Model-based software development

Lecture VIII.

Constraints, Syntax, Semantics

Dr. Semeráth Oszkár

Syntax és Semantics

I. Constraints

II. Concrete and Abstract syntax

III. Editors

IV. Semantics

What are the components of a Domain-Specific Language?

- What do we need for a DSL?

  - Language Structure (e.g. Metamodel)
  - Constraints

  → Abstract Syntax

  - Views and Editors

  → Concrete Syntax

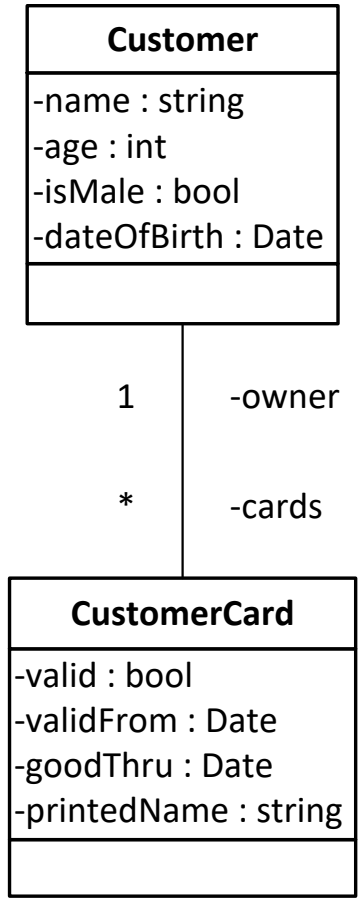
  - Meaning

  → Semantics

Constraints: Motivation

Challenge: How to describe complex structural patterns?

- Example: **Object-oriented programming**
  - > Each object is responsible for the consistency of itself
  - > *"**name** is not empty"*
  - > *"**age** cannot be negative"*
  - > *"**printedName** on card needs to be the first part of **name**"*
- Example: Domain-Specific Language
  - > Global constraints are more typical, responsibility is not on the object
  - > *"**validFrom** and **printedName** needs to be unique"*
  - > *"**Customer** can have at most one valid card"*



About constraints

- What is a constraint?

  A restriction that filters unwanted models.

  A transformation that maps models to a set of error messages.

- Sometimes can be enforced by metamodel, but:
  - It can become difficult
  - Or impssible

- How to define constraints?

Object Constraint Language (OCL)

- Allows precise UML/metamodel definitions
- OMG standard
- Features
  - **OCL constraints are declarative:** they specify what is correct and not what should be done
  - **OCL constraints have no side effects:** evaluating OCL expressions does not change the state of the system
  - **OCL constraints have formal syntax and semantics:** their interpretation is unambiguous and can be automated
- Metamodel extension = more options
- Extending constraints = fewer options

Context

- **Context:** The model element for which the OCL term is defined
  - > class, interface, data type, component, operation, instance

```
context Customer inv: self.name <> ''
```

- **Context Type:** the type of model element for which the expression is evaluated
- If the context is a type, it is the same as the type of the context

```
context Customer inv:…
```

$$\forall\, x : \boldsymbol{Customer}(x) \Rightarrow \cdots$$

- **Context instance:** the specific model element for which the expression is evaluated
  - > Referenced by the keyword „*self*”

Expression types by context

- Invariant
  - > *„Every student has a Neptun code”*
- Pre- and post-condition
  - > *„It is dark before/after the sun rises/sets”*
- Initial value
  - > „The c*ar has run 0 km at the time of production”*
- Derived value
  - > *„The final grade is the average of the midterm and the final exam”*
- Method body
  - > *“The number of books in the library is the sum of the number of books on the shelves”*

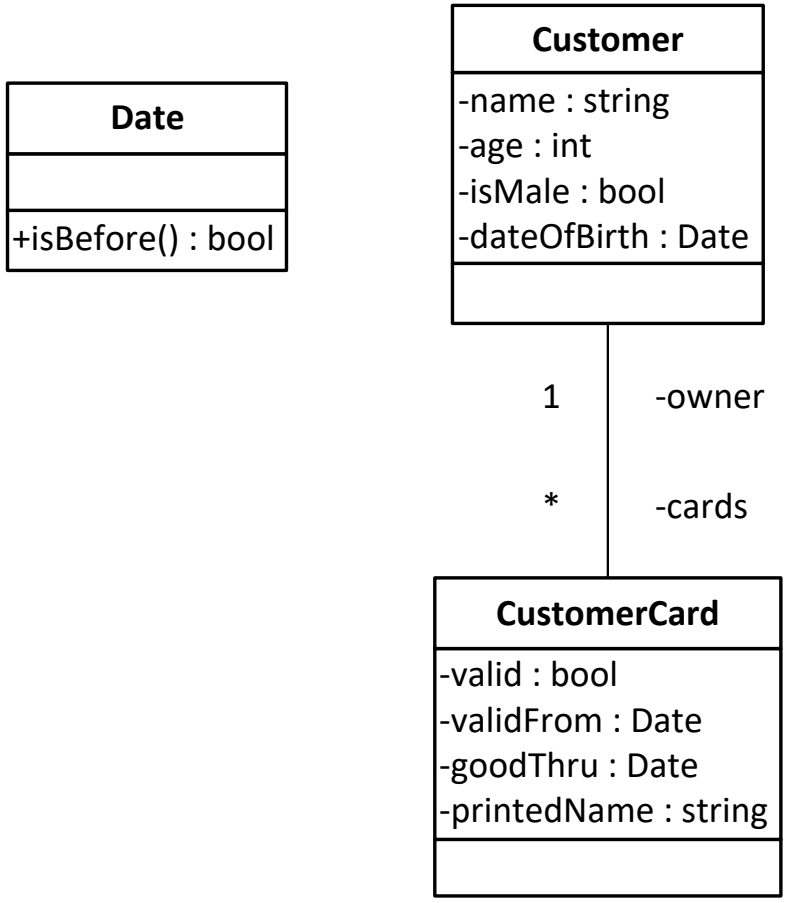
Invariants

- Constraint defined on a *metamodel element*
- A logical expression that must be true for all instances of a metamodel element at all times

```
context <metaelement>
inv [<contraint name>]: <logical expression>
```

Invariant examples

```
context Customer inv:
      self.name <>''

context Customer inv:
      age >= 18

context CustomerCard inv
checkDates:
validFrom.isBefore(goodThru)

context Customer inv:
Customer.allInstances()->
forAll(c1, c2 | c1<>c2 implies
c1.name <> c2.name)
```



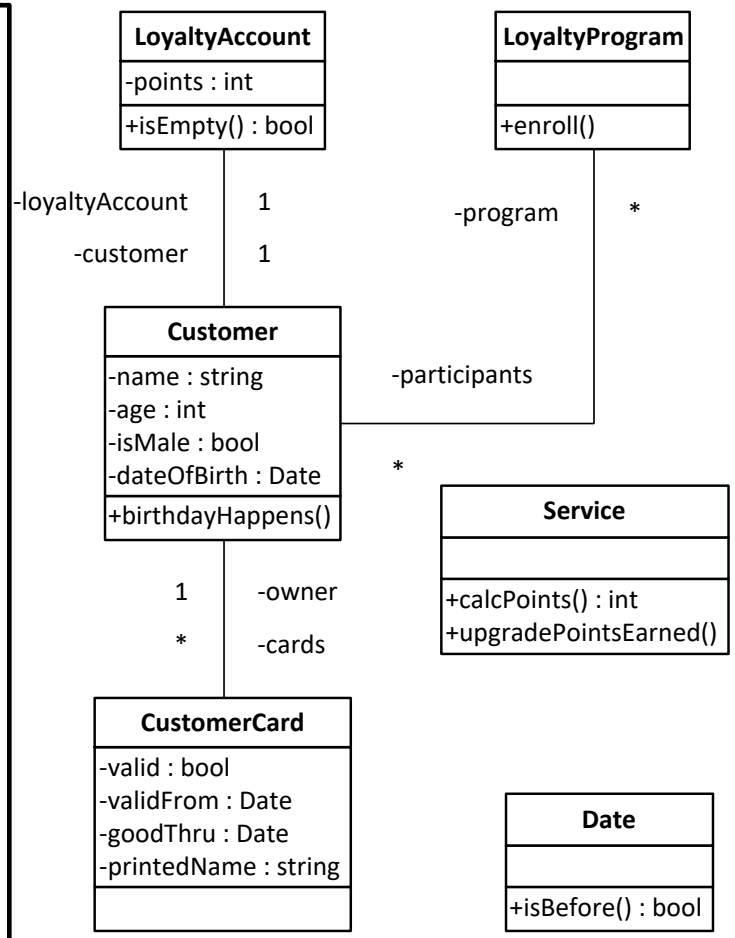
Pre- and post-conditions

- Constraint defined for an *operation*
- Focuses on the effect of the operation regardless of algorithm or implementation
  - > Precondition: a condition true at the last moment before an operation is performed
  - > Postcondition: condition true at the first moment after an operation is performed

```
context <metaelement>::<operation> (<parameters>)
pre[<contraint name>]: <logical expression>

context <metaelement>::<operation> (<parameters>)
post[<contraint name>]: <logical expression>
```

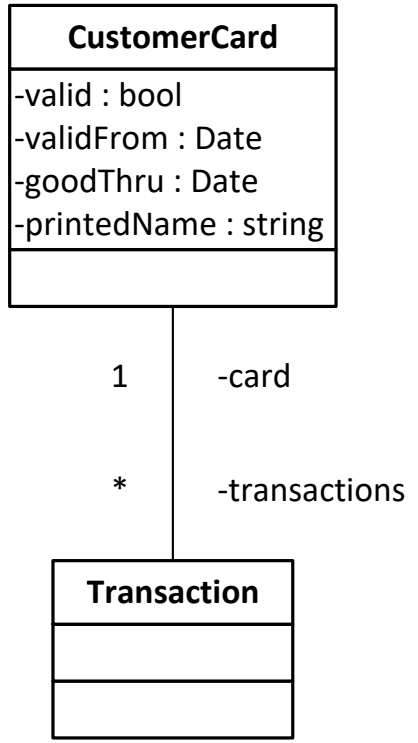
Pre- and post-condition examples

```
context LoyaltyAccount::
                isEmpty(): Boolean
post: result = (points = 0)

context Customer::birthdayHappens()
post: age = age@pre + 1

context Service::
upgradePointsEarned(amount: Integer)
post: calcPoints() = calcPoints@pre() +
amount

context LoyaltyProgram::
                enroll(c : Customer)
pre: c.name <> ' '
post: participants = participants@pre->
 including(c)
```



Initial value

- Constraint defined on an *attribute* or *association*
- A value taken by the attribute or association at the moment of context instance creation



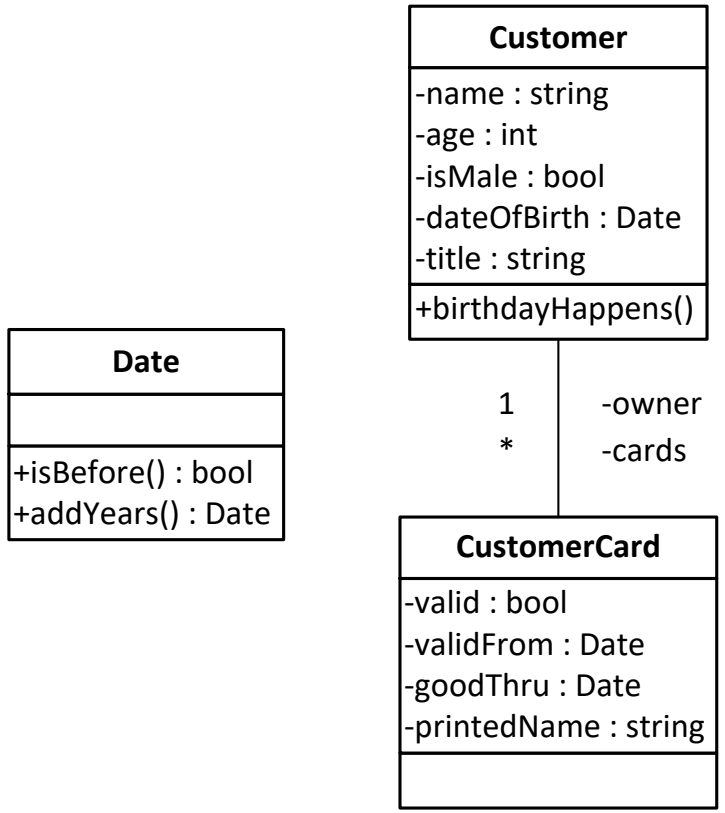

```
context <metaelement>
init: <logical expression>
```

```
context CustomerCard::transactions : Set(Transaction)
init: Set{}

context CustomerCard::valid : Boolean
init: true
```

Derived value

- Constraint defined on an *attribute* or *association*
- A derived element is not a value in itself, it is always defined by other elements

```
context CustomerCard::printedName
derive: owner.title.concat(owner.name)

context CustomerCard::goodThru
derive: if ( owner.isMale='male' )
        then validFrom.addYears(5)
        else validFrom.addYears(3)
        endif
```



Query operations body

- Constraint defined for an *operation*
- There are operations that retrieve a specific value, with no other side effects
- A constraint can be used to specify what exactly they should return

```
context LoyaltyAccount::getCustomerName() : String
body: Membership.card.owner.name

context CustomerCard::getTransactions(from : Date, until: Date )
                                      : Set(Transaction)
body: transactions->select(date.isAfter( from ) and
                           date.isBefore( until ) )

context Department::getSalary(): int
body: member.salary->
       iterate(member: Member; sum: Integer = 0 | sum + salary)
```

Constraints and inheritance

- Constraints are also inherited
- The **invariant** is inherited by the derived class. The derived class may tighten the constraint, but it may not relax it.
  - > VIPCustomer needs to have a name as well. Name needs to start with “VIP-”
- The **precondition** can be relaxed in a redefined operation of the derived class. It cannot be tightened.
- The **postcondition** can be tightened in the redefined operation of the derived class. It cannot be relaxed.

EXTRA: Types and Boole algebra in OCL

- All OCL expressions are typed
  - `OclAny`:
    The type that includes all others. E.g. `x, y : OclAny`
  - `x = y`
    x and y are the same object.
  - `x <> y`
    `not (x = y)`.
  - `x.oclType()`
    The type of x.
  - `x.isKindOf ( T )`
    True if T is a supertype (transitive) of the type of x.
  - `T.allInstances()` : Collection
    All the instances of type T.
- Boolean operators:
  - b and b2, b or b2,
    b xor b2, not b
    If any part of a Boolean expression fully determines the result, then it does not matter
    if some other parts of that expression have unknown or undefined results.
  - b implies b2
    True if b is false or if b is true and b2 is true.
  - if b then e1 else e2 endif
    If b is true the result is the value of e1; otherwise, the result is the value of e2.

EXTRA: Overview of Collection Valued Terms

- Size / aggregation:
  - `c->size()`: Integer
    Number of elements in the collection; for a bag or sequence, duplicates are counted as separate items.
  - `c->sum()`: Integer
    Sum of elements in the collection. Elements must be numbers
  - `c->count(e)`: Integer
    The number of times that e is in c.
  - `c->isEmpty()`: Boolean
    Same as `c->size() = 0`.
  - `c->notEmpty()`: Boolean
    Same as `not c->isEmpty()`.
- Equality
  - `c = c2` : Boolean
- Collection membership
  - `c->includes(e)`: Boolean;
    `c->exists ( x | x = e )`.
  - `c->excludes(e)`: Boolean;
    `not c->includes( e )`.
  - `c->includesAll(c2)`: Boolean;
    c includes all the elements in c2.
  - `c->including(e)`: Collection
    The collection that includes all of c as well as e.
  - `c->excluding(e)`: Collection
    The collection that includes all of c except e.

EXTRA: Overview of Collection Valued Terms

- Existential quantifier:
  - `c->exists(x | P )`: Boolean;
    there is at least one element in c, named x, for which predicate P is true.
  - Equivalent notation is:
    `c->exists(P),`
    `c->exists(x:Type | P(x))`
- Universal quantifier:
  - `c->forAll(x | P)`: Boolean;
    for every element in c, named x, predicate P is true.
  - Equivalent notation is:
    `c->forAll(P)`
    `c->forAll(x:Type | P)`
- Selection:
  - `c->select(x | P)`: Collection
    The collection of elements in c for which P is true.
  - Equivalent is: `c->select(P)`
- Filtering:
  - `c->reject(x | P)`: Collection
    `c->select(x | not P)`.
  - Equivalent is: `c->reject(P)`
- Collection:
  - `c->collect(x | E)` : Bag
    The bag obtained by applying E to each element of c, named x.
  - `c.attribute` : Collection
    The collection(of type of c)

Syntax és Semantics

I. Constraints

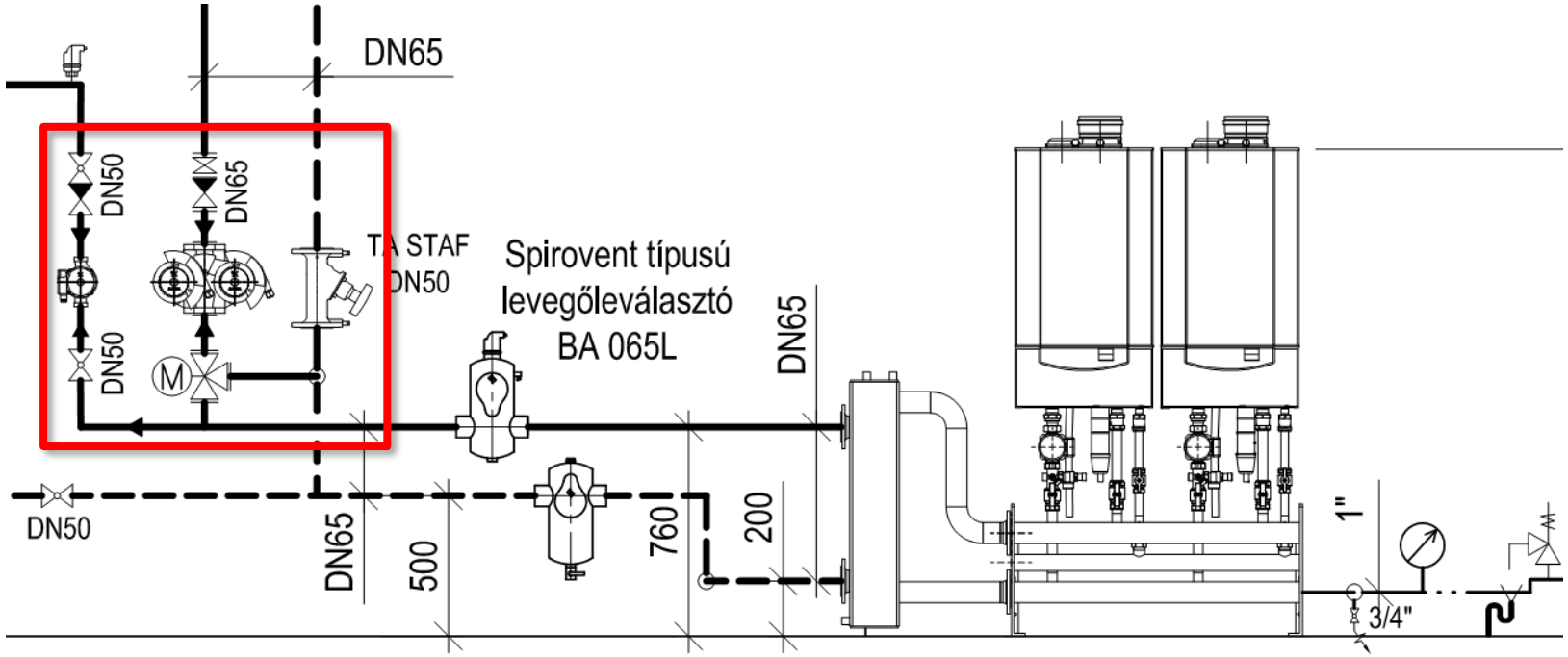
II. Concrete and Abstract syntax

III. Editors

IV. Semantics

Concrete syntax

- How models look like?



Honeywell keverőcsap DN50 K_{VS} 40

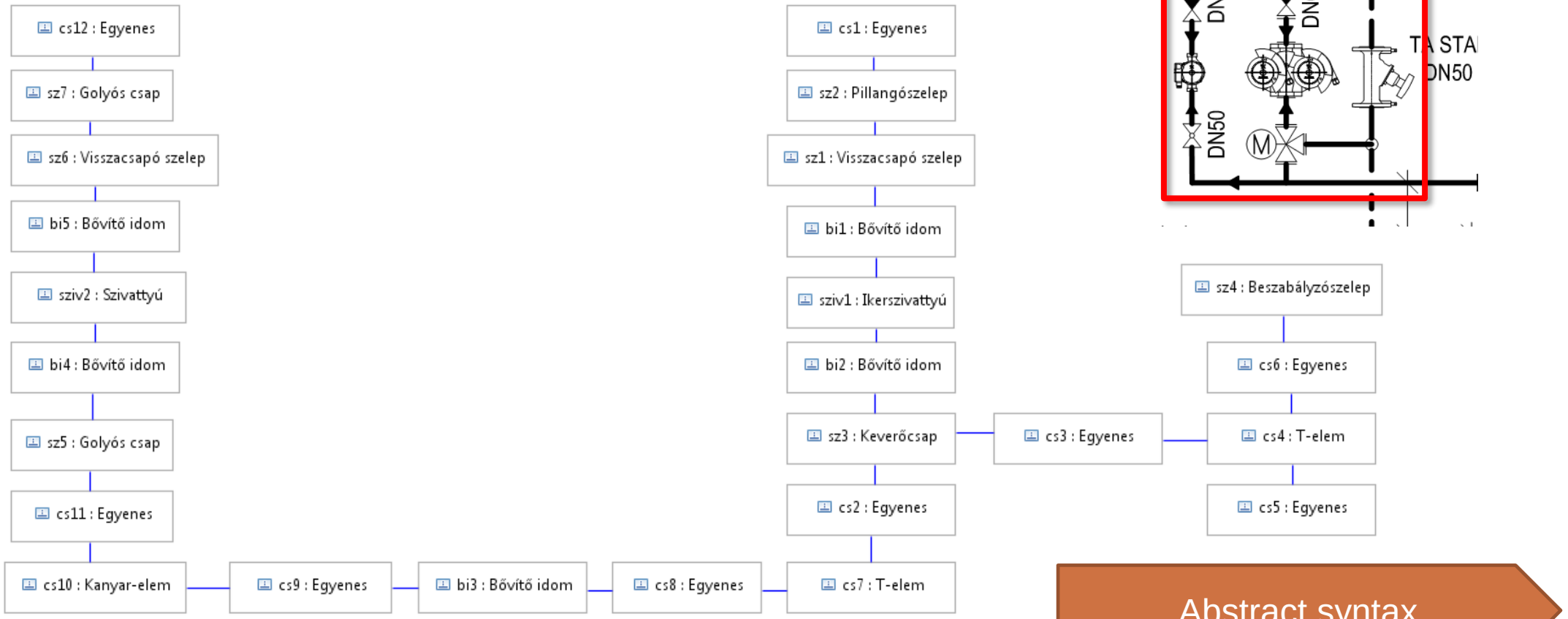
Spirovent típusú iszapleválasztó BE 065L

Remeha Quinta kaszkád rendszer hidrauliku

Concrete syntax

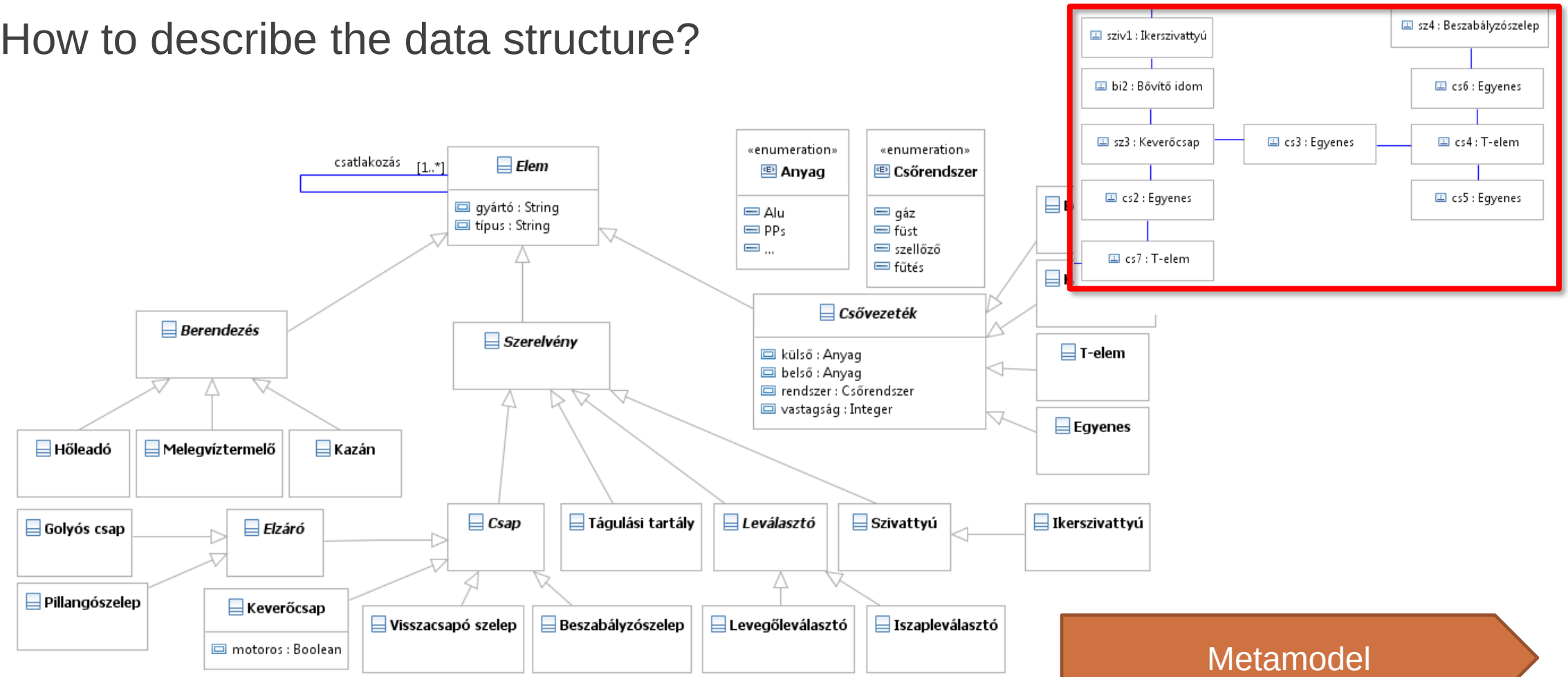
Abstract syntax: model

- How to represent models in a computer?



Abstract syntax: description of the model

- How to describe the data structure?



Definitions

- Separate models to **representation-independent** and **representation-specific** parts.

- Definition **[Abstract syntax]:** is the (abstract) data structure of a model, which excludes representation-specific details.
- Definition **[Concrete syntax]:** is the complete representation of a model (which includes representation-specific details).

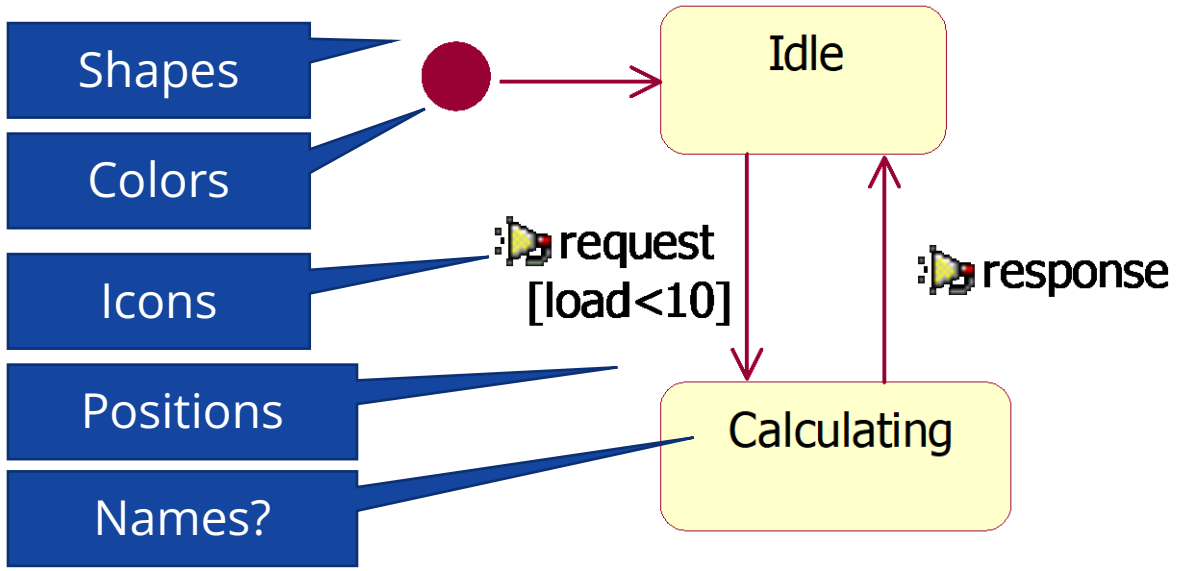
- Abstract and concrete syntax may also denote the representation technique of a modeling language.

„In the concrete syntax of the Yakindu modeling language states are represented by rounded rectangles..”

Concrete syntax

- What is part of the concrete syntax, but not part of the abstract?

Concrete graphical syntax



Concrete textual syntax

```
String state = "idle";
request() {
   if (state == "idle" &&
   this.load<10)
   state = "calculating";
}
response() { /* … */ }
```

Formatting

Keywords

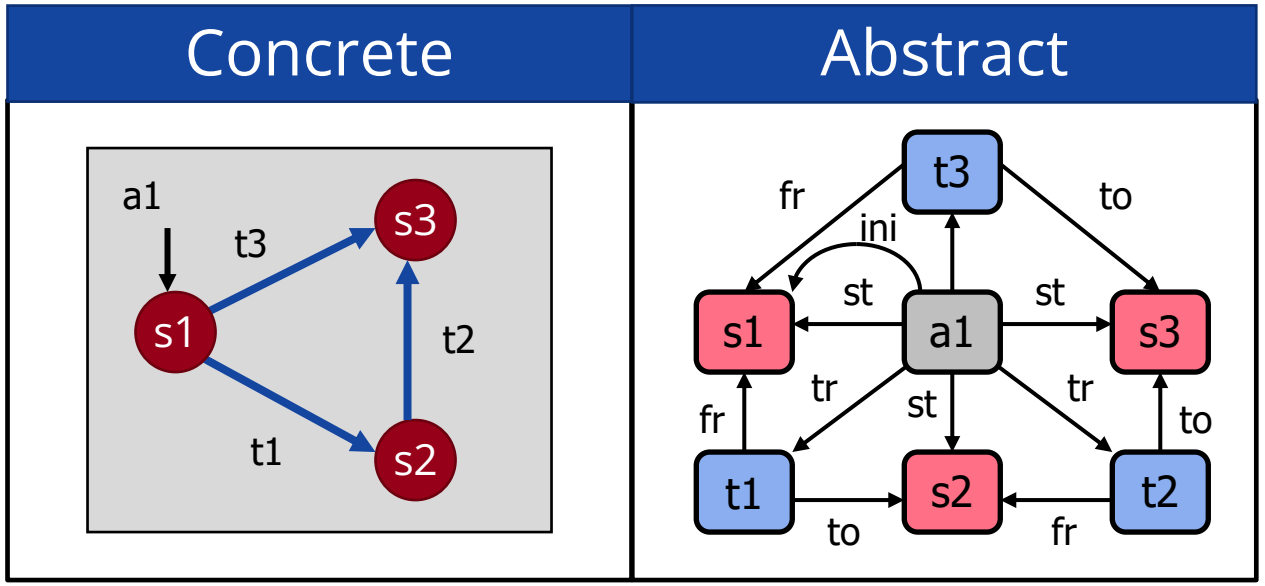
Grammar?

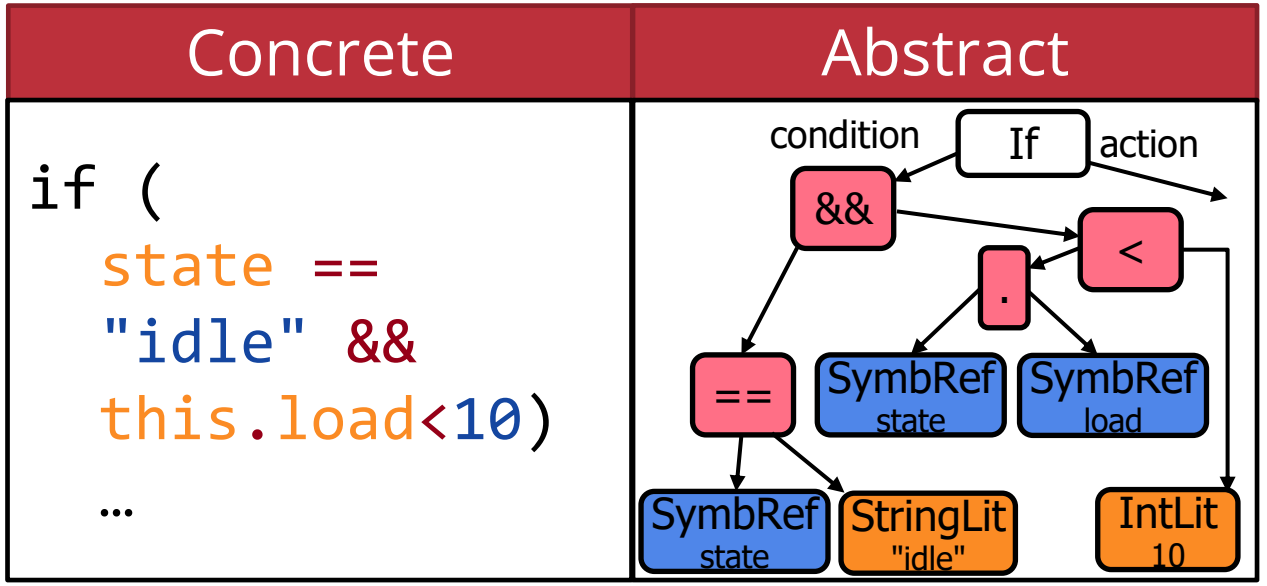
Comments?

Syntax highlight?

Abstract syntax

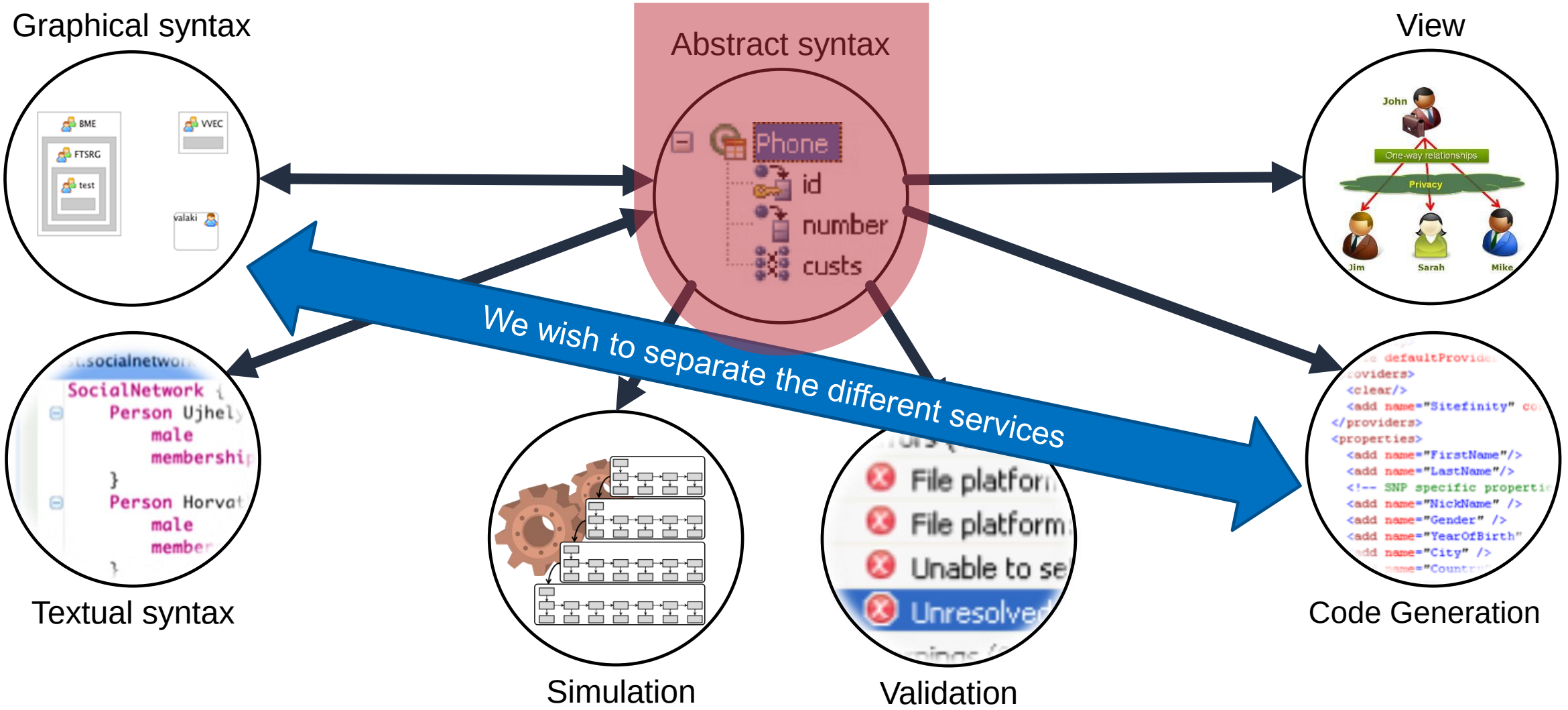
- How to capture the abstract syntax of a model?



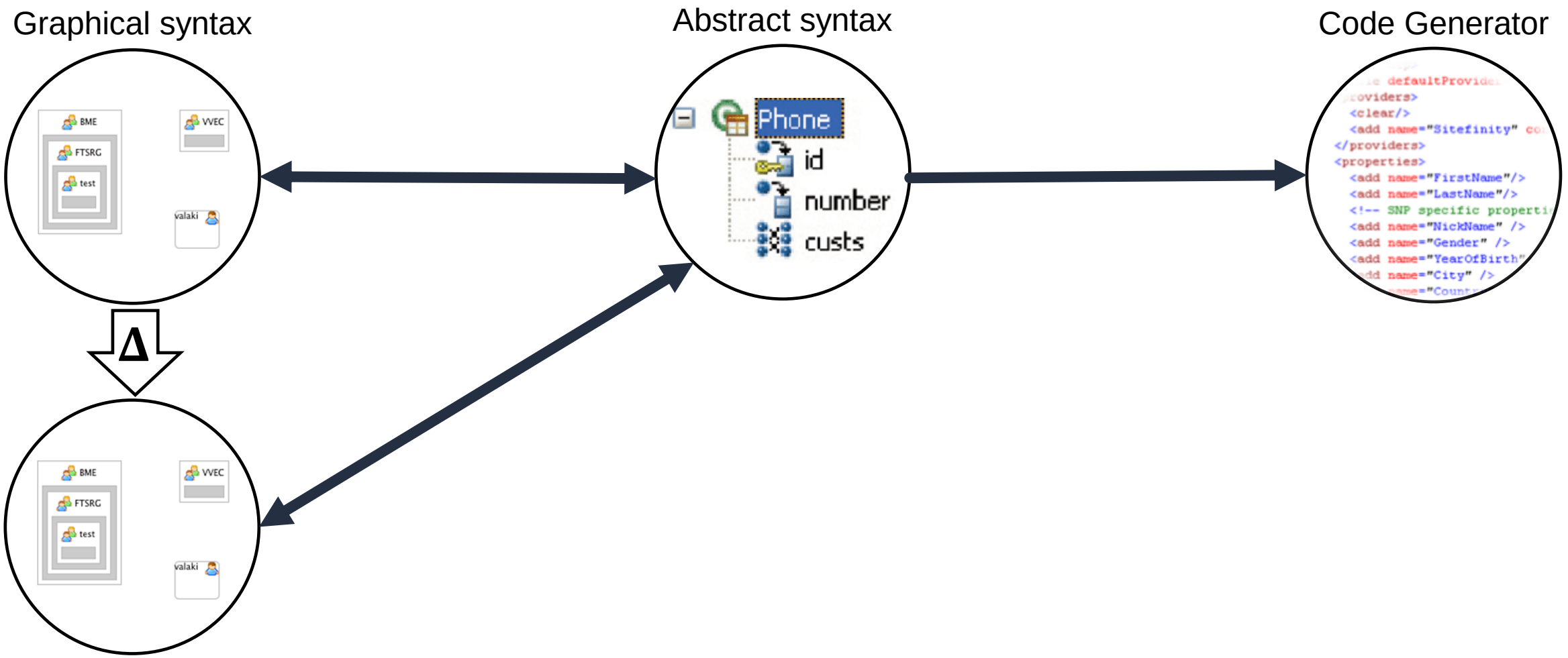


- **Abstract syntax:** typically a graph-based structure.

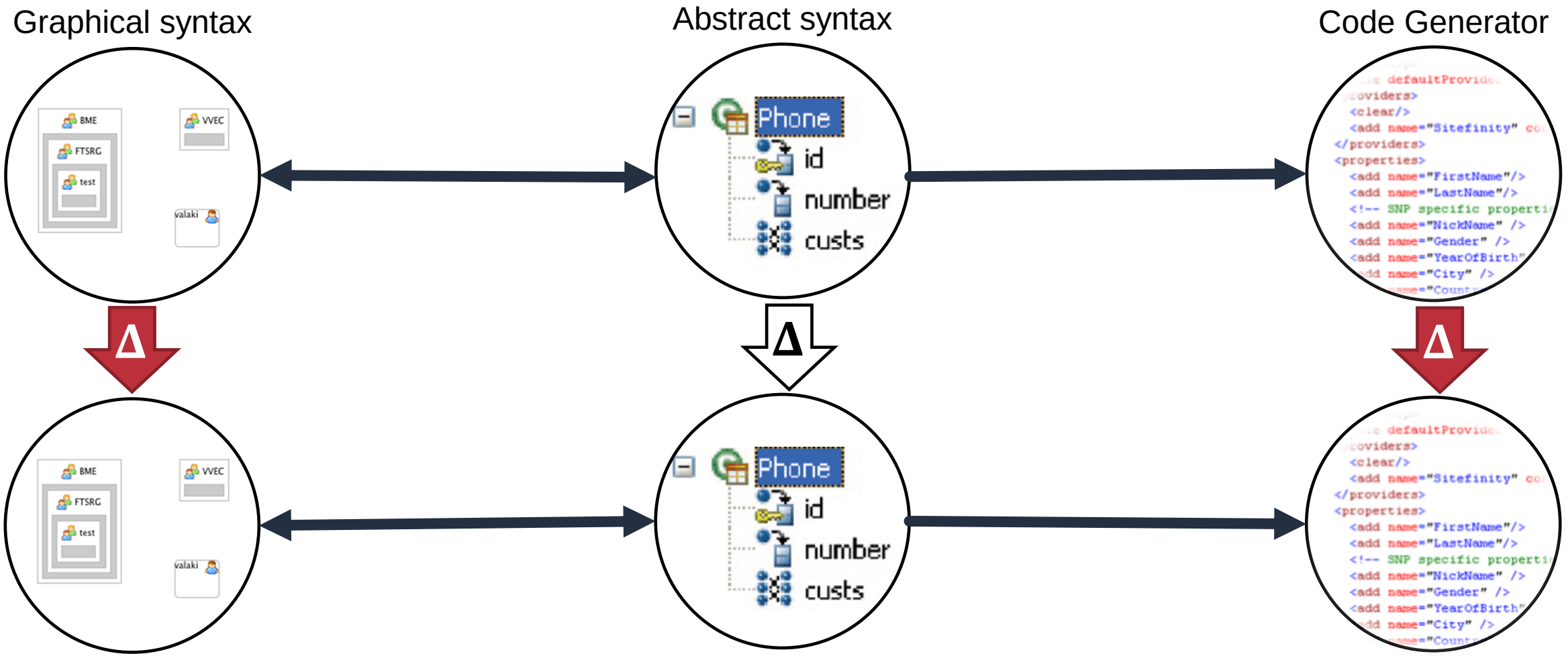
How to separate concrete and abstract syntax?



Change the concrete syntax, but not the abstract syntax



Change the abstract syntax



Goal: separate the concrete ans abstract syntax without the modeling services interfering

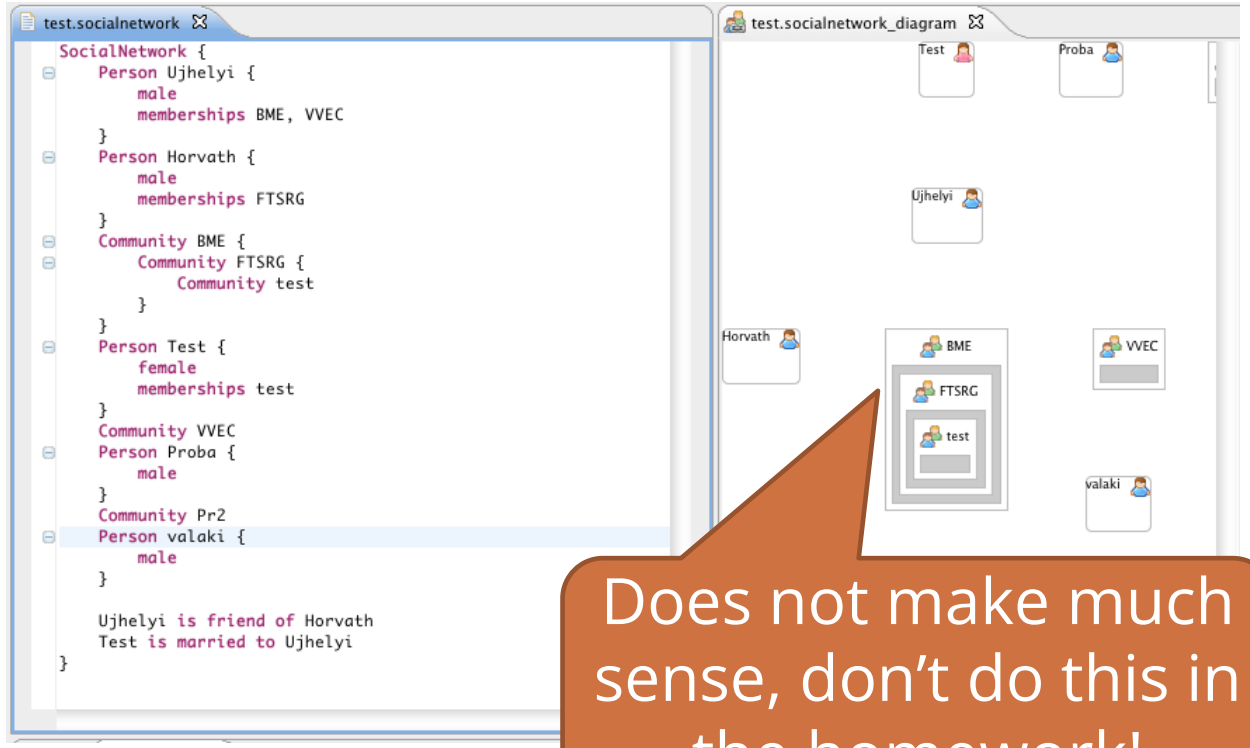
Multiplicity of Notations

- 1 abstract syntax → many textual and visual notations
  - Human-readable-writable textual or visual syntax
  - Textual syntax for exchange or storage (typically XML)
  - In case of UML, each diagram is only a partial view
- 1 abstract model → many concrete forms in 1 syntax!
  - Whitespace, diagram layout
  - Comments
  - Syntactic sugar
- 1 semantic interpretation → many abstract models
  - e.g. UML2 Attribute vs. one-way Association

Textual + Graphical

Same model, two syntaxes

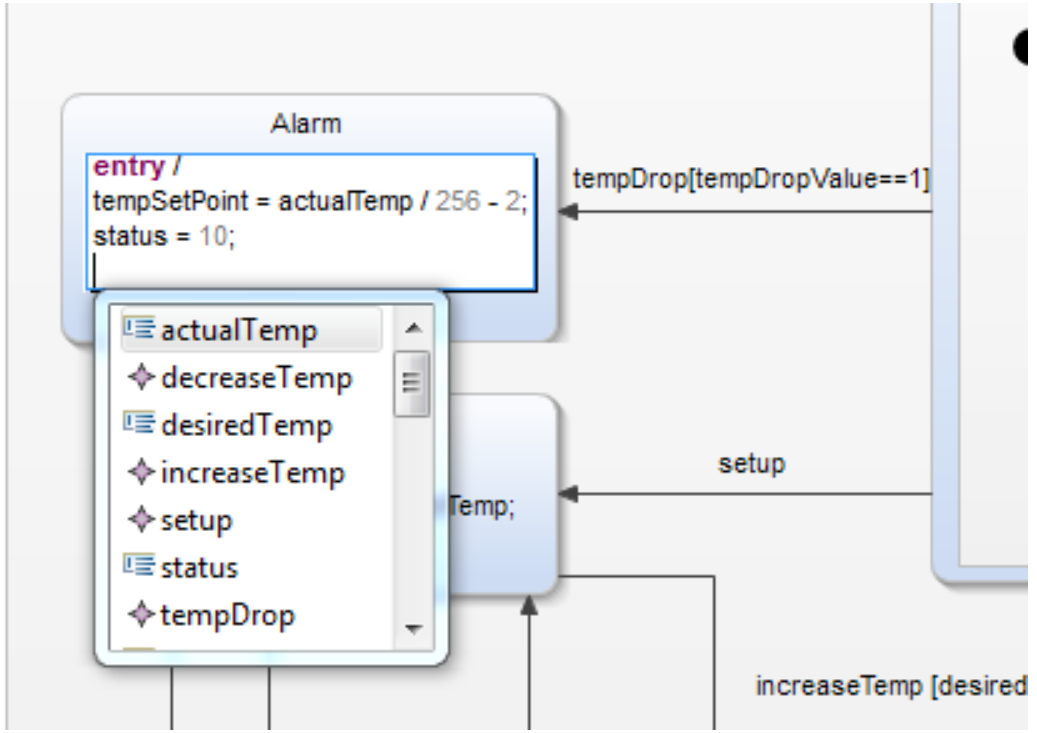
- Text editor + graphical view
- Xtext Generic Viewer



Does not make much sense, don't do this in the homework!

Different aspects of model

- Diagram with text fields
- Embedded Xtext support



Syntax és Semantics

I. Constraints

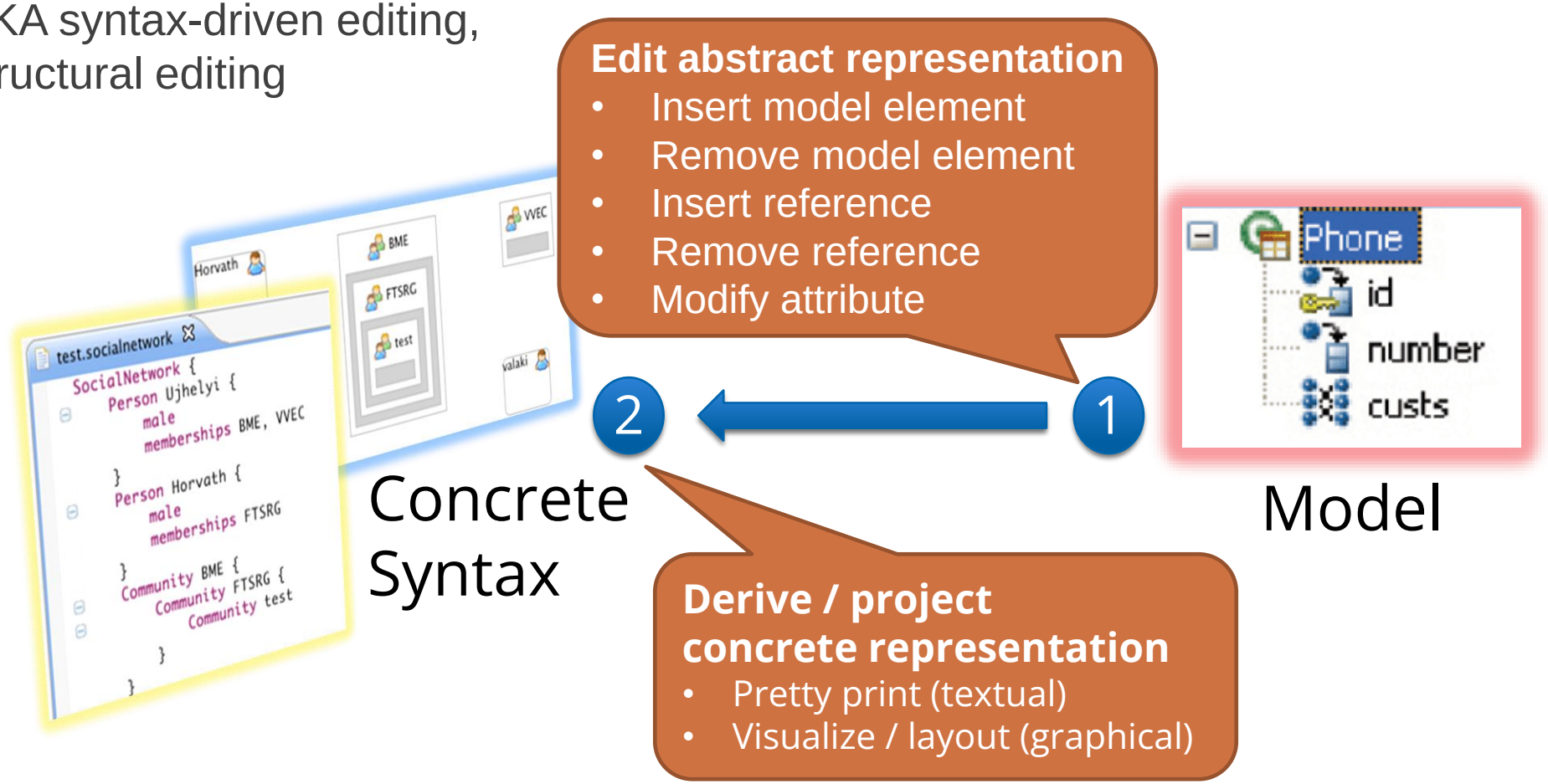
II. Concrete and Abstract syntax

III. Editors

IV. Semantics

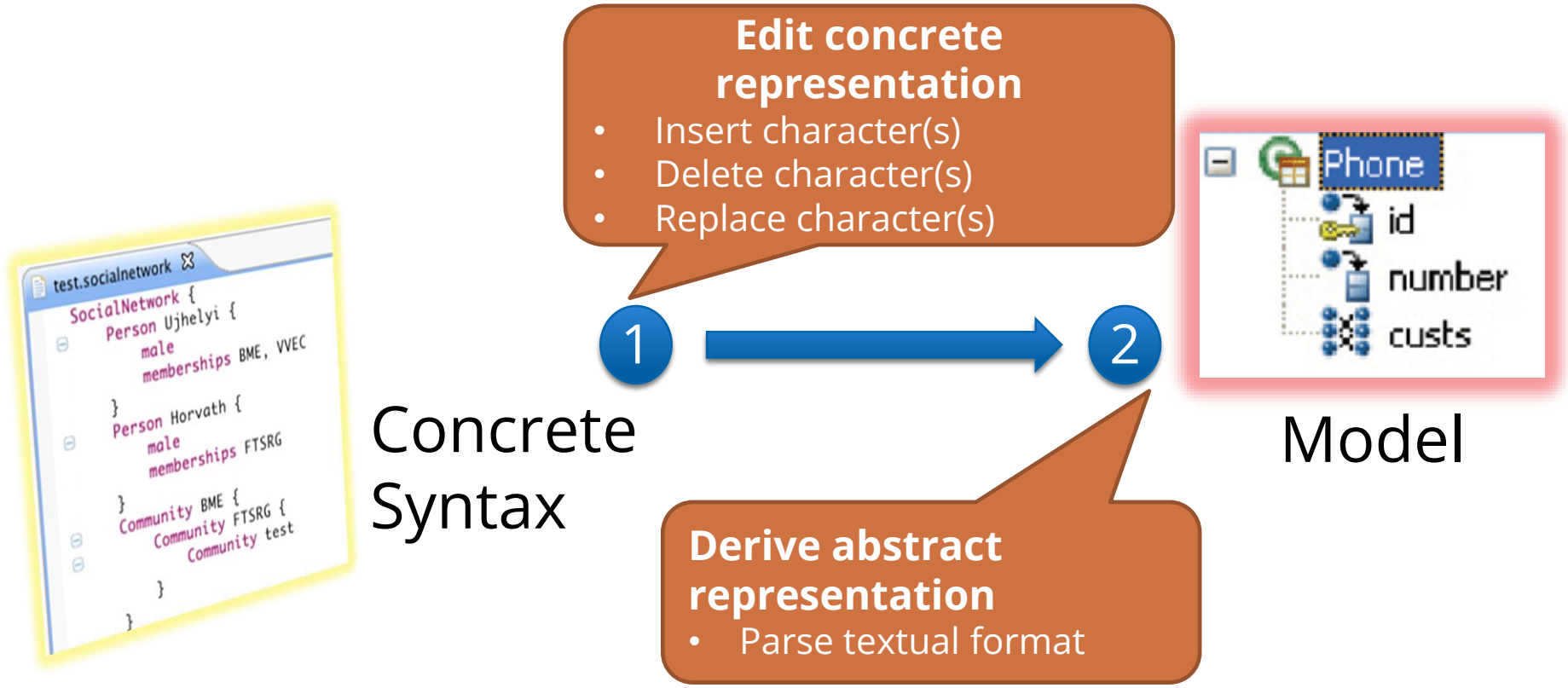
Projectional vs Raw editing

- Workflow 1: **projectional editing**
  - > AKA syntax-driven editing, structural editing



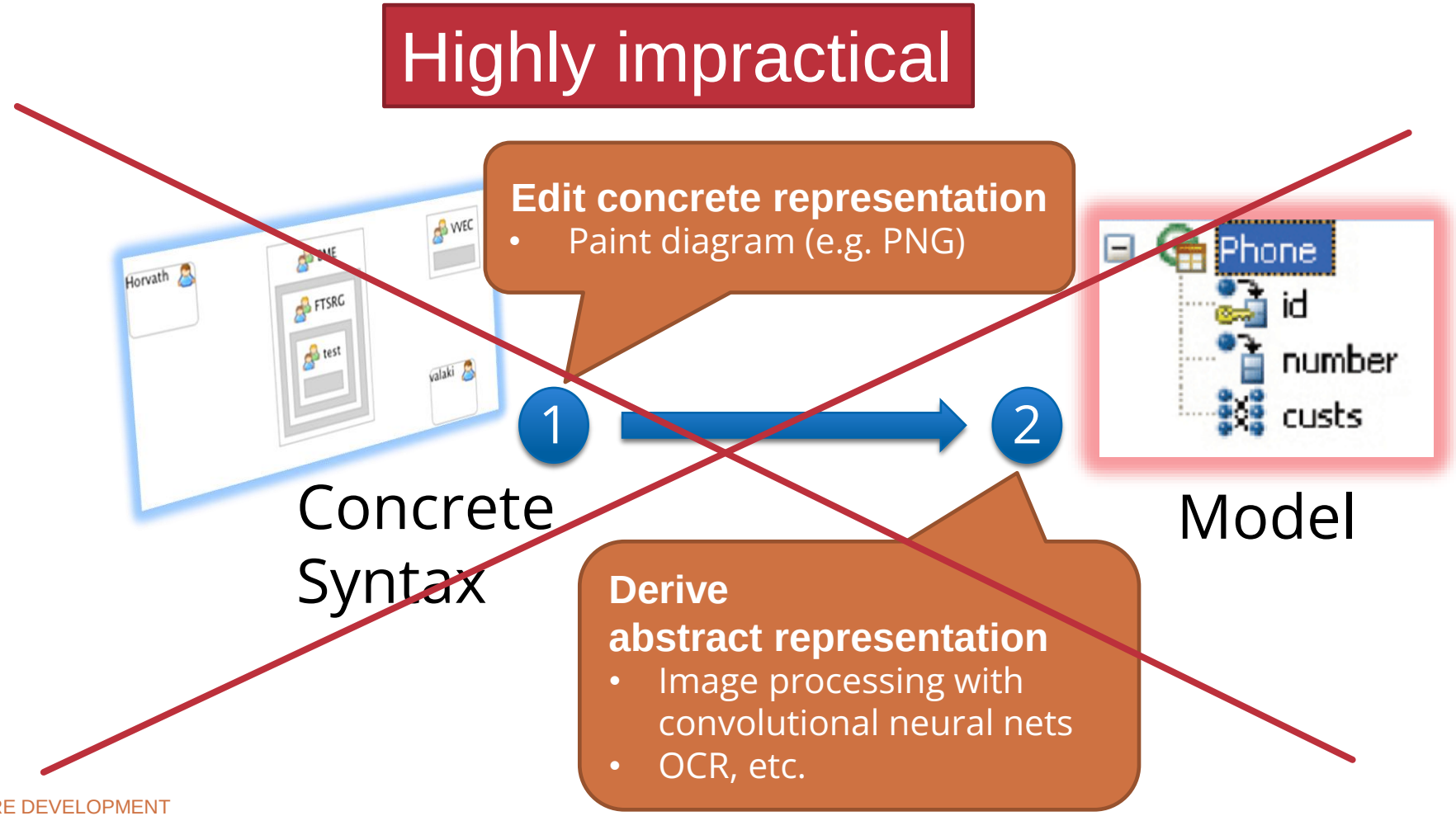
Projectional vs Raw editing

- Workflow 2: **raw editing** (w. textual syntax)
  - > AKA source editing



Projectional vs Raw editing

- Workflow 2: **raw editing** (w. graphical syntax)

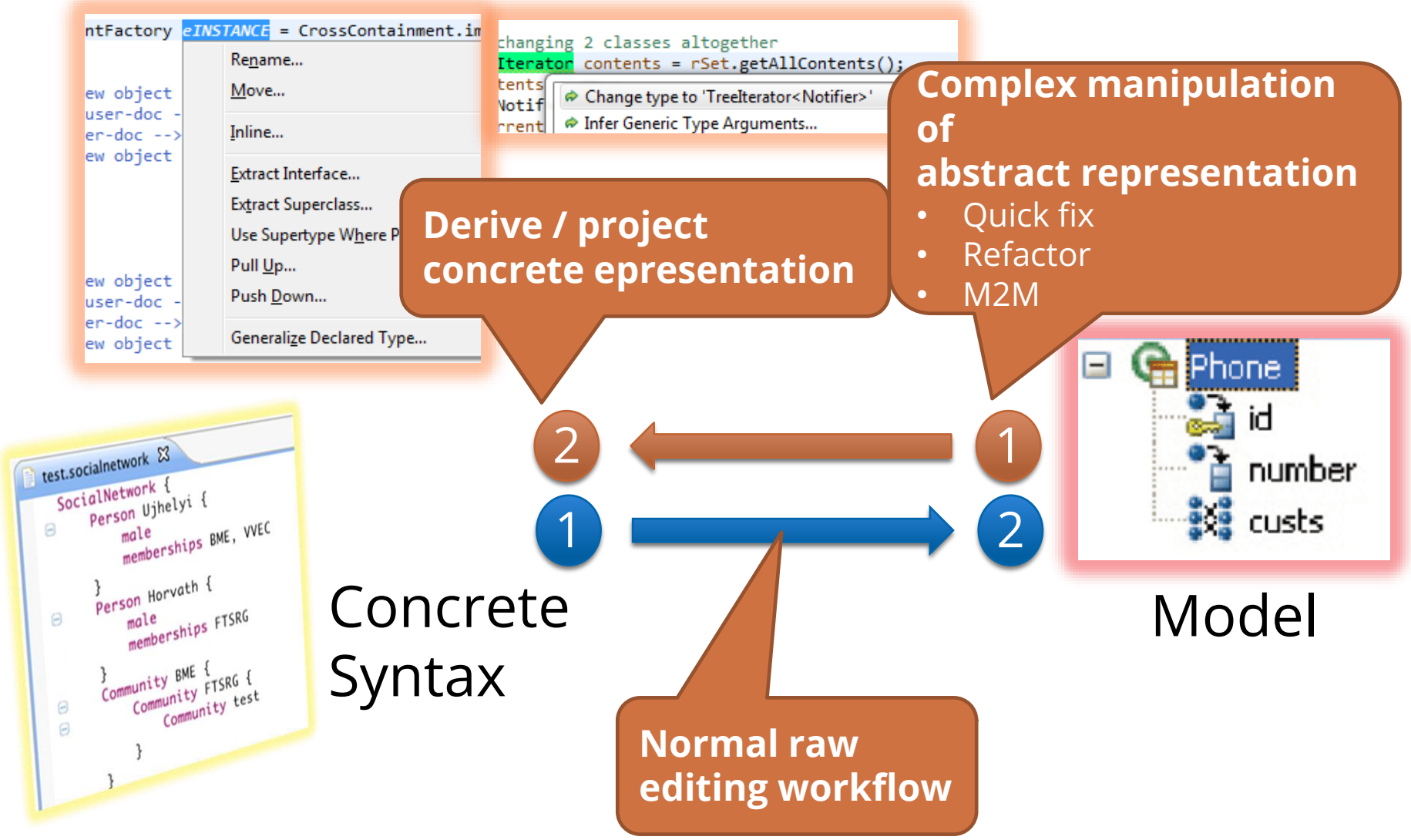


Projectional vs Raw editing

„Feature matrix” + examples

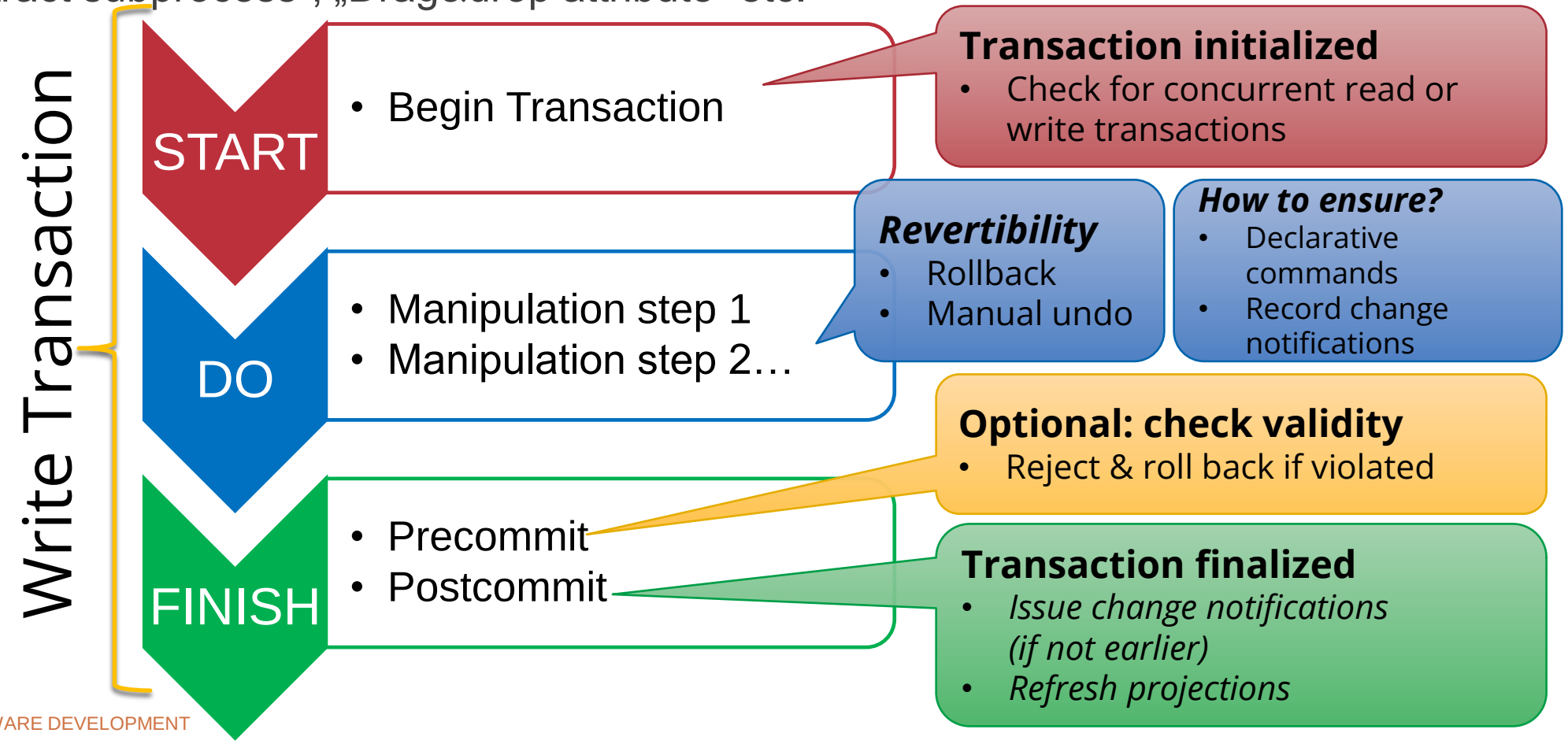
	Graphical syntax	**Textual syntax**
Raw editing	~~X~~	Typical 
Projectional editing	Typical 	Rare 

Mixed workflow



Transactions in projectional editing

- Complex manipulation sequence as single action
  - > „Extract subprocess”, „Drag&drop attribute” etc.

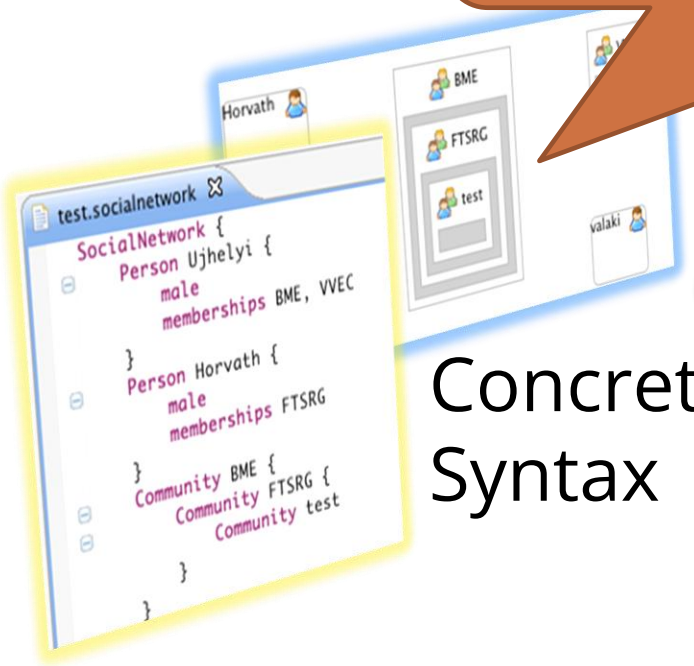


Superfluous notational parameters

- Workflow 1: **projectional editing**

Must include *notational parameters:*
- Whitespace and comments, etc. (textual)
- Layout, edge routing, size, shape, etc. (graphical)

...even though not domain information



Concrete Syntax

2

1

Model

Derive / project concrete representation
- Pretty print (textual)
- Visualize / layout (graphical)

Deriving notational parameters

- Notational parameters can be…
  - …”baked into” projection code
    - e.g. all lines are black, all fonts are 10pt (graphical)
    - e.g. apply this code formatting template (textual)
  - …derived from domain information
    - e.g. shape determined by type, color by visibility

Problem 1:
Editable parameters cannot be a function of the domain model, must be stored

Problem 2:
Providing sane values is difficult for some parameters e.g. position in diagram

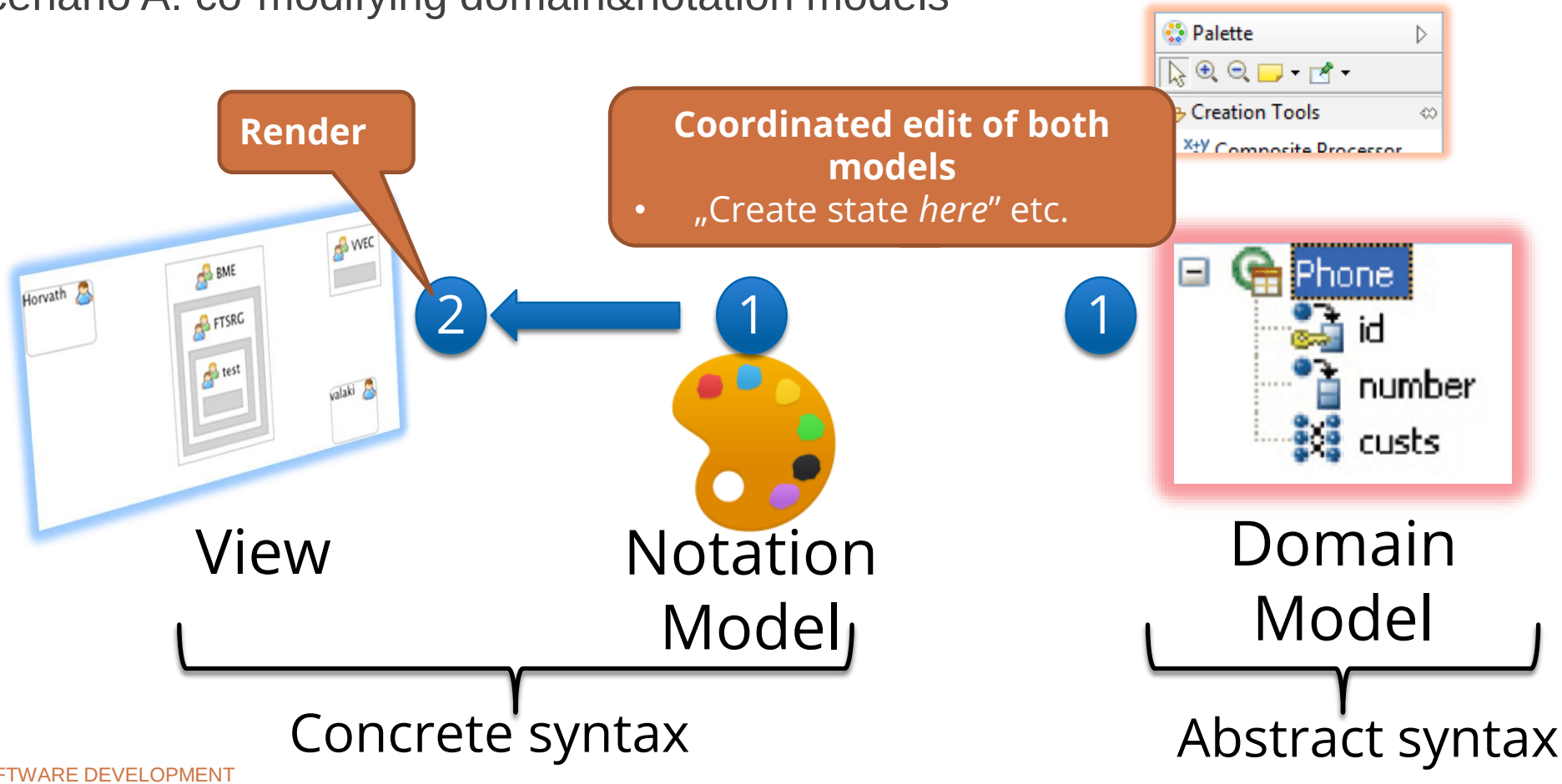
  - …**stored in the model**

Notation/view models

- Decompose model:
  - Domain / Semantic model (abstract syntax)
  - **Notation model** (view model): presentation state
    - may be editable by user
    - but still needs derivable defaults → see layouting
- Generic implementation in GMF and Graphiti
  - Based on EMF, in fact
- Often stored in external files
  - Separation of concerns
  - E.g. code generator not interested in view information

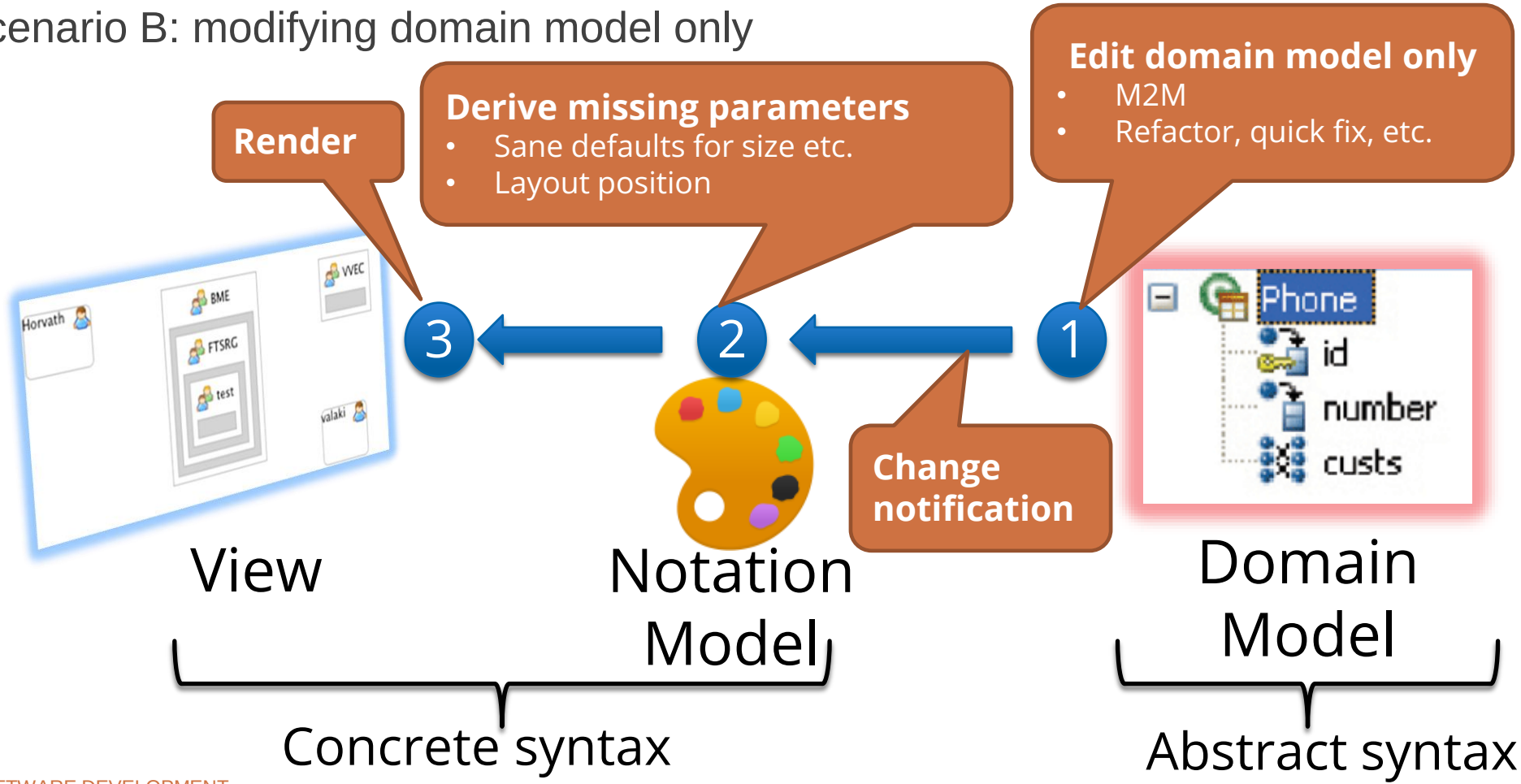
Editing workflow with notation models

- Workflow 1: **projectional editing**
  - > Scenario A: co-modifying domain&notation models



Editing workflow with notation models

- Workflow 1: **projectional editing**
  - > Scenario B: modifying domain model only



Syntax és Semantics

I. Constraints

II. Concrete and Abstract syntax

III. Editors

IV. Semantics

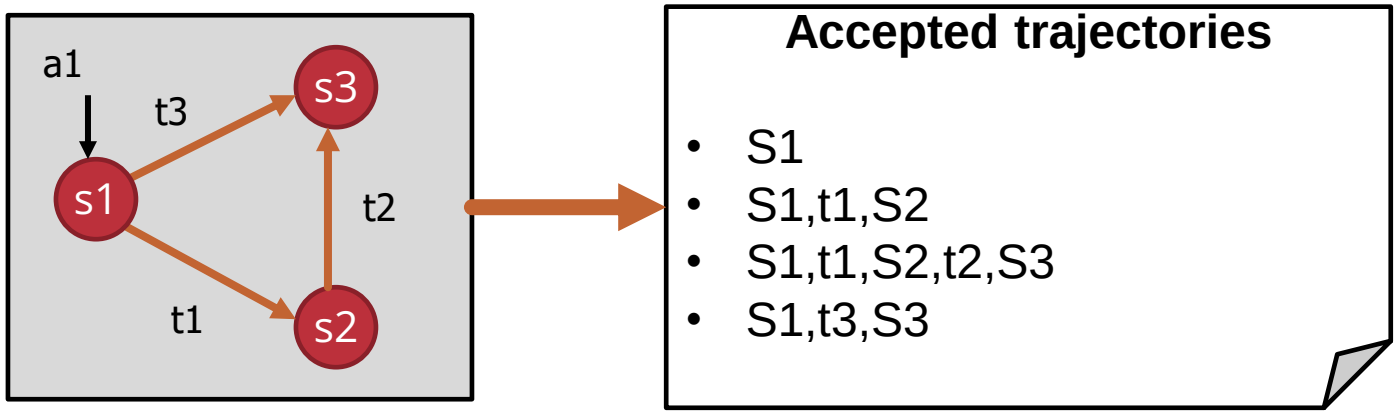
Semantics

- We talked a lot about syntax.
- Semantics: the meaning of concepts in a language
  - Static: what does a snapshot of a model mean?
  - Dynamic: how does the model change/evolve/behave?
- The semantics of a modeling language maps models to a real-world or a mathematical semantic domain.

Types of semantics

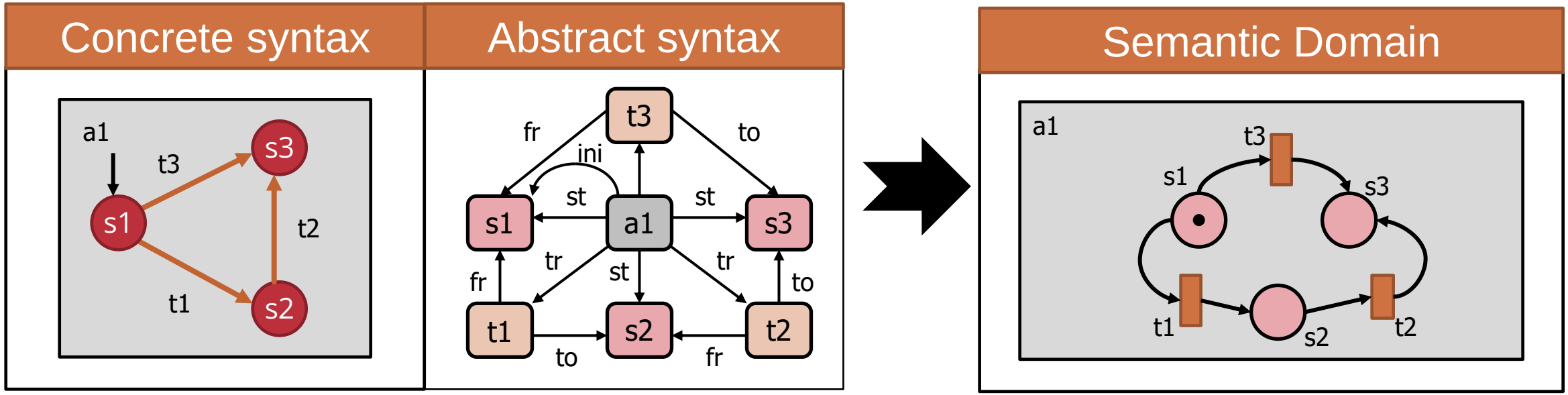
- Static Semantics
  - > Interpretation of metamodel elements: meaning of concepts in the abstract syntax

- **Formal**: mathematical statements about the interpretation
- Semantics is captured by mathematical axioms



Example: denotational semantics

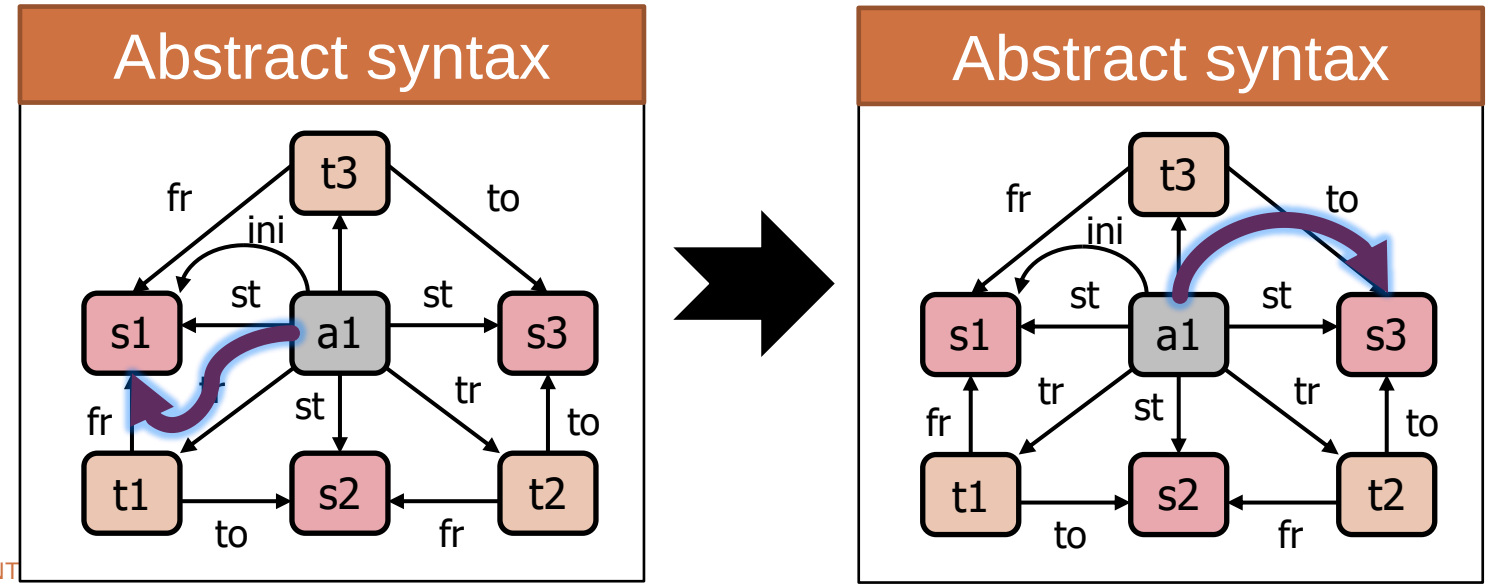
- **Denotational** (Translational)
  - > translating concepts in one language to another language (called **semantic domain**)
  - > „compiled”
  - > E.g. explaining state machines as Petri-net



Operational Semantics

- **Operational**
  - > Modeling the operational behavior of language concepts
  - > „interpreted”
  - > e.g. defining how the finite automaton may change state at run-time
  - > Sometimes dynamic features are introduced only for formalizing dynamic sematics

Dynamic feature : current



Thank you for your attention