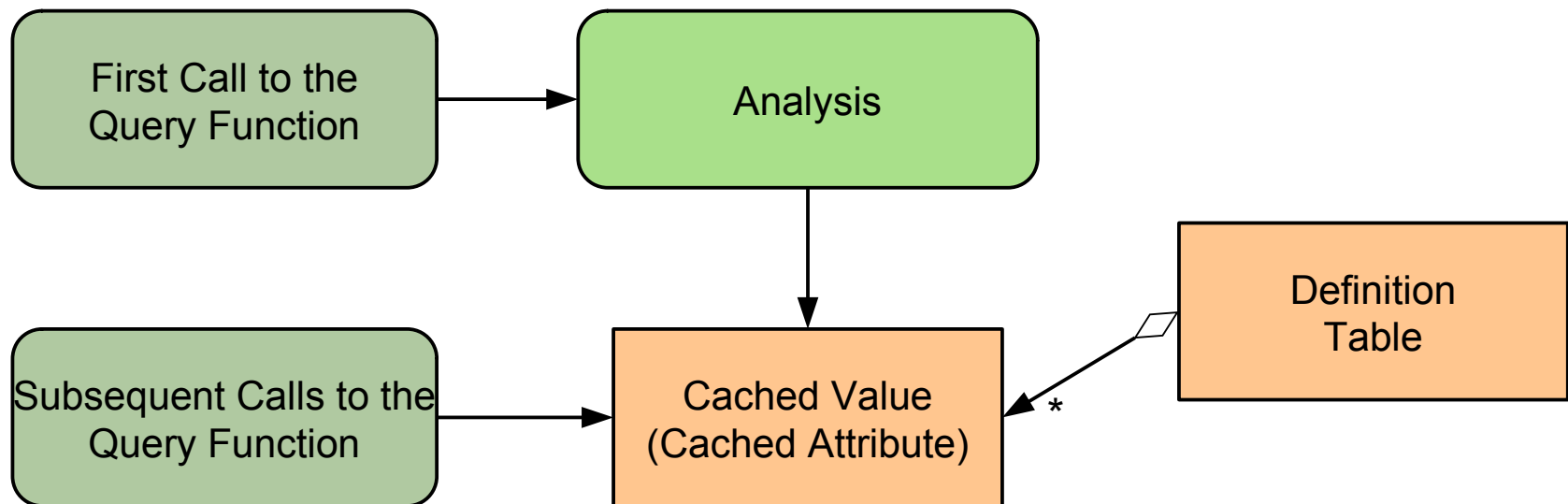
# Definition of Attribution, Access and Query Functions

From the metamodel, we can define **access**, **query** and **attribution functions**, functions to access, query **attributes** or **neighbors**:

- ▶ **(Local) Attribute access functions:**
  - `ModelElement.hasName()`
  - `ModelElement.getDeclaringType()`
- ▶ **Neighbor access functions (via references):**
  - `Class.getPackage()`: for neighbor Package
  - `Class.getUpperClass()`: get the direct upper class
  - `Class.getDeclaresMethod()`: for contained Method
- ▶ **Query functions** looking up information in the abstract syntax graph (ASG) or model:
  - `Expr.getUsedTypes()`: search all types which are used in Expr (type analysis, type resolution)
  - `Name.getType()`: search the type object to the Name
  - `Name.getMeaning()`: search the definition of the Name
  - `Stmt.getProcedure()`: search out to find the procedure of the Stmt
- ▶ **Pattern match functions** assemble all matching redexes of a pattern

# Name and Type Analysis: Caching a Query Function

- ▶ Some values of query functions change never, once they have been determined
  - The values can be cached
- ▶ **Attribute caching** is a mechanism to cache semantic attributes in an ASG or model for faster access
- ▶ A **definition table (often called symbol table)** is a set of cached attributes.



## 30.2.1 .QL – Relational Queries on Source Code in Technical Space Java

Courtesy to Florian Heidenreich and  
<http://semml.com>



DRESDEN  
concept  
Exzellenz aus  
Wissenschaft  
und Kultur

# SQL-Like Code Query Language .QL

- ▶ **.QL** is an object-oriented query language in the spirit of SQL and Datalog
  - Developed in the group of Prof. Oege de Moor (Oxford)
  - Marketed by Semmler.com
- ▶ Queries, metrics, visualizations are supported
  - Repositories with Java and Objective-C code
  - Works also now on C/C++
- ▶ Metamodel is EMOF-like (single inheritance, references)
  - Classes, Methods, Blocks are interpreted as **relational tables** (sets of tuples over member entries) resp. **Predicates** (telling whether a tuple exists)
  - Definition of access functions:
    - `Class.getDeclaredMethod()`: for neighbor Method
    - `Class.getPackage()`: for neighbor Package
    - `ModelElement.hasName()`: get the Name
    - `ModelElement.getDeclaringType()`: get the Type



# Code Display

The screenshot shows the Eclipse IDE interface. The title bar reads "Java - PatternPainter.java - Eclipse SDK". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The Package Explorer on the left shows a project named "jhotdraw" with a tree structure including packages like "org.jhotdraw.samples.javawdraw" and classes like "PatternPainter" and "draw". The Quick query window displays a query:

```
from Class clazz, Method method
where
  clazz.getPackage().getName().matches("org.jhotdraw.samples%")
  and method = clazz.getAMethod()
  and not(clazz.isAnonymous())
select clazz.getPackage(), clazz, method, method.getAMethodParamType()
```

The main editor shows the code for "PatternPainter.java". The "draw" method is highlighted:

```
public void draw(Graphics g, DrawingView view) {
    drawPattern(g, fImage, view);
}

/**
 * Draws a pattern background pattern by replicating an image.
 */
private void drawPattern(Graphics g, Image image, DrawingView view) {
    int iwidth = image.getWidth(view);
    int iheight = image.getHeight(view);
    Dimension d = view.getSize();
    int x = 0;
    int y = 0;

    while (y < d.height) {
        while (x < d.width) {
            g.drawImage(image, x, y, view);
            x += iwidth;
        }
        y += iheight;
    }
}
```

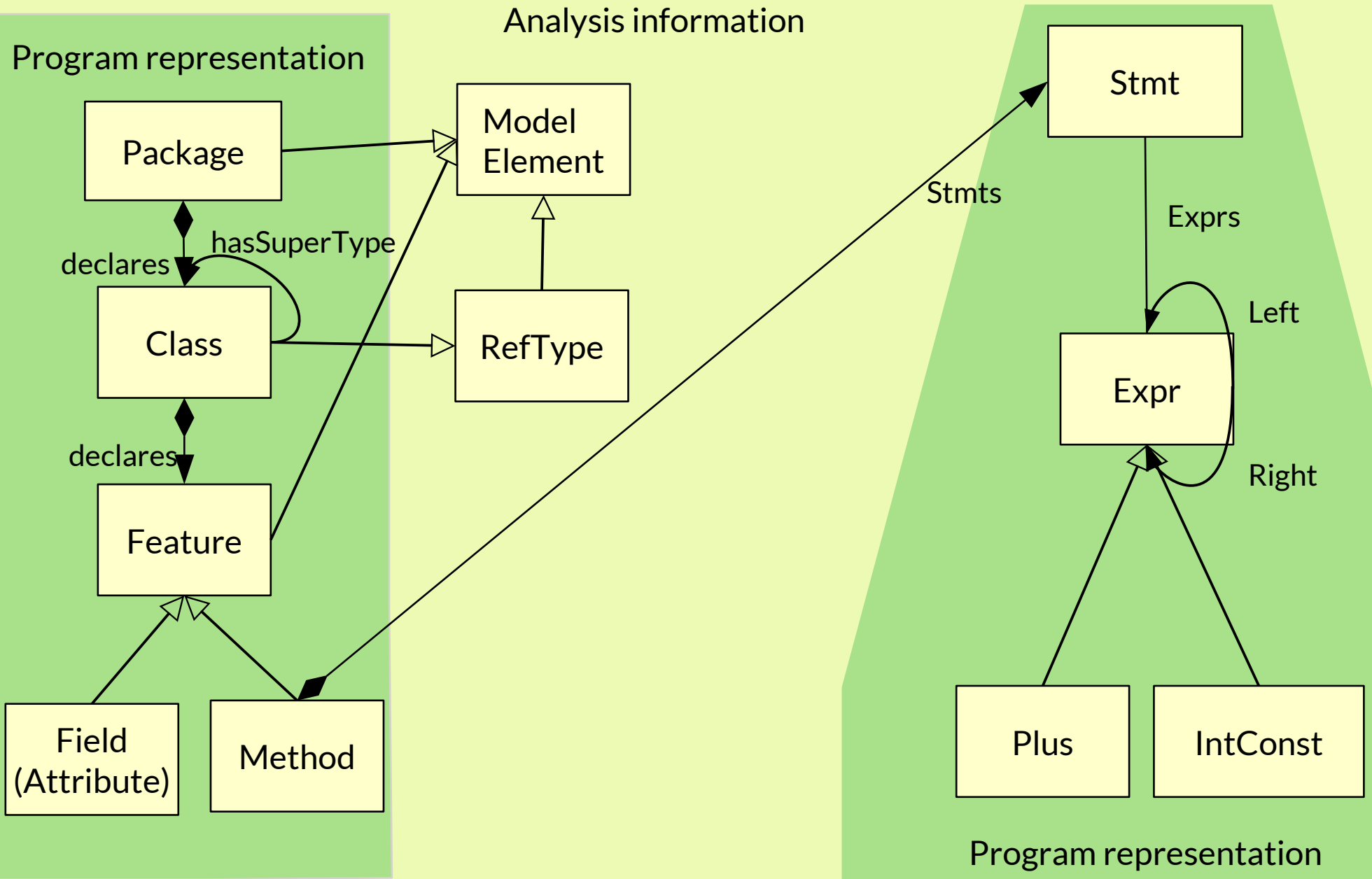
The bottom status bar shows "Problems", "Javadoc", "Declaration", "Search", "Console", and "Progress".



# A Simple Model (Schema) of Semmle-DDL in EMOF

37

Model-Driven Software Development in Technical Spaces (MOST)



- ▶ Query examples:
  - Select Statements on classes, methods, statements, expressions
- ▶ Language features:
  - Local Variables in queries
  - Non-deterministic methods returning sets and streams
  - Casts
  - Chaining
  - User defined query classes
- ▶ Metric examples:
  - Aggregation functions
  - SLOC
  - #Methods
- ▶ Expressions:
  - FROM <classes> WHERE <conditions> SELECT <variables>

# Select Statements (1)

- ▶ Find all classes *c* implementing `compareTo`, but do not overwrite `equals`
- ▶ Find their packages

```
from Class c
where
    c.declaresMethod("compareTo")
    and not (c.declaresMethod("equals"))
select
    c.getPackage(), c
```

# Select Statements (2)

- ▶ Find all **main**-methods declared in a package ending with „demo“
- ▶ Also called **pattern matching**

```
from Method m
where
    m.hasName("main")
    and m.getDeclaringType().getPackage().getName().matches("%demo")
select
    m.getDeclaringType().getPackage(),
    m.getDeclaringType(),
    m
```

# Definition of New Functions and Predicates

- ▶ Definition of new **query functions** by declaring query functions/methods in a class (note: this is similar to attributions in JastAdd)
- ▶ Methods may be indeterministic, i.e., return collections of objects

```
class Classinfo {  
    Method findMethod(Class c) {  
        c.declaresMethod("sumUpBill")  
    }  
}
```

- ▶ Definition of new **predicates** as methods in a class, using a domain-specific language language extension of Java
- ▶ Testing on or-conditions:

```
predicate isJDKMethod (Method m) {  
    m.hasName("equals")  
    or m.hasName("hashCode")  
    or m.hasName("toString")  
    or m.hasName("clone")  
}
```

# Definition of New Predicates

- ▶ Use of Kleene Star for transitive closure on predicates/edges:

```
predicate upperClass(RefType down, RefType up) {  
    down.hasSupertype*(up)  
}
```

- ▶ Complicated path expressions

```
predicate inTheMiddle(RefType down, RefType middle, RefType up) {  
    down.hasSupertype*(middle) and  
    middle.hasSupertype*(up)  
}
```



# Local Variables in Queries

Find all methods calling `System.exit(...)`

Sysexit is a local variable

```
from Method m, Method sysexit, Class system
where
    system.hasQualifiedName("java.lang", "System")
    and sysexit.hasName("exit")
    and sysexit.getDeclaringType() = system
    and m.getACall() = sysexit
select m
```

# The Use of Non-deterministic Methods

- ▶ Synthesize a call graph between the methods of two packages
  - Call graph is returned as a set of tuples of (caller, callee)
- ▶ getARefType and getACallable are indeterministic, i.e., return collections of objects

```
from Package caller, Package callee
where caller.getARefType().getACallable().calls(
    callee.getARefType().getACallable())
    and caller.fromSource()
    and callee.fromSource()
    and caller != callee
select caller, callee
```

# Chaining (Multiple Source - Multiple Target Graph Reachability Problem, MSMT)

Find all Pairs (s,t) such that

- t is a direct superclass of s
- s and t are superclasses of `org.jfree.data.gantt.TaskSeriesCollection`
- and t is not `java.lang.Object`

```
from RefType tsc, RefType s, RefType t
where
    tsc.hasQualifiedName("org.jfree.data.gantt","TaskSeriesCollection")
    and s.hasSubtype*(tsc)
    and t.hasSubtype(s)
    and not(t.hasName("Object"))
select s,t
```

# .QL-Query Classes (Dynamic Classes/Sets)

- ▶ **Query classes** in .QL are sets described by special predicates and nested other predicates
  - They define “synthetic” objects and “truths” about the model
  - Their constructors define restrictions of metaclasses

```
// definition of a query class as subclass of a metaclass
```

```
class VisibleInstanceField extends Field {
```

```
  VisibleInstanceField() {
```

```
    not (this.hasModifier("private")) and
```

```
    not (this.hasModifier("static"))
```

```
  }
```

```
predicate readExternally() {
```

```
  exists (FieldRead fr |
```

```
    fr.getField()=this and
```

```
    fr.getSite().getDeclaringType()  
      != this.getDeclaringType())
```

```
  }
```

```
}
```

```
// use of a query class
```

```
from VisibleInstanceField vif
```

```
where vif.fromSource() and not
```

```
  (vif.readExternally())
```

```
select vif.getDeclaringType().getPackage(),
```

```
  vif.getDeclaringType(),
```

```
  vif
```

## 30.2.2 Metrics with .QL



# Aggregation Functions for Computing Metrics

- ▶ Compute the average number of methods per type and package
  - Other aggregation functions: count, sum, max, min, avg
- ▶ Employs „Eindhoven Quantifier Notation“ (Dijkstra et al.)
  - $C \mid \langle \text{predicate} \rangle$
- ▶ Query: „**Compute the average number of methods in all type c of a package p**“

```
from Package p
where p.fromSource()
select p, avg(RefType c |
    c.getPackage() =p |
    c.getNumberOfMethods())
```

# Aggregation Functions for Computing SLOC Metrics

- ▶ “Calculate a SLOC metrics on package “Billing” in the current compilation unit”
- ▶ Grammar rules:
- ▶ Aggr ::= aggregationFunction '('  
    localvars // FROM  
    '|' condition // WHERE  
    '|' aggregatedValue ')' // SELECT
- ▶ AggregationFunction ::= 'sum' | 'count' | 'avg' | 'max' | 'min'

```
from Package pkg  
where pkg.hasName("Billing")  
select sum(CompilationUnit comp | //FROM  
          comp.getPackage()=pkg | // WHERE  
          comp.getNumberOfLines()) // SELECT
```

# Statistics (Metrics) Uses Aggregation Functions

51

Model-Driven Software Development in Technical Spaces (MOST)

The screenshot shows the Eclipse IDE interface. On the left is the Package Explorer showing a project structure for 'org.jhotdraw.samples'. The main editor displays a SQL query using aggregation functions. Below the query, a window titled 'Average Method/Constructor Fan-In (jhotdraw)' displays a 3D bar chart. The chart shows the average fan-in for various packages, with the highest values for 'org.jhotdraw.framework' and 'org.jhotdraw.util'.

```
from Package p, float average
where p.fromSource ()
    and average = avg(Class c, Callable member
                    | c.getPackage() = p and
                    | member.getDeclaringType() = c
                    | member.getFanIn())
select p, average order by average desc
```

**Average Method/Constructor Fan-In (jhotdraw)**

Package	Average Fan-In
org.jhotdraw.applet	1.1
org.jhotdraw.application	1.4
org.jhotdraw.contrib	1.1
org.jhotdraw.contrib.dnd	1.1
org.jhotdraw.contrib.html	0.8
org.jhotdraw.contrib.zoom	1.0
org.jhotdraw.figures	1.2
org.jhotdraw.framework	2.5
org.jhotdraw.samples.javadraw	0.6
org.jhotdraw.samples.minimap	0.4
org.jhotdraw.samples.net	0.7
org.jhotdraw.samples.nothing	0.3
org.jhotdraw.samples.pert	0.8
org.jhotdraw.standard	1.6
org.jhotdraw.util	2.5
org.jhotdraw.util.collections.jdk11	0.2
org.jhotdraw.util.collections.jdk12	0.1





## 30.2.3. Graph Querying with GReQL

- ▶ Open source, from University of Koblenz-Landau, Prof. Ebert
- ▶ Applicable to a subset of UML (GrUML)



# TGreQL is similar to .QL

- ▶ But uses a relational notation, from-with-report clauses

```
from RefType tsc, RefType s, RefType t
```

```
where
```

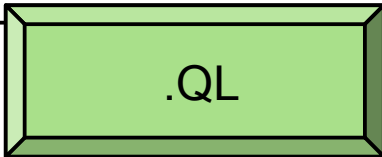
```
  tsc.hasQualifiedName("org.jfree.data.gantt", "TaskSeriesCollection")
```

```
  and s.hasSubtype*(tsc)
```

```
  and t.hasSubtype(s)
```

```
    and not(t.hasName("Object"))
```

```
select s,t
```

.QL

```
from RefType tsc, RefType s, RefType t
```

```
with
```

```
  s hasSubtype*->tsc,
```

```
  tsc.hasQualifiedName("org.jfree.data.gantt", "TaskSeriesCollection"),
```

```
  t hasSubtype->s,
```

```
    not t.hasName("Object")
```

```
report s,t
```

TGreQL

# The Query Language TGreQL

- ▶ TgreQL style is very similar to Xcerpt
- ▶ Implements F-Datalog incl. Transitive closure operator
- ▶ Prof. J. Ebert U Koblenz

## Operators:

- \* Transitive closure operator
- + positive transitive closure
- → ← navigation direction
- [] optional path
- ( ) sequence of paths or edges
- | alternative path

```
// construct a call graph
From caller, callee: V{Method}
With caller (
    {isStatementIn}
    [ {isReturnValueOf} ]
    {isActualParameterOf} *
    {isCalleeOf}
) + callee
Report
    caller.name as „Caller“
    callee.name as „Callee“
```

## Result (example):

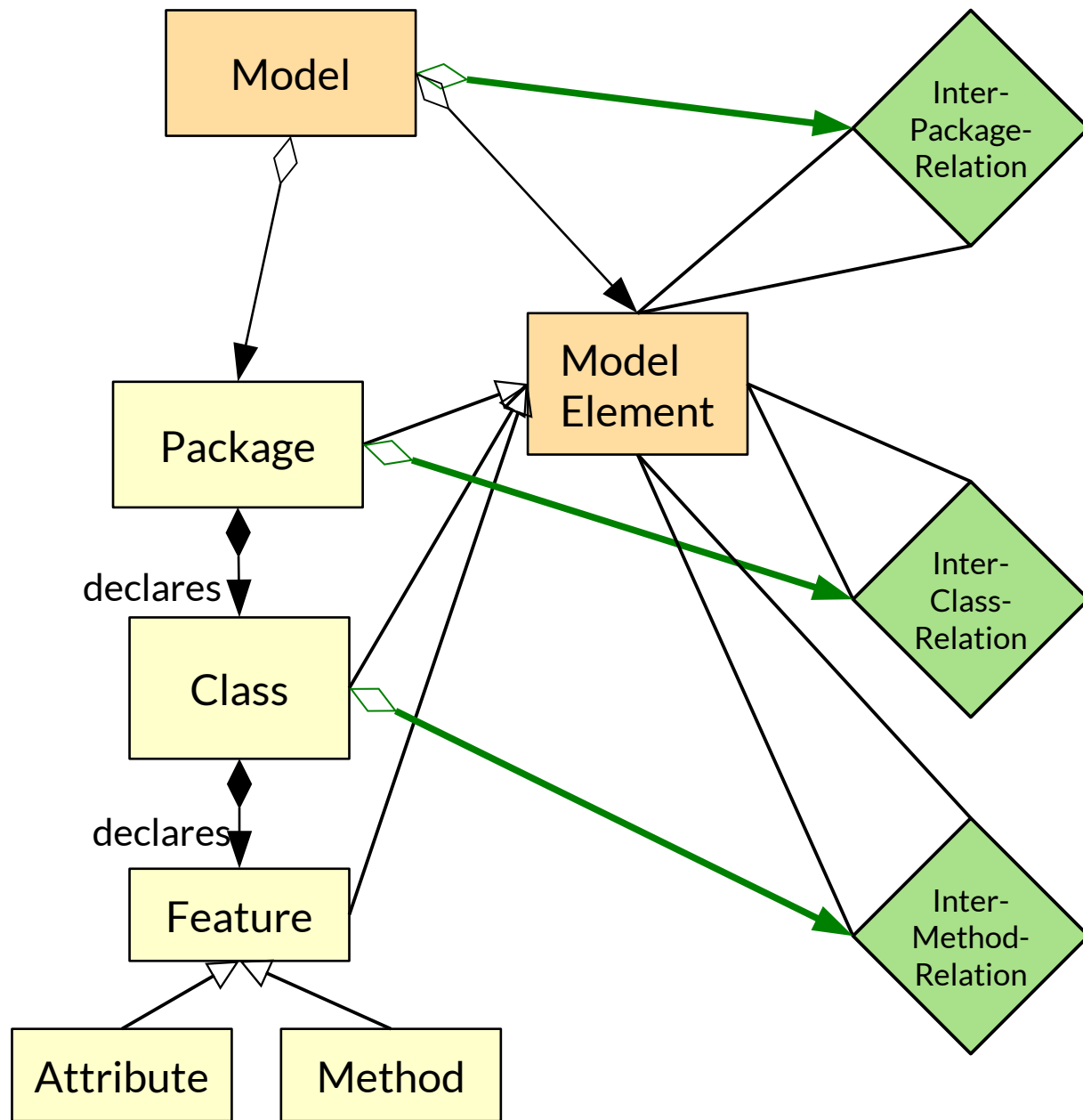
Caller	Callee
main	System.out.println
main	compute
main	twice
main	add
compute	twice
compute	add

## 30.2.4. Lifting Information Up the Containment Hierarchy



# Lifting Information Along the Block Containment Structure (Scope Structure)

- ▶ Languages are block-structured, i.e., live in a **containment hierarchy**.
- ▶ A model has **model elements**
- ▶ A class has **inter-method relationship** (e.g., the call graph)
- ▶ A package has **inter-class relationships** between these model elements
- ▶ A model has **inter-package relationships** between these model elements

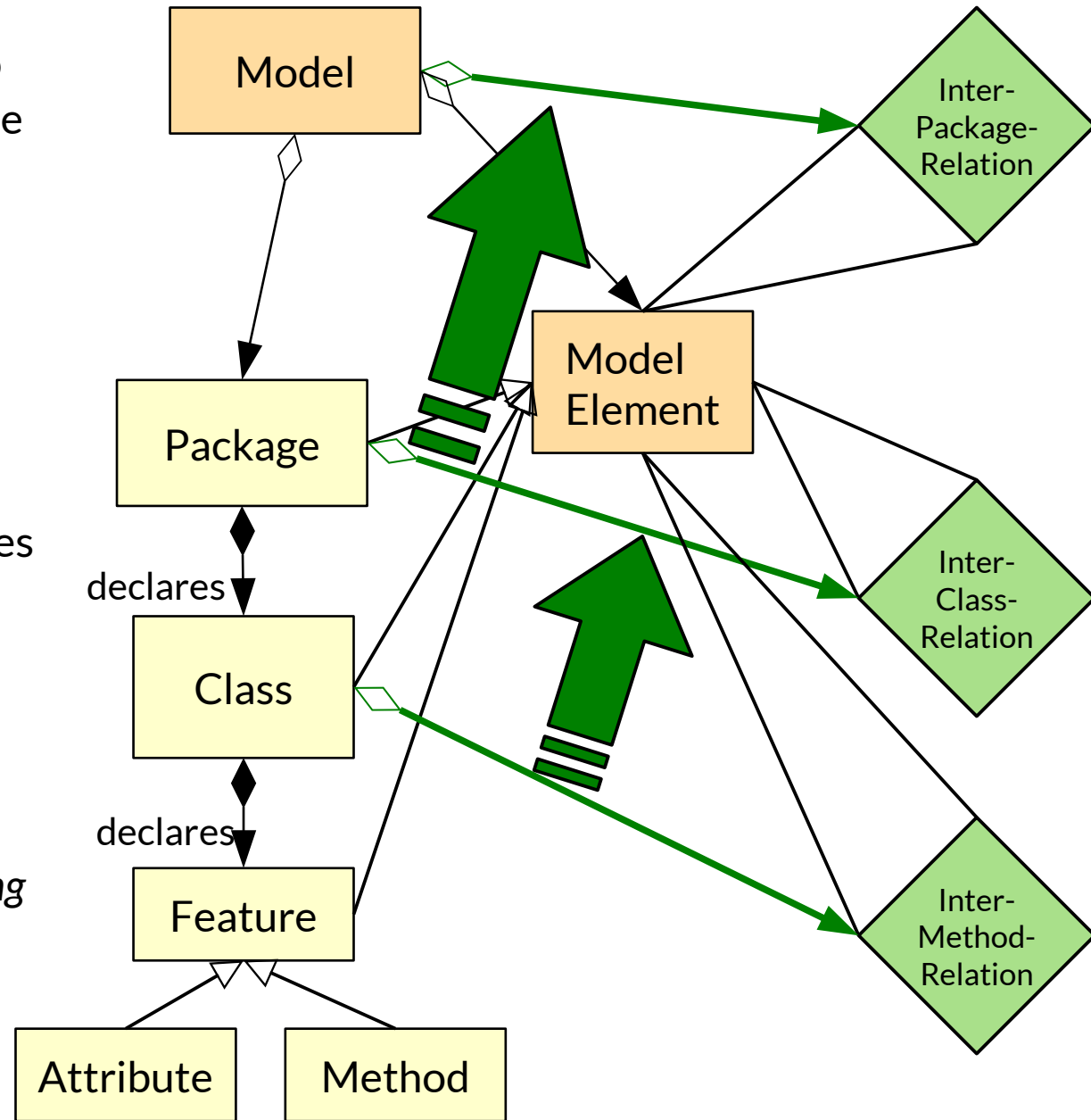


**A macromodel builds on graphs, at least on link trees, no longer on trees**



# Lifting Information Along the Block Containment Structure

- ▶ **Dependency lifting** means to lift information **up** in along the containment hierarchy
  - from an inter-method relationship to a inter-class relationship
  - from an inter-class relationship to a inter-package relationship
- ▶ Dependency lifting propagates information **up** the abstract syntax tree and the containment tree
- ▶ Dependency lifting is an important process to *summarize dependencies among siblings in containment hierarchies in models*



# Dependency Lifting Information Along the Block Containment Structure

- ▶ **Dependency lifting** lifts dependency information up the containment structure in a model, thereby summarizing the dependencies at the level of the model
- ▶ result is implicitly defined default return parameter

```
// Lifting a pair of method dependencies
// on a pair of classes
class Method {
  Class getDependentClass() {
    exists (Method m |
      depends(this.getClass(),m)
      and result = m.getClass()
    )
    and result != this
  }
}
```

```
// Lifting a pair of class dependencies to
// a pair of packages
class Class {
  Package getDependentPackage() {
    exists (Class cl |
      depends(this.getPackage(),cl)
      and result = cl.getPackage()
    )
    and result != this
  }
}
```

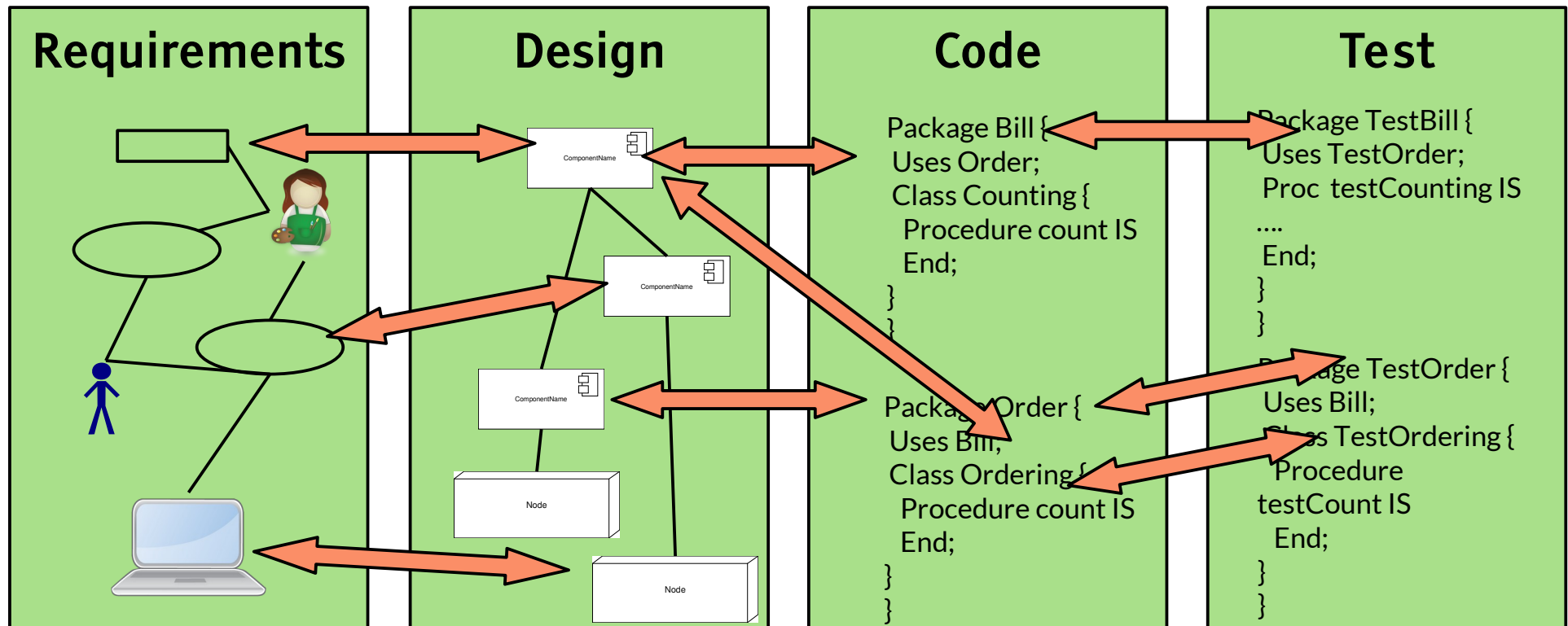
## 30.3 Macromodel Dependency Analysis

- Remember: A **macromodel** is a multimodel with consistent dependencies



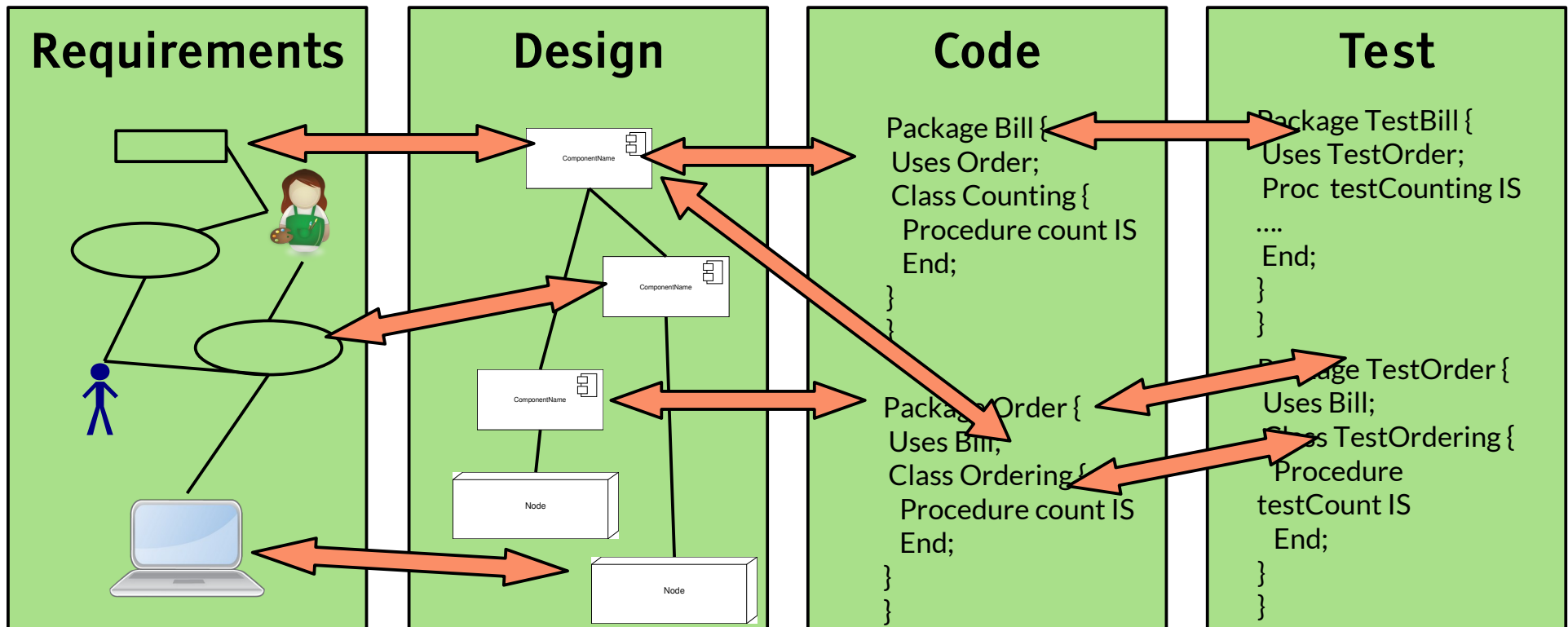
# Q12: The ReDeCT Problem and its Macromodel

- ▶ The **ReDeCT problem** is the problem how requirements, design, code and tests are related (→ V model)
- ▶ Mappings between the Requirements model, Design model, Code, Test cases
- ▶ A **ReDeCT macromodel** has maintained mappings between all 4 models



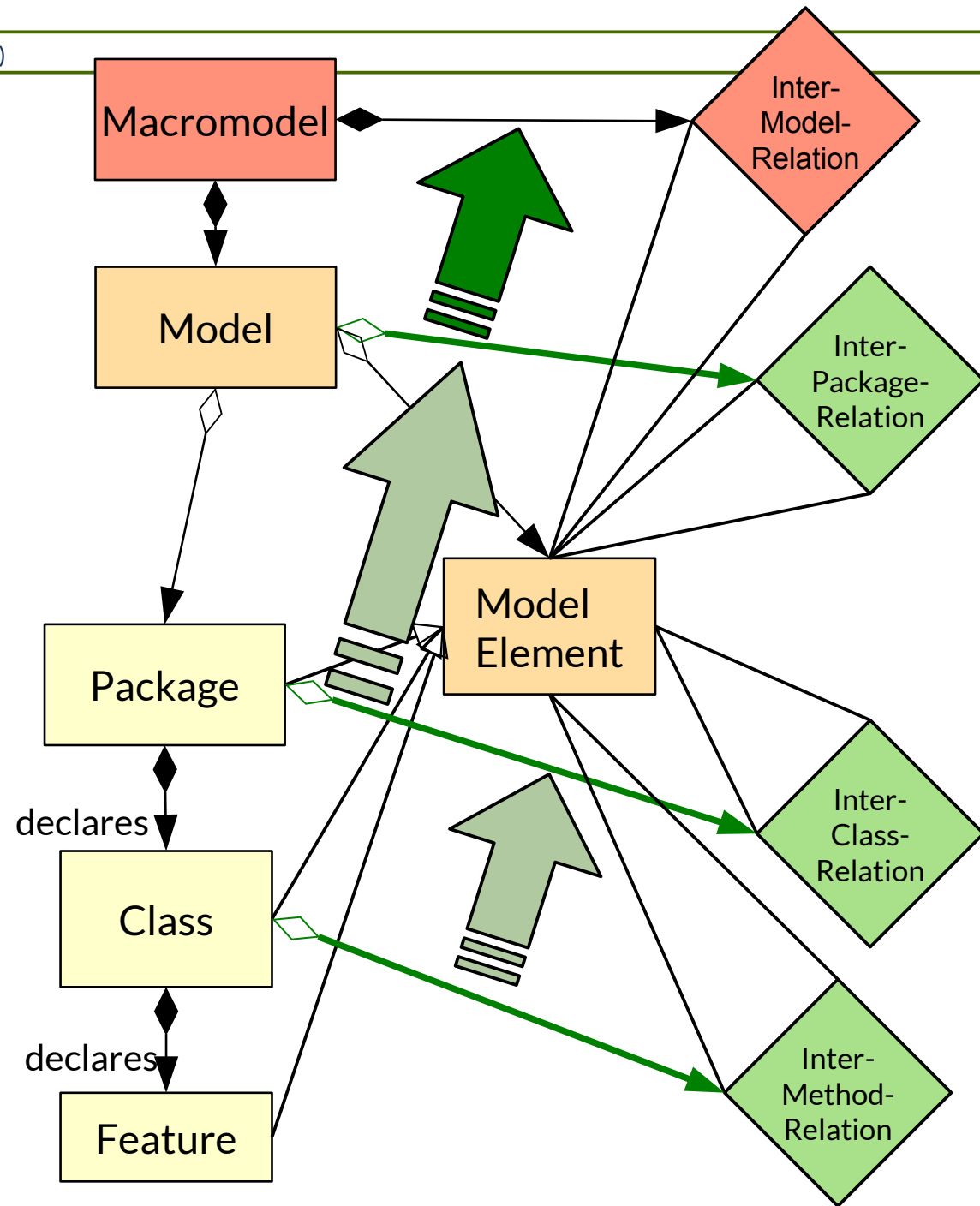
# Inter-Model Relationships in The ReDeCT Macromodel

- ▶ An **inter-model relationship** is a relationship between model elements of different models (usually link or graph relationship)
  - Here: expresses mapping between the Requirements model, Design model, Code, Test cases
- ▶ The **ReDeCT macromodel** relies on inter-model relationships between all 4 models



# Lifting Information Along the Block Containment Structure Between Models in the Macromodel

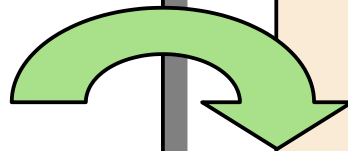
- ▶ **Macromodel-Dependency**  
Lifting means to lift information **up** in along the containment hierarchy **from between the packages of a model to between the models of the macromodel**
  - from an intra-model relationships to a inter-model relationship
- ▶ Megamodel-Dependency- Lifting propagates information **up** into the megamodel
- ▶ Megamodel-Dependency- Lifting is an important process to *summarize dependencies among models*
- ▶ Result: a **macromodel**



# Megamodel Dependency Lifting in Semmle .QL

- ▶ The lifting procedure also works for lifting package dependencies within a model to model dependencies.
  - Consider models as “normal” objects in the repository
  - Formulate queries about model-element relationships and lift them to model relationships

```
// Lifting a pair of class dependencies to  
// a pair of packages  
class Class {  
  Package getDependentPackage() {  
    exists (Class cl |  
      depends(this.getPackage(),cl)  
      and result = cl.getPackage()  
    )  
    and result != this  
  }  
}
```



```
// Lifting a pair of package  
dependencies to  
// a pair of models  
class Package {  
  Model getDependentModel() {  
    exists (Model mod |  
      depends(this.getModel(),mod)  
      and result = mod.getModel()  
    )  
    and result != this  
  }  
}
```

# How to Discover Dependencies Between Models in a Multimodel

- ▶ After analysis of all models, **lift the information up the containment hierarchy into the multimodel**
  - Construct inter-model relationships by lifting from inter-package relationships
- ▶ This turns the multimodel into a *macromodel*, because a multimodel with model-element constraints is called a macromodel
- ▶ The lifted dependencies allow for discovering dependencies between models in a multimodel
  - The precise detailed dependencies give tracing to update models in a multimodel, if something changes

**Macromodel dependency analysis** consists of lifting model-level dependency analysis to inter-model relationships

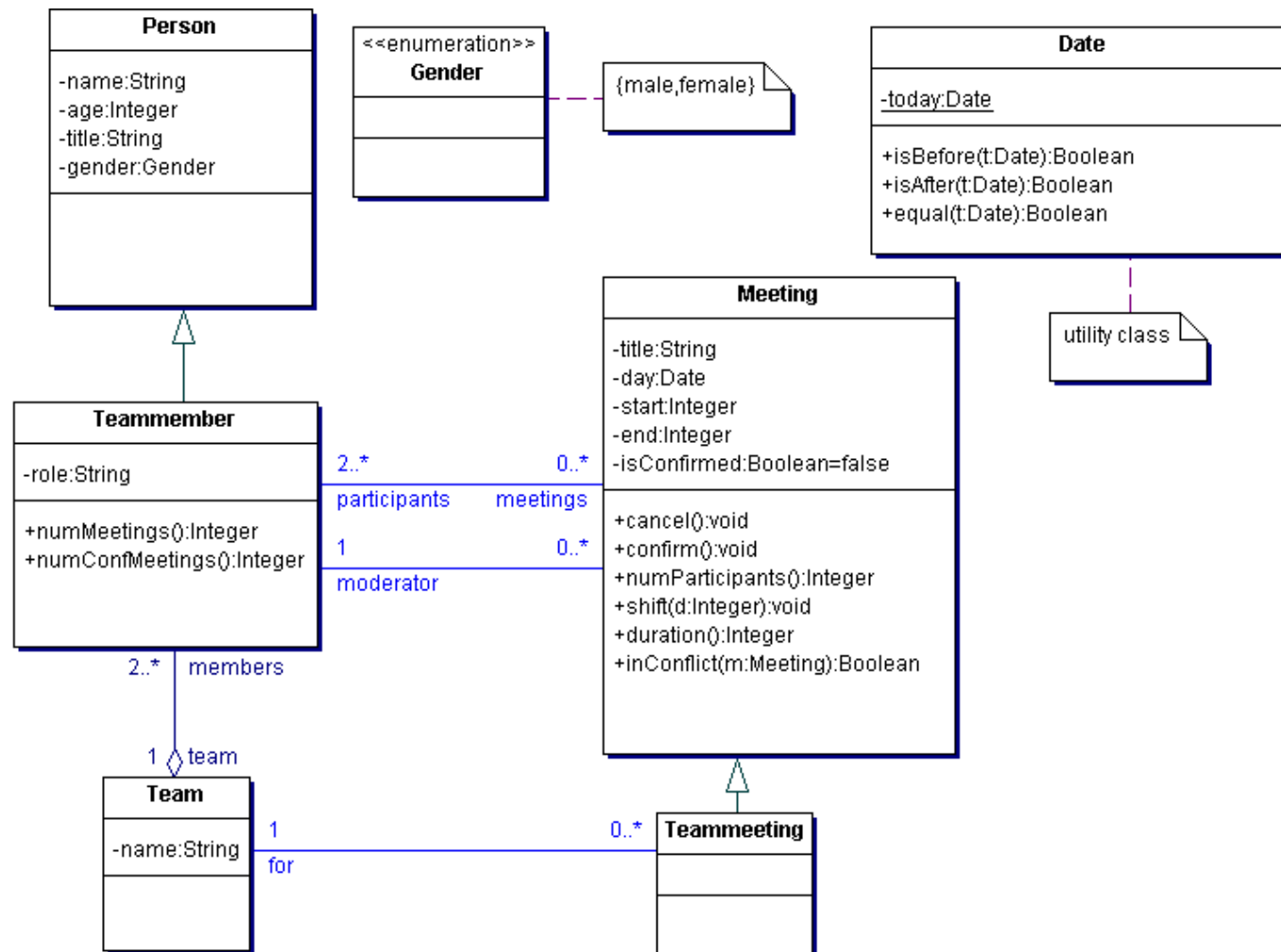
**Macromodel consistency** consists of updating all inter-model relationships and all induced model-level dependencies

## 30.4. Writing Model Constraints by Graph Querying with OCL

- The DDL of OCL is MOF
- .QL is for Java and other GPL
- OCL is for UML-CD

# OCL for Invariants in UML-Class Diagrams

► → course Softwaretechnologie-II



# Examples OCL Invariants

- ▶ OCL queries usually start at a specific class; their results define *invariants* on the objects of the class
  - All attributes of a class are visible by default in OCL.
  - Relations between classes define functions
- ▶ Query language uses expressions over these functions

Example of Invariant:

```
context Meeting inv: self.end > self.start
```

Equivalent:

```
context Meeting inv: end > start
```

```
-- self is the context of the query, from which processing starts
```

Equivalent named constraint:

```
context Meeting inv startEndConstraint:
```

```
self.end > self.start
```

```
-- Constraints can constrain attribute values
```

- ▶ FROM and SELECT clauses are modeled via functions:

**Selection constraint:**

```
context Person inv searchForPerson:
```

```
allInstances() ->select (p:Person | p.name.StartsWith („Uwe“))
```

```
-- FROM clause is modeled via allInstances() function
```

```
-- SELECT clause is modeled via select() function
```



# Examples OCL Invariants

- ▶ **Selection constraint:**

```
context Person inv searchNames:
```

```
allInstances() ->collect(name)
```

```
context Person inv countNames:
```

```
allInstances() ->collect(name) ->size()
```

- ▶ **Multiplicity constraint:**

```
context Person inv countNames:
```

```
allInstances() ->collect(name) ->size() < 15
```

- ▶ More on OCL: → Course Softwaretechnologie-II, Ch. “Konsistenzprüfung mit OCL”, Dr. Birgit Demuth
- ▶ [Www.dresden-ocl.de](http://www.dresden-ocl.de)

# The End

- ▶ Why does ERD and MOF help to define link-consistent link trees?
- ▶ Explain why TgreQL and Xcerpt have similar query styles
- ▶ Why does a megamodel usually build on graphs, not on trees?
- ▶ Why do we need graph query and transformation languages?

## 30.4.2 Model Mappings with Query-View-Transformations (QVT)

The language of the OMG for model transformations within MDA

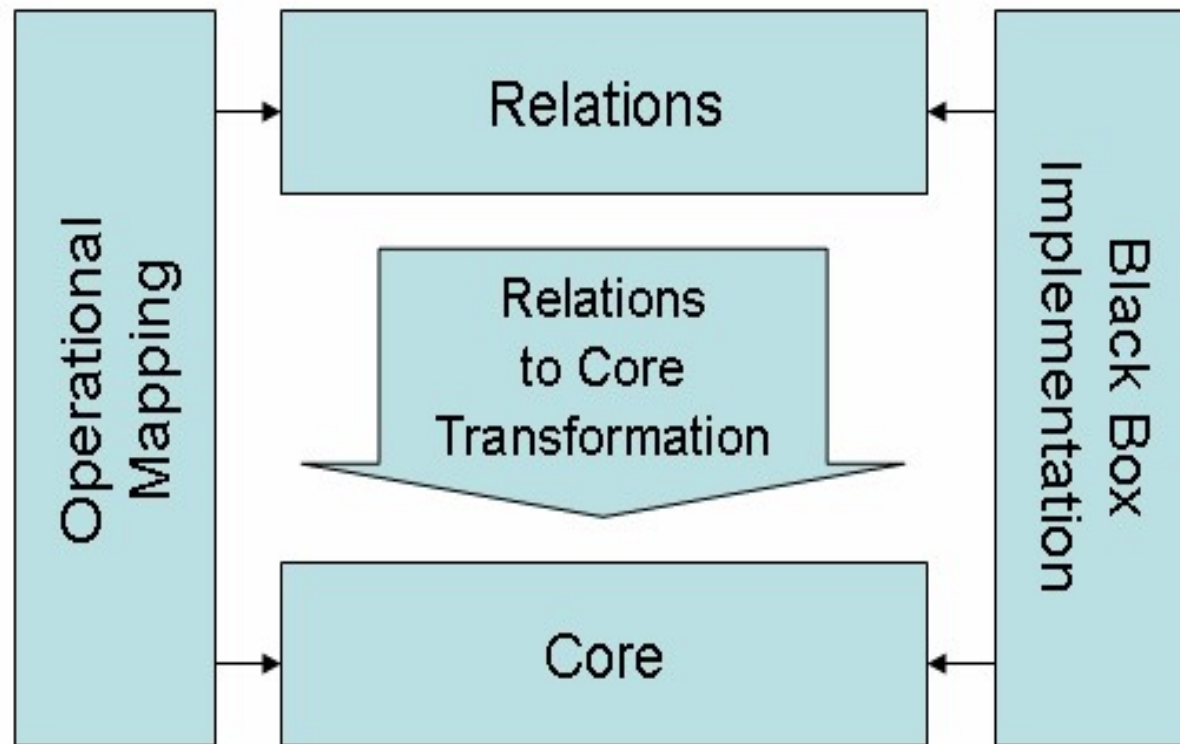
OMG: MOF 2.0 Query / Views / Transformations RFP. ad/2002-04-10. Needham, MA: Object Management Group, April 2002.

<http://www.omg.org/cgi-bin/doc?ad/2002-4-10>



DRESDEN  
concept  
Exzellenz aus  
Wissenschaft  
und Kultur

# QVT Dialects



From: [[https://de.wikipedia.org/wiki/Datei:QVT-Language-Architecture\\_591x387.jpg](https://de.wikipedia.org/wiki/Datei:QVT-Language-Architecture_591x387.jpg)]

# Transitive Closure with QVT Relations

- ▶ **QVT relations** uses logic expressions on base and derived relations (graph-logic isomorphism)

```
// Transitive Closure in QVT relations,  
// Modeled with recursive relation  
"transitiverelation"
```

```
relation transitiverelation {  
  domain node:Node {  
    // matching attributes  
    name = sameName;  
  }  
  domain node2:Node {  
    // node2 must have the  
    // same name as node  
    name = sameName;  
  }  
  domain node3:Node {  
    // node3 must also  
    // have the same name  
    name = sameName;  
  }  
}
```

```
when {  
  // conditions: base relation must exist  
  baserelation(node,node2) or  
  // or a transitive relation to a base relation  
  (transitiverelation(node,neighbor)  
  and baserelation(neighbor,node2));  
}  
where { // Aufruf einer Transformation  
  makeNodeSound(node);  
}  
}
```

# QVT Tools

Tool			
Eclipse M2M Project	Operational	<a href="http://www.eclipse.org/m2m/">http://www.eclipse.org/m2m/</a>	
Magic Draw	Operational		
MediniQVT	Relational	<a href="http://projects.ikv.de/qvt/wiki">http://projects.ikv.de/qvt/wiki</a>	



# QVT-R uses OCL for Model Search, Query, and Mapping

- ▶ OCL can be called within QVT scripts
  - Two different DQL are combined within a single language

```
// this is QVT
rule checkNoDoubleFeatureInSuperClasses(name:String) {
  from node: Class (
    -- OCL query
    node->TransitiveClosure()->collect().exists(s | s.name() = name);
  )
  to
  System.out.println("Error: super class has doubly defined feature:
+s.name());
}
```

## 30.4.3. Graph Invariant Specification with Spider Diagrams





# Spider Diagrams

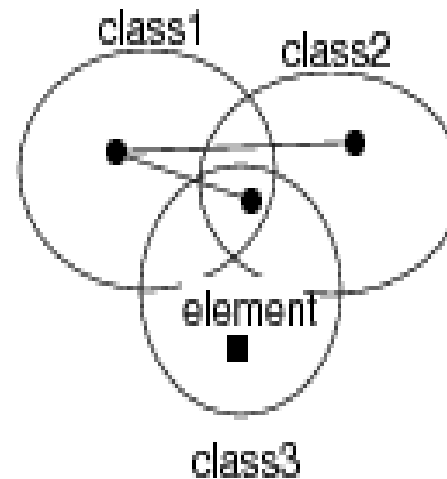
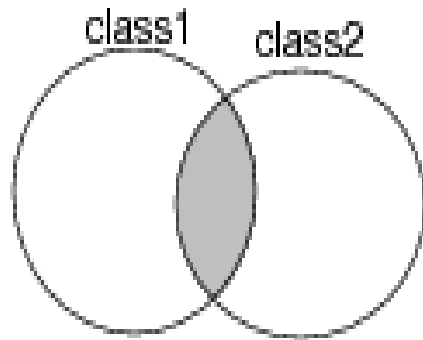
- ▶ [http://en.wikipedia.org/wiki/Spider\\_diagram](http://en.wikipedia.org/wiki/Spider_diagram)
- ▶ S. Kent. Constraint Diagrams: Visualizing Invariants in OO Modelling. Proceedings of OOPSA 97, ACM Press, Oct. 97, pp. 327-341.
- ▶ S. Kent and J. Howse. Mixing Visual and Textual Constraint Languages, UML 99, IEEE press, Oct 1999.
- ▶ Spider-Diagramme are equivalent to monadic second-order logic 2. Stufe (MSOL).
  - They include OCL (first-order logic)
- ▶ Source of diagrams: J. Lövdahl, Towards a Visual Editing Environment for the Semantic Web. Linköpings universitet, 2002.

# Simple Spider Diagrams are Extended Venn Diagrams

- ▶ Classes are visualized as venn ellipsoids
- ▶ Set algebra is expressed by intersection of ellipsoids
- ▶ Existential Logic (propositional logic with existential quantifiers) is expressed by **spiders** (hyperedges)

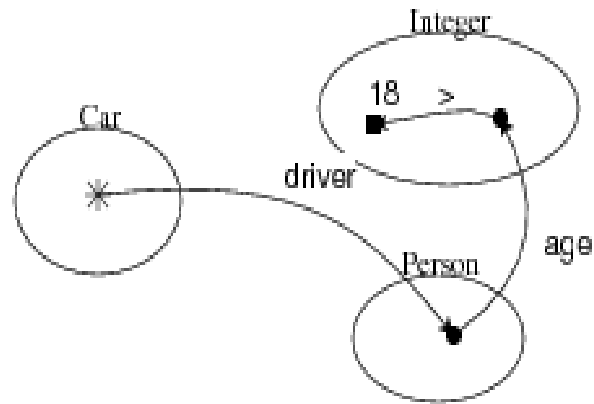
Result =  
 $class1 \wedge class2$

An object of class1 has an object of class2  
and an object in  $class1 \wedge class2 \wedge class3$   
and  $class3 \setminus class1 \setminus class2$  is not empty

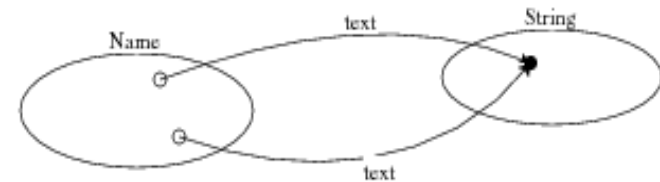


- ▶ All quantifiers are possible (star symbol)

All cars must be driven  
by a person older than 18

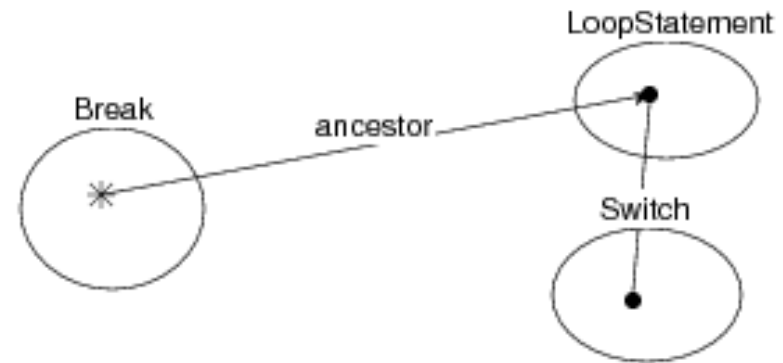


There are no two names that have the same string

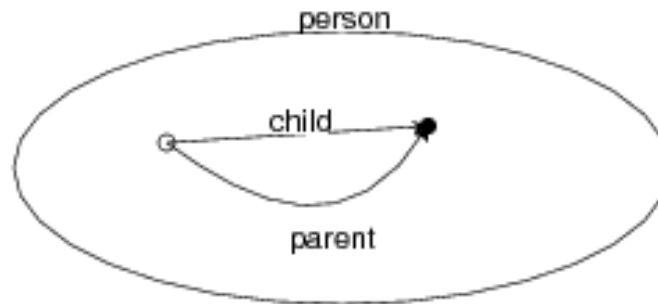


# Other constraints

All Break statements must have a LoopStatement as ancestor, which is related to a Switch state

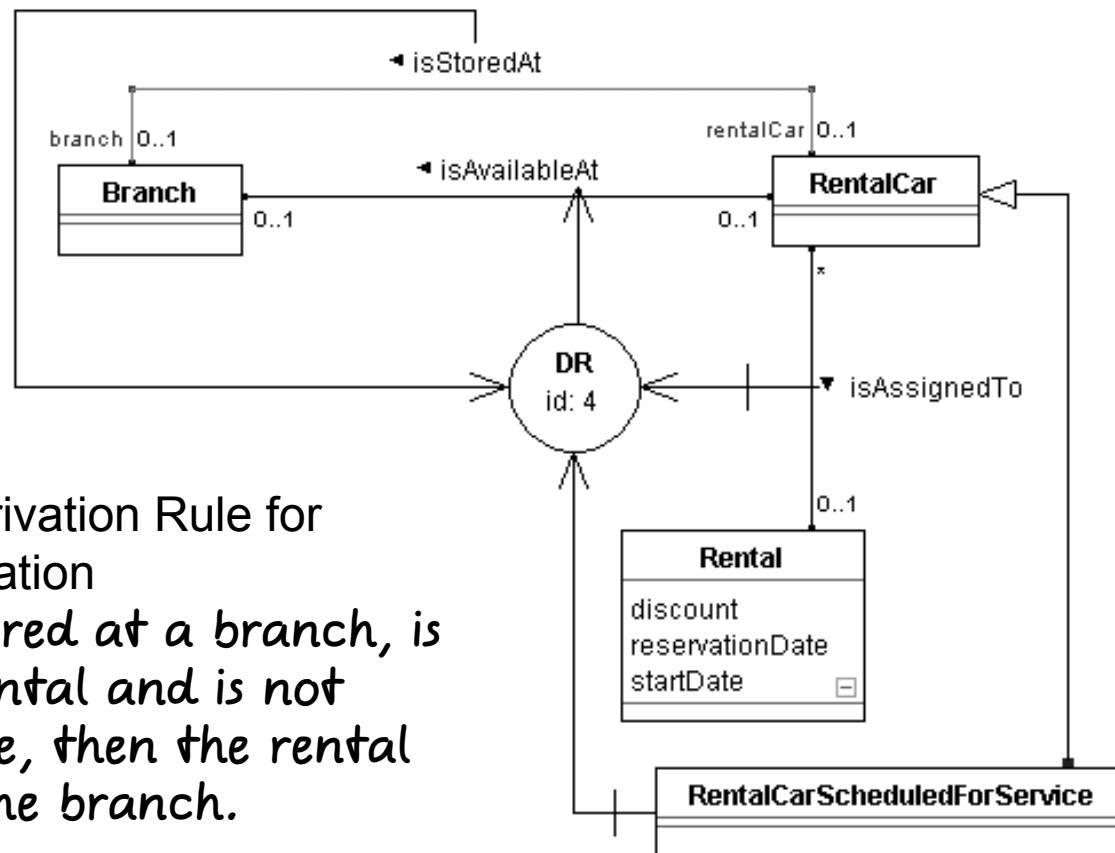


For every person, there is no child that has no parent



## 30.4.4. URML – A UML-like Spider Notation

- ▶ URML <http://oxygen.informatik.tu-cottbus.de/reverse-i1/?q=URML>
- ▶ Emilian Pascalau and Adrian Giurca. Can URML model successfully Drools rules? Proceedings of the 2nd East European Workshop on Rule-Based Applications (RuleApps 2008) at the 18th European Conference on Artificial Intelligence. Patras, Greece, July 23, 2008.
  - <http://ceur-ws.org/Vol-428/paper5.pdf>

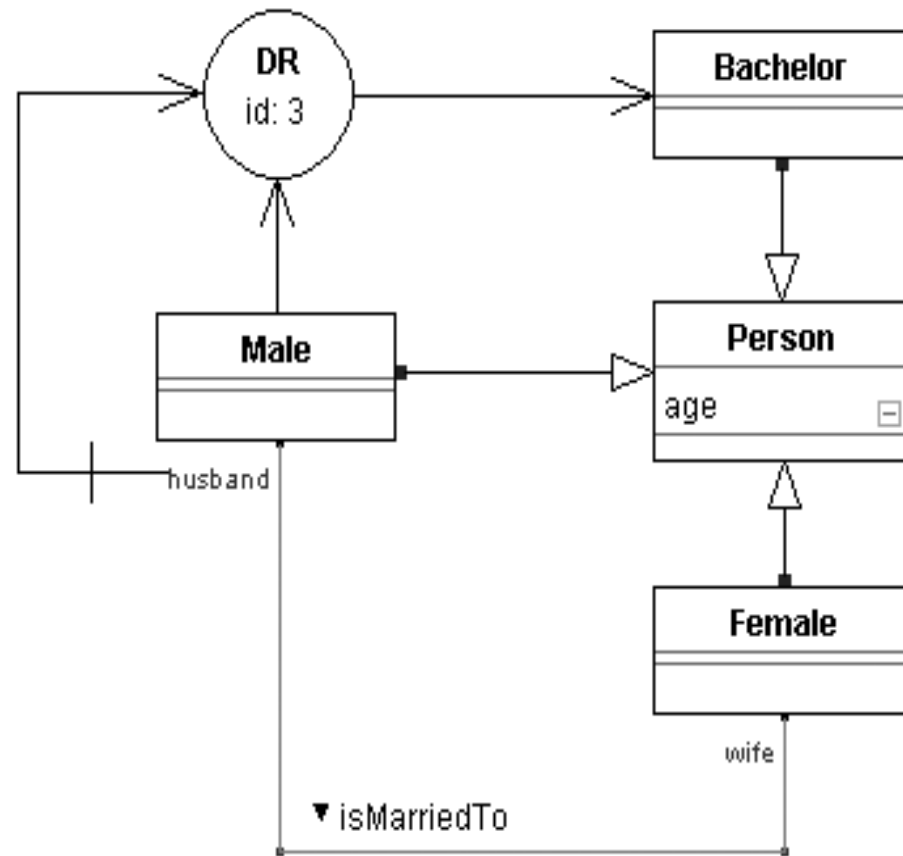


- ▶ Ex: Modeling a Derivation Rule for Defining an Association

*If a rental car is stored at a branch, is not assigned to a rental and is not scheduled for service, then the rental car is available at the branch.*

# Modeling a Derivation Rule with a Role Condition

*A bachelor is a male that is not a husband.*



# The End

- ▶ Why does ERD and MOF help to define link-consistent link trees?
- ▶ Explain why TgreQL and Xcerpt have similar query styles
- ▶ Why does a megamodel usually build on graphs, not on trees?
- ▶ Why do we need graph query and transformation languages?

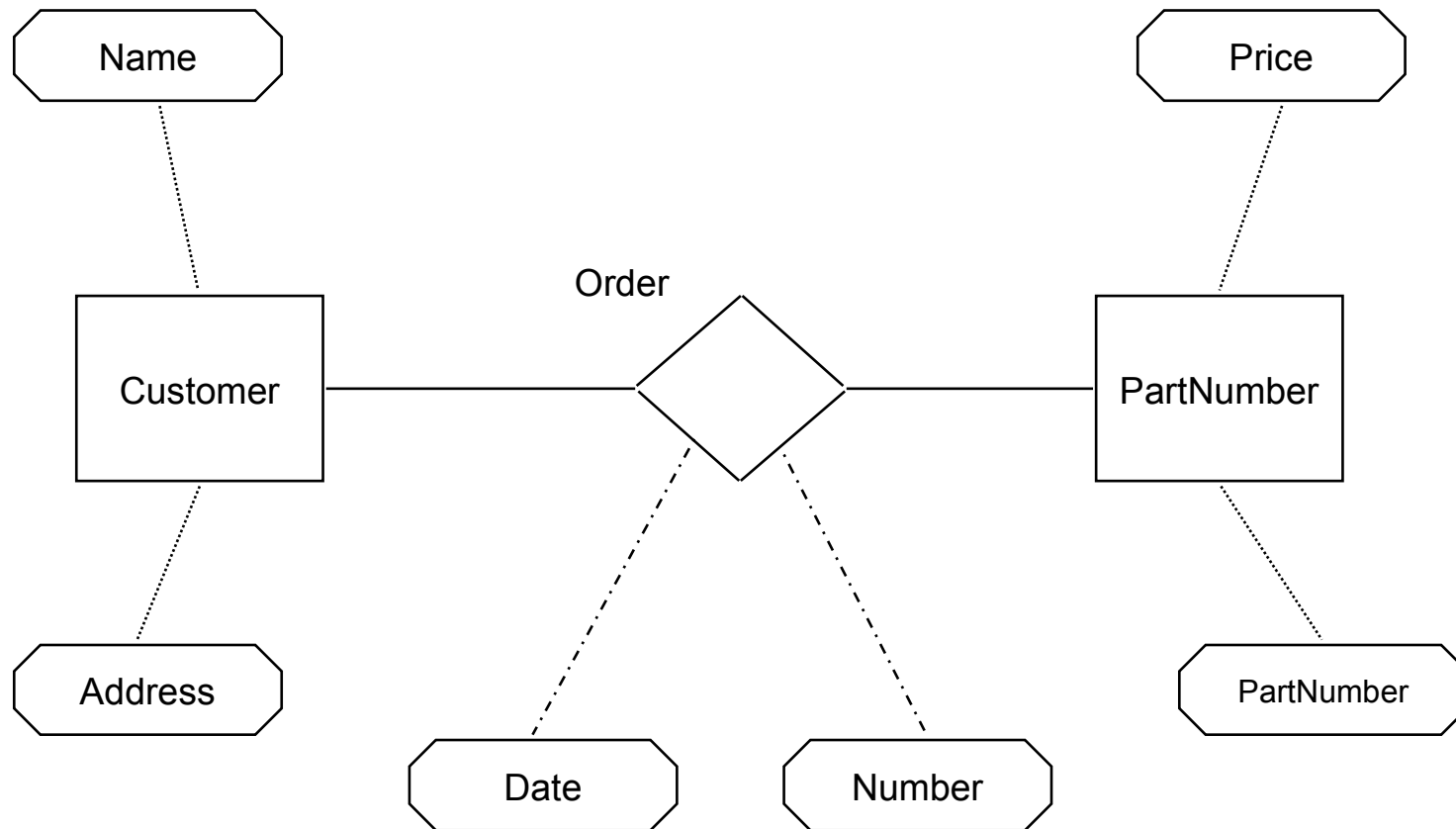


# Appendix


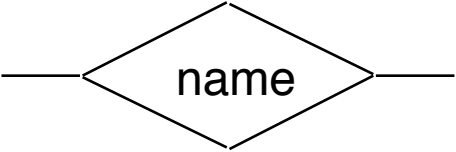



# A Simple ER-Model

- ▶ All “entities” (classes) are represented as “entity-”tables



# ERD Model Elements [Chen]

Notation	Meaning
	<b>Entity type:</b> Set of objects
	<b>Relationship type:</b> Set of relations between entity types
	<b>Attribute:</b> Describes a function or a predicate over an entity
1, n 0 < n	<b>Cardinality</b> of a relationship type: minimum and maximum amount of neighbors in a relation