Modellbasierte Softwareentwicklung (MODSOFT)

Part II

Domain Specific Languages

Notations and Representations (Introductions & xText)

Prof. Joachim Fischer /
Dr. Markus Scheidgen / Dipl.-Inf. Andreas Blunk

{fischer,scheidge,blunk}@informatik.hu-berlin.de

LFE Systemanalyse, III.310

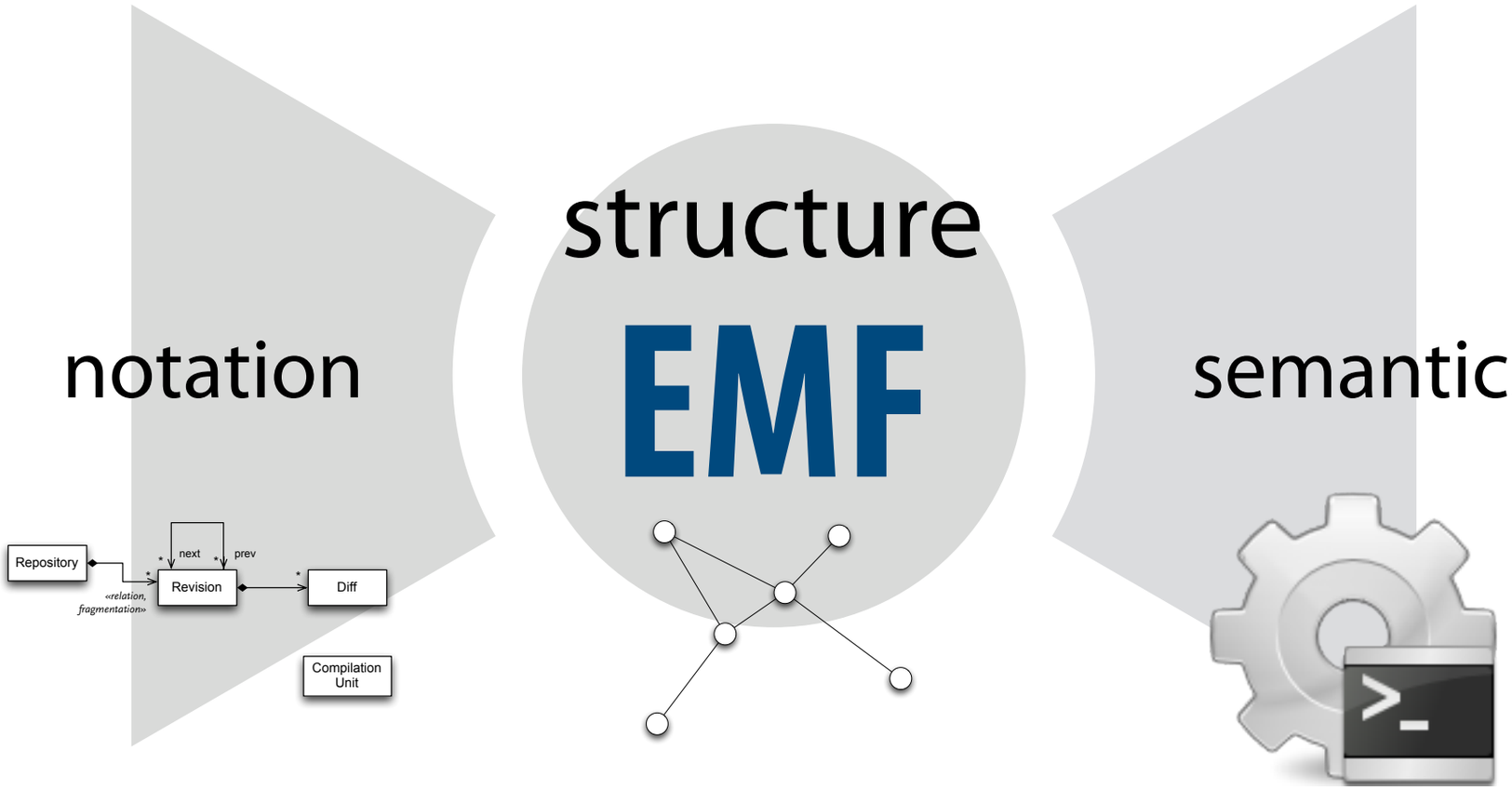
Agenda

prolog (1 VL)	**Introduction:** languages and their aspects, modeling vs. programming, meta-modeling and the 4 layer model
0. (2 VL)	**Eclipse/Plug-ins:** eclipse, plug-in model and plug-in description, features, *p2*-repositories, *RCP*s
1. (2 VL)	**Structure:** *Ecore*, *genmodel*, working with generated code, constraints with *Java* and *OCL*, *XML/XMI*
➡ 2. (3 VL)	**Notation:** Customizing the tree-editor, textural with *XText*, graphical with *GEF* and *GMF*
3. (4 VL)	**Semantics:** interpreters with Java, code-generation with *Java* and *XTend*, model-transformations with *Java* and *ATL*
epilog (2 VL)	**Tools:** persisting large models, model versioning and comparison, model evolution and co-adaption, modular languages with *XBase*, *Meta Programming System* (MPS)

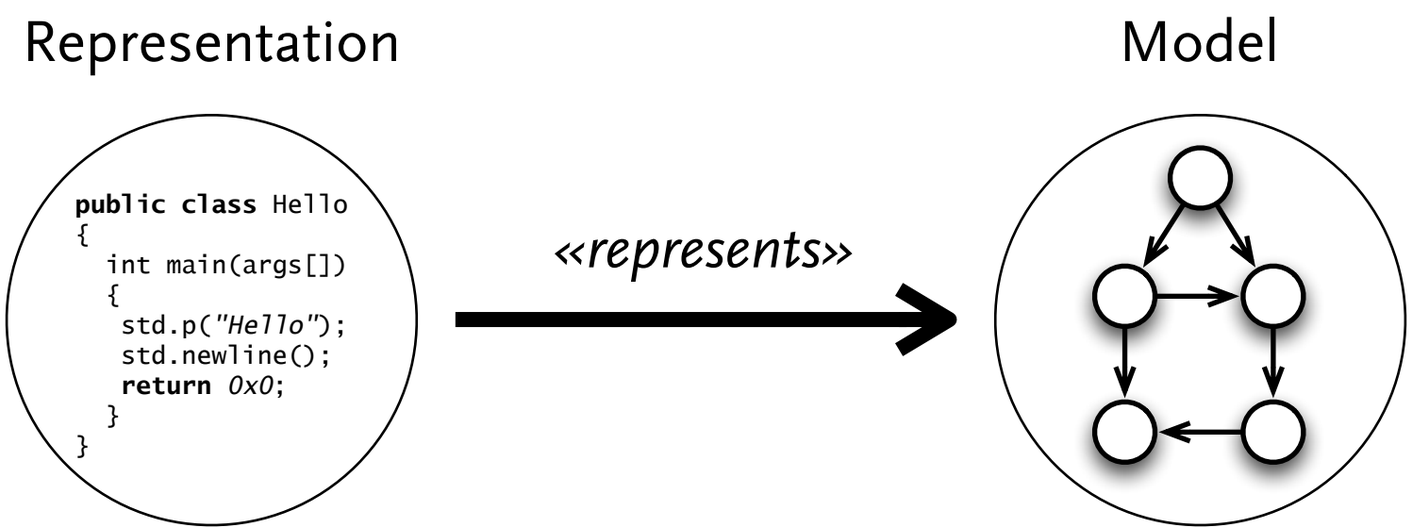
Notations and Representations

Introduction

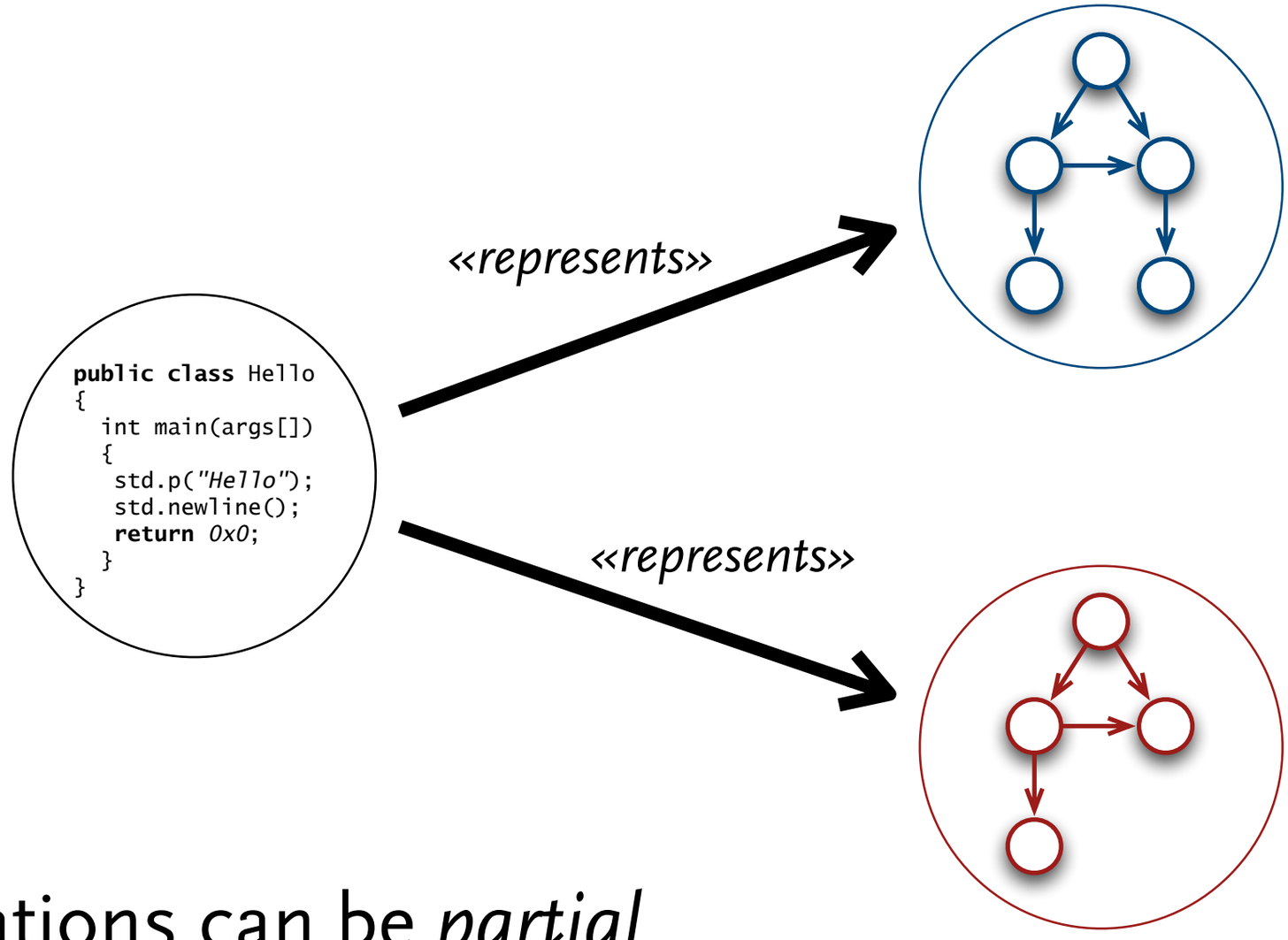
Eclipse Modeling Framework



Representation

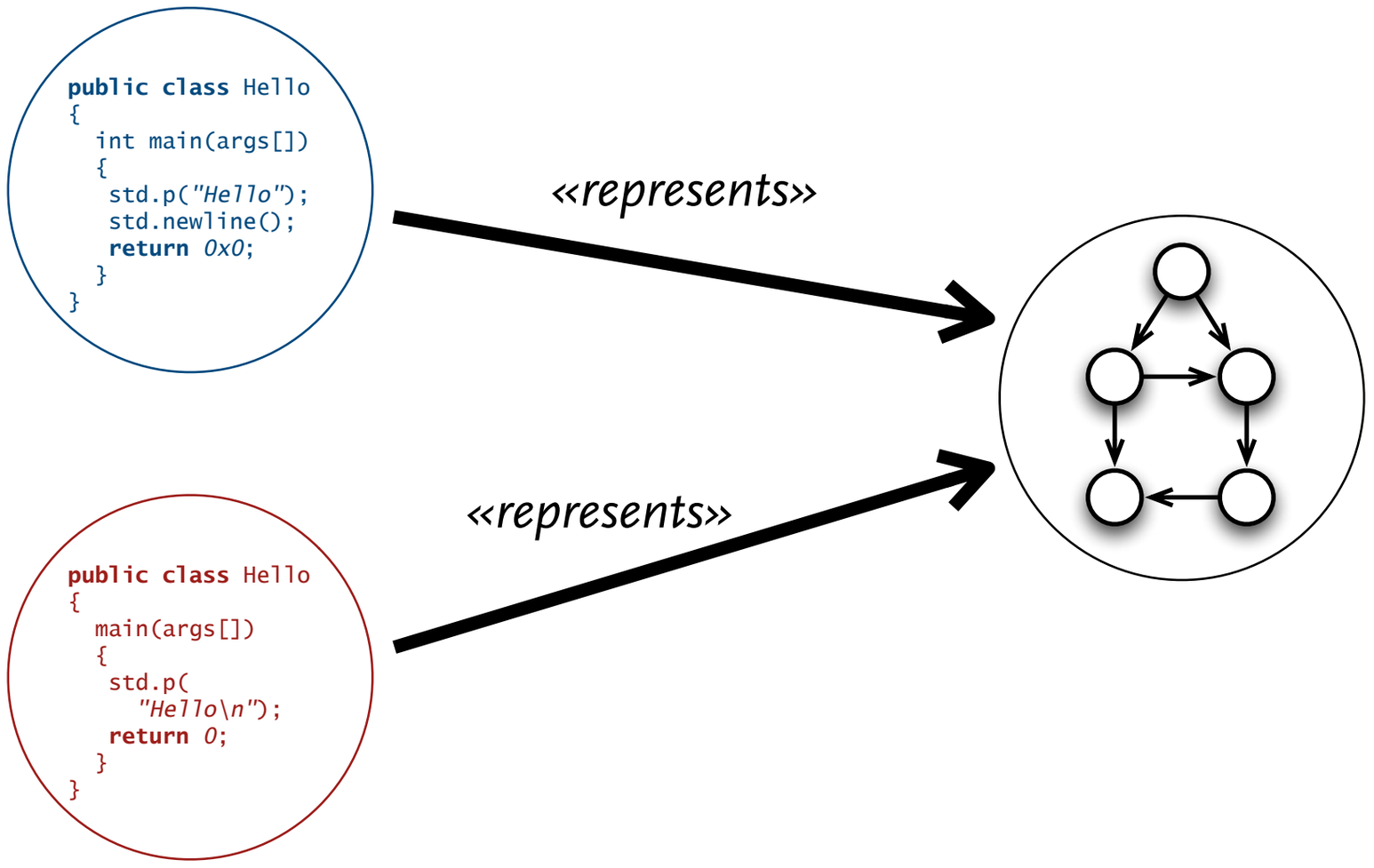


Ambiguous Representation



- representations can be *partial*
- representations can be *abstract*
- can be **ambiguous**

Multiple Representations

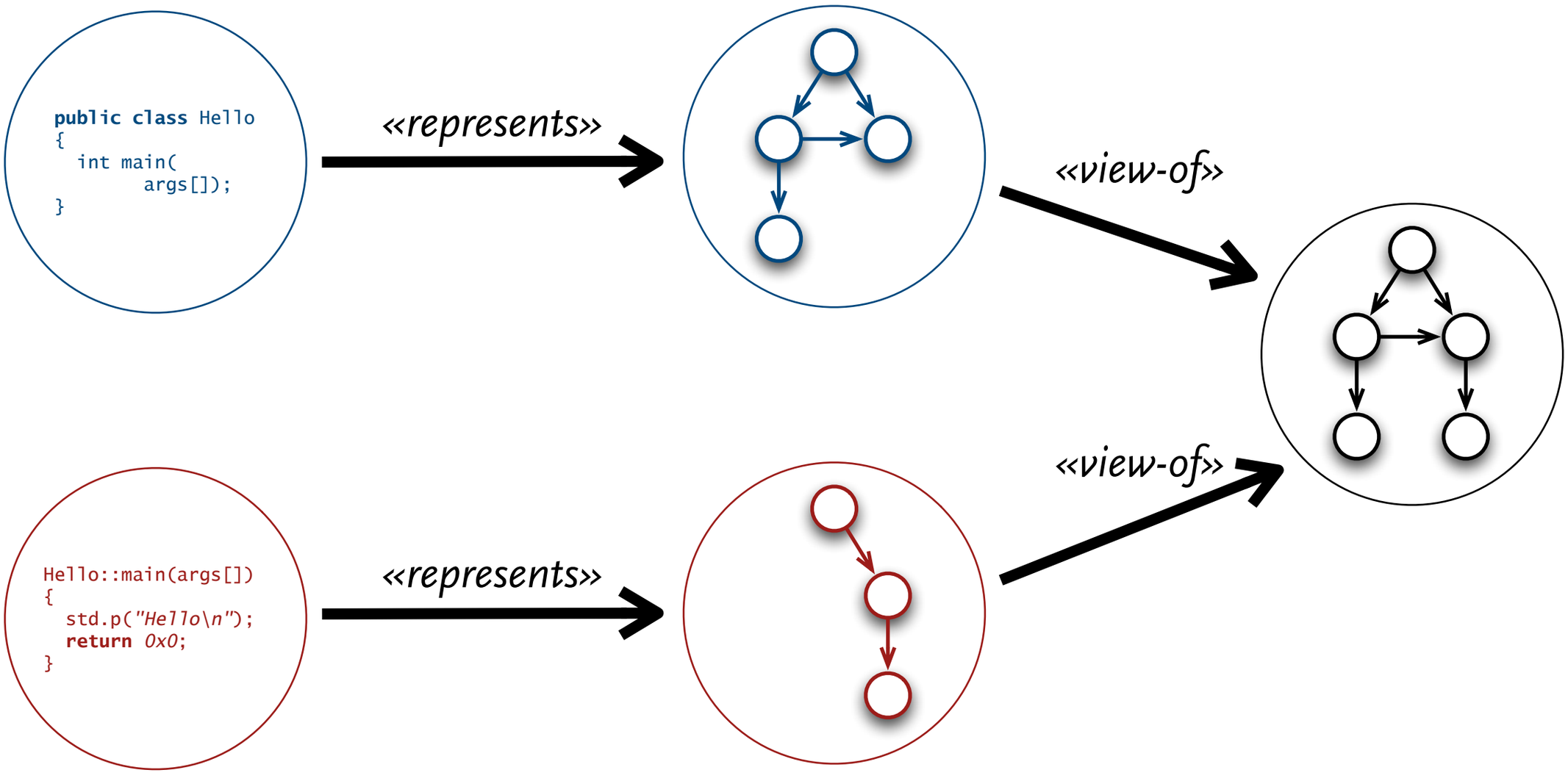


- there can multiple different representations for the same model (even in the same notation)

“Secondary Notation”

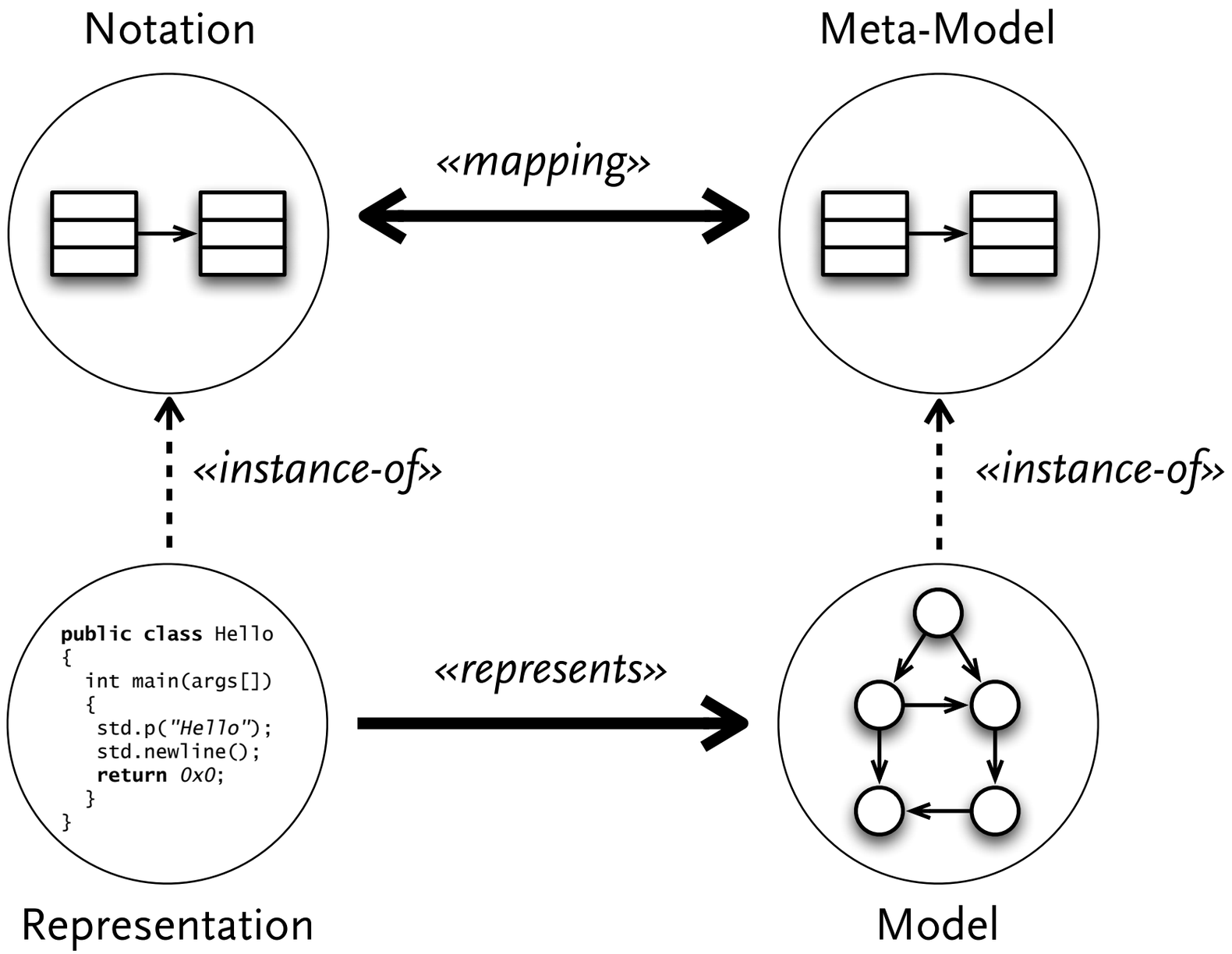
- Representations often contain elements that do not represent elements in the model.
- Classifiers for such elements are called secondary notations.
- Examples are
  - white spaces
  - documentation
  - position, sizes, colors in diagrams
  - order without meaning in the model
  - coding conventions
  - *but not* names
- Secondary notation contains meaningful information (at least for the authors/readers) and are worth keeping.

Views



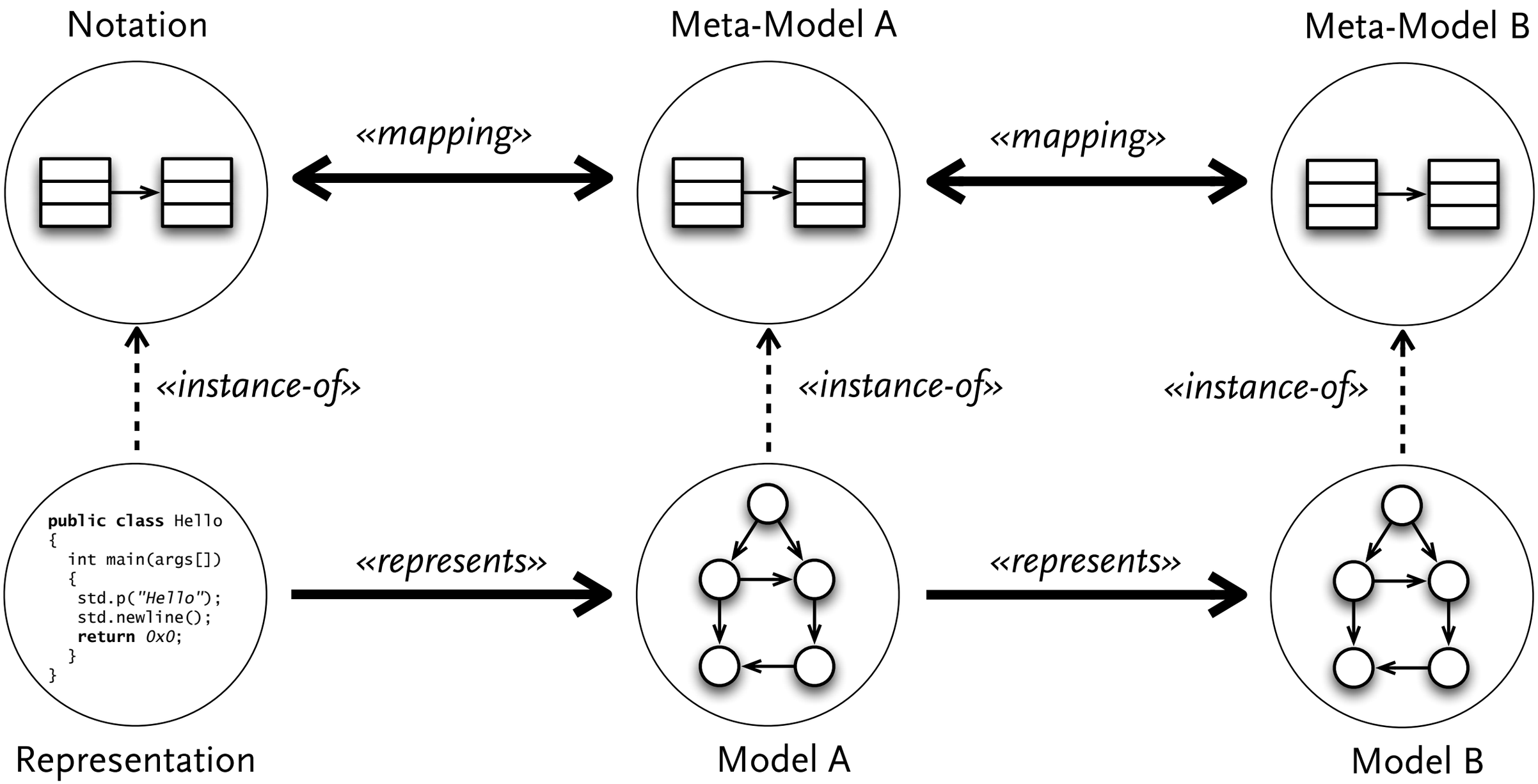
- multiple views represent different aspects of a model (multiple views, again not necessarily ambiguous)

Notations

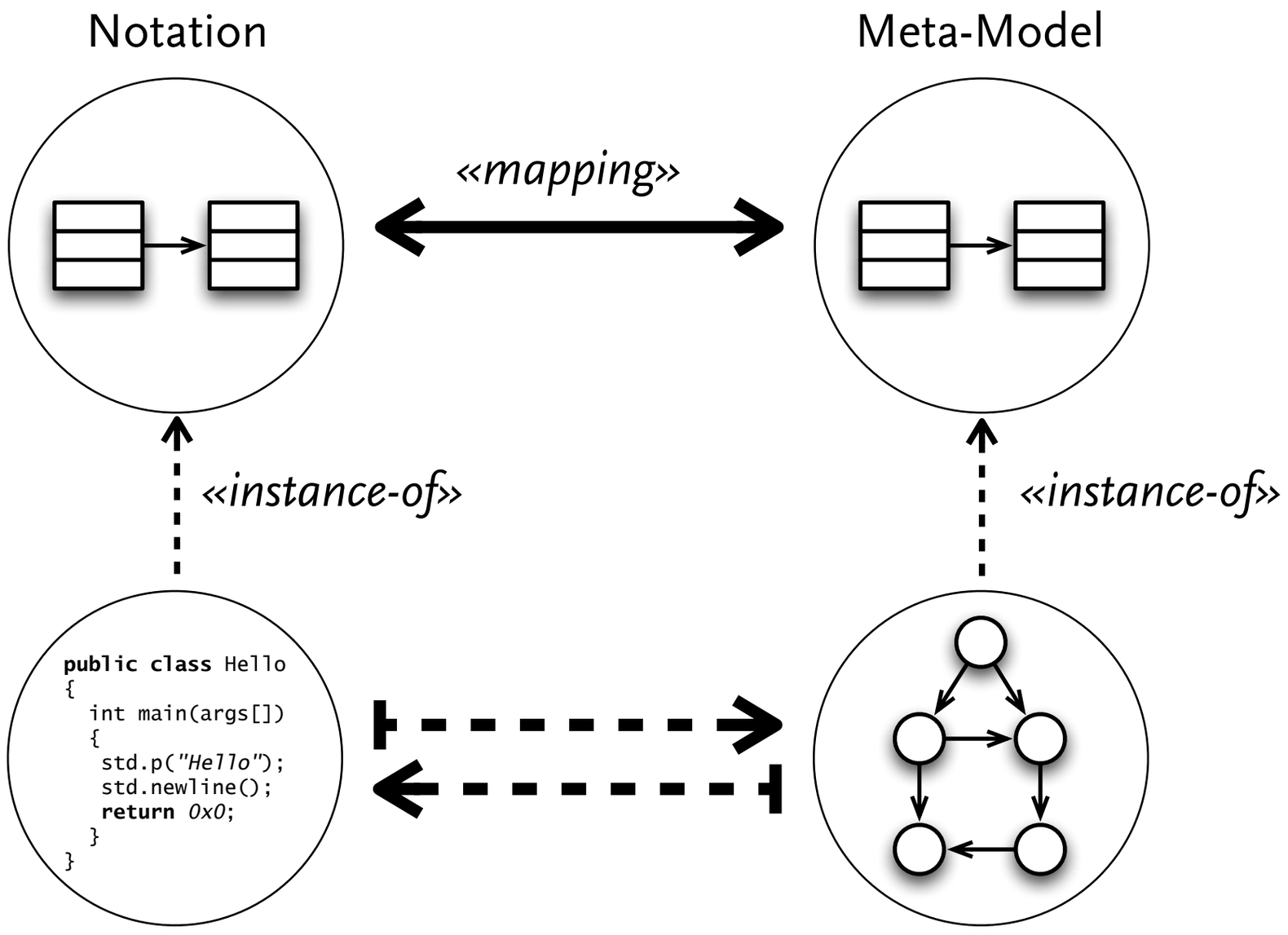


- Notations are descriptions of representations
- Notations depend on a meta-model

Other Language as Notation



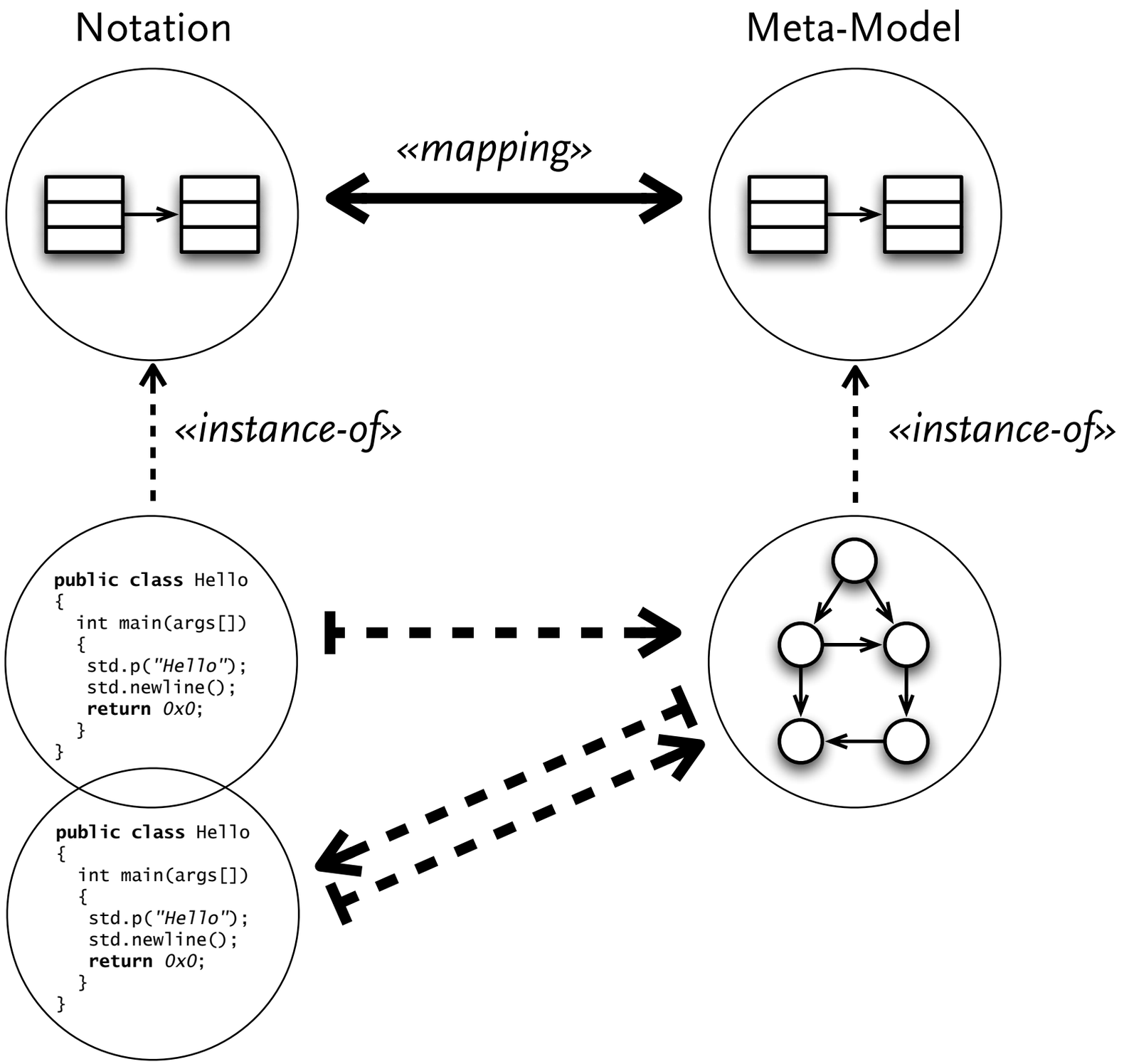
Mappings and Transformations



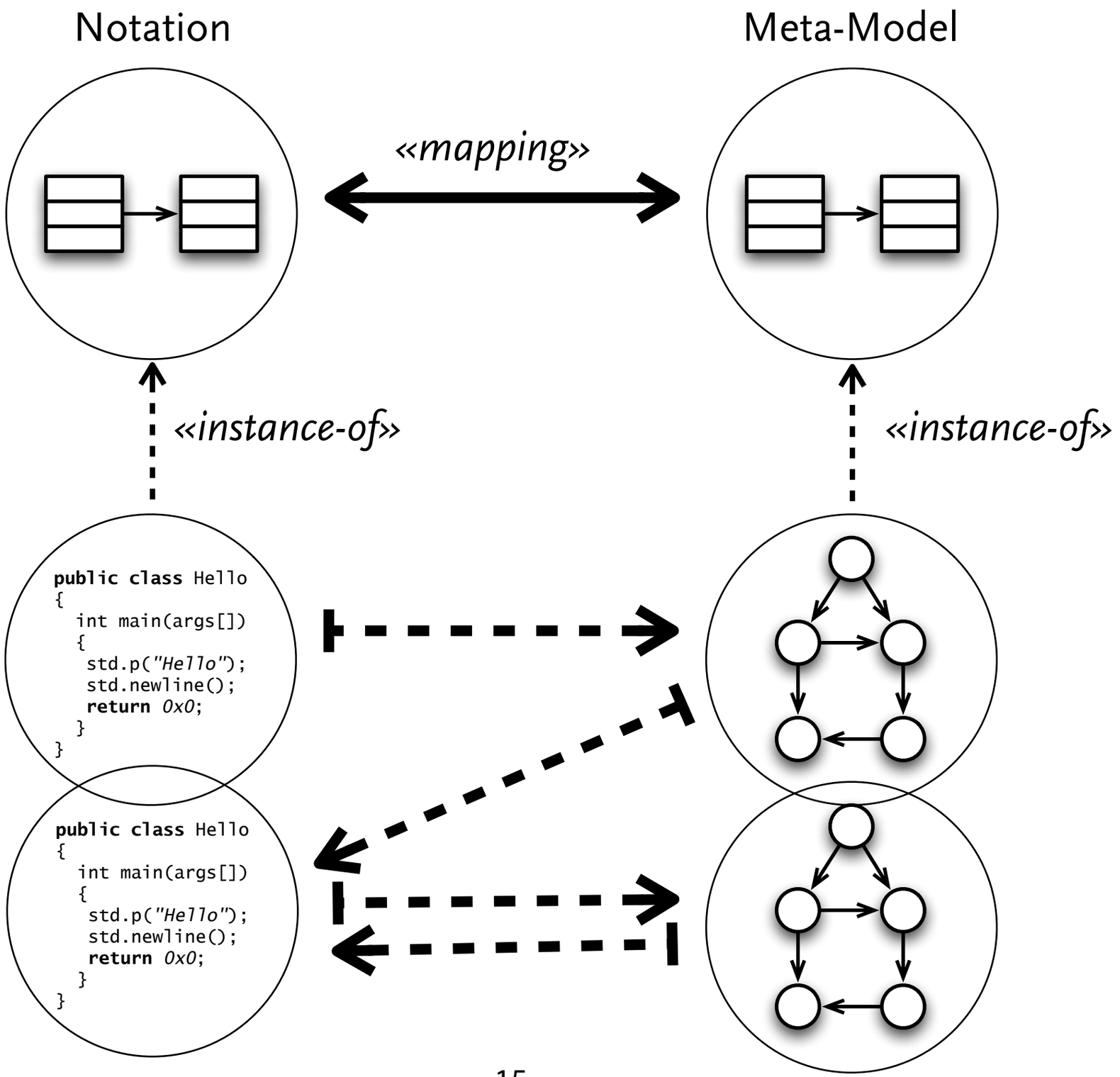
Unidirectional Mappings (1)

- Bidirectional mappings in the sense of a bijective function are often meaning less, they imply a homomorphism and hence equivalence, i.e. no
  - abstraction
  - views
  - additions (secondary notation)
- Unidirectional mappings are often easy to describe and implements, but two unidirectional mappings do not make a bidirectional mapping.

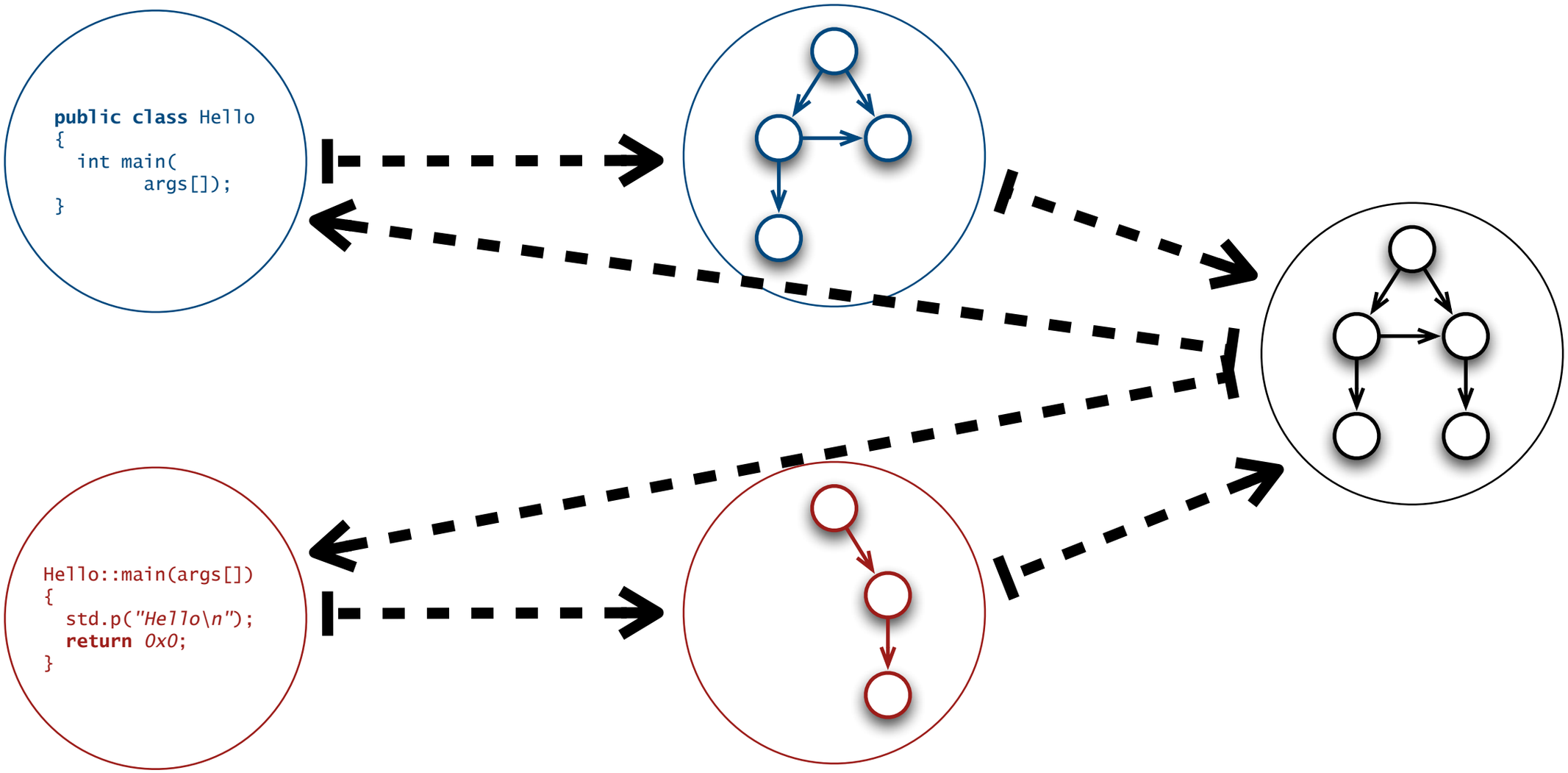
Unidirectional Mappings (2)



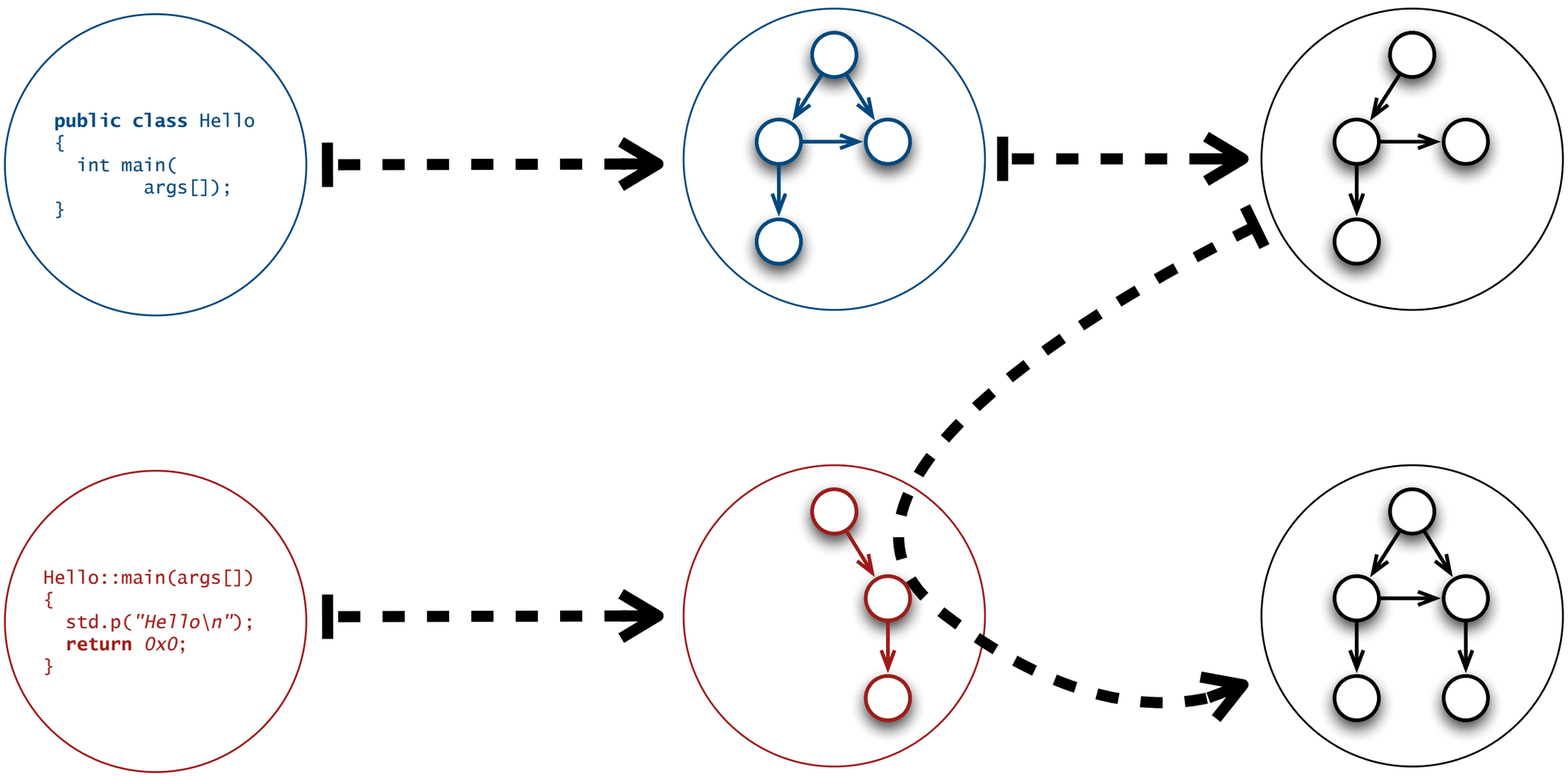
Unidirectional Mappings (3)



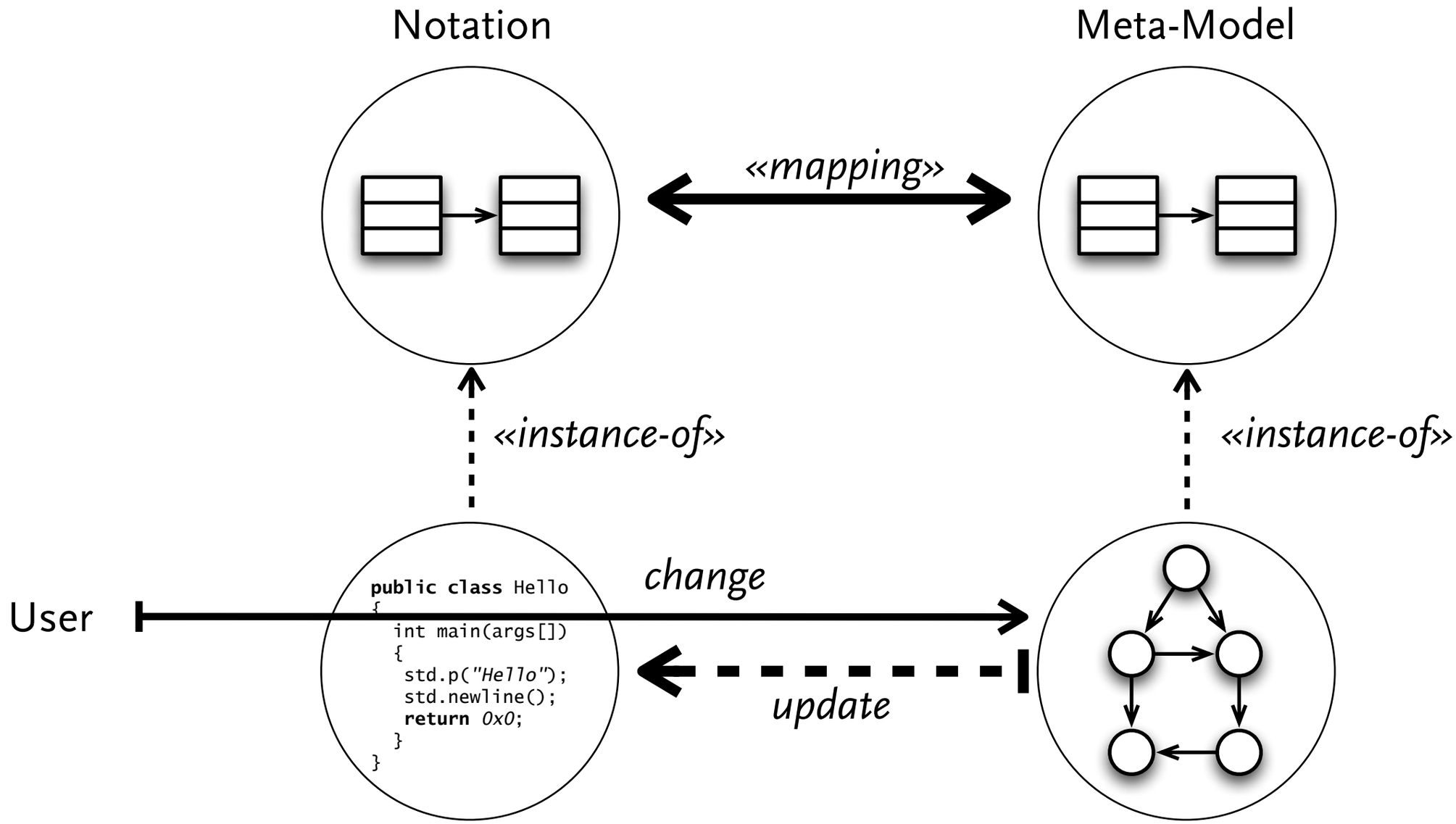
Synchronization (1)



Synchronization (1) – Asymetric Mappings

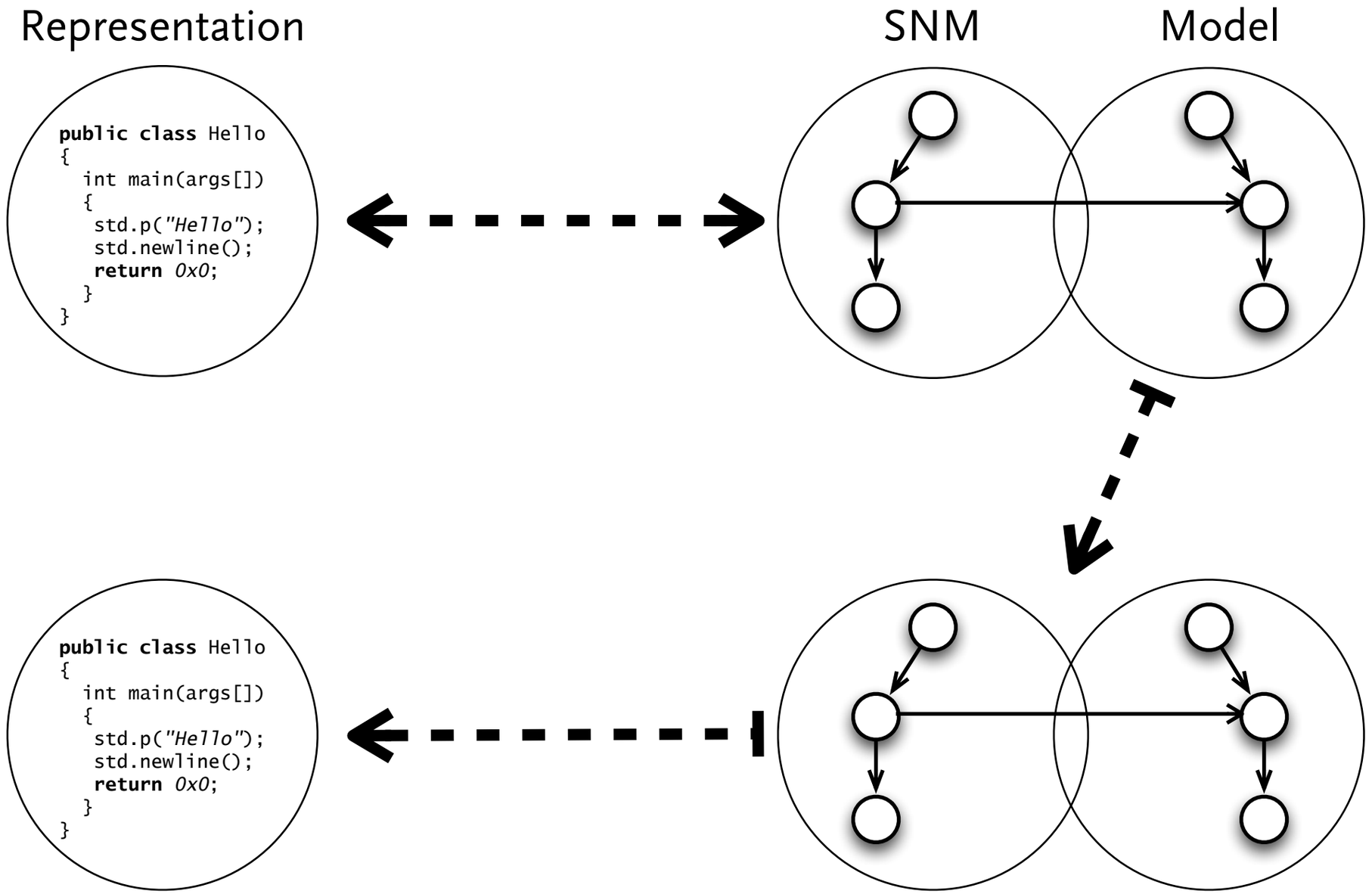


Model-View-Controller



- the reasonable example of bijective mappings, if secondary notation is part of the model

Representation, Secondary Notation Model + Model



Textual Notations

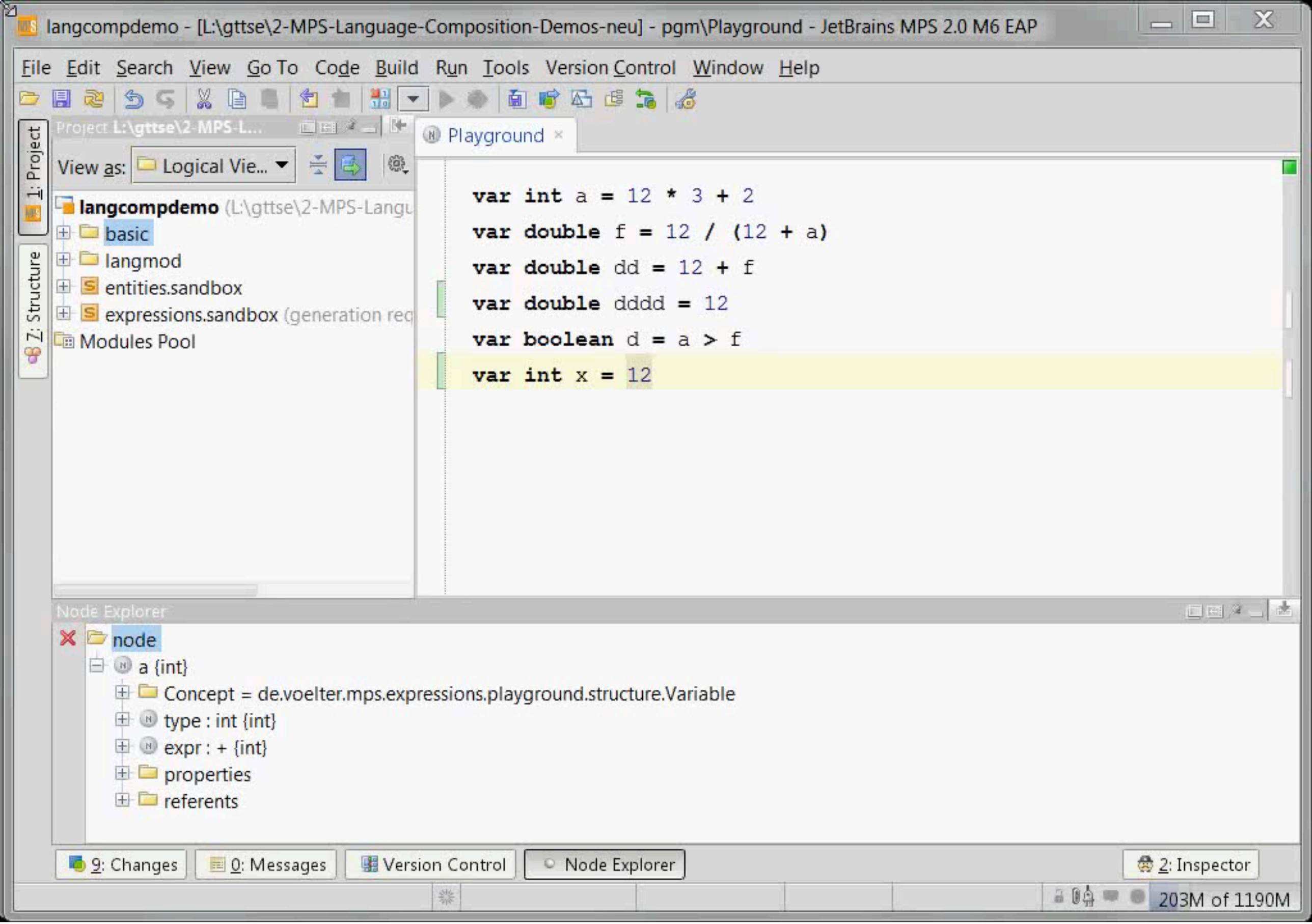
Mapping between Meta-Modeling Formalisms

	OO-Meta-Modeling	**XML**	**Rel. DBs.**	**Grammars**	**Automata**
M3	MOF	Schema-Schema	Rel. Alg.	Math	Math
M2	UML	Schema	E.-R.-M.	Grammar	Automata
M1	Model	XML	DB	Program	Word
M0	Things	Data	Data	Thing	Things

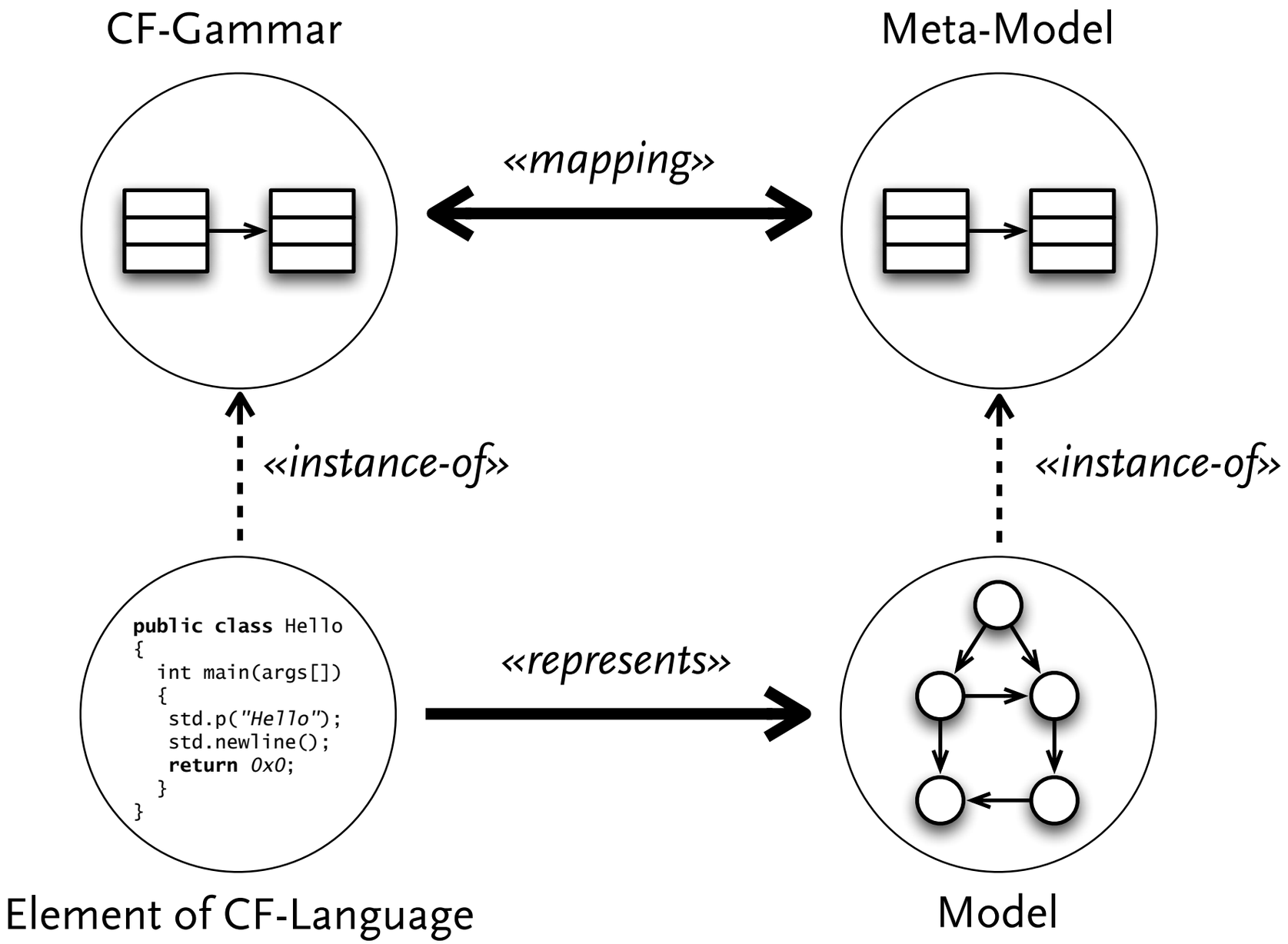
Strategies: Projection vs. Parsing

- Traditional context-free grammar based parsing of text documents (Parsing)
  - create a new model when parsing
  - and synchronization is difficult
- Model-View-Controller style use of text elements that look like a text document (Projection)
  - SNM+Model approach allows bijective mappings
  - does not feel like text editing
  - you can only type syntactically correct models

Projection with MPS


langcompdemo - [L:\gttse\2-MPS-Language-Composition-Demos-neu] - pgm\Playground - JetBrains MPS 2.0 M6 EAP
File Edit Search View Go To Code Build Run Tools Version Control Window Help
Project L:\gttse\2-MPS-L...
View as: Logical Vie...
langcompdemo (L:\gttse\2-MPS-Langu
basic
langmod
entities.sandbox
expressions.sandbox (generation req
Modules Pool
1: Project
7: Structure
Playground
var int a = 12 * 3 + 2
var double f = 12 / (12 + a)
var double dd = 12 + f
var double dddd = 12
var boolean d = a > f
var int x = 12
Node Explorer
node
a {int}
Concept = de.voelter.mps.expressions.playground.structure.Variable
type : int {int}
expr : + {int}
properties
referents
9: Changes
0: Messages
Version Control
Node Explorer
2: Inspector
203M of 1190M

Parsing (1)


CF-Gammar
Meta-Model
«mapping»
«instance-of»
«instance-of»
public class Hello
{
int main(args[])
{
std.p("Hello");
std.newline();
return 0x0;
}
}
«represents»
Element of CF-Language
Model

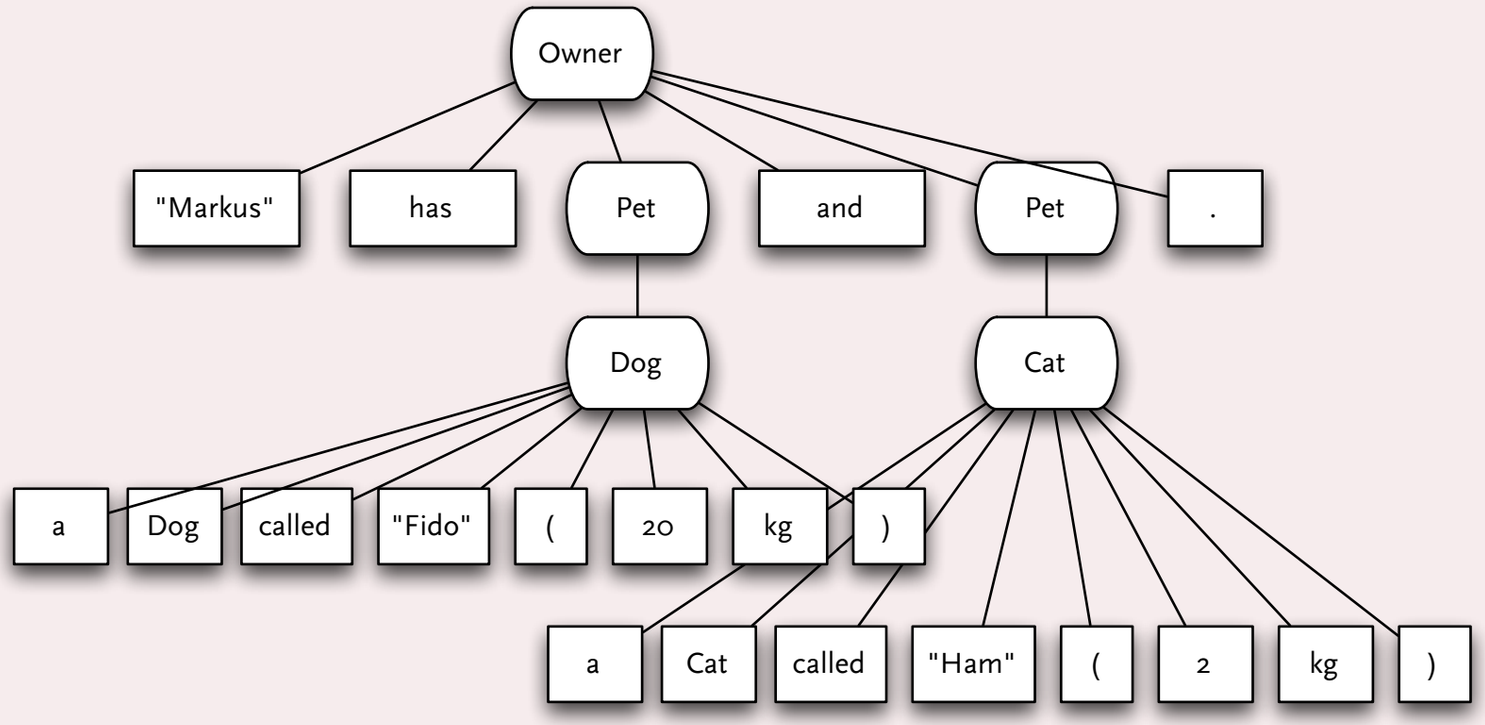
Context-Free Grammars

```
Owner : STRING ‘has’ Pet (‘and’ Pet)* ‘.’

Pet= Dog | Cat

Dog : ‘a’ ‘Dog’ ‘called’ STRING ‘(‘ INT ‘kg’ ‘)’
Cat : ‘a’ ‘Cat’ ‘called’ STRING ‘(‘ INT ‘kg’ ‘)’
```

grammar

```
“Markus” has a Dog “Fido” (20kg) and a Cat “Ham” (3kg).
```

text



AST

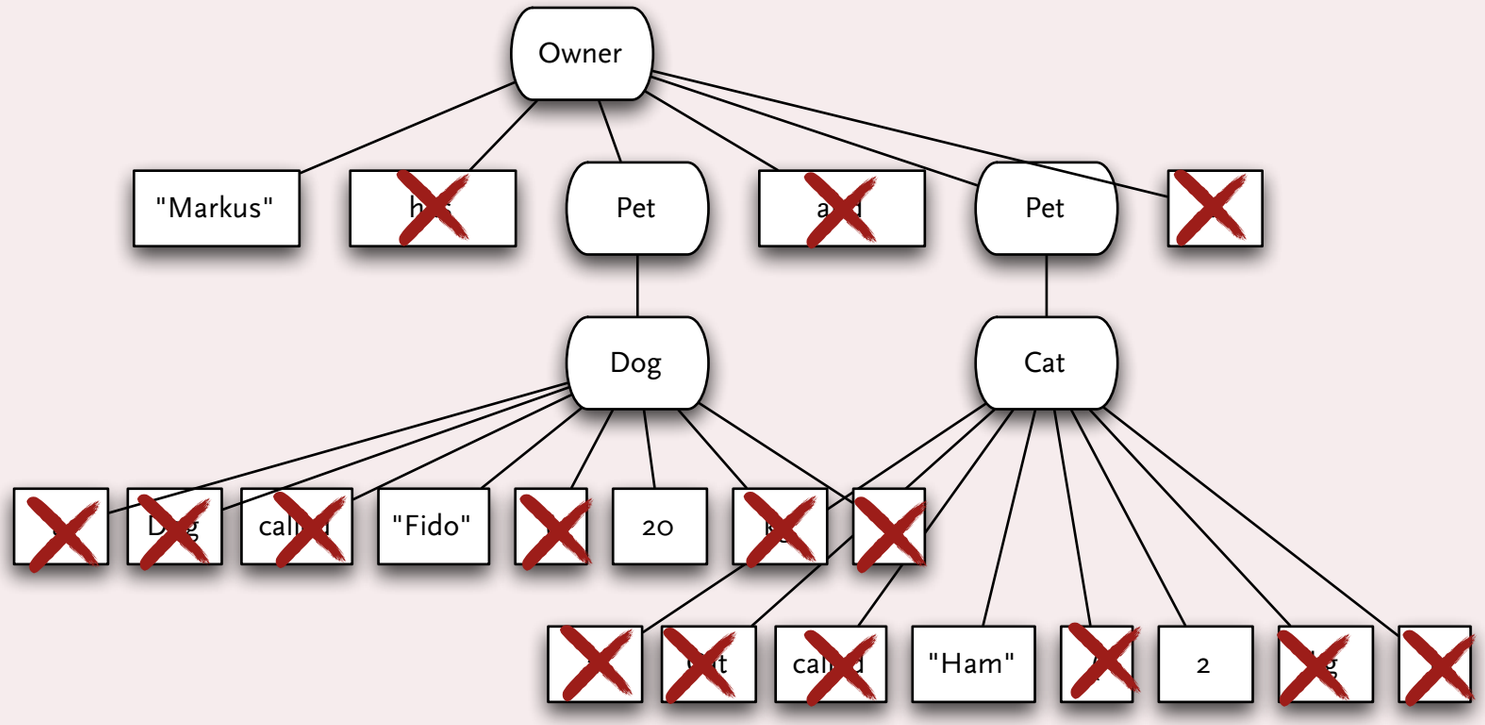
Context-Free Grammars

```
Owner : STRING ‘has’ Pet (‘and’ Pet)* ‘.’

Pet= Dog | Cat

Dog : ‘a’ ‘Dog’ ‘called’ STRING ‘(‘ INT ‘kg’ ‘)’
Cat : ‘a’ ‘Cat’ ‘called’ STRING ‘(‘ INT ‘kg’ ‘)’
```

grammar

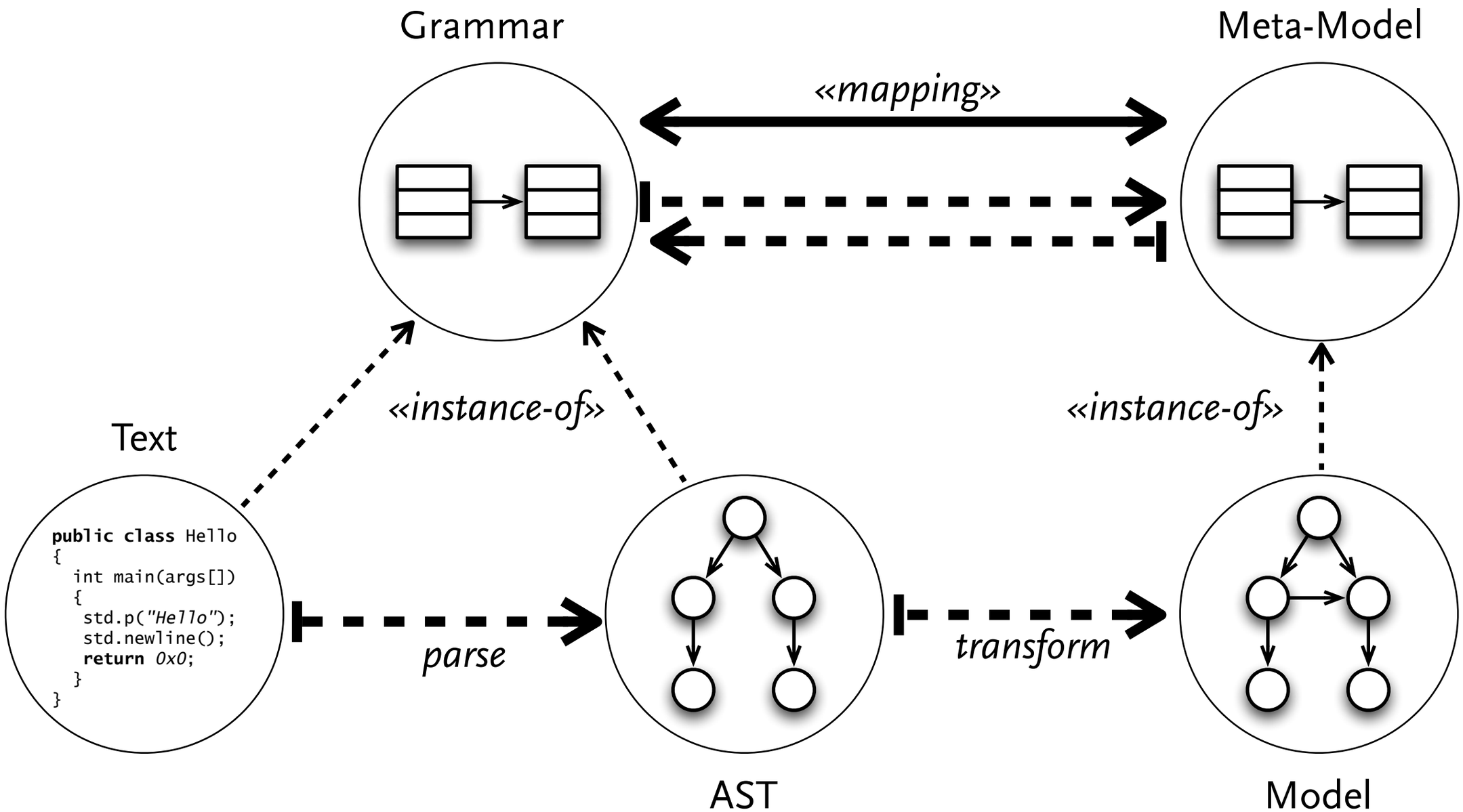
```
“Markus” has a Dog “Fido” (20kg) and a Cat “Ham” (3kg).
```

text

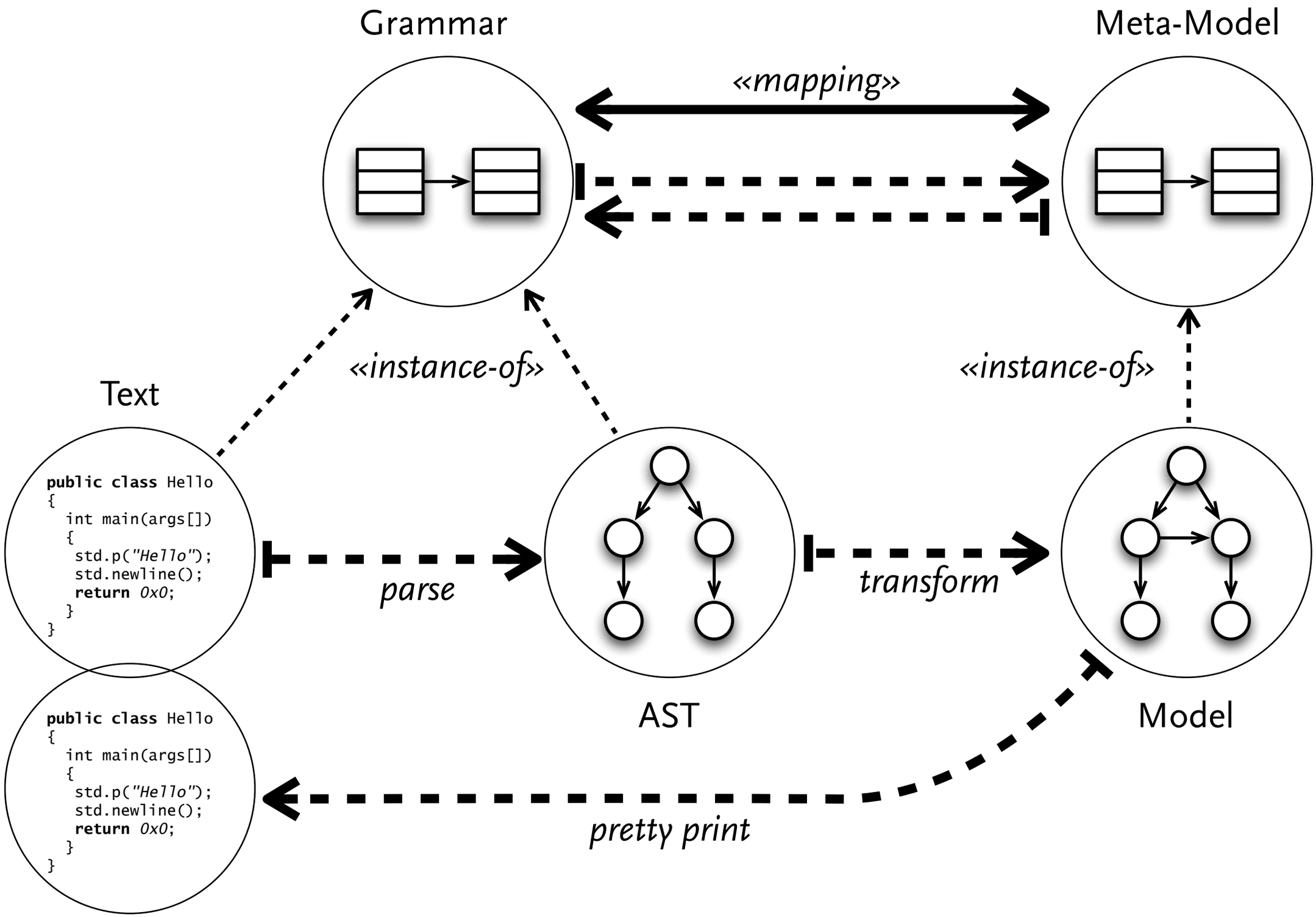


AST

Parsing (2)


Grammar
Meta-Model
«mapping»
«instance-of»
«instance-of»
Text
public class Hello
{
int main(args[])
{
std.p("Hello");
std.newline();
return 0x0;
}
}
parse
transform
AST
Model

Parsing (2)


Grammar
Meta-Model
«mapping»
«instance-of»
«instance-of»
Text
public class Hello
{
int main(args[])
{
std.p("Hello");
std.newline();
return 0x0;
}
}
parse
AST
transform
Model
public class Hello
{
int main(args[])
{
std.p("Hello");
std.newline();
return 0x0;
}
}
pretty print

Abstract Syntax Tree vs. EMF-Model

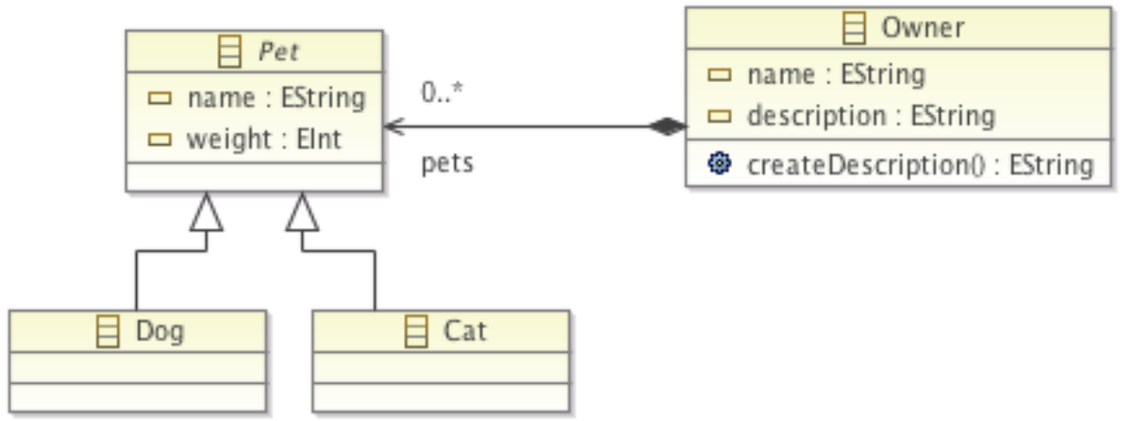
- AST is a representation of the model's containment hierarchy
  - typically grammars are defined structurally equivalent to meta-models
- AST does only contain *unresolved* cross-reference links
  - references via *Ids* (e.g. names)
  - depend on *scoping* (i.e. depend on namespaces, inheritance, imports, etc.)
  - references can be *broken*
- AST is not validated
- AST might contain secondary notation (i.e. annotations, comments)

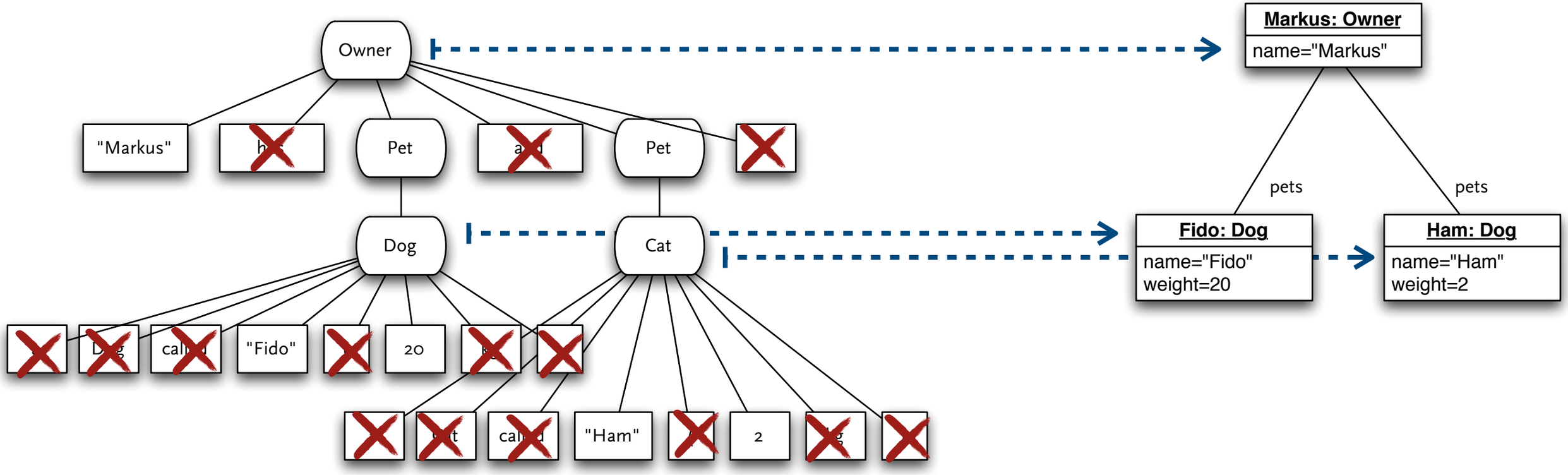
Grammar-Meta-Model Mapping (1)

- *hypothesis*: representations are structurally similar (or even equivalent) to their corresponding models
  - otherwise more complex mappings are required
- Symbols -> Classifier and -> AST-Nodes-> objects/values
- RHS positions -> Features | AST-child relations -> value sets

Grammar-Meta-Model Mapping (2)

```
Owner : STRING ‘has’ Pet (‘and’ Pet)* ‘.’

Pet= Dog | Cat

Dog : ‘a’ ‘Dog’ ‘called’ STRING ‘(‘ INT ‘kg’ ‘)’
Cat : ‘a’ ‘Cat’ ‘called’ STRING ‘(‘ INT ‘kg’ ‘)’
```

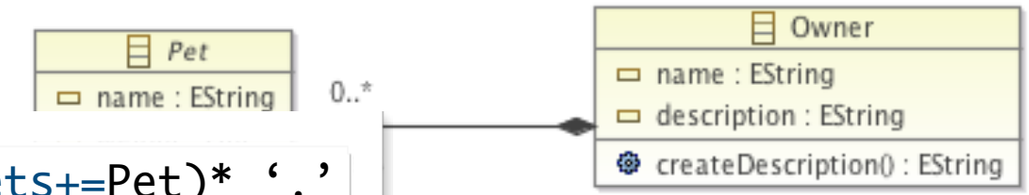


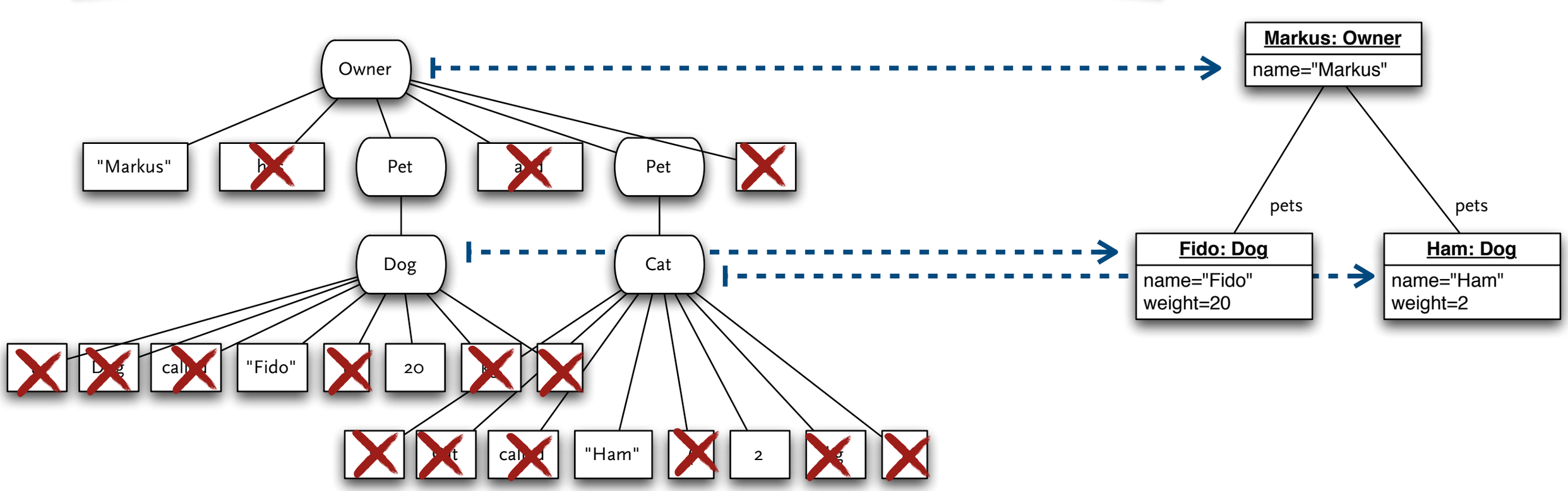


Grammar-Meta-Model Mapping (2)

```
Owner : STRING ‘has’ Pet (‘and’ Pet)* ‘.’

Pe

Do
Ca
```

```
Owner->Owner: name=STRING ‘has’ pets+=Pet (‘and’ pets+=Pet)* ‘.’

Pet: Dog | Cat

Dog->Dog: ‘a’ ‘Dog’ ‘called’ name=STRING ‘(‘ weight=INT ‘kg’ ‘)’
Cat->Cat: ‘a’ ‘Cat’ ‘called’ name=STRING ‘(‘ weight=INT ‘kg’ ‘)’
```





Declarative vs. Imperative

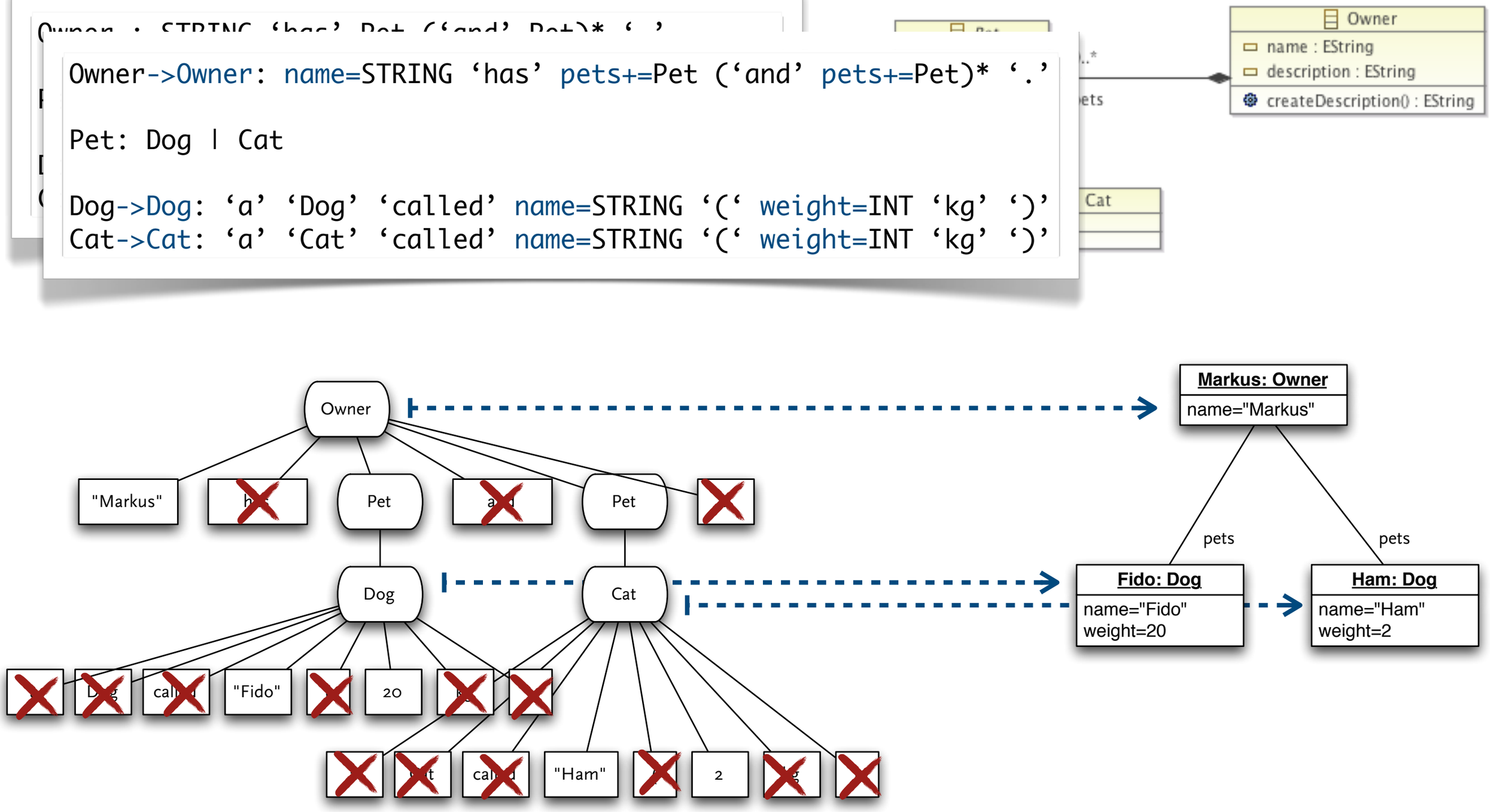
- Mappings describe transformations. There are two possible ways:
- Declarative descriptions of transformations define conditions and constraints that two models/ASTs must meet.
- Imperative descriptions of transformations define commands that create a target model/AST from a source model/AST.
- Imperative transformation descriptions can implement declarative descriptions when the conditions and constraints in the declarative description are met for all possible pairs of source and target models/ASTs. In practice often hard to achieve and impossible to validate.
- Eventually imperative implementations have to be derived from declarative descriptions (machines work imperative).

Grammar-Meta-Model-Mappings – Declarative vs. Imperative

- Grammars are declarative, they describe if a word is part of the language generated by the grammar and they declaratively describe a mapping between words and ASTs.
- Parsers are imperative implementations (using a stack automaton). Result of transformation from source word is target AST.
- Grammar-Meta-Model-Mappings seem declarative, but are actually imperative. Transformation is executed via traversing the AST (depth first), classifier and feature annotations are used to execute create objects and assign values to value sets commands.

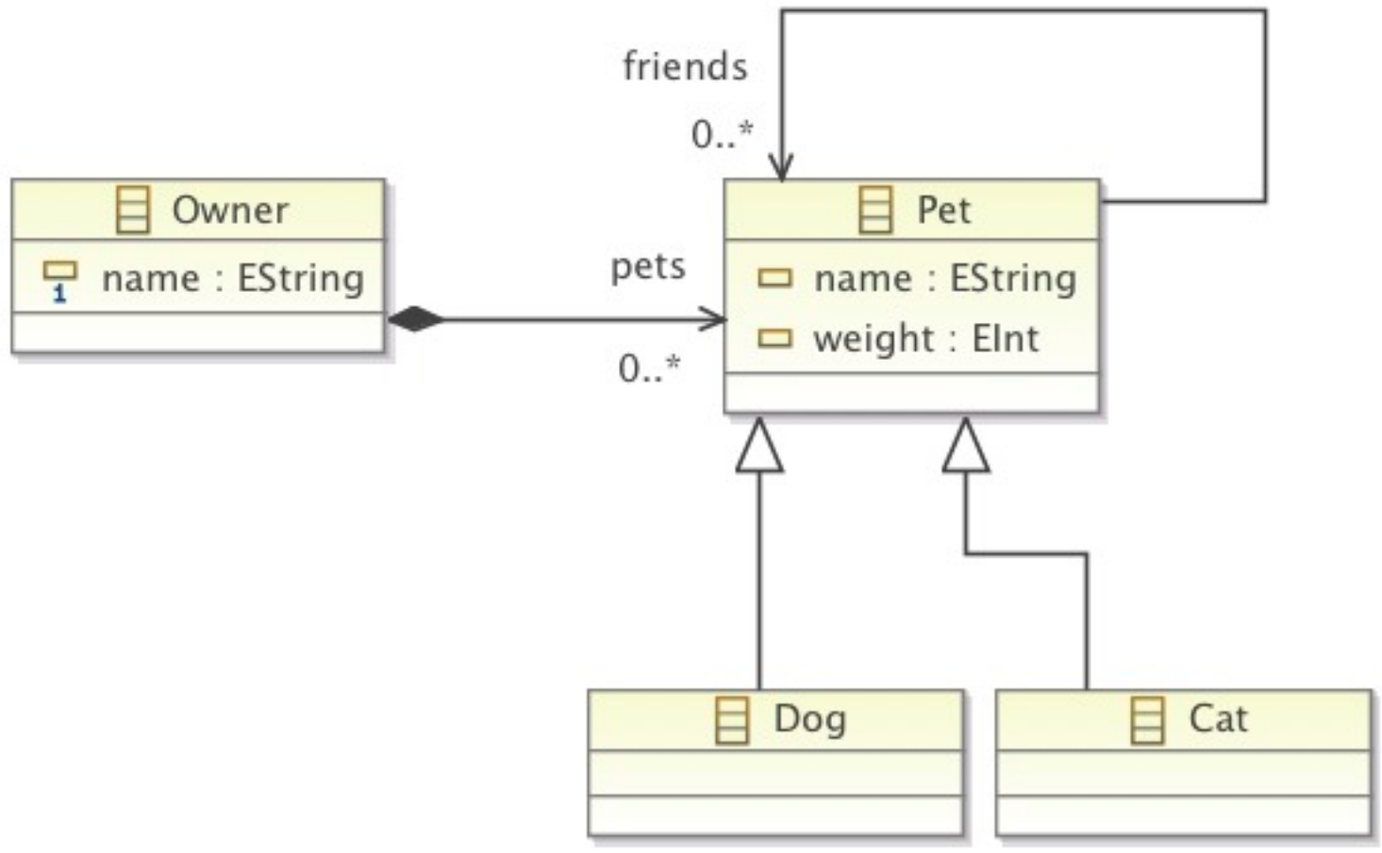
Grammar-Meta-Model Mapping – Imperative

```
Owner->Owner: name=STRING 'has' pets+=Pet ('and' pets+=Pet)* '.'

Pet: Dog | Cat

Dog->Dog: 'a' 'Dog' 'called' name=STRING '(' weight=INT 'kg' ')'
Cat->Cat: 'a' 'Cat' 'called' name=STRING '(' weight=INT 'kg' ')'
```



Grammar-Meta-Model Mapping – Cross References

```
Owner->Owner :
    name=STRING
    'has'
    pets+=Pet ('and' pets+=Pet)*
    '.'

Pet: Dog | Cat

Dog->Dog:
    'a' 'Dog' 'called'
    name=STRING
    ( 'and' 'is' 'friend' 'of' friends+=
    '(' weight=INT 'kg' ')'

Cat->Cat:
    'a' 'Cat' 'called'
    name=STRING
    '(' weight=INT 'kg' ')'
```

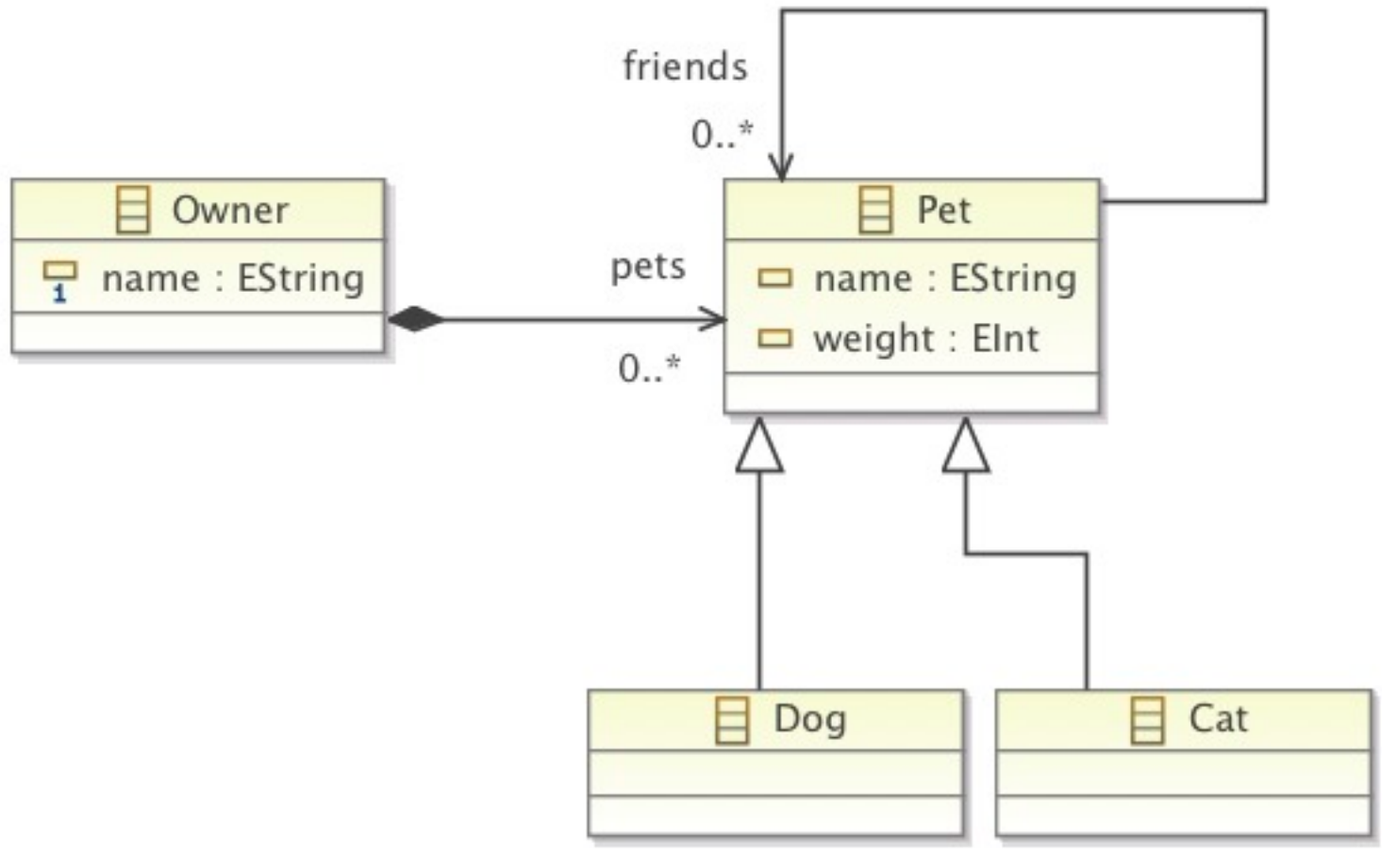



```
"Markus" has a Dog called "Fido" and is friend
of "Ham" (20kg) and a Cat called "Ham" (2kg).
```

- To what attribute refers the reference?
- Where to find possibly references Pets?
- What is with name collisions?

Grammar-Meta-Model Mapping – Cross References

```
Owner->Owner :
    name=STRING
    'has'
    pets+=Pet ('and' pets+=Pet)*
    '.'

Pet: Dog | Cat

Dog->Dog:
    'a' 'Dog' 'called'
    name=STRING
    ( 'and' 'is' 'friend' 'of' friends+=
    '(' weight=INT 'kg' ')'

Cat->Cat:
    'a' 'Cat' 'called'
    name=STRING
    '(' weight=INT 'kg' ')'
```



"Markus" has a Dog called "Fido" and is friend of "Ham" (20kg) and a Cat called "Ham" (2kg).

- To what attribute refers the reference?
- Where to find possibly references Pets?
- What is with name collisions?

Resolving Cross References – Scopes

- `scope(EObject, EReference)->List<EObject>` = all elements that could be linked from a specific object using a specific reference
  - `scope(Fido, friends) = { Fido, Ham, Maestro, Angela }`
  - `scope(Fido, friends) = { Ham, Maestro, Angela }`
  - `scope(Fido, friend) = { Ham }`
- scope has to be computable based on containment hierarchy only
- Scopes are also useful for implementing code completion

```
“Markus” has a Dog called “Fido” (20kg) and a Cat called “Ham” (2kg).
“Kathi” has a Cat called “Maestro” (1kg) and a Cat called “Angela” (3kg).
```

Identifier in Complex Scoping Systems, e.g. Java

- In most programming languages we use non globally unique identifier and scoping to uniquely identify referenced elements.
  - System.out.println(“Hello World.”);
- Identifier can be more than a name, e.g. can also comprise static types of arguments.
- A qualified identifier is a globally unique identifier
  - java.lang.System.out
  - java.io.PrintWriter#println(String)
- What elements an identifier identifies depends on its scope.
- One identifier can identify multiple elements in the same scope (e.g. hidden names, but not overloading)
- Complex scopes through: imports, inheritance, overloading, overwriting, hiding

Resolving Cross References – Scoping with Identifier

- scope(EObject, EReference) -> List<EObject>
- qualifiedId(EObject) -> Id
- isSuffix(Id,Id) -> boolean
- compare(Id,Id) -> int

- This could be described with OCL or similar languages. No real declarative mapping is known.

Resolving Cross References – Scoping with Identifier

```
“Markus” has a Dog called “Fido” and friends with “Maestro” (20kg) and a
Cat called “Ham” (2kg).
“Kathi” has a Cat called “Maestro” (1kg) and a Cat called “Angela” (3kg).
```

- scope(EObject, EReference) -> List<EObject>
- qualifiedId(EObject) -> Id
- isSuffix(Id,Id) -> boolean
- compare(Id,Id) -> int
- This could be described with OCL or similar languages. No real declarative mapping is known.

Resolving Cross References – Scoping with Identifier

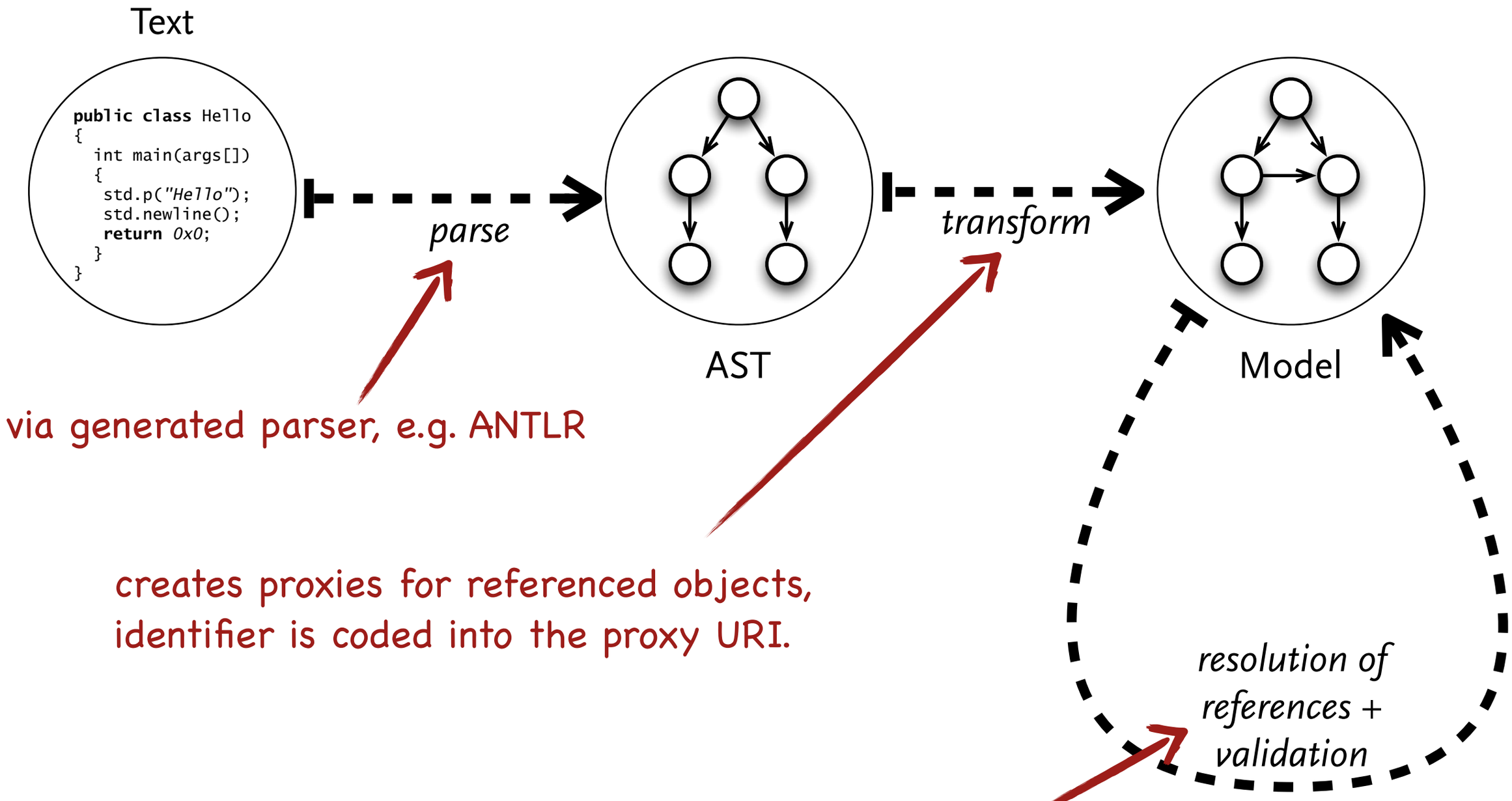
```
“Markus” has a Dog called “Fido” and friends with “Maestro” (20kg) and a
Cat called “Ham” (2kg).
“Kathi” has a Cat called “Maestro” (1kg) and a Cat called “Angela” (3kg).
```

- scope(EObject, EReference) -> List<EObject>
- qualifiedId(EObject) -> Id
- isSuffix(Id,Id) -> boolean
- compare(Id,Id) -> int
- This could be described with OCL or similar languages. No real declarative mapping is known.

Validation

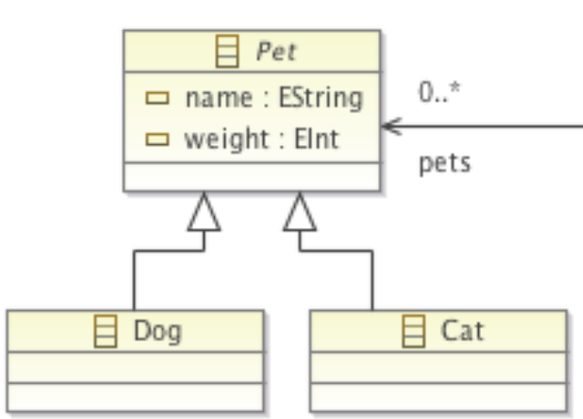
- as usual, EMF validation framework, Java, OCL, or other languages
- type system is a typical part of validation rules (static type safety)
- type system and scoping can depend on each other
  - scope depends on type, e.g. in `out.println("Hello")` type of *out* is relevant
  - type can only be determined if identifier is resolved, e.g. need to know what *out* refers two
  - validation before cross-reference resolution or vice versa?

Text to Model Transformation + Resolution + Validation



Mapping between Grammar Formalism and Ecore

- A meta-model can be generated from a grammar

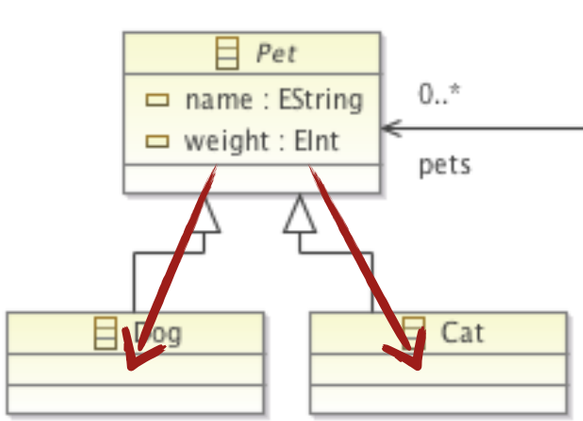



```
Owner->Owner: name=STRING ‘has’ pets+=Pet (‘and’ pets+=Pet)* ‘.’

Pet: Dog | Cat

Dog->Dog: ‘a’ ‘Dog’ ‘called’ name=STRING ‘(‘ weight=INT ‘kg’ ‘)’
Cat->Cat: ‘a’ ‘Cat’ ‘called’ name=STRING ‘(‘ weight=INT ‘kg’ ‘)’
```

- A grammar can be generated from a meta-model

Mapping between Grammar Formalism and Ecore

- A meta-model can be generated from a grammar

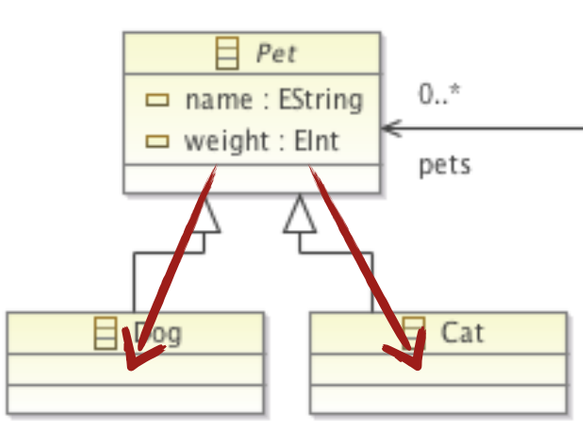



```
Owner->Owner: name=STRING 'has' pets+=Pet ('and' pets+=Pet)* '.'

Pet: Dog | Cat

Dog->Dog: 'a' 'Dog' 'called' name=STRING '(' weight=INT 'kg' ')'
Cat->Cat: 'a' 'Cat' 'called' name=STRING '(' weight=INT 'kg' ')'
```

- A grammar can be generated from a meta-model

Mapping between Grammar Formalism and Ecore

- A meta-model can be generated from a grammar




```
Owner->Owner: name=STRING ‘has’ pets+=Pet (‘and’ pets+=Pet)* ‘.’

Pet: Dog | Cat

Dog->Dog: ‘a’ ‘Dog’ ‘called’ name=STRING ‘(‘ weight=INT ‘kg’ ‘)’
Cat->Cat: ‘a’ ‘Cat’ ‘called’ nam
```

- A grammar can be generated from

```
Owner->Owner: ‘{’
    ‘name’ ‘:’ name=STRING
    ‘pets’ ‘:’ ‘[‘ pets+=Pet ‘]’
‘}’

Pet: Dog | Cat

Dog->Dog: ‘{‘
    ‘name’ ‘:’ name=STRING
    ‘weight’ ‘:’ weight=INT
‘}’
...
```

And what about?

- Representations that consist of multiple texts? Like multiple files?
  - containment hierarchies over multiple resources
  - reference resolution, scoping over multiple resources
  - imports as secondary notation
- Non structural equivalent representations
  - AST and model containment hierarchies do not match: `“Fido” is a Dog (20kg) and “Ham” is a Cat (2kg). “Markus” owns “Fido” and “Ham”.`
  - Complex mapping chain with intermediate model and model-to-model transformation necessary.
- Inheritance in the meta-model?

Summary

- Projection and Parsing
- Mapping between Grammar and Meta-Model allows Text-to-Model transformation
- Reference resolution outside simple grammar-meta-model mapping
- Scoping can be used to describe reference resolution
- Validation based on the model

xText

Java - Eclipse SDK

File Edit Source Refactor Refactor Navigate Search Run Project Window Help

CVS Reposito... Team Synchr... Debug Java

Packa JUnit Naviga Hierar

- Other Projects
- schulung
  - schulung.example
  - schulung.example-xvar
  - schulung.example-xweave
  - schulung.generator
  - schulung.generator-v2
  - schulung.mm
  - schulung.platform
  - schulung.smtest
  - schulung.smtest.editor
  - schulung.uml.example

Outline

An outline is not available.

Problems Javadoc Declaration Console Properties Search Recipes History Error Log

<terminated> New_configuration [Eclipse Application] C:\Programme\Java\jre1.5.0_06\bin\javaw.exe (03.09.2007 12:19:39)

Start 3 S... E... J... 100% 12:20

xText Basics

xText Grammar vs. Meta-Model

xText Grammar

Validation and Extensions in xText

Dependency Injection

Extending xText

Eclipse Text Editors

Eclipse Text-Editors

Graphical Notations

GEF, GMF, Sirius, Graphity