

Model-Based Software Engineering

Lecture 06 – Concrete Syntax

Prof. Dr. Joel Greenyer



May 24, 2016



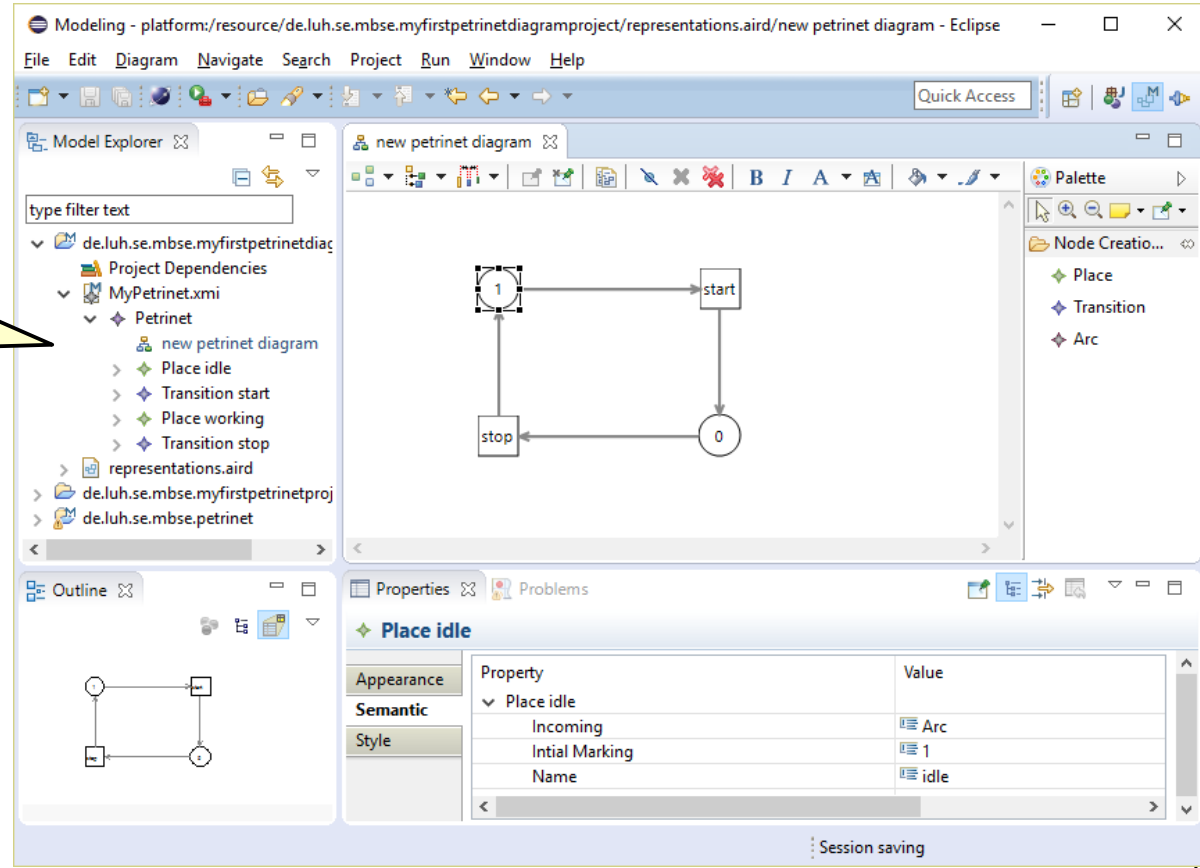
Acknowledgment

- The slides of this lecture are inspired by lecture slides from
 - *Ekkart Kindler*: Course on Advanced Topics in Software Engineering, DTU Compute, 2015.
 - <http://www2.imm.dtu.dk/courses/02265/f15/schedule.shtml>
 - *Ina Schäfer, Christoph Seidl*: Modellbasierte Softwareentwicklung, TU Braunschweig, 2015.
 - *Steffen Becker*: Model-Driven Software Development, Universität Paderborn, 2013
 - The Eclipse Open Model CourseWare (OMCW) Project:
 - <https://eclipse.org/gmt/omcw/>

in the last lecture...

- Eclipse **Sirius** works by *interpreting* a graphical mapping of the model
 - no code generation required

example: Petri net editor created in 15 Minutes!



in the last lecture...

4.2. Textual syntax

Excursion on Compilers – Syntax Analysis

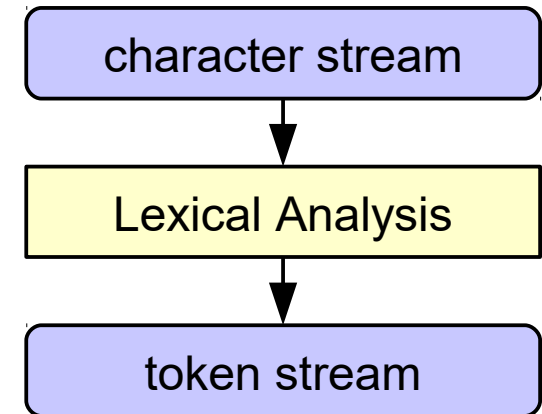
in the last lecture...

- Steps in the textual syntax analysis:

character stream

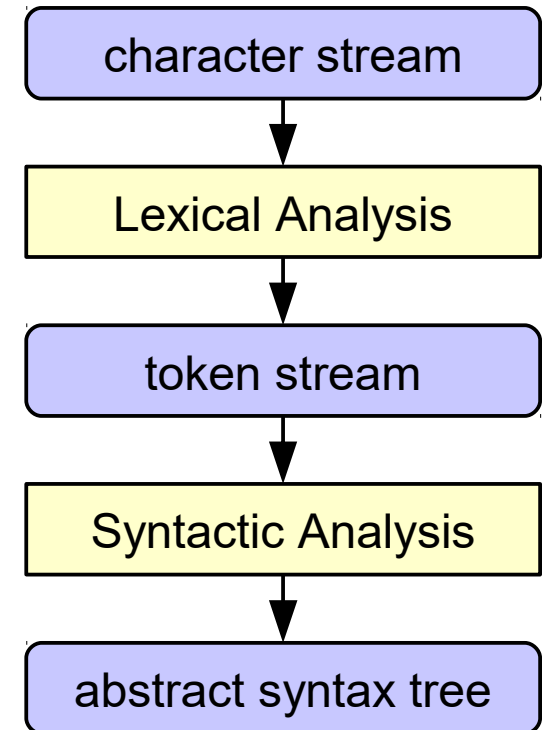
in the last lecture...

- Steps in the textual syntax analysis:
 - Lexical Analysis
 - partitions character stream into tokens (removes whitespaces, identifies keywords, identifiers, ...)



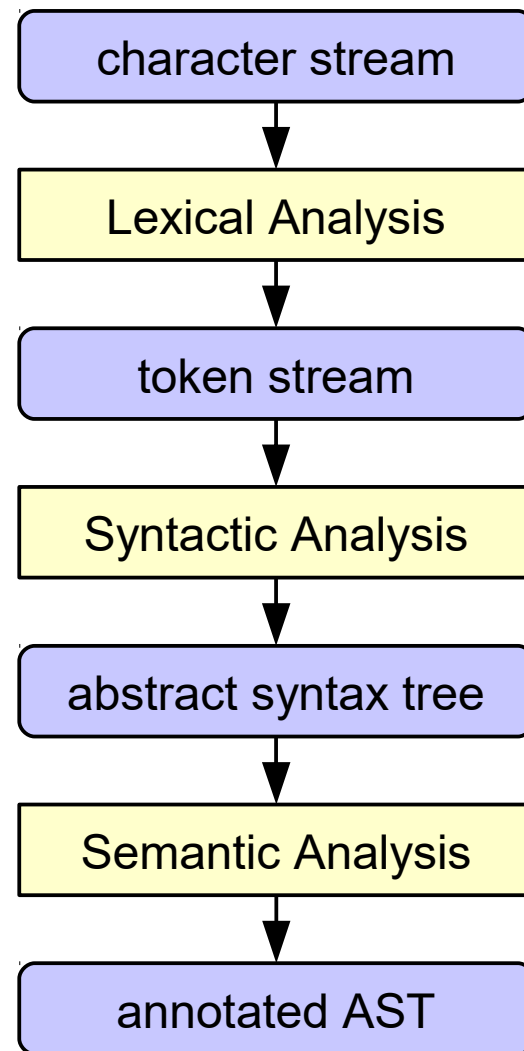
in the last lecture...

- Steps in the textual syntax analysis:
 - Lexical Analysis
 - partitions character stream into tokens (removes whitespaces, identifies keywords, identifiers, ...)
 - Syntax Analysis
 - context-free analysis identifies abstract syntax tree (AST) structure



in the last lecture...

- Steps in the textual syntax analysis:
 - Lexical Analysis
 - partitions character stream into tokens (removes whitespaces, identifies keywords, identifiers, ...)
 - Syntax Analysis
 - context-free analysis identifies abstract syntax tree (AST) structure
 - Semantic Analysis
 - analyze cross-references of AST elements (variable scoping, type conformance, ...)



Parser generators – Example: ANTLR

in the last lecture...

- There exist **frameworks for constructing compilers** that can generate lexer and parser components (and other things) from a language definition in the form of a grammar
- a popular example: **ANTLR**
 - takes as input a context-free grammar in Extended Backus–Naur Form (EBNF)

Parser generators – Example: ANTLR

in the last lecture...

- ANTLRWorks Interpreter visualizes the result of the syntactic analysis:

The screenshot displays the ANTLRWorks Interpreter interface. The top panel shows the grammar definition for 'grammar T;'. The rule 'def' is highlighted, showing its definition: 'def : modifier+ 'int' ID '=' INT ';' | modifier+ 'int' ID ';''. The bottom panel shows the syntax diagram for the rule 'def', which is currently selected in the dropdown menu. The diagram illustrates the hierarchical structure of the rule, showing the root node '<grammar T>' branching into 'def', which then branches into 'modifier', 'int', 'x', '=', '3', and ';'. The 'modifier' node further branches into 'public' and 'static'. The input text 'public static int x = 3;' is visible on the left. The bottom status bar indicates '5 rules', '5:33', and 'Writable'.

Xtext – a Framework for Building Textual Languages in Eclipse

in the last lecture...

- **Xtext** is a framework for building textual languages and editors within Eclipse (also IntelliJ IDEA and web browser)
- The language is built based on a grammar definition similar to EBNF
 - but with extra features for referencing a corresponding Ecore metamodel
 - Generation of an Ecore metamodel from a grammar is also supported
- Xtext can create the Ecore metamodel, lexer/parser, and editor from the grammar definition
 - The editor supports syntax checking, highlighting, and code completion, renaming/refactoring, and has extensions for implementing quick-fixes, and other functionality

Xtext – Simple Example: Company

in the last lecture...

```
// automatically generated by Xtext
grammar de.luh.se.mbse.company.cml.CML with org.eclipse.xtext.common.Terminals
```

```
import "platform:/resource/de.luh.se.mbse.company/model/company.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

```
Company returns Company:
    { Company }
    'Company'
    name=EString
    '{'
        ('department' '{' department+=Department ( "," department+=Department)* '}' )?
    '}';
```

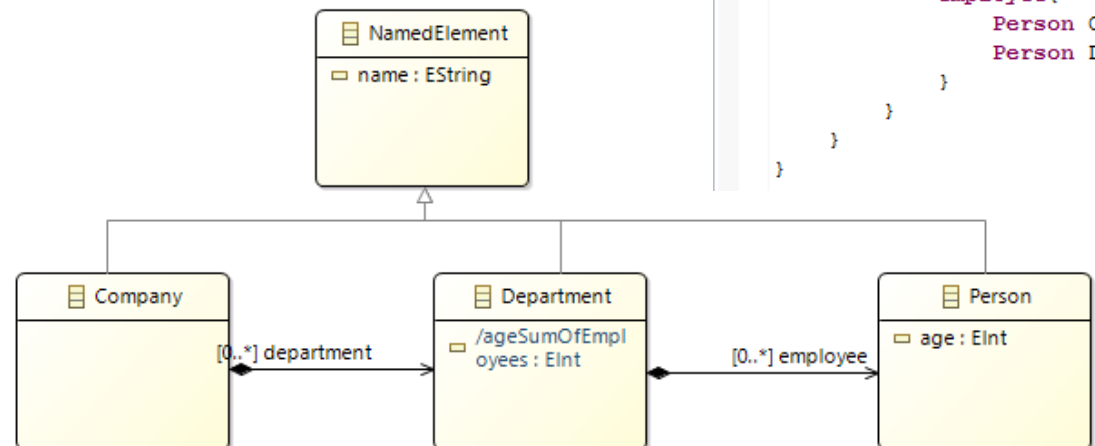
```
EString returns ecore::EString:
    STRING | ID;
```

```
Department returns Department:
    { Department }
    'Department'
    name=EString
    '{'
        ('employee' '{' employee+=Person ( "," employee+=Person)* '}' )?
    '}';
```

```
Person returns Person:
    { Person }
    'Person'
    name=EString
    '{'
        ('age' age=EInt)?
    '}';
```

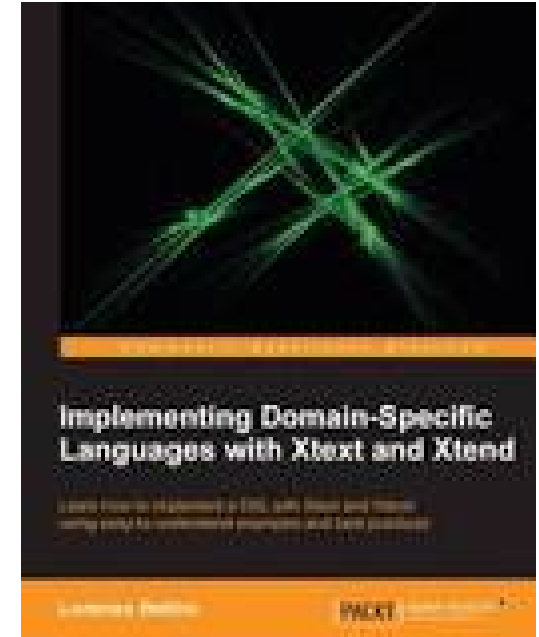
```
EInt returns ecore::EInt:
    '-'? INT;
```

```
mycompany.cml
Company MyCompany{
    department {
        Department Finance{
            employee{
                Person Alison {age 21},
                Person Beverly {age 34}
            }
        },
        Department Marketing{
            employee{
                Person Charlie {age 43},
                Person Dave {age 39}
            }
        }
    }
}
```



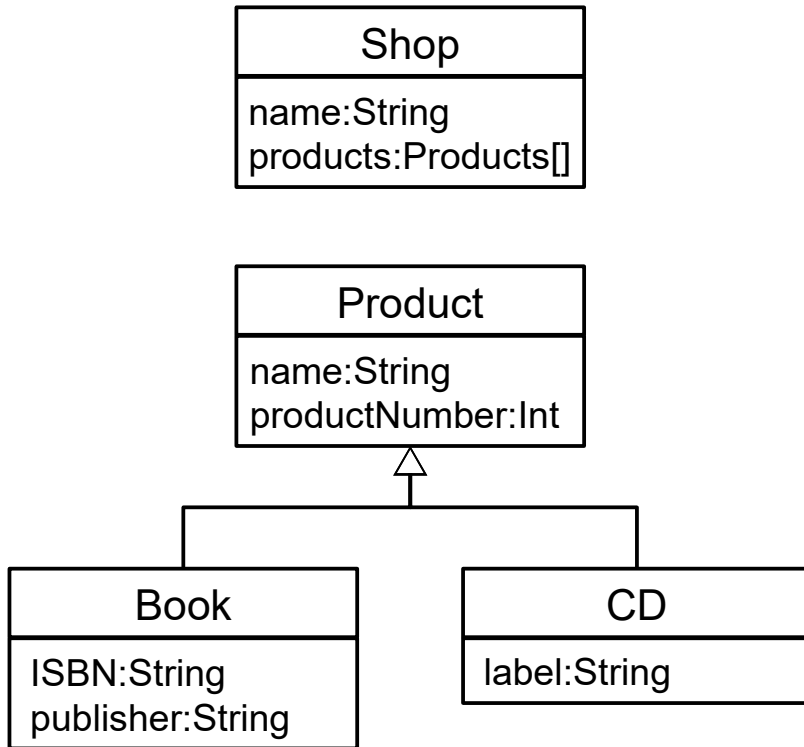
4.3. Xtext

- Bettini, Lorenzo: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, 2013.
- The *Entities* example (coming next) is taken from this book.



Example: Defining a Language for Entities

- Entities: A simple class-model-like DSL



graphical syntax

```

entity Shop{
    String name;
    Product[] product;
}

entity Product{
    String name;
    Int productNumber;
}

entity Book extends Product{
    String ISBN;
    String publisher;
}

entity CD extends Product{
    String label;
}
  
```

textual syntax
(this is what we are going to define)

Example: Defining a Language for Entities

- We start with a very simple version of the language

```
grammar de.luh.se.mbse.entities.Entities  
with org.eclipse.xtext.common.Terminals
```

```
generate entities "http://www.luh.de/se/mbse/entities/Entities"
```

Model:

```
entities+=Entity*;
```

Entity:

```
'entity' name=ID;
```


Example: Defining a Language for Entities

- We start with a very simple version of the language

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "http://www.luh.de/se/mbse/entities/Entities"

Model:

entities+=Entity*;

Entity:

'entity' name=ID;

name of the grammar /
language

Example: Defining a Language for Entities

- We start with a very simple version of the language

```
grammar de.luh.se.mbse.entities.Entities
with org.eclipse.xtext.common.Terminals
```

```
generate entities "http://www.luh.de/se/mbse"
```

```
Model:
```

```
    entities+=Entity*;
```

```
Entity:
```

```
    'entity' name=ID;
```

reuses the grammar
Terminals, which
defines strings,
numbers, comments

Example: Defining a Language for Entities

- We start with a very simple version of the language

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "<http://www.luh.de/se/mbse/entities/Entities>"

Model:

entities+=Entity*;

Entity:

'entity' name=ID;

Will generate a
corresponding Ecore
model with the given
nsURI

Example: Defining a Language for Entities

- We start with a very simple version of the language

```
grammar de.luh.se.mbse.entities.Entities
```

```
with org.eclipse.xtext.common.Terminals
```

```
generate entities "http://www.luh.de/se/mbse/entities/Entities"
```

Model:

```
entities+=Entity*;
```

Entity:

```
'entity' name=ID;
```

Example: Defining a Language for Entities

- We start with a very simple version of the language

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "<http://www.luh.de/se/mbse/entities/Entities>"

Model:

entities+=Entity*;

a **rule**: when a rule is applied in the parsing process, a corresponding EObject is created

Entity:

'entity' name=ID;

Example: Defining a Language for Entities

- We start with a very simple version of the language

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "<http://www.luh.de/se/mbse/entities/Entities>"

Model:

entities+=Entity*;

a **rule**: when a rule is applied in the parsing process, a corresponding EObject is created

Entity:

'entity' name=ID;

the first rule is the
start rule

Example: Defining a Language for Entities

- We start with a very simple version of the language

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "<http://www.luh.de/se/mbse/entities/Entities>"

Model:

entities+=Entity*;

Entity:

'entity' name=ID;

an **assignment**: assigns the parsed information to a feature of the currently produced object

Example: Defining a Language for Entities

- We start with a very simple version of the language

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "<http://www.luh.de/se/mbse/entities/Entities>"

Model:

entities+=Entity*;

Entity:

'entity' name=ID;

First rule:

- Model is the type of the root element of the AST model (Every Xtext rule corresponds to an EClass in the corresponding Ecore model)
- Model contains a collection entities of zero or many Entity elements.

Example: Defining a Language for Entities

- We start with a very simple version of the language

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "<http://www.luh.de/se/mbse/entities/Entities>"

Model:

entities+=Entity*;

Entity:

+= operator:
entities is a
collection (many-
valued EReference)

First rule:

- Model is the type of the root element of the AST model (Every Xtext rule corresponds to an EClass in the corresponding Ecore model)
- Model contains a collection entities of zero or many Entity elements.

Example: Defining a Language for Entities

- We start with a very simple version of the language

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "<http://www.luh.de/se/mbse/entities/Entities>"

Model:

entities+=Entity*;

Entity:

'entity' name=ID;

An Entity statement starts with the keyword "entity", followed by a name

IDs from `org.eclipse.xtext.common.Terminals`

- ID is defined in `org.eclipse.xtext.common.Terminals`

Entity:

```
'entity' name=ID ;
```

IDs from `org.eclipse.xtext.common.Terminals`

- ID is defined in `org.eclipse.xtext.common.Terminals`

Entity:

```
'entity' name=ID ;
```

- A look into `Terminals` shows what valid IDs are:

IDs from `org.eclipse.xtext.common.Terminals`

- ID is defined in `org.eclipse.xtext.common.Terminals`

Entity:

```
'entity' name=ID ;
```

- A look into `Terminals` shows what valid IDs are:

`terminal` ID:

IDs from `org.eclipse.xtext.common.Terminals`

- ID is defined in `org.eclipse.xtext.common.Terminals`

Entity:

```
'entity' name=ID ;
```

- A look into `Terminals` shows what valid IDs are:

terminal ID:

```
'^'?('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

IDs from org.eclipse.xtext.common.Terminals

- ID is defined in org.eclipse.xtext.common.Terminals

Entity:

```
'entity' name=ID ;
```

- A look into Terminals shows what valid IDs are:

terminal ID:

```
'^'?('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Any sequence of lower- and upper-case letters, underscores, and digits

- but no leading digits
- leading '^' possible

Example: Defining a Language for Entities (tool-specific: generate code)

The screenshot shows an IDE's project explorer on the left and a context menu on the right. The project structure is as follows:

- de.luh.se.mbse.entities
 - src
 - de.luh.se.mbse.entities
 - Entities.xtext
 - EntitiesRuntimeModule.xtend
 - EntitiesStandaloneSetup.xtend
 - GenerateEntities.mwe2** (selected)
 - de.luh.se.mbse.entities.gen
 - de.luh.se.mbse.entities.sco
 - de.luh.se.mbse.entities.val
 - src-gen
 - xtend-gen
 - JRE System Library [JavaSE-1.8]
 - Plug-in Dependencies
 - .settings
 - META-INF
 - model
 - generated
 - Entities.ecore
 - Entities.genmodel
 - .antlr-generator-3.2.0-patch.jar
 - .classpath
 - .project
 - build.properties
 - plugin.xml
 - plugin.xml_gen
- de.luh.se.mbse.entities.ide
- de.luh.se.mbse.entities.tests
- de.luh.se.mbse.entities.ui
- de.luh.se.mbse.entities.ui.tests

The context menu for 'GenerateEntities.mwe2' includes the following options:

- New
- Open (F3)
- Open With
- Show In (Alt+Shift+W)
- Copy (Ctrl+C)
- Copy Qualified Name
- Paste (Ctrl+V)
- Delete
- Remove from Context (Ctrl+Alt+Shift+Down)
- Mark as Landmark (Ctrl+Alt+Shift+Up)
- Build Path
- Refactor (Alt+Shift+T)
- Import...
- Export...
- Refresh (F5)
- Assign Working Sets...
- Run As** (highlighted)
- Debug As

The 'Run As' option is expanded, showing a sub-menu with the following options:

- 1 MWE2 Workflow** (highlighted)
- Run Configurations...

Example: Defining a Language for Entities (tool-specific: generate code)

- ▼ de.luh.se.mbse.entities
 - ▼ src
 - ▼ de.luh.se.mbse.entities
 - Entities.xtext
 - EntitiesRuntimeModule.xtend
 - EntitiesStandaloneSetup.xtend
 - GenerateEntities.mwe2
 - de.luh.se.mbse.entities.gen
 - de.luh.se.mbse.entities.sco
 - de.luh.se.mbse.entities.val
 - src-gen
 - xtend-gen
 - JRE System Library [JavaSE-1.8]
 - Plug-in Dependencies
 - .settings
 - META-INF
 - ▼ model
 - ▼ generated
 - Entities.ecore
 - Entities.genmodel
 - .antlr-generator-3.2.0-patch.jar
 - .classpath
 - .project
 - build.properties
 - plugin.xml
 - plugin.xml_gen
 - de.luh.se.mbse.entities.ide
 - de.luh.se.mbse.entities.tests
 - de.luh.se.mbse.entities.ui
 - de.luh.se.mbse.entities.ui.tests

xtext grammar file

- New >
- Open F3
- Open With >
- Show In Alt+Shift+W >
- Copy Ctrl+C
- Copy Qualified Name
- Paste Ctrl+V
- Delete Delete
- Remove from Context Ctrl+Alt+Shift+Down
- Mark as Landmark Ctrl+Alt+Shift+Up
- Build Path >
- Refactor Alt+Shift+T >
- Import...
- Export...
- Refresh F5
- Assign Working Sets...
- Run As >
- Debug As >

- 1 MWE2 Workflow
- Run Configurations...

Example: Defining a Language for Entities (tool-specific: generate code)

The screenshot shows an IDE project explorer on the left and a context menu on the right. The project explorer displays the following structure:

- de.luh.se.mbse.entities
 - src
 - de.luh.se.mbse.entities
 - Entities.xtext
 - EntitiesRuntimeModule.xtend
 - EntitiesStandaloneSetup.xtend
 - GenerateEntities.mwe2
 - de.luh.se.mbse.entities.gen
 - de.luh.se.mbse.entities.sco
 - de.luh.se.mbse.entities.val
 - src-gen
 - xtend-gen
 - JRE System Library [JavaSE-1.8]
 - Plug-in Dependencies
 - .settings
 - META-INF
 - model
 - generated
 - Entities.ecore
 - Entities.genmodel
 - .antlr-generator-3.2.0-patch.jar
 - .classpath
 - .project
 - build.properties
 - plugin.xml
 - plugin.xml_gen
- de.luh.se.mbse.entities.ide
- de.luh.se.mbse.entities.tests
- de.luh.se.mbse.entities.ui
- de.luh.se.mbse.entities.ui.tests

The context menu is open over the 'GenerateEntities.mwe2' file. The menu items are:

- New
- Open
- Open With
- Show In
- Alt+Shift+W
- Copy
- Ctrl+C
- Copy Qualified Name
- Paste
- Ctrl+V
- Delete
- Delete
- Remove from Context
- Ctrl+Alt+Shift+Down
- Mark as Landmark
- Ctrl+Alt+Shift+Up
- Build Path
- Refactor
- Alt+Shift+T
- Import...
- Export...
- Refresh
- F5
- Assign Working Sets...
- Run As
- 1 MWE2 Workflow
- Debug As
- Run Configurations...

Two callout boxes provide additional information:

- A yellow callout box points to 'Entities.xtext' with the text: **xtext grammar file**
- A yellow callout box points to 'GenerateEntities.mwe2' with the text: **workflow file: execute to generate a number of plug-ins, models, and Java-classes**

Example: Defining a Language for Entities

- How does the Ecore model look that is generated by Xtext?

```
grammar de.luh.se.mbse.entities.Entities
with org.eclipse.xtext.common.Terminals

generate entities
"http://www.luh.de/se/mbse/entities/Entities"
```

```
Model:
    entities+=Entity*;
```

```
Entity:
    'entity' name=ID;
```

Example: Defining a Language for Entities

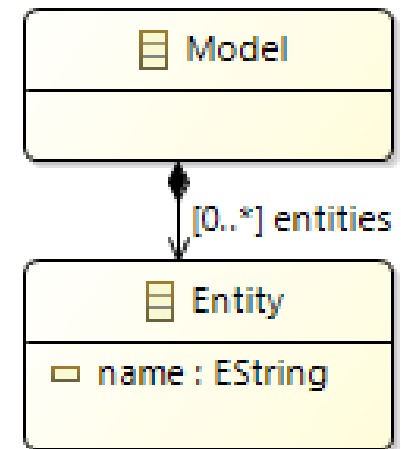
- How does the Ecore model look that is generated by Xtext?

```
grammar de.luh.se.mbse.entities.Entities
with org.eclipse.xtext.common.Terminals

generate entities
"http://www.luh.de/se/mbse/entities/Entities"
```

```
Model:
    entities+=Entity*;

Entity:
    'entity' name=ID;
```



Example: Defining a Language for Entities

- Let's extend the language to allow attributes of entities and entities that extend other entities:

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities ["http://www.luh.de/se/mbse/entities/Entities"](http://www.luh.de/se/mbse/entities/Entities)

Model:

entities+=Entity*;

Entity:

'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';

Attribute:

type=[Entity] (array?='[]')? name=ID ';';

Example: Defining a Language for Entities

- Let's extend the language to allow attributes of entities and entities that extend other entities:

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "<http://www.luh.de/se/mbse/entities/Entities>"

Model:

entities+=Entity*;

Entity:

'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';

Attribute:

type=[Entity] (array?='[]')? name=ID ';;';

Example: Defining a Language for Entities

- Let's extend the language to allow attributes of entities and entities that extend other entities:

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "http://www.luh.de/se/mbse/entities/Entities"

Model:

entities+=Entity*;

Entity:

'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';

An Entity statement starts with the keyword "entity", followed by a name

Attribute:

type=[Entity] (array?='[]')? name=ID ';;';

Example: Defining a Language for Entities

- Let's extend the language to allow attributes of entities and entities that extend other entities:

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "http://www.luh.de/se/mbse/entities/Entities"

Model:

entities+=Entity*;

Entity:

'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';

An Entity statement starts with the keyword "entity", followed by a name

It can be followed by the keyword "extends" and a reference called supertype to an existing element of type Entity

- optionality is denoted by ?
- referring to a type in square brackets [] expresses a cross reference to an existing element of the given type

Attribute:

type=[Entity] (array?=

Example: Defining a Language for Entities

- Let's extend the language to allow attributes of entities and entities that extend other entities:

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "http://www.luh.de/se/mbse/entities/Entities"

Model:

entities+=Entity*;

Entity:

'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';

An Entity statement starts with the keyword "entity", followed by a name

Then, enclosed in the keywords/symbols "{" and "}", follows a collection of attributes.

It can be followed by the keyword "extends" and a reference called supertype to an existing element of type Entity

- optionality is denoted by ?
- referring to a type in square brackets [] expresses a cross reference to an existing element of the given type

Example: Defining a Language for Entities

- Let's extend the language to allow attributes of entities and entities that extend other entities:

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities ["http://www.luh.de/se/mbse/entities/Entities"](http://www.luh.de/se/mbse/entities/Entities)

Model:

entities+=Entity*;

Entity:

'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';

Attribute:

type=[Entity] (array?='[]')? name=ID ';' ;

Example: Defining a Language for Entities

- Let's extend the language to allow attributes of entities and entities that extend other entities:

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities ["http://www.luh.de/se/mbse/entities/Entities"](http://www.luh.de/se/mbse/entities/Entities)

Model:

entities+=Entity*;

Entity:

'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';

Attribute:

type=[Entity] (array?='[]')? name=ID ';' ;

cross reference to an
existing entity (as above)

Example: Defining a Language for Entities

- Let's extend the language to allow attributes of entities and entities that extend other entities:

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities ["http://www.luh.de/se/mbse/entities/Entities"](http://www.luh.de/se/mbse/entities/Entities)

Model:

entities+=Entity*;

Entity:

'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';

Attribute:

type=[Entity] (array?='[]')? name=ID ';' ;

cross reference to an
existing entity (as above)

attribute array, ? defines it as EBoolean. If
optional '[]' is provided, set array to true.

Example: Defining a Language for Entities

- Let's extend the language to allow attributes of entities and entities that extend other entities:

grammar de.luh.se.mbse.entities.Entities

with org.eclipse.xtext.common.Terminals

generate entities "http://www.luh.de/se/mbse/entities/Entities"

Model:

entities+=Entity*;

Entity:

'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';

Attribute:

type=[Entity] (array?='[]')? name=ID ';' ;

Then a name, and
finally a semicolon ';' ;

cross reference to an
existing entity (as above)

attribute array, ? defines it as EBoolean. If
optional '[]' is provided, set array to true.

Example: Defining a Language for Entities

- How does the Ecore model look that is generated by Xtext?

```
grammar de.luh.se.mbse.entities.Entities
with org.eclipse.xtext.common.Terminals
generate entities
"http://www.luh.de/se/mbse/entities/Entities"
```

```
Model:
    entities+=Entity*;
```

```
Entity:
    'entity' name=ID
    ('extends' supertype=[Entity])? '{'
    attributes+=Attribute*
    '}';
```

```
Attribute:
    type=[Entity] (array?='[]')? name=ID ';' ;
```

Example: Defining a Language for Entities

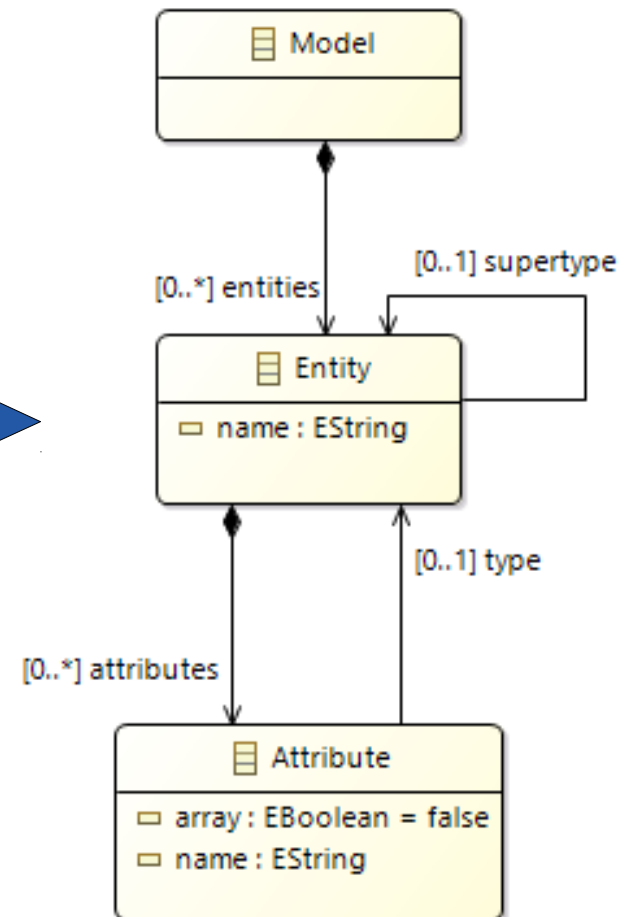
- How does the Ecore model look that is generated by Xtext?

```
grammar de.luh.se.mbse.entities.Entities
with org.eclipse.xtext.common.Terminals
generate entities
"http://www.luh.de/se/mbse/entities/Entities"
```

```
Model:
    entities+=Entity*;

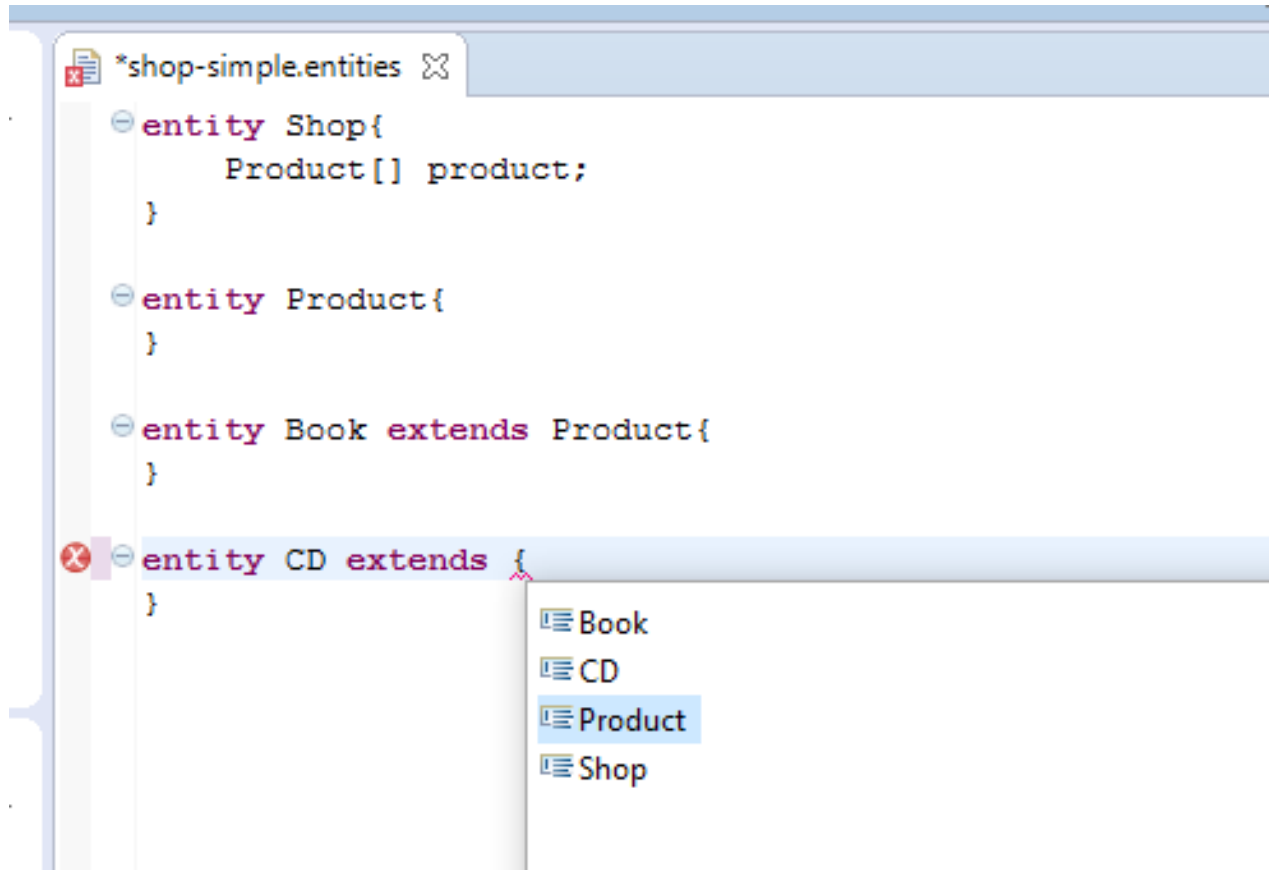
Entity:
    'entity' name=ID
    ('extends' supertype=[Entity])? '{'
    attributes+=Attribute*
    '}';

Attribute:
    type=[Entity] (array?='[]')? name=ID ';' ;
```



Example: Defining a Language for Entities

- Let's try the editor:



```
*shop-simple.entities

entity Shop{
    Product[] product;
}

entity Product{
}

entity Book extends Product{
}

entity CD extends {
}
```

Book
CD
Product
Shop

Example: Defining a Language for Entities

- Let's try the editor:

syntax highlighting

```
*shop-simple.entiti
entity Shop{
    Product[] product;
}

entity Product{
}

entity Book extends Product{
}

entity CD extends {
}
```

- Book
- CD
- Product
- Shop

Example: Defining a Language for Entities

- Let's try the editor:

The screenshot shows a code editor window titled `*shop-simple.entiti`. The code defines four entities: `Shop`, `Product`, `Book`, and `CD`. The `Shop` entity has a `Product[] product;` attribute. The `Product` entity is a base class. The `Book` entity extends `Product`. The `CD` entity extends `Product` and is currently being edited, with a dropdown menu showing the inheritance hierarchy: `Book`, `CD`, `Product` (selected), and `Shop`.

Two callouts highlight features of the editor:

- syntax highlighting**: Points to the `entity` keyword in the `Shop` definition.
- nice indentation ("pretty-printing") supported during editing, also auto-format function provided and customizable**: Points to the indentation of the `Product[] product;` line in the `Shop` definition.

```
entity Shop{
    Product[] product;
}

entity Product{
}

entity Book extends Product{
}

entity CD extends {
}
```

Book
CD
Product
Shop

Example: Defining a Language for Entities

- Let's try the editor:

The screenshot shows a code editor window titled `*shop-simple.entiti`. The code defines several entities:

```

entity Shop{
    Product[] product;
}

entity Product{
}

entity Book extends Product{
}

entity CD extends {

```

Annotations highlight editor features:

- syntax highlighting**: Points to the `entity` keyword in the first line.
- nice indentation ("pretty-printing") supported during editing, also auto-format function provided and customizable**: Points to the indentation of the `Product[] product;` line.
- error markers**: Points to a red 'X' icon next to the `entity CD extends {` line, indicating a syntax error.

A dropdown menu is visible on the right, showing a list of entities: `Book`, `CD`, `Product` (highlighted), and `Shop`.

Example: Defining a Language for Entities

- Let's try the editor:

The screenshot shows an IDE window titled `*shop-simple.entiti` containing the following code:

```
entity Shop{
    Product[] product;
}

entity Product{
}

entity Book extends Product{
}

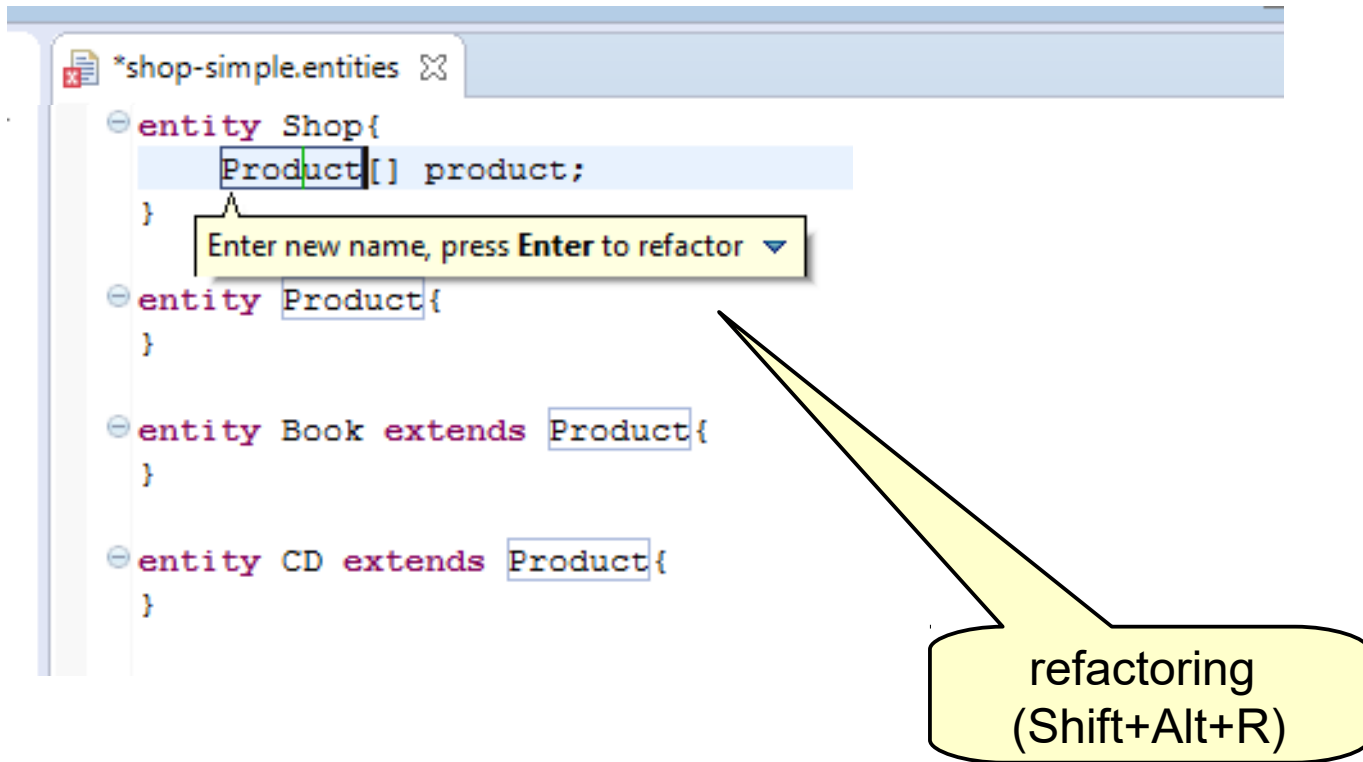
entity CD extends {
```

Callouts highlight the following features:

- syntax highlighting**: Points to the `entity` keyword in the first line.
- nice indentation ("pretty-printing") supported during editing, also auto-format function provided and customizable**: Points to the indentation of the `Product[] product;` line.
- error markers**: Points to a red 'X' icon next to the `entity CD extends {` line.
- auto-complete functionality (on Ctrl+Space) here: select an existing entity**: Points to a dropdown menu showing the list of existing entities: `Book`, `CD`, `Product`, and `Shop`.

Example: Defining a Language for Entities

- Let's try the editor:



Example: Defining a Language for Entities (extending the language)

- What about String and Integer attributes?

```
entity Shop{
    String name;
    Product[] product;
}

entity Product{
    String name;
    Int productNumber;
}

entity Book extends Product{
    String ISBN;
    String publisher;
}

entity CD extends Product{
    String label;
}
```

Example: Defining a Language for Entities (extending the language)

...

Model:

```
entities+=Entity*;
```

Entity:

```
'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';
```

Attribute:

```
type=AttributeType name=ID ';' ;
```

AttributeType:

```
elementType=ElementType (array?='[' (length=INT)? ']')?;
```

ElementType:

```
EntityType | BasicType;
```

EntityType:

```
entity=[Entity];
```

BasicType:

```
typename=('String' | 'Boolean' | 'Int');
```

Example: Defining a Language for Entities (extending the language)

...

Model:

```
entities+=Entity*;
```

Entity:

```
'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';
```

An Attribute has a type

Attribute:

```
type=AttributeType name=ID ';' ;
```

AttributeType:

```
elementType=ElementType (array?='[' (length=INT)? ']')?;
```

ElementType:

```
EntityType | BasicType;
```

EntityType:

```
entity=[Entity];
```

BasicType:

```
typename=('String' | 'Boolean' | 'Int');
```


Example: Defining a Language for Entities (extending the language)

...

Model:

```
entities+=Entity*;
```

Entity:

```
'entity' name=ID ('extends' supertype=[Entity])? '{'
attributes+=Attribute*
'}';
```

Attribute:

```
type=AttributeType name=ID ';';
```

AttributeType:

```
elementType=ElementType (array?='[' (length=INT)? ']')?;
```

ElementType:

```
EntityType | BasicType;
```

EntityType:

```
entity=[Entity];
```

BasicType:

```
typename=('String' | 'Boolean' | 'Int');
```

An Attribute has a type

The AttributeType specifies and element type and whether it is an array of a specific length

Example: Defining a Language for Entities (extending the language)

...

Model:

```
entities+=Entity*;
```

Entity:

```
'entity' name=ID ('extends' supertype=[Entity])? '{'  
attributes+=Attribute*  
'}';
```

Attribute:

```
type=AttributeType name=ID ';;';
```

AttributeType:

```
elementType=ElementType (array?='[' (length=INT)? '']')?;
```

ElementType:

```
EntityType | BasicType;
```

EntityType:

```
entity=[Entity];
```

BasicType:

```
typename=('String' | 'Boolean' | 'Int');
```

An Attribute has a type

The AttributeType specifies and element type and whether it is an array of a specific length

The ElementType can be an entity type or (operator |) a basic (data) type

Example: Defining a Language for Entities (extending the language)

...

Model:

```
entities+=Entity*;
```

Entity:

```
'entity' name=ID ('extends' supertype=[Entity])? '{'  
attributes+=Attribute*  
'}';
```

Attribute:

```
type=AttributeType name=ID ';;'
```

AttributeType:

```
elementType=ElementType (array?='[' (length=INT)? '']')?;
```

ElementType:

```
EntityType | BasicType;
```

EntityType:

```
entity=[Entity];
```

BasicType:

```
typename=('String' | 'Boolean' | 'Int');
```

An Attribute has a type

The AttributeType specifies and element type and whether it is an array of a specific length

The ElementType can be an entity type or (operator |) a basic (data) type

an entity type consists of a reference to an existing Entity

Example: Defining a Language for Entities (extending the language)

...

Model:

```
entities+=Entity*;
```

Entity:

```
'entity' name=ID ('extends' supertype=[Entity])? '{'  
attributes+=Attribute*  
'}';
```

Attribute:

```
type=AttributeType name=ID ';;';
```

AttributeType:

```
elementType=ElementType (array?='[' (length=INT)? ']')?;
```

ElementType:

```
EntityType | BasicType;
```

EntityType:

```
entity=[Entity];
```

BasicType:

```
typename=('String' | 'Boolean' | 'Int');
```

An Attribute has a type

The AttributeType specifies and element type and whether it is an array of a specific length

The ElementType can be an entity type or (operator |) a basic (data) type

an entity type consists of a reference to an existing Entity

a basic type has a string attribute typename, which can be either "String", "Boolean" or "Int"

Example: Defining a Language for Entities

- How does the Ecore model look?

Model:

```
entities+=Entity*;
```

Entity:

```
'entity' name=ID
('extends' supertype=[Entity])? '{'
attributes+=Attribute* '}';
```

Attribute:

```
type=AttributeType name=ID ';';
```

AttributeType:

```
elementType=ElementType
(array?='[' (length=INT)? '']?);
```

ElementType:

```
EntityType | BasicType;
```

EntityType:

```
entity=[Entity];
```

BasicType:

```
typename=('String' | 'Boolean' | 'Int');
```

Example: Defining a Language for Entities

- How does the Ecore model look?

Model:

```
entities+=Entity*;
```

Entity:

```
'entity' name=ID
('extends' supertype=[Entity])? '{'
attributes+=Attribute* '}';
```

Attribute:

```
type=AttributeType name=ID ';';
```

AttributeType:

```
elementType=ElementType
(array?='[' (length=INT)? ''])?;
```

ElementType:

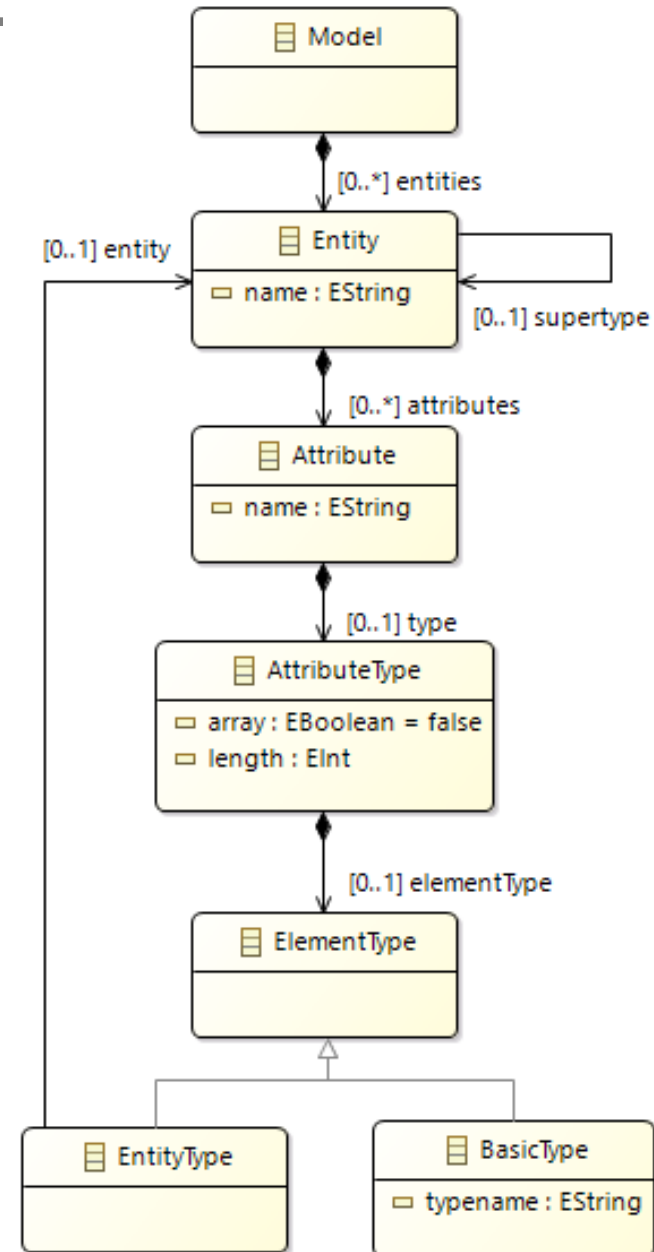
```
EntityType | BasicType;
```

EntityType:

```
entity=[Entity];
```

BasicType:

```
typename=('String' | 'Boolean' | 'Int');
```



Example: Defining a Language for Entities

- We can use Enums, too.

...

Attribute:

```
type=AttributeType name=ID ';' ;
```

AttributeType:

```
elementType=ElementType  
(array?='[' (length=INT)? '']?);
```

ElementType:

```
EntityType | BasicType;
```

EntityType:

```
entity=[Entity];
```

BasicType:

```
kind=BasicTypeEnum;
```

enum BasicTypeEnum:

```
String | Boolean | Int;
```

Example: Defining a Language for Entities

- We can use Enums, too.

...

Attribute:
type=AttributeType name=ID ';' ;

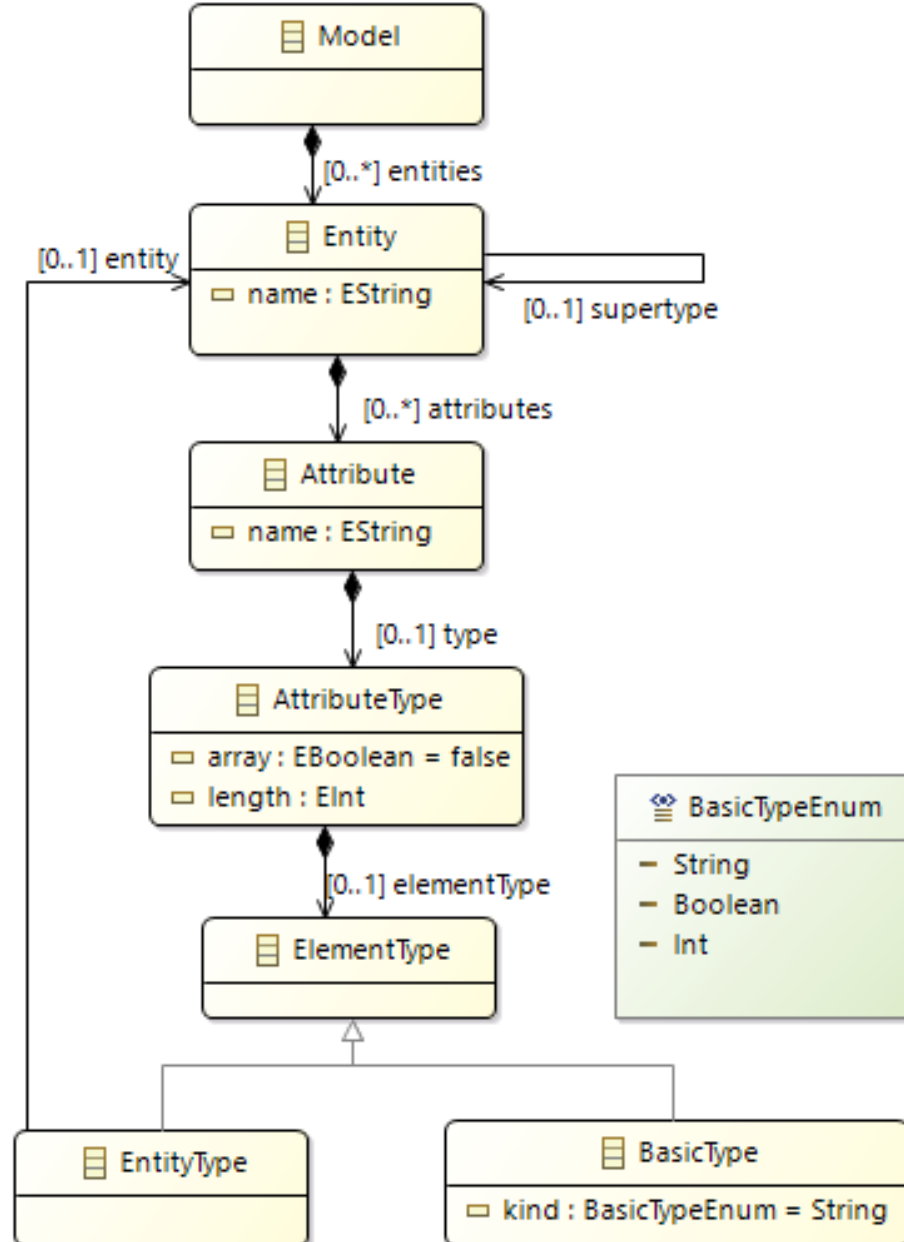
AttributeType:
elementType=ElementType
(array?='[' (length=INT)? '']')?;

ElementType:
EntityType | BasicType;

EntityType:
entity=[Entity];

BasicType:
kind=BasicTypeEnum;

enum BasicTypeEnum:
String | Boolean | Int;



Referencing an Existing Ecore Model

- An Xtext grammar can also refer to an existing Ecore model:

```
// automatically generated by Xtext
grammar de.luh.se.mbse.petrinet.pn1.PNL
with org.eclipse.xtext.common.Terminals

//import "http://de.luh.se.mbse.petrinet/petrinet"
import "platform:/resource/de.luh.se.mbse.petrinet/model/petrinet.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

Petrinet:

```
'Petri net' name=ID '{'
    element+=NetElement*
'}';
```

NetElement:

```
Node | Arc;
```

Node:

```
Place | Transition;
```

Place:

```
'Place' name=ID;
```

...

Referencing an Existing Ecore Model

- An Xtext grammar can also refer to an existing Ecore model:

```
// automatically generated by Xtext
grammar de.luh.se.mbse.petrinet.pn1.PNL
with org.eclipse.xtext.common.Terminals

//import "http://de.luh.se.mbse.petrinet/petrinet"
import platform:/resource/de.luh.se.mbse.petrinet/model/petrinet.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

Petrinet:

```
'Petri net' name=ID '{'
    element+=NetElement*
'}';
```

NetElement:

```
Node | Arc;
```

Node:

```
Place | Transition;
```

Place:

```
'Place' name=ID;
```

...

Referencing an Existing Ecore Model

- An Xtext grammar can also refer to an existing Ecore model:

```
// automatically generated by Xtext
grammar de.luh.se.mbse.petrinet.pn1.PNL
with org.eclipse.xtext.common.Terminals

//import "http://de.luh.se.mbse.petrinet/petrinet"
import platform:/resource/de.luh.se.mbse.petrinet/model/petrinet.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

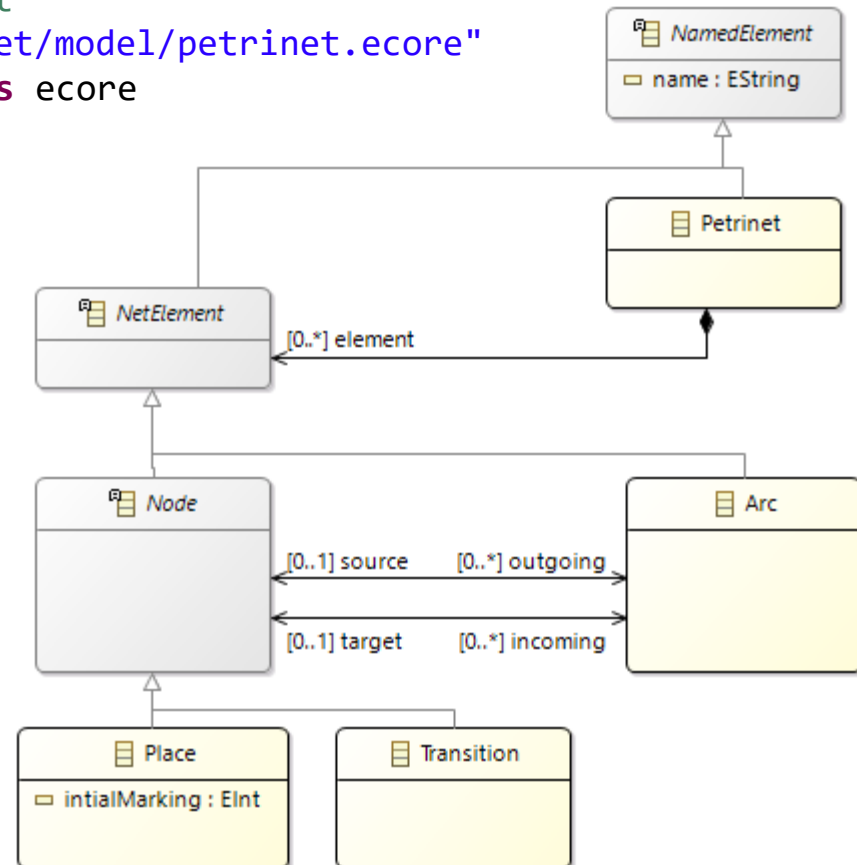
```
Petrinet:
    'Petri net' name=ID '{'
        element+=NetElement*
    '}';
```

```
NetElement:
    Node | Arc;
```

```
Node:
    Place | Transition;
```

```
Place:
    'Place' name=ID;
```

...



Some common Xtext issues

- “...may consume non empty input without object instantiation”

Petrinet:

```
'Petri net' '{' element+=NetElement* '}';
```

The entry rule 'Petrinet' may consume non empty input without object instantiation. Add an action to ensure object creation, e.g. '{Petrinet}'.

Some common Xtext issues

- “...may consume non empty input without object instantiation”

Petrinet:

```
'Petri net' '{' element+=NetElement* '}';
```

The entry rule 'Petrinet' may consume non empty input without object instantiation. Add an action to ensure object creation, e.g. '{Petrinet}'.

If the element list is empty, the parser will parse the keywords without creating a Petrinet object. (Object creation happens when the first assignment is executed, e.g. name=ID.)

Some common Xtext issues

- “...may consume non empty input without object instantiation”

Petrinet:

```
'Petri net' '{' element+=NetElement* '}';
```

The entry rule 'Petrinet' may consume non empty input without object instantiation. Add an action to ensure object creation, e.g. '{Petrinet}'.

If the element list is empty, the parser will parse the keywords without creating a Petrinet object. (Object creation happens when the first assignment is executed, e.g. name=ID.)

Petrinet: // solution 1: Add name assignment

```
'Petri net' name=ID '{' element+=NetElement* '}';
```

Some common Xtext issues

- “...may consume non empty input without object instantiation”

Petrinet:

```
'Petri net' '{' element+=NetElement* '}';
```

The entry rule 'Petrinet' may consume non empty input without object instantiation. Add an action to ensure object creation, e.g. '{Petrinet}'.

If the element list is empty, the parser will parse the keywords without creating a Petrinet object. (Object creation happens when the first assignment is executed, e.g. name=ID.)

Petrinet: // solution 1: Add name assignment

```
'Petri net' name=ID '{' element+=NetElement* '}';
```

Some common Xtext issues

- “...may consume non empty input without object instantiation”

Petrinet:

```
'Petri net' '{' element+=NetElement* '}';
```

The entry rule 'Petrinet' may consume non empty input without object instantiation. Add an action to ensure object creation, e.g. '{Petrinet}'.

If the element list is empty, the parser will parse the keywords without creating a Petrinet object. (Object creation happens when the first assignment is executed, e.g. name=ID.)

Petrinet: // solution 1: Add name assignment

```
'Petri net' name=ID '{' element+=NetElement* '}';
```

Petrinet: // solution 2: require a non-empty list

```
'Petri net' '{' element+=NetElement+ '}';
```


Some common Xtext issues

- “...may consume non empty input without object instantiation”

Petrinet:

```
'Petri net' '{' element+=NetElement* '}';
```

The entry rule 'Petrinet' may consume non empty input without object instantiation. Add an action to ensure object creation, e.g. '{Petrinet}'.

If the element list is empty, the parser will parse the keywords without creating a Petrinet object. (Object creation happens when the first assignment is executed, e.g. name=ID.)

Petrinet: // solution 1: Add name assignment

```
'Petri net' name=ID '{' element+=NetElement* '}';
```

Petrinet: // solution 2: require a non-empty list

```
'Petri net' '{' element+=NetElement+ '}';
```

Some common Xtext issues

- “...may consume non empty input without object instantiation”

Petrinet:

```
'Petri net' '{' element+=NetElement* '}';
```

The entry rule 'Petrinet' may consume non empty input without object instantiation. Add an action to ensure object creation, e.g. '{Petrinet}'.

If the element list is empty, the parser will parse the keywords without creating a Petrinet object. (Object creation happens when the first assignment is executed, e.g. name=ID.)

Petrinet: // solution 1: Add name assignment

```
'Petri net' name=ID '{' element+=NetElement* '}';
```

Petrinet: // solution 2: require a non-empty list

```
'Petri net' '{' element+=NetElement+ '}';
```

Petrinet: // solution 3: Specify object creation explicitly

```
{Petrinet}
```

```
'Petri net' '{' element+=NetElement* '}';
```

Some common Xtext issues

- “...may consume non empty input without object instantiation”

Petrinet:

```
'Petri net' '{' element+=NetElement* '}';
```

The entry rule 'Petrinet' may consume non empty input without object instantiation. Add an action to ensure object creation, e.g. '{Petrinet}'.

If the element list is empty, the parser will parse the keywords without creating a Petrinet object. (Object creation happens when the first assignment is executed, e.g. name=ID.)

Petrinet: // solution 1: Add name assignment

```
'Petri net' name=ID '{' element+=NetElement* '}';
```

Petrinet: // solution 2: require a non-empty list

```
'Petri net' '{' element+=NetElement+ '}';
```

Petrinet: // solution 3: Specify object creation explicitly

```
{Petrinet}
```

```
'Petri net' '{' element+=NetElement* '}';
```

Some common Xtext issues

Arc:

```
'Arc' source=[Place] '->' target=[Transition]  
| 'Arc' source=[Transition] '->' target=[Place] ;
```

Idea: use Xtext grammar to constrain
valid Petri net edges

Some common Xtext issues

Arc:

```
'Arc' source=[Place] '->' target=[Transition]  
| 'Arc' source=[Transition] '->' target=[Place] ;
```

Idea: use Xtext grammar to constrain
valid Petri net edges

- “warning(200): ... Decision can match input such as
"Arc' RULE_ID '->' RULE_ID" using multiple alternatives...”

Some common Xtext issues

Arc:

```
'Arc' source=[Place] '->' target=[Transition]
| 'Arc' source=[Transition] '->' target=[Place] ;
```

Idea: use Xtext grammar to constrain valid Petri net edges

- “warning(200): ... Decision can match input such as "Arc' RULE_ID '->' RULE_ID" using multiple alternatives...”
 - when the parser reads the tokens representing the source and target nodes, it just reads the ID strings, without knowing yet whether they refer to a place or a transition.

Some common Xtext issues

Arc:

```
'Arc' source=[Place] '->' target=[Transition]
| 'Arc' source=[Transition] '->' target=[Place] ;
```

Idea: use Xtext grammar to constrain valid Petri net edges

- “warning(200): ... Decision can match input such as "Arc' RULE_ID '->' RULE_ID" using multiple alternatives...”
 - when the parser reads the tokens representing the source and target nodes, it just reads the ID strings, without knowing yet whether they refer to a place or a transition.
 - therefore, the grammar is **ambiguous**

Some common Xtext issues

Arc:

```
'Arc' source=[Place] '->' target=[Transition]
| 'Arc' source=[Transition] '->' target=[Place] ;
```

Idea: use Xtext grammar to constrain valid Petri net edges

- “warning(200): ... Decision can match input such as "Arc' RULE_ID '->' RULE_ID" using multiple alternatives...”
 - when the parser reads the tokens representing the source and target nodes, it just reads the ID strings, without knowing yet whether they refer to a place or a transition.
 - therefore, the grammar is **ambiguous**

Arc: // solution 1: add keywords

Some common Xtext issues

Arc:

```
'Arc' source=[Place] '->' target=[Transition]
| 'Arc' source=[Transition] '->' target=[Place] ;
```

Idea: use Xtext grammar to constrain valid Petri net edges

- “warning(200): ... Decision can match input such as "Arc' RULE_ID '->' RULE_ID" using multiple alternatives...”
 - when the parser reads the tokens representing the source and target nodes, it just reads the ID strings, without knowing yet whether they refer to a place or a transition.
 - therefore, the grammar is **ambiguous**

Arc: // solution 1: add keywords

```
'Arc' 'pl' source=[Place] '->' 'tr' target=[Transition]
```

Some common Xtext issues

Arc:

```
'Arc' source=[Place] '->' target=[Transition]
| 'Arc' source=[Transition] '->' target=[Place] ;
```

Idea: use Xtext grammar to constrain valid Petri net edges

- “warning(200): ... Decision can match input such as "Arc' RULE_ID '->' RULE_ID" using multiple alternatives...”
 - when the parser reads the tokens representing the source and target nodes, it just reads the ID strings, without knowing yet whether they refer to a place or a transition.
 - therefore, the grammar is **ambiguous**

Arc: // solution 1: add keywords

```
'Arc' 'pl' source=[Place] '->' 'tr' target=[Transition]
| 'Arc' 'tr' source=[Transition] '->' 'pl' target=[Place] ;
```

Integration with OCL constraints on the Ecore metamodel

Arc:

```
'Arc' source=[Place] '->' target=[Transition]
| 'Arc' source=[Transition] '->' target=[Place] ;
```

Arc: // solution 2: use OCL constraint in the Ecore metamodel

```
'Arc' source=[Node] '->' target=[Node];
```

```
Petri net DayAndNight {
  Place day
  Transition sunset
  Place night
  Transition sunrise
  Arc day -> night
}
```

The 'NoArcsBetweenNodesOfTheSameKind' constraint is violated on
'de.luh.se.mbse.petrinet.impl.ArcImpl@60cefcf8{platform:/resource/de.luh.se.mbse.myfirstpetrinetproject/
myfirstpetrinet2.pnl#//@element.4}'

Custom Validation

Arc:

```
'Arc' source=[Place] '->' target=[Transition]
| 'Arc' source=[Transition] '->' target=[Place] ;
```

Arc: // solution 3: implement a custom Xtext validation function

```
'Arc' source=[Node] '->' target=[Node];
```

```
Petri net DayAndNight {
    Place day
    Transition sunset
    Place night
    Transition sunrise
    Arc day -> night
}
```



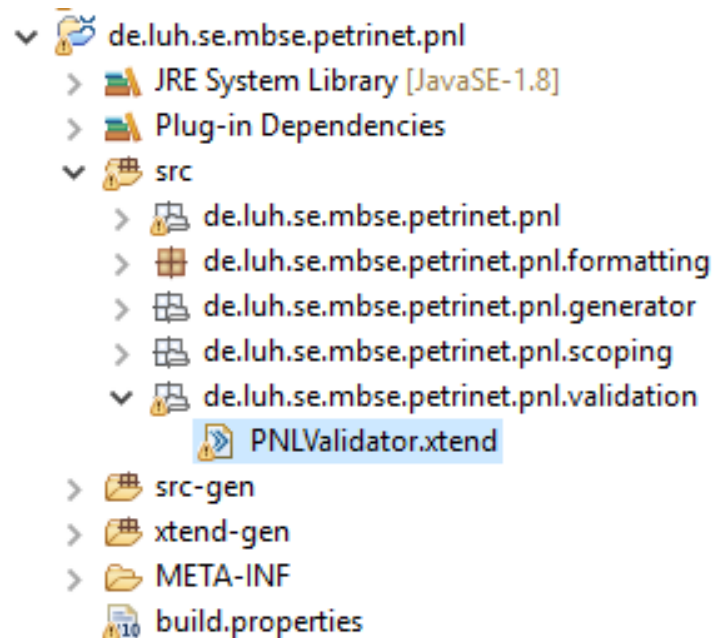
An arc can only connect a place to transition or a transition to a place

Custom Validation

- You can add **custom validation functions** by implementing specific check methods in the **validator class**

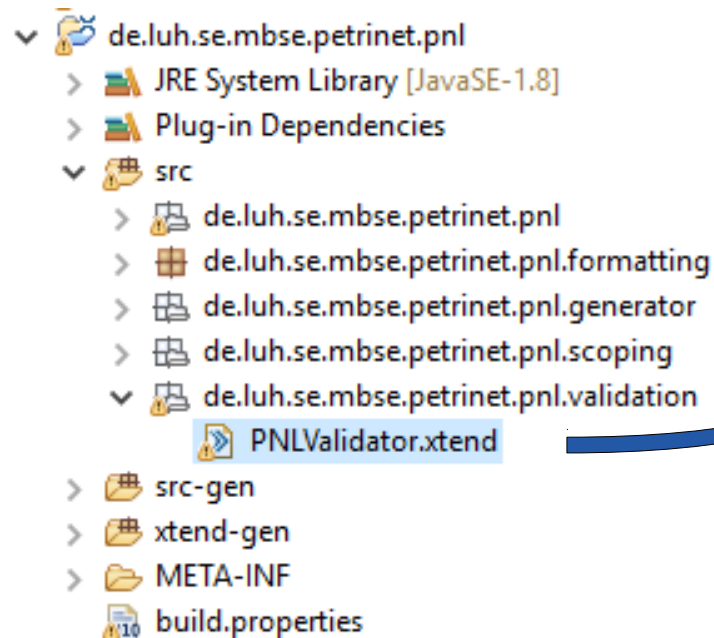
Custom Validation

- You can add **custom validation functions** by implementing specific check methods in the **validator class**



Custom Validation

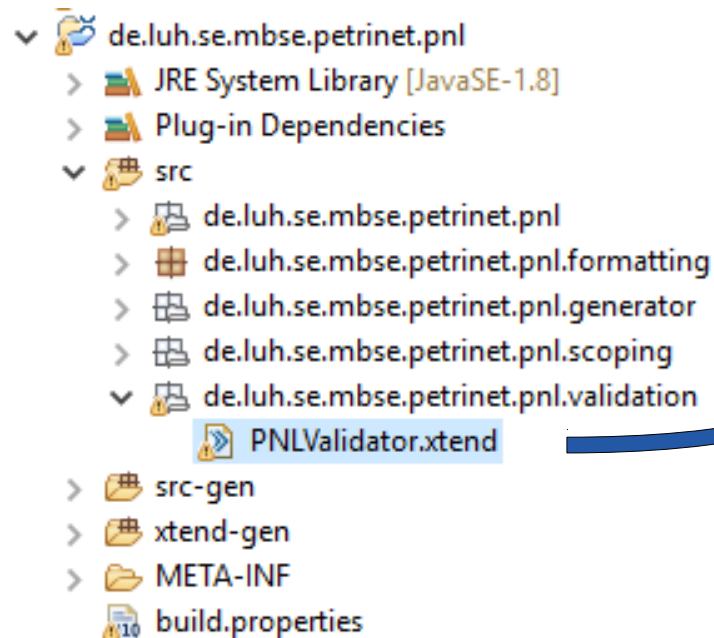
- You can add **custom validation functions** by implementing specific check methods in the **validator class**



```
@Check
def checkValidArc(Arc arc){
    if(
        !( arc.source instanceof Place
          && arc.target instanceof Transition
          || arc.source instanceof Transition
          && arc.target instanceof Place)
    ){
        error('An arc can only connect ...',
              PetrinetPackage.Literals.ARC__SOURCE
        )
    }
}
```

Custom Validation

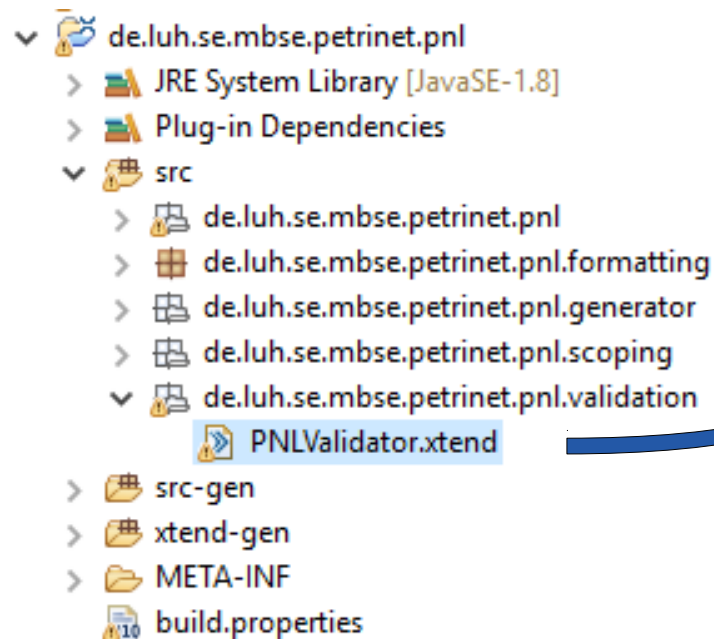
- You can add **custom validation functions** by implementing specific check methods in the **validator class**
 - implementation in Xtend, a Java-like programming language



```
@Check
def checkValidArc(Arc arc){
    if(
        !( arc.source instanceof Place
          && arc.target instanceof Transition
          || arc.source instanceof Transition
          && arc.target instanceof Place)
    ){
        error('An arc can only connect ...',
              PetrinetPackage.Literals.ARC__SOURCE
        )
    }
}
```


Custom Validation

- You can add **custom validation functions** by implementing specific check methods in the **validator class**
 - implementation in Xtend, a Java-like programming language
 - Xtend: less verbose than Java, easy to learn if you know Java



```
@Check
def checkValidArc(Arc arc){
    if(
        !( arc.source instanceof Place
          && arc.target instanceof Transition
          || arc.source instanceof Transition
          && arc.target instanceof Place)
    ){
        error('An arc can only connect ...',
              PetrinetPackage.Literals.ARC__SOURCE
        )
    }
}
```

Summary Xtext

- define a textual language by using an EBNF-style grammar

Summary Xtext

- define a textual language by using an EBNF-style grammar
- generate Ecore model or import existing one

Summary Xtext

- define a textual language by using an EBNF-style grammar
- generate Ecore model or import existing one
- generates rich editor functionality

Summary Xtext

- define a textual language by using an EBNF-style grammar
- generate Ecore model or import existing one
- generates rich editor functionality
- allows extensions:

Summary Xtext

- define a textual language by using an EBNF-style grammar
- generate Ecore model or import existing one
- generates rich editor functionality
- allows extensions:
 - validation

Summary Xtext

- define a textual language by using an EBNF-style grammar
- generate Ecore model or import existing one
- generates rich editor functionality
- allows extensions:
 - validation
 - quick fixes

Summary Xtext

- define a textual language by using an EBNF-style grammar
- generate Ecore model or import existing one
- generates rich editor functionality
- allows extensions:
 - validation
 - quick fixes
 - scoping

Summary Xtext

- define a textual language by using an EBNF-style grammar
- generate Ecore model or import existing one
- generates rich editor functionality
- allows extensions:
 - validation
 - quick fixes
 - scoping
 - formatting

Summary Xtext

- define a textual language by using an EBNF-style grammar
- generate Ecore model or import existing one
- generates rich editor functionality
- allows extensions:
 - validation
 - quick fixes
 - scoping
 - formatting
- supports importing other grammars: combine an existing language into your DSL

Summary Xtext

- Supports easy development of JVM-compatible languages using Xbase, including compiler-support
 - grammar needs to map DSL-concepts to JVM concepts
 - see extended Entities example: https://eclipse.org/Xtext/documentation/104_jvmdomainmodel.html

The screenshot displays the Eclipse IDE with the Xtext DSL editor and the Outline view.

Person.dmodel Editor:

```
import java.util.List

package my.model {

    entity Person {
        name: String
        firstName: String
        friends: List<Person>
        address : Address
        op getFullName() : String {
            return firstName + " " + name;
        }

        /**
         * @return a view on all {@link #friends} sorted
         * using their {@link #getFullName()}
         */
        op getFriendsSortedByFullName() : List<Person> {
            return friends.sortBy[ f | f.fullName ]
        }
    }

    entity Address
    // ...
}
```

Outline View:

- Person
 - <unnamed>
 - java.util.List
 - my.model
 - Person
 - name
 - firstName
 - friends
 - address
 - getFullName
 - getFriendsSortedByFullName
 - Address

Hover Information:

friends : List<Person> - Person

friendsSortedByFullName : List<Person> - Person

List<Person> Person.getFriendsSortedByFullName()

Returns:
a view on all `#friends` using their `#getFullName()`

Summary Xtext Strengths and Weaknesses

- Strengths

Summary Xtext

Strengths and Weaknesses

- Strengths
 - Minimal effort for building DSLs

Summary Xtext

Strengths and Weaknesses

- Strengths
 - Minimal effort for building DSLs
 - EMF/Eclipse integration

Summary Xtext

Strengths and Weaknesses

- Strengths
 - Minimal effort for building DSLs
 - EMF/Eclipse integration
 - EBNF like grammar concepts are relatively easy to learn

Summary Xtext

Strengths and Weaknesses

- Strengths
 - Minimal effort for building DSLs
 - EMF/Eclipse integration
 - EBNF like grammar concepts are relatively easy to learn
 - rich editor “for free”

Summary Xtext

Strengths and Weaknesses

- Strengths
 - Minimal effort for building DSLs
 - EMF/Eclipse integration
 - EBNF like grammar concepts are relatively easy to learn
 - rich editor “for free”
 - easily extensible

Summary Xtext

Strengths and Weaknesses

- Strengths
 - Minimal effort for building DSLs
 - EMF/Eclipse integration
 - EBNF like grammar concepts are relatively easy to learn
 - rich editor “for free”
 - easily extensible
- Weaknesses

Summary Xtext

Strengths and Weaknesses

- Strengths
 - Minimal effort for building DSLs
 - EMF/Eclipse integration
 - EBNF like grammar concepts are relatively easy to learn
 - rich editor “for free”
 - easily extensible
- Weaknesses
 - relies on ANTLR framework, which only supports LL(k) grammars: not possible to parse all kinds of languages

Summary Xtext

Strengths and Weaknesses

- Strengths
 - Minimal effort for building DSLs
 - EMF/Eclipse integration
 - EBNF like grammar concepts are relatively easy to learn
 - rich editor “for free”
 - easily extensible
- Weaknesses
 - relies on ANTLR framework, which only supports LL(k) grammars: not possible to parse all kinds of languages
 - heavy framework may be too much for your needs, if you do not need or want the rich Eclipse editor support

Summary Xtext

Strengths and Weaknesses

- Strengths
 - Minimal effort for building DSLs
 - EMF/Eclipse integration
 - EBNF like grammar concepts are relatively easy to learn
 - rich editor “for free”
 - easily extensible
- Weaknesses
 - relies on ANTLR framework, which only supports LL(k) grammars: not possible to parse all kinds of languages
 - heavy framework may be too much for your needs, if you do not need or want the rich Eclipse editor support
 - Building a language is hard or impossible if the Ecore model does not fit the structure of the grammar

Summary Concrete Syntax

Summary Concrete Syntax

- textual vs. graphical: different advantages and disadvantages

Summary Concrete Syntax

- textual vs. graphical: different advantages and disadvantages
- rich frameworks exist for building graphical and textual languages

Summary Concrete Syntax

- textual vs. graphical: different advantages and disadvantages
- rich frameworks exist for building graphical and textual languages
- Important principle: separate abstract and concrete syntax!

Some notes on the exam

Some notes on the exam

- The core Xtext language concepts discussed in this lecture are relevant for the exam

Some notes on the exam

- The core Xtext language concepts discussed in this lecture are relevant for the exam
- Possible exam questions:

Some notes on the exam

- The core Xtext language concepts discussed in this lecture are relevant for the exam
- Possible exam questions:
 - extend a given grammar

Some notes on the exam

- The core Xtext language concepts discussed in this lecture are relevant for the exam
- Possible exam questions:
 - extend a given grammar
 - given a grammar and an object model diagram, write the textual syntax representation

Some notes on the exam

- The core Xtext language concepts discussed in this lecture are relevant for the exam
- Possible exam questions:
 - extend a given grammar
 - given a grammar and an object model diagram, write the textual syntax representation
 - given a grammar and a textual syntax document, draw the corresponding object model diagram

Some notes on the exam

- The core Xtext language concepts discussed in this lecture are relevant for the exam
- Possible exam questions:
 - extend a given grammar
 - given a grammar and an object model diagram, write the textual syntax representation
 - given a grammar and a textual syntax document, draw the corresponding object model diagram
 - Infer Ecore model from a given Xtext grammar