Model-Based Software Engineering

Lecture 06 – Concrete Syntax

Prof. Dr. Joel Greenyer

May 24, 2016

Leibniz Universität Hannover

Acknowledgment

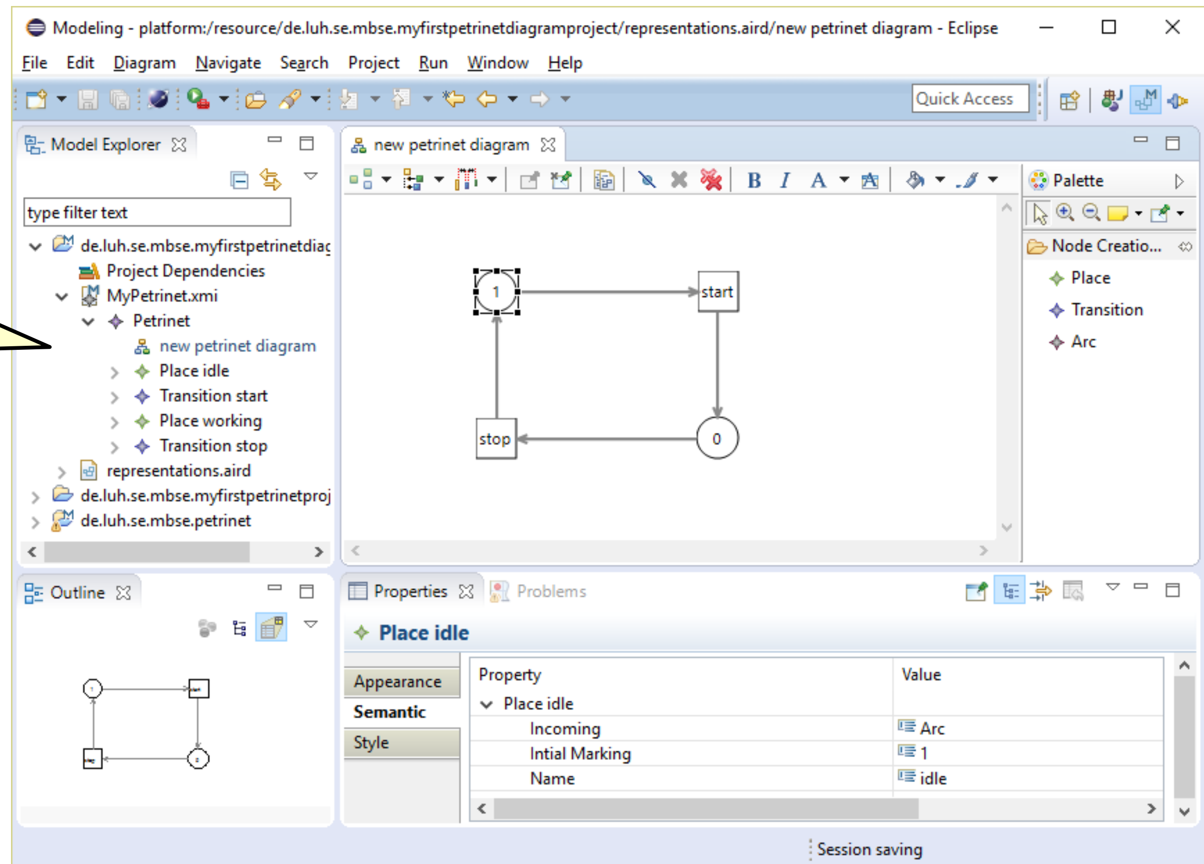
- The slides of this lecture are inspired by lecture slides from
  - *Ekkart Kindler*: Course on Advanced Topics in Software Engineering, DTU Compute, 2015.
    - http://www2.imm.dtu.dk/courses/02265/f15/schedule.shtml
  - *Ina Schäfer, Christoph Seidl*: Modellbasierte Software-entwicklung, TU Braunschweig,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

Eclipse Sirius

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!



see https://eclipse.org/sirius/
and https://eclipse.org/sirius/getstarted.html

4.2. Textual syntax

Excursion on Compilers – Syntax Analysis

in the last lecture...

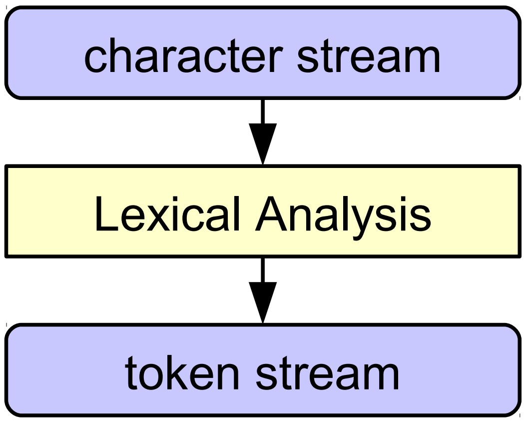
- Steps in the textual syntax analysis:

character stream

Excursion on Compilers – Syntax Analysis

in the last lecture...

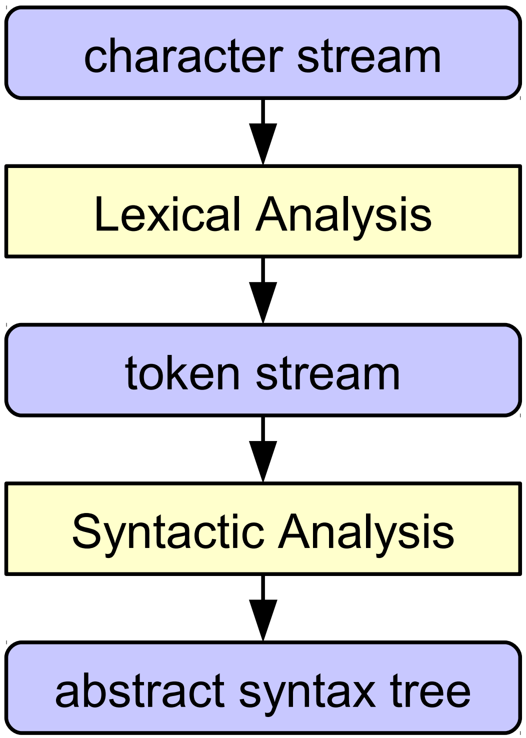
- Steps in the textual syntax analysis:
  - Lexical Analysis
    - partitions character stream into tokens (removes whitespaces, identifies keywords, identifiers, ...)



Excursion on Compilers – Syntax Analysis

in the last lecture...

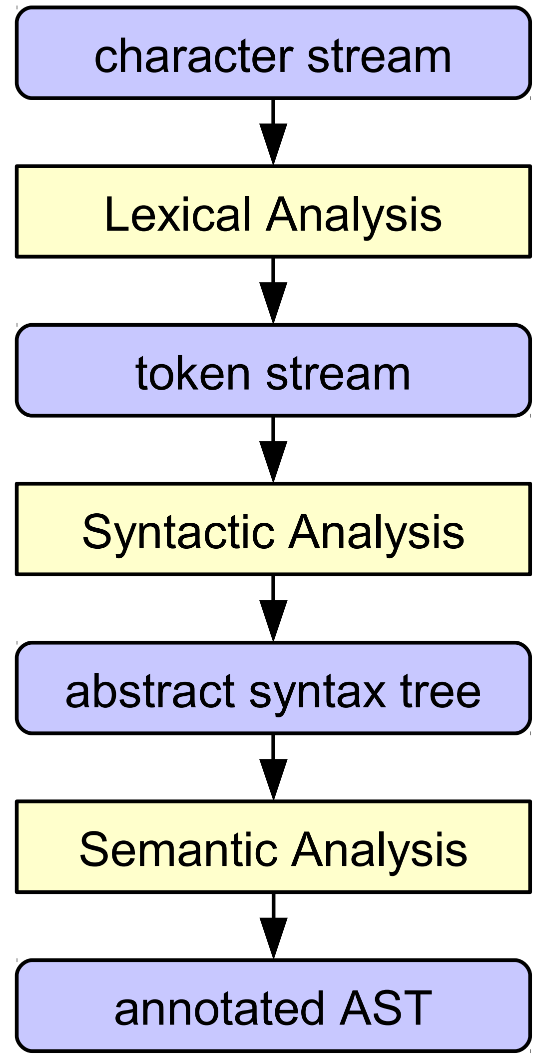
- Steps in the textual syntax analysis:
  - Lexical Analysis
    - partitions character stream into tokens (removes whitespaces, identifies keywords, identifiers, ...)
  - Syntax Analysis
    - context-free analysis identifies abstract syntax tree (AST) structure



Excursion on Compilers – Syntax Analysis

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...

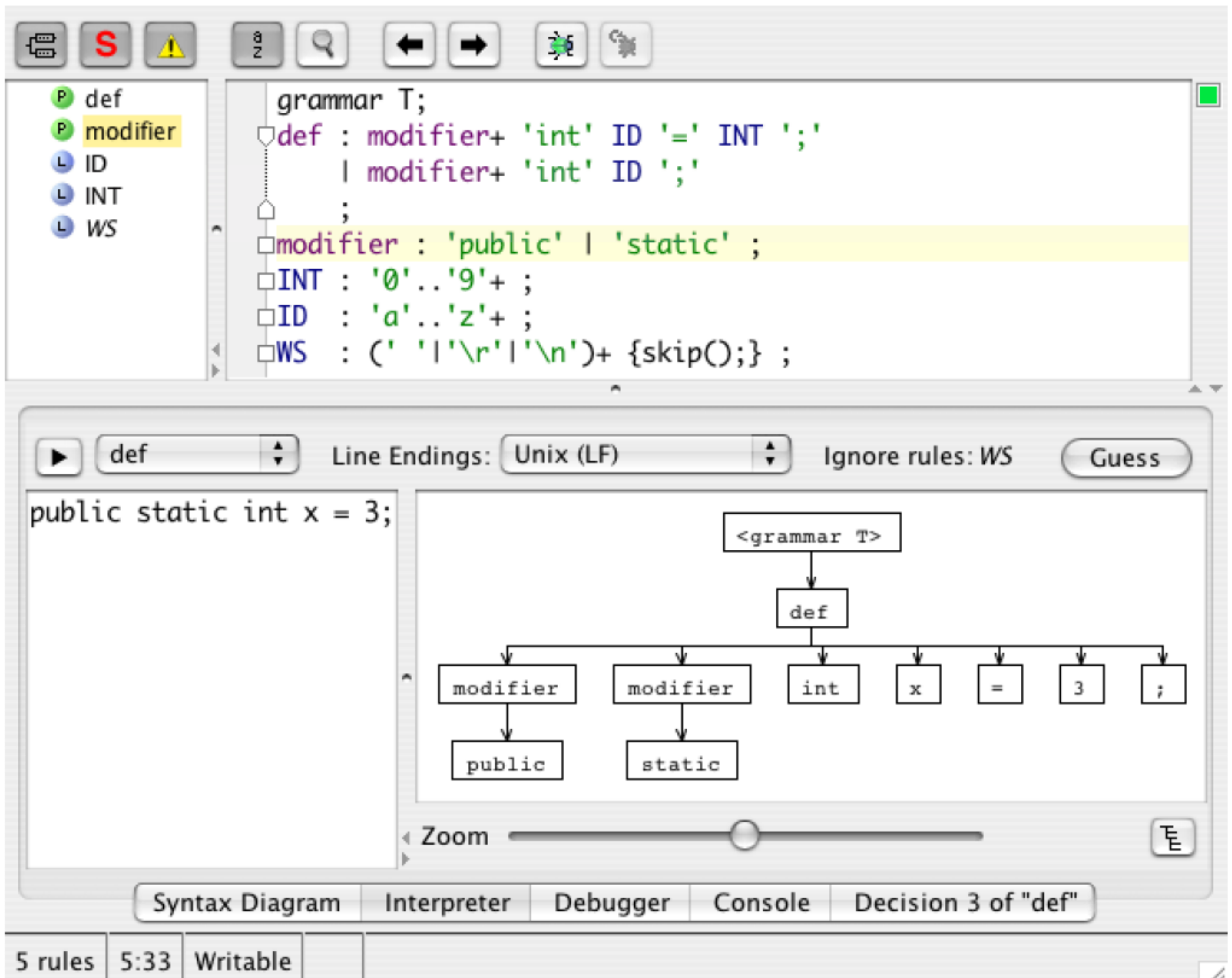
- There exist **frameworks for constructing compilers** that can generate lexer and parser components (and other things) from a language definition in the form of a grammar

- 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:



see http://www.antlr.org/papers/antlrworks-draft.pdf

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

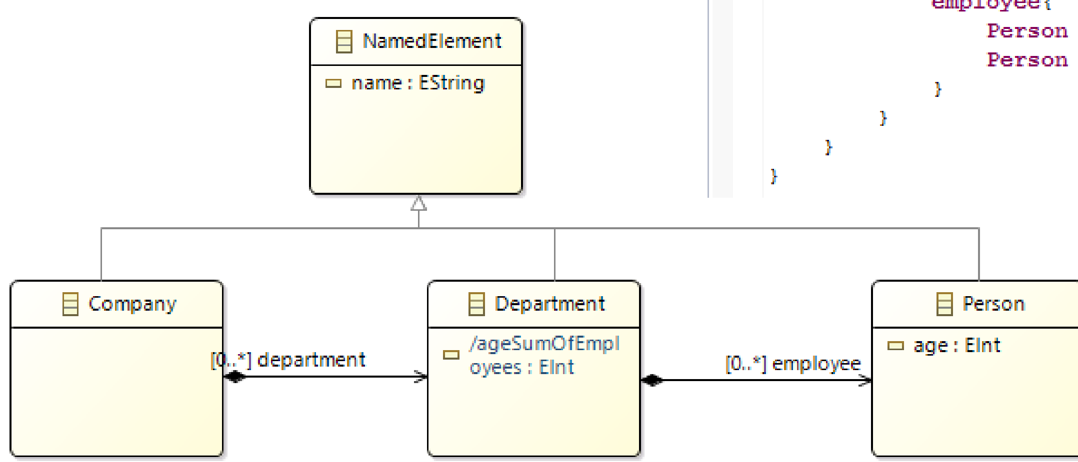
Xtext – Simple Example: Company

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

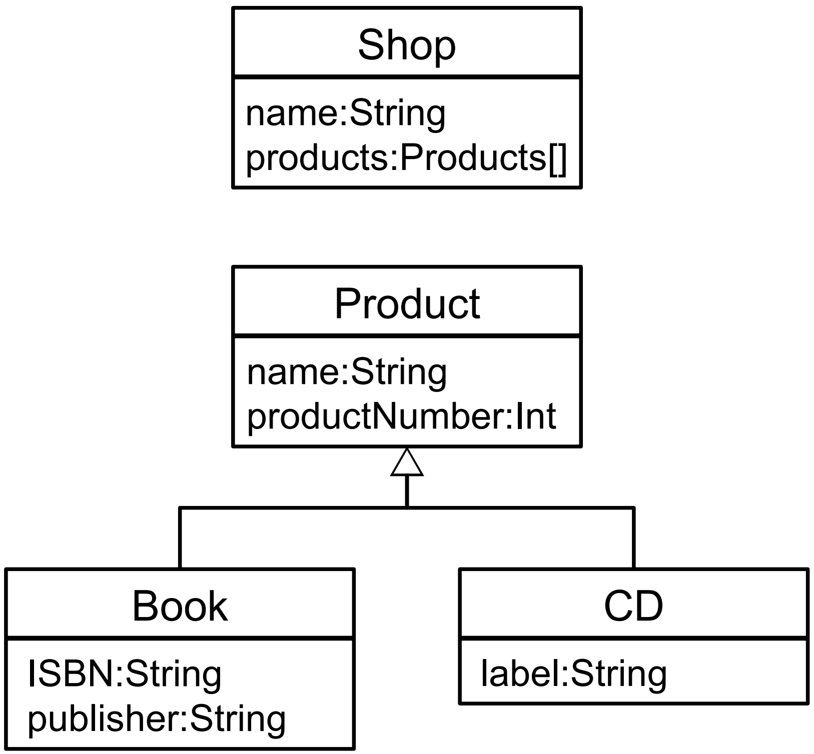
Literature

- 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




```
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;
}
```

graphical syntax

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*;

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

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

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;
```

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

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 `ID`s 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 `ID`s 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 `ID`s 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`

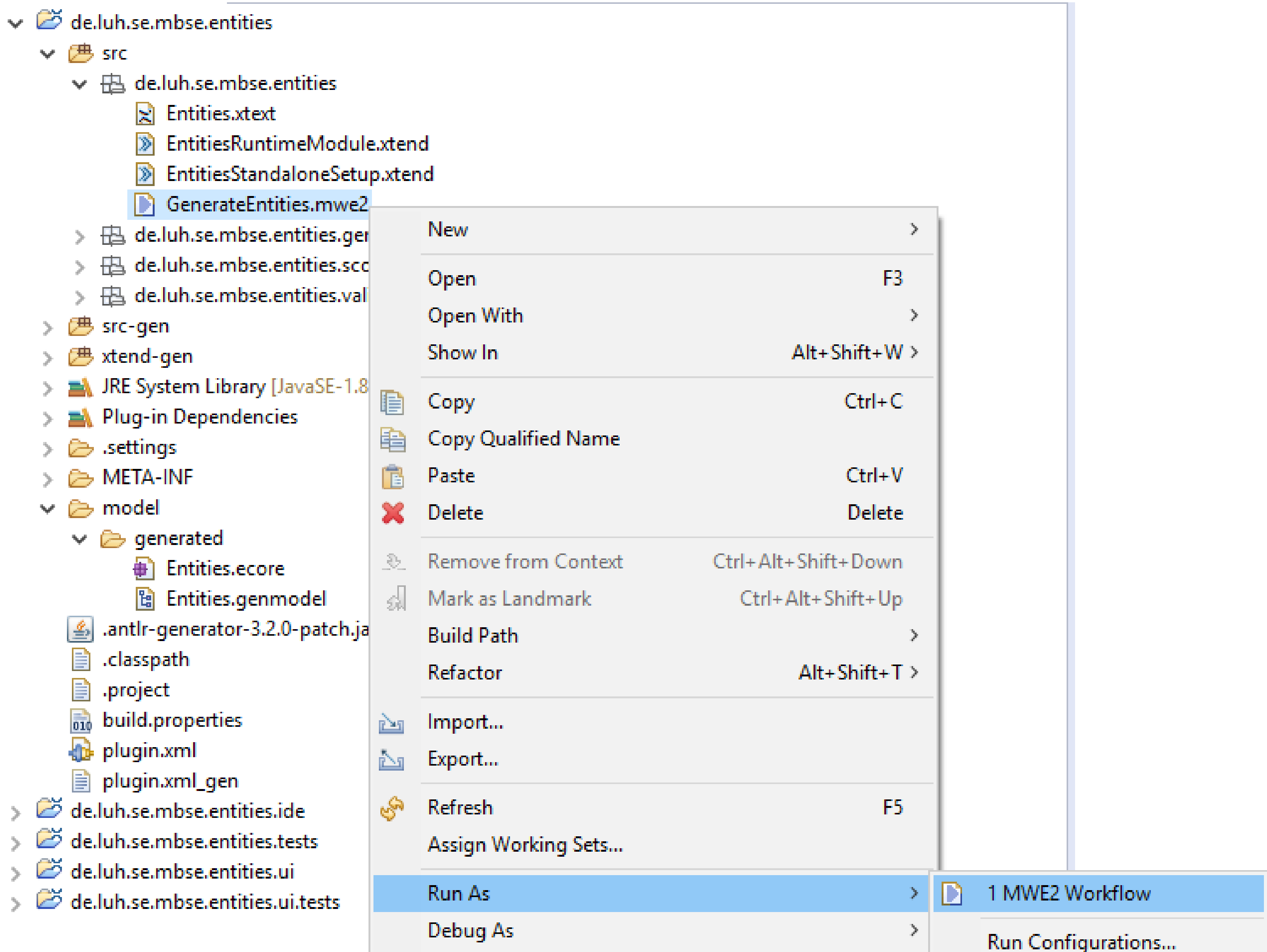
```
Entity:
        'entity' name=ID ;
```

- A look into `Terminals` shows what valid `ID`s are:

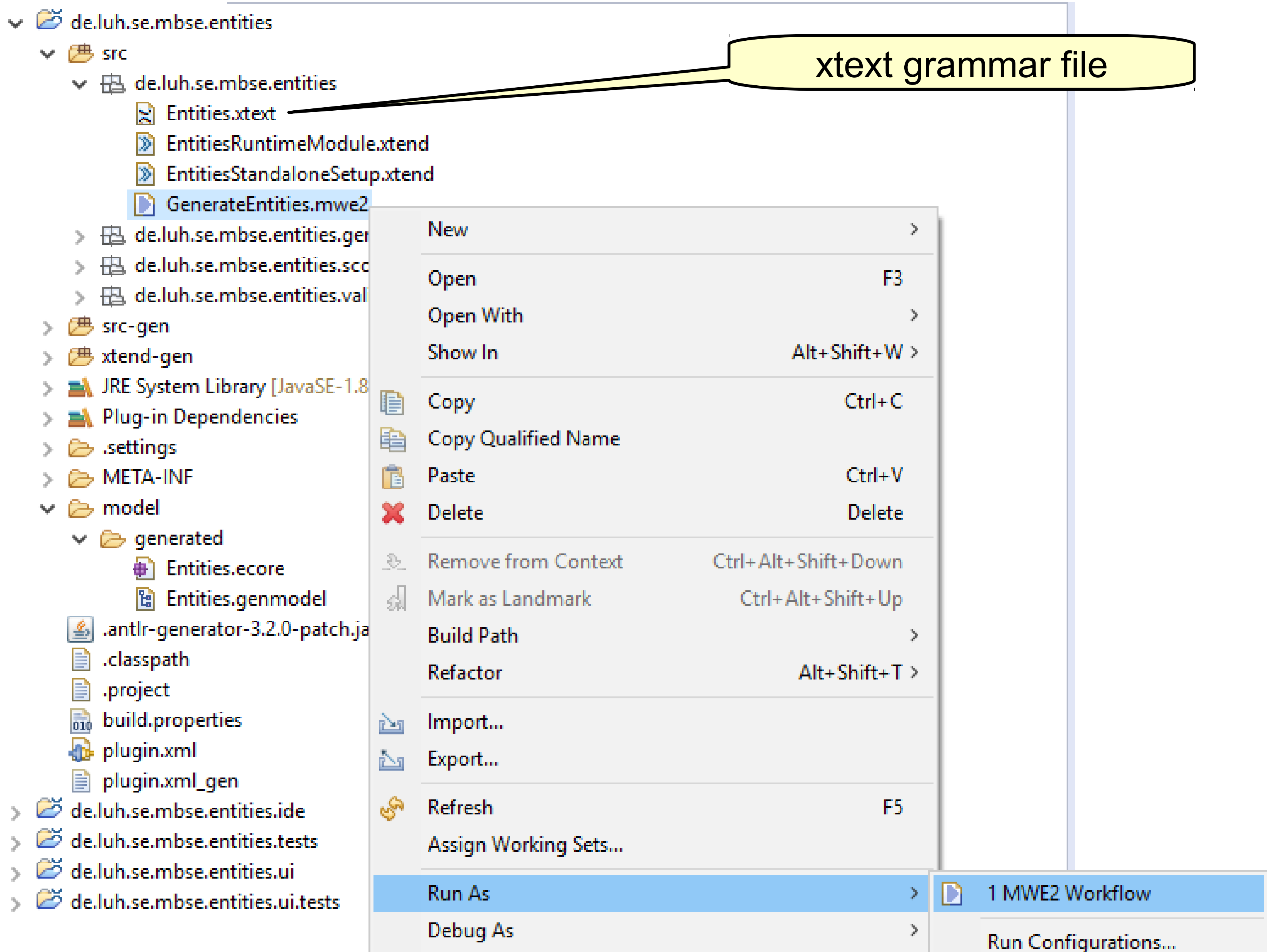
```
terminal ID:
        '^'?('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Any sequence of lower- and upper-case letters, underscores, and digits
- but no leading digits
- leading '^' possible

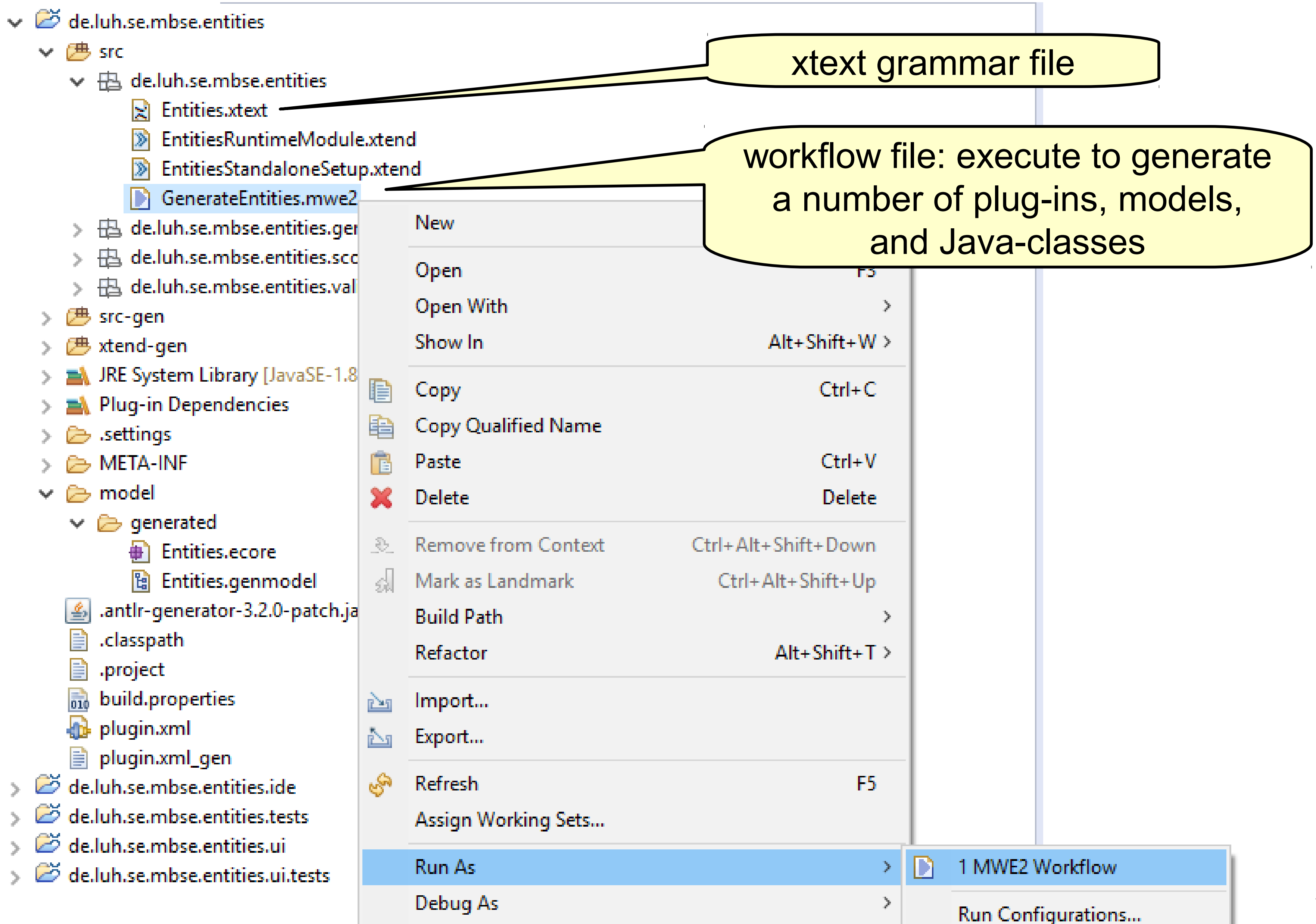
Example: Defining a Language for Entities (tool-specific: generate code)



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



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



Example: Defining a Language for Entities

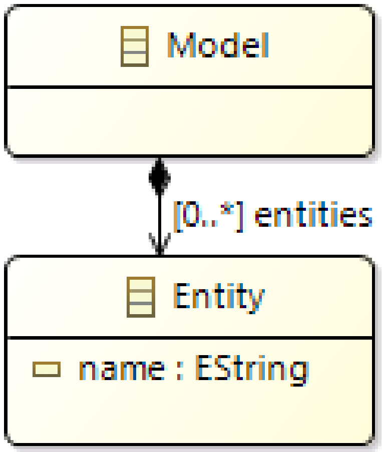
- How does the Ecore model look that is generated by Xtext?

```
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"

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*
	'}';

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

An `Entity` statement starts with the keyword “entity”, followed by a name

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?=
```

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

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

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

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*
	'}';

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"

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

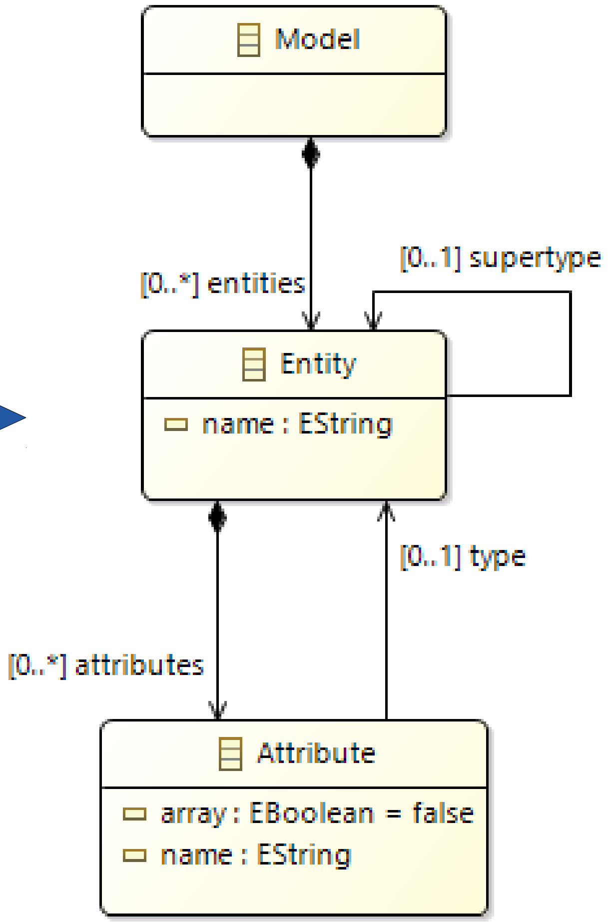
- How does the Ecore model look that is generated by Xtext?

```
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

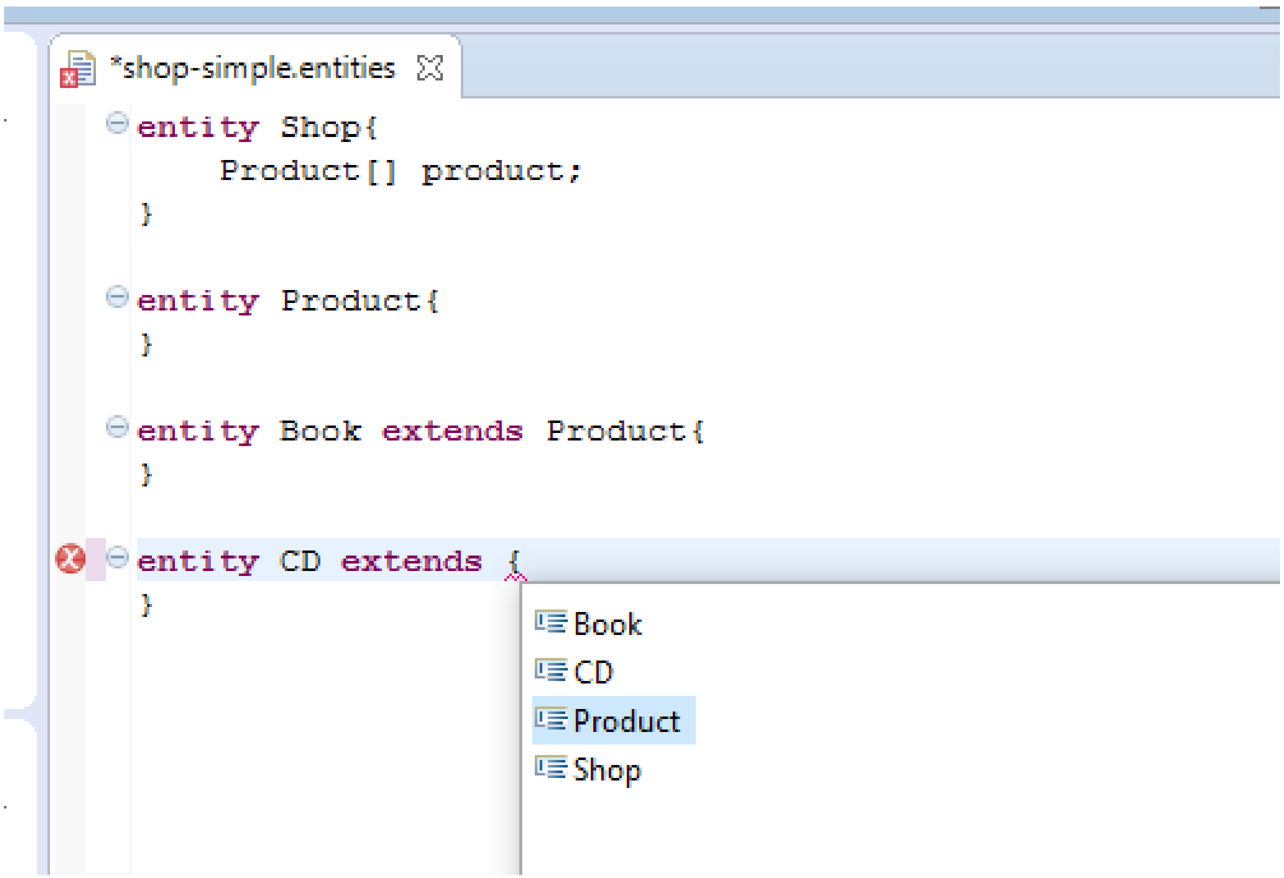
- How does the Ecore model look that is generated by Xtext?

```
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 ';';
```



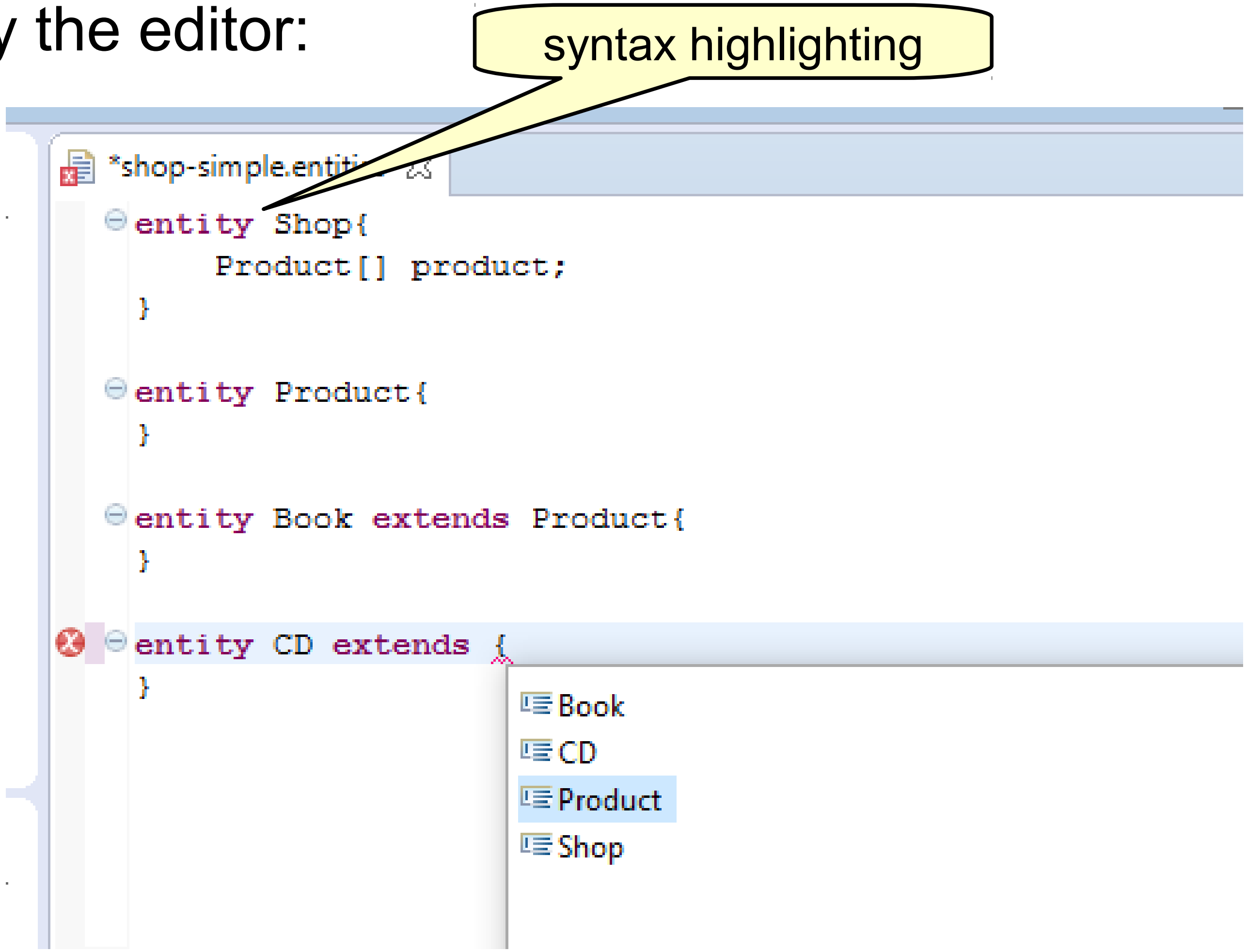
Example: Defining a Language for Entities

- Let's try the editor:



Example: Defining a Language for Entities

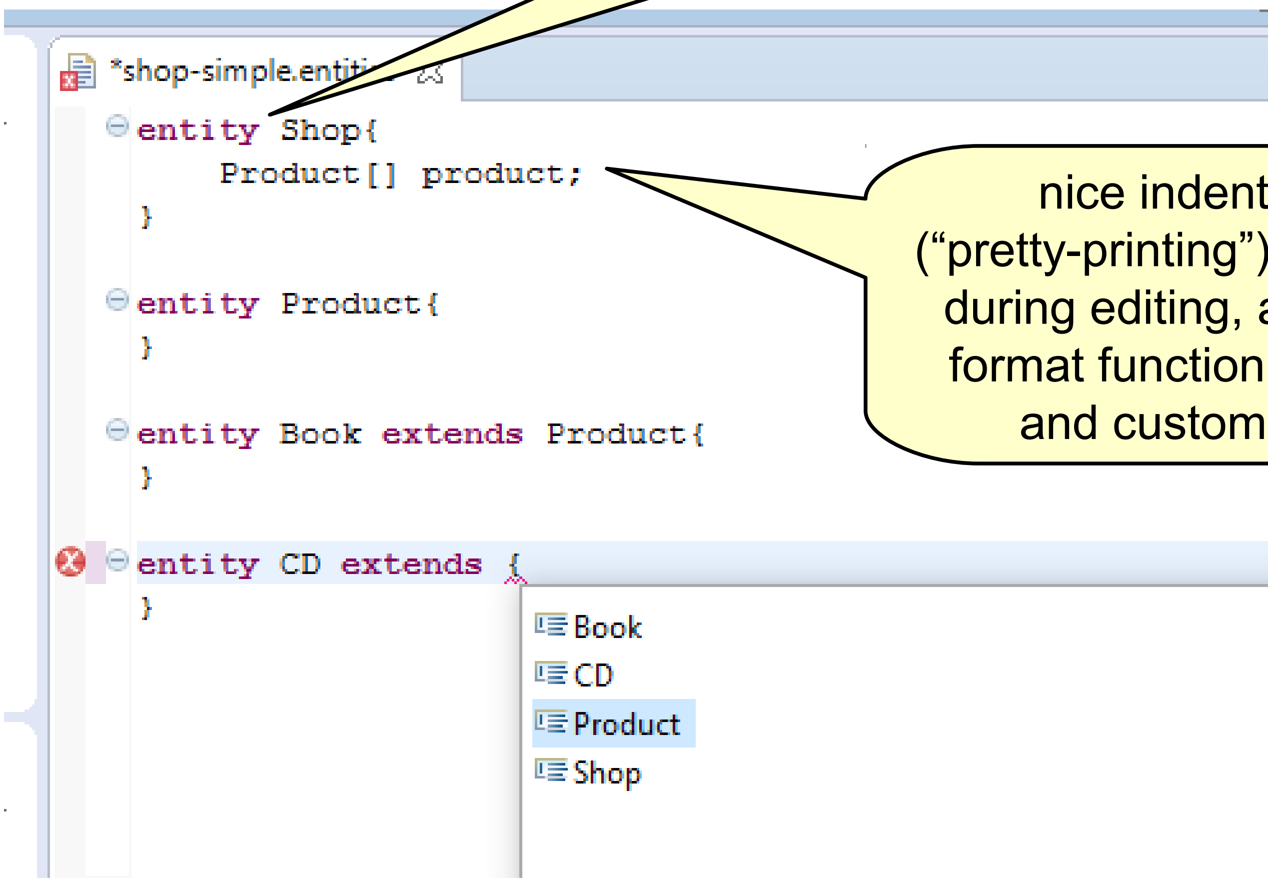
- Let's try the editor:



Example: Defining a Language for Entities

- Let's try the editor:

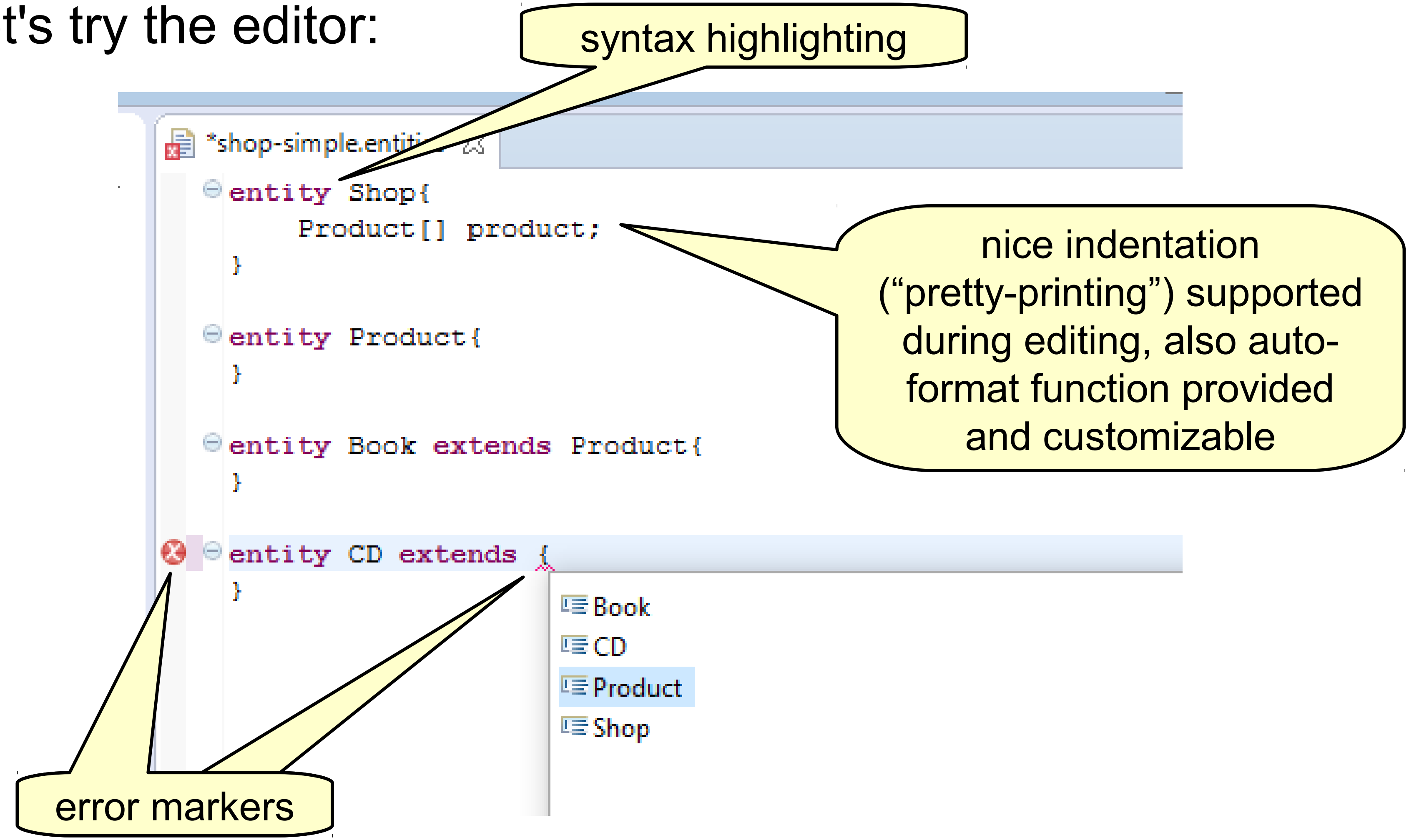
syntax highlighting



nice indentation (“pretty-printing”) supported during editing, also auto-format function provided and customizable

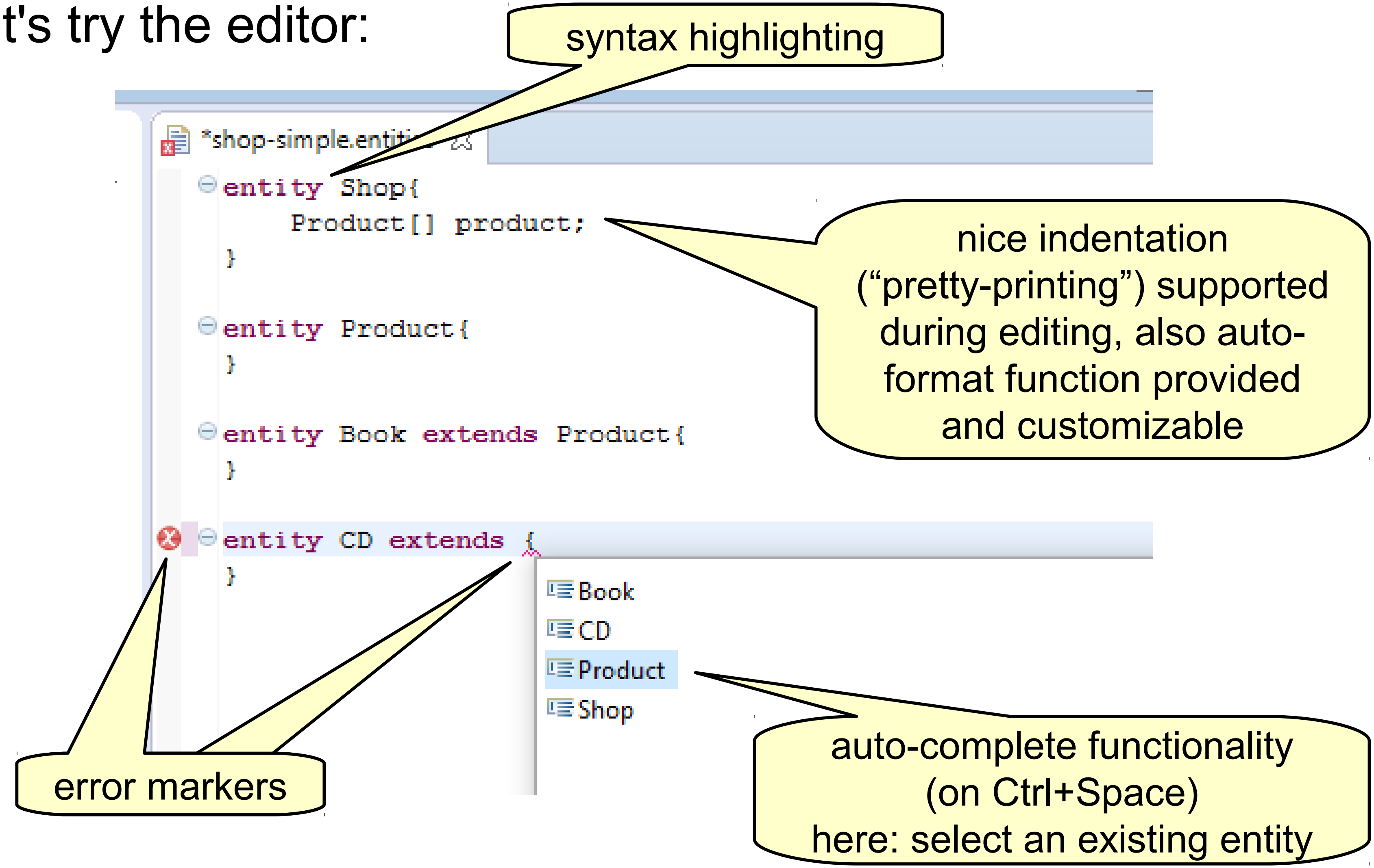
Example: Defining a Language for Entities

- Let's try the editor:



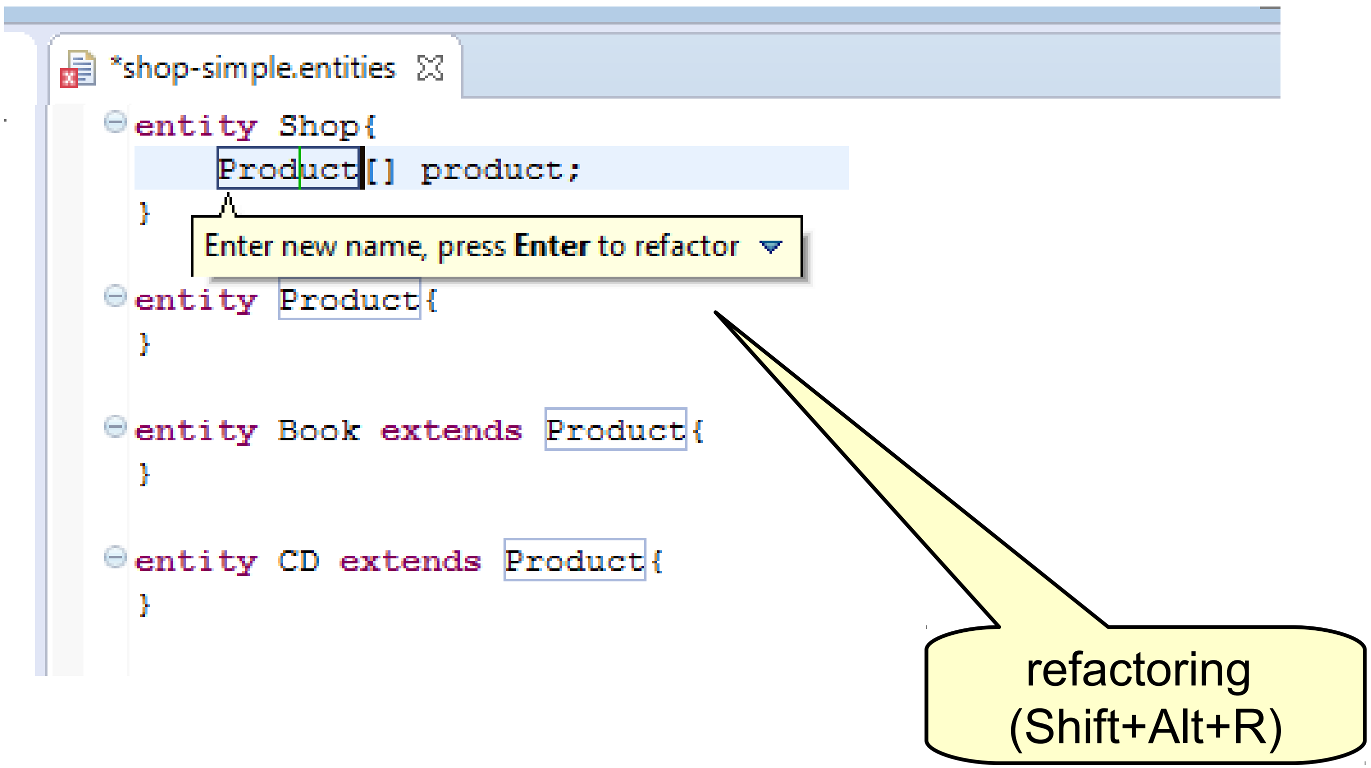
Example: Defining a Language for Entities

- Let's try the editor:



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*
     '}';

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

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

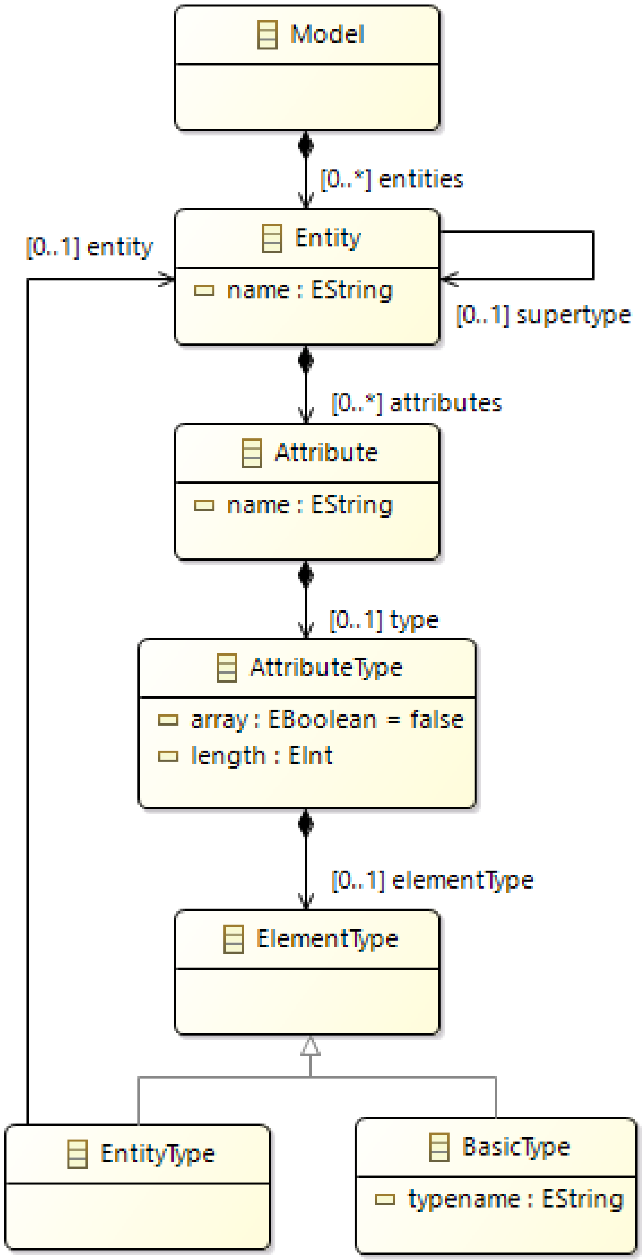
- How does the Ecore model look?

```
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

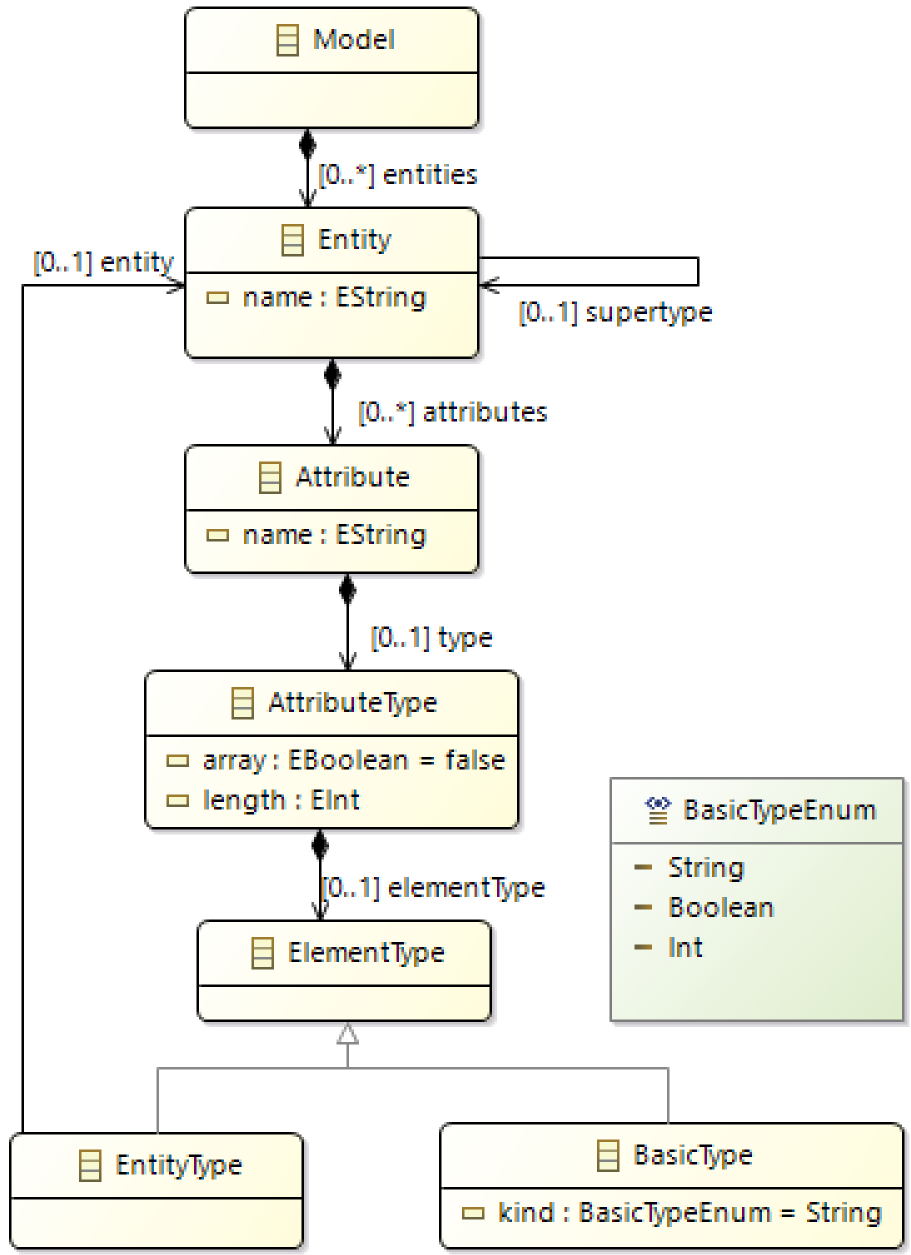
- We can use Enums, too.

```
...

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.pnl.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

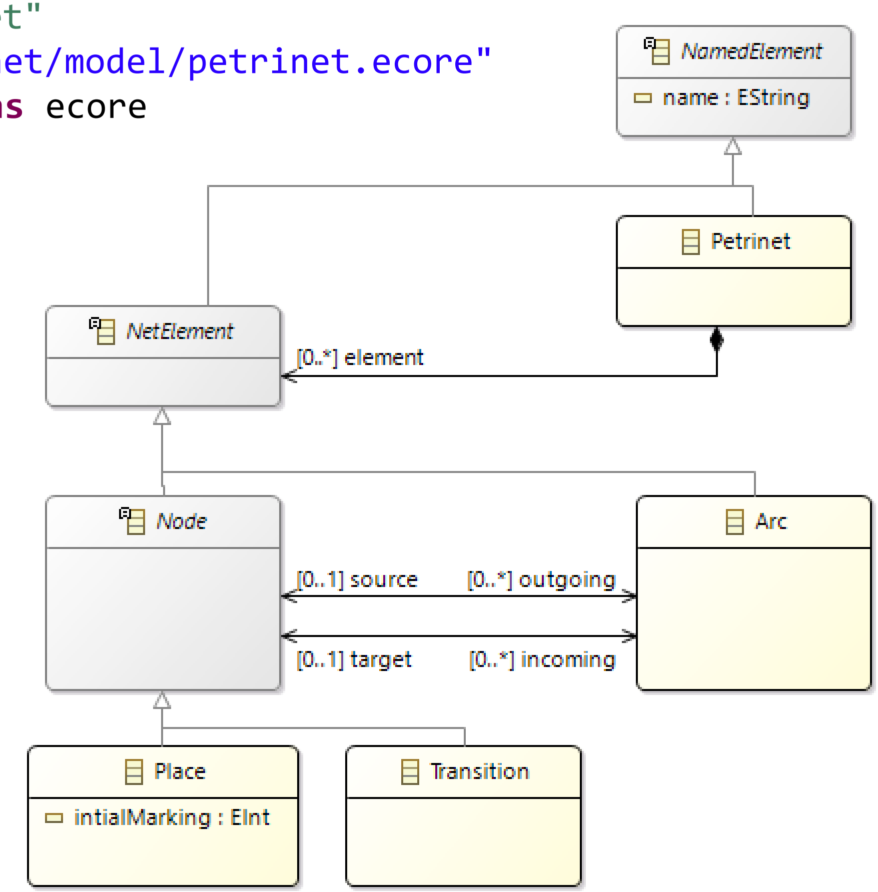
- An Xtext grammar can also refer to an existing Ecore model:

```
// automatically generated by Xtext
grammar de.luh.se.mbse.petrinet.pnl.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.pnl.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
}
```

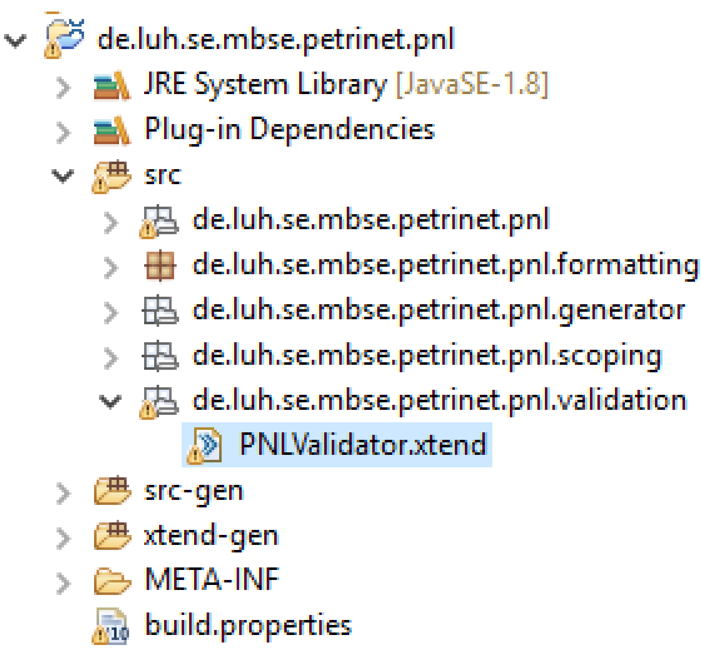
An arc can only connect a place to transition or a transition to a place

Custom Validation

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

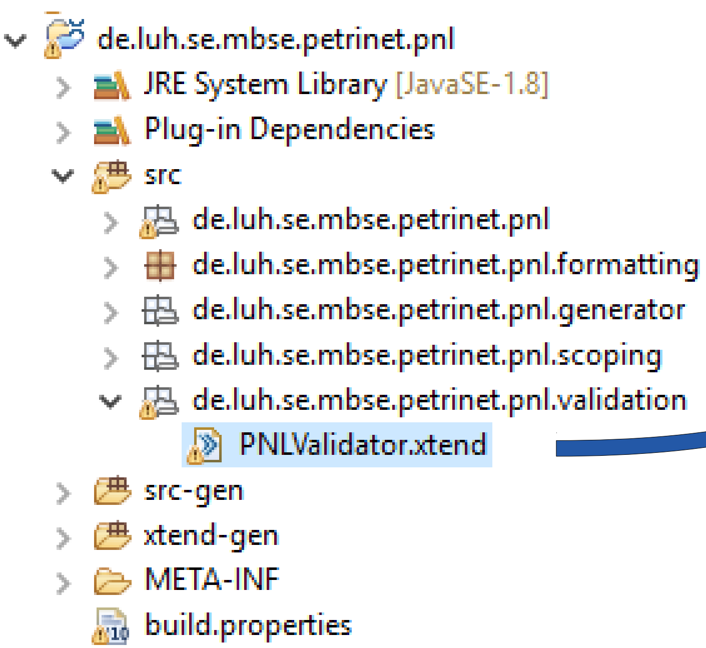
Custom Validation

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



Custom Validation

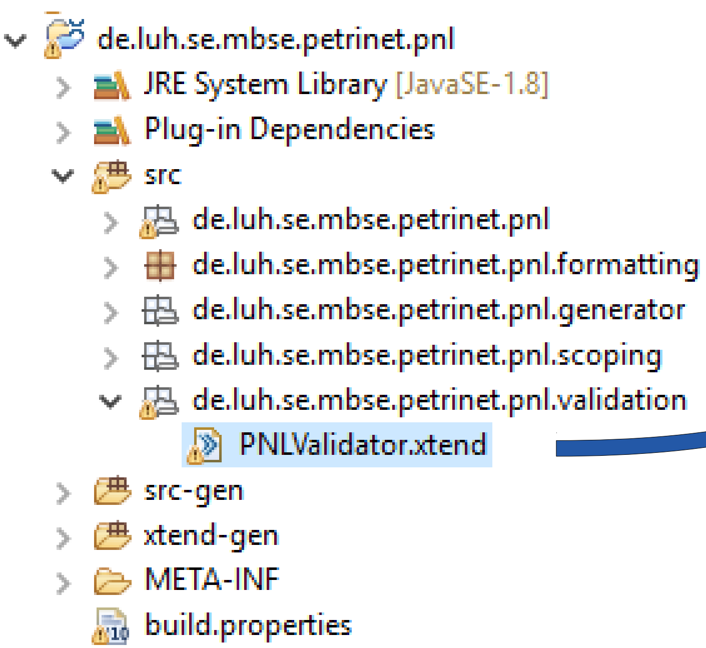
- You can add **custom validation functions** by implementing specific check methods in the **validator class**




```
@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

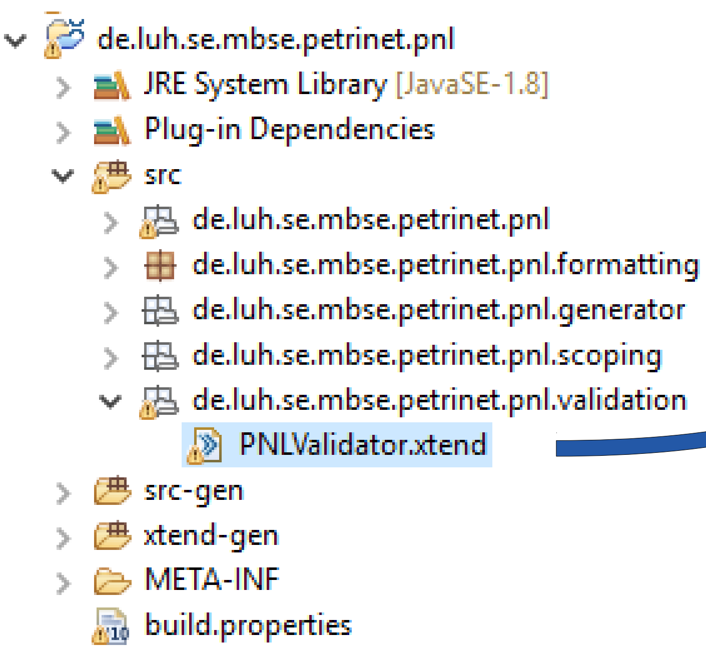
- You can add **custom validation functions** by implementing specific check methods in the **validator class**
  - implementation in Xtend, a Java-like programming language




```
@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

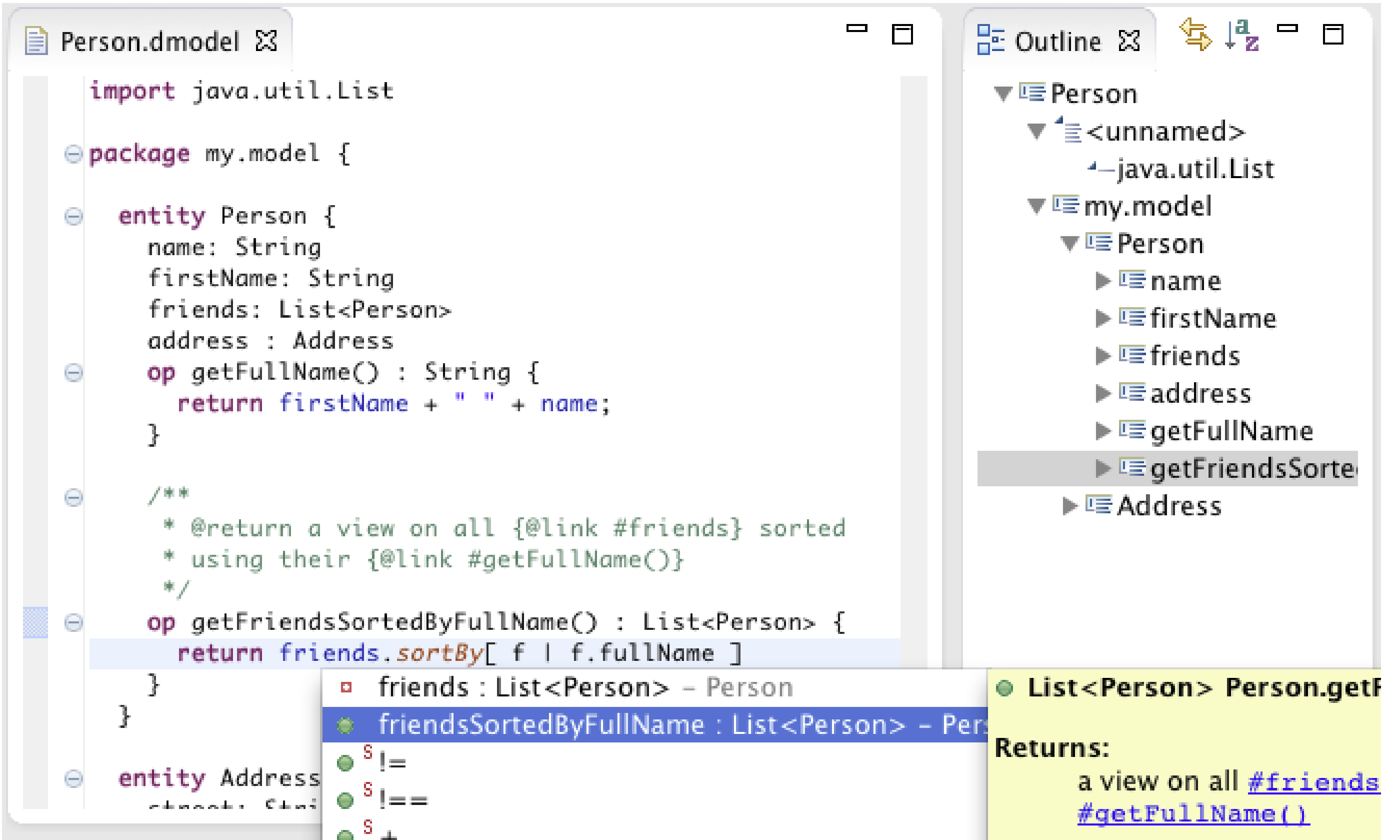
- define a textual language by using an EBNF-style grammar
- generate Ecore model or import existing one
- generates rich editor functionality
- allows extensions:
  - validation
  - quick fixes
  - scoping
  - formatting

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



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 an 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 an a textual syntax document, draw the corresponding object model diagram
  - Infer Ecore model from a given Xtext grammar