ESRG

Embedded Systems Research Group

CENTROALGORITMI

Domain-Specific Languages with Xtext and Xtend

Group 1: José Martins, Nicolas Almeida, Ricardo Teixeira, Pedro Fernandes, Miguel Araújo, Nelson Pinto e Rita Martins

Universidade do Minho
Escola de Engenharia

COMPETE

QREN Quadro de Referência Estratégico Nacional

União Europeia
Fundo Europeu de Desenvolvimento Regional

FCT Fundação para a Ciência e a Tecnologia
Ministério da Ciência, Tecnologia e Ensino Superior

Agenda

- Implementing a DSL
  - Defining a DSL
  - DSL and GPL
  - DSL vs XML
  - Parsing
  - Grammar
  - AST
  - Integration with IDE
  - Creating a Project in Eclipse
- Creating your first Xtext language
  - Creating a simple language
  - Terminal Rules
  - EBNF
  - Try editor
  - Xtext Generator
  - Eclipse Modeling Framework
  - Improvements to the DSL
  - Dealing with types

DSL

- What is a DSL?
  - **Domain Specific Languages** are programming languages or specification languages that target a specific problem domain;
  - Implementing a DSL means developing a program that is able to read text written in that DSL, parse it, process it, and then possibly interpret it or generate code in another language;
  - Examples include HTML, SQL, Mathematica, etc.

GPL

- What is a GPL?
  - **General-Purpose Language** is a programming language designed to be used for writing software in a wide variety of application domains;
  - Examples include Java, C, Pascal, etc.

- A program written in a DSL can be interpreted or compiled in a GPL.

Why use DSL?

- XML
  - –The XML tags insert syntax noise to the information;
  - –Not straightforward.

```
<people>
  <person>
    <name>James</name>
    <surname>Smith</surname>
    <age>50</age>
  </person>
  <person employed="true">
    <name>John</name>
    <surname>Anderson</surname>
    <age>40</age>
  </person>
</people>
```

- DSL
  - –More human readable;
  - –Less noise;
  - –Easier to grasp;
  - –More compact specification.

```
person {
  name=James
  surname=Smith
  age=50
}
person employed {
  name=John
  surname=Anderson
  age=40
}
```

```
James Smith (50)
John Anderson (40) employed
```

Implementing a DSL

- The implementation has to make sure that the lexic and syntax are respected;
- **Lexical analysis** is the process of converting a sequence of characters into a sequence of tokens using regular expressions syntax;
- The program that performs this analysis is called **lexer** or simply a **scanner**.
- Each token is an atomic element:
  - Keyword;
  - Identifier;
  - Symbol name.

Parsing

- In the **syntactic analysis**, the sequence of tokens must form a valid statement in the language;

- In this given example, the input must respect the following structure:
  - two literal strings
  - the operator (
  - one integer literal
  - the operator)
  - the optional keyword employed.

```
James Smith (50)
John Anderson (40) employed
```

- The parser relies on the lexer.

Grammar

- A **grammar** is a set of rules that describe the form of the elements that are valid according to the language syntax;

- It is not necessary to implement a parser by hand because there are already tools to deal with this problem. These tools are called **parser generators** or **compiler-compilers**.

Grammar

- Flex
  - Lexical structure that generates the lexer in C.
- Bison
  - Syntactic structure that generates the parser;
  - Implementation of Yacc (Yet Another Compiler-Compiler).

Grammar

- ANTLR (ANother Tool for Language Recognition)
  - Allows the programmer to specify the grammar in one single file;
  - Doesn't separate the syntactic and lexical specifications in different files.

```
expression
   : INT
   | expression '*' expression
   | expression '+' expression
;
```

Abstract Syntax Tree

- **Abstract Syntax Tree**, which is build during parsing, stores a representation of the parsed program;

- The AST is stored in memory so that it can be later used for **semantic analysis** (i.e. type checking) and code generation, without needing to parse the same text every time.

Abstract Syntax Tree

- For building the AST we need two things:
  - code for node representation;

```
interface Expression { }

class Literal implements Expression {
  Integer value;
  // constructor and set methods...
}

class BinaryExpression implements Expression {
  Expression left, right;
  String operator;
  // constructor and set methods...
}
```

Abstract Syntax Tree

- For building the AST we need two things:
  - grammar specification with the actions.

```
expression:
  INT { $value = new Literal(Integer.parseInt($INT.text)); }
| left=expression '*' right=expression {
  $value = new BinaryExpression($left.value, $right.value);
  $value.setOperator("*");
}
| left=expression '+' right=expression {
  $value = new BinaryExpression($left.value, $right.value);
  $value.setOperator("+");
}
;
```

Integration with the IDE

- A DSL should have a good IDE support, to make the adoption easier by programmers. Some useful IDE features include:
  - **Syntax highlighting**: gives immediate feedback concerning the syntax correctness and colors the different elements;
  - **Background parsing**: the programming environment should not let the programmer realize about errors too late;
  - **Error markers**: it should highlight the parts of the program with errors directly in the editor and fill the view Problem with the errors;

Integration with the IDE

- **Content assist:** the feature that automatically, or on demand, provides suggestions on how to complete the statement the programmer just typed;
- **Hyperlink:** makes it possible to navigate between references in a program;
- **Quickfixes:** when the programmer makes a mistake and the DSL implementation is able to fix it;

Integration with the IDE

- **Outline**: clicking on an element of the outline should bring the programmer directly to the corresponding source line in the editor;
- **Automatic build**: when a file is modified and saved, the IDE will automatically compile that file and all of its dependencies.

- All the features of the Eclipse Java editor are based on the Eclipse framework.

Xtext

- **Xtext** is an Eclipse framework for implementing programming languages and DSLs;
- **Automatically generates** the lexer, the parser, the AST models, the construction of the AST and the Eclipse editor with all the IDE features;
- It **only needs a grammar specification** similar to ANTLR; it doesn't need to annotate the rules with actions to build the AST, since its creation is handled automatically by Xtext itself.

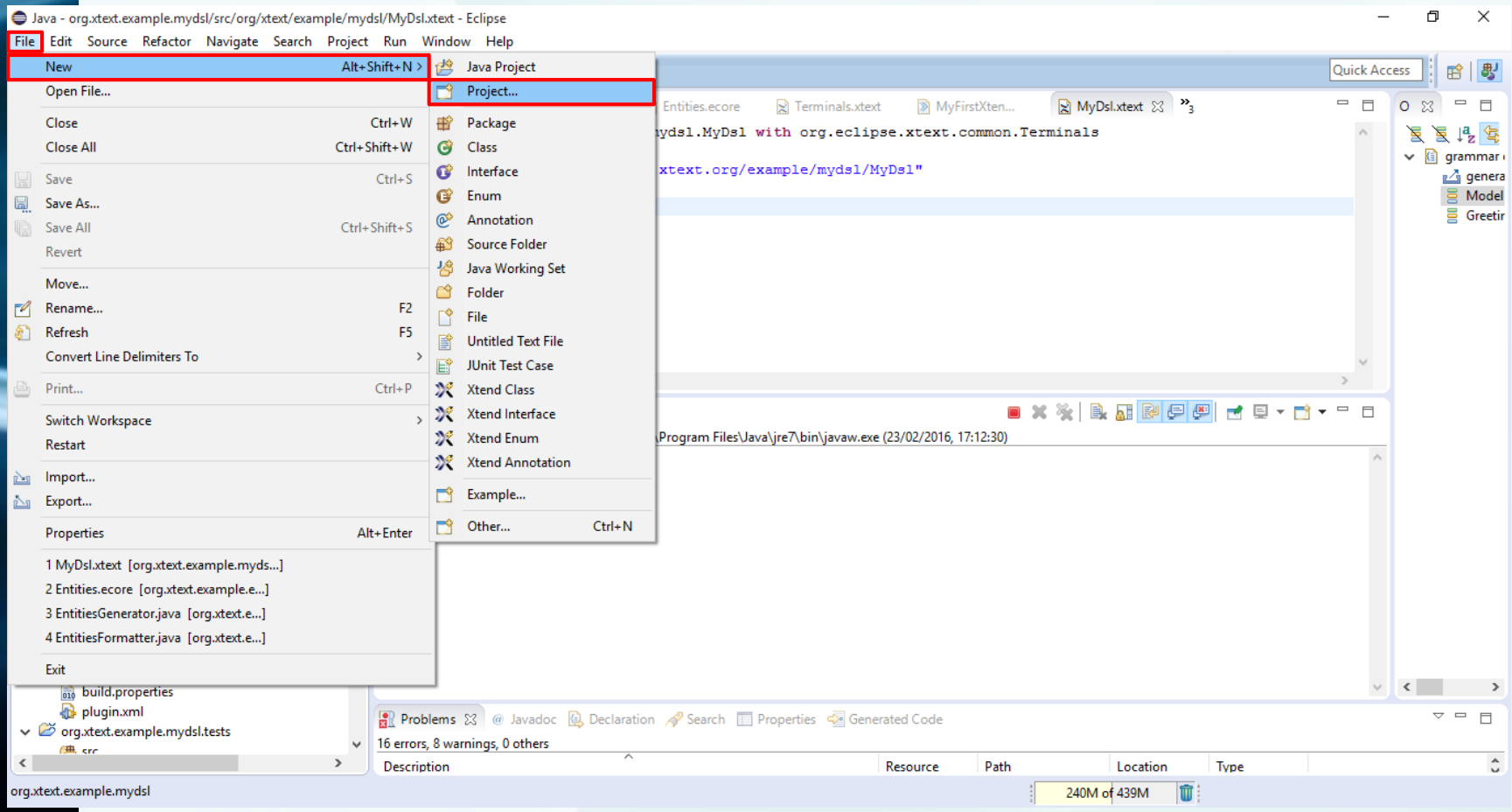
Installing Xtext

- First method
  - In Eclipse enter Help | Install new software…;
  - Copy this link to 'Work with' text box:
    - http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases
  - Press [ENTER] and wait until the search is complete;
  - Then choose Xtend SDK 2.4.2 and Xtext SDK 2.4.2. (or newer version).

- Second method
  - Download Eclipse IDE for Java and DSL Developers available at http://www.eclipse.org/downloads

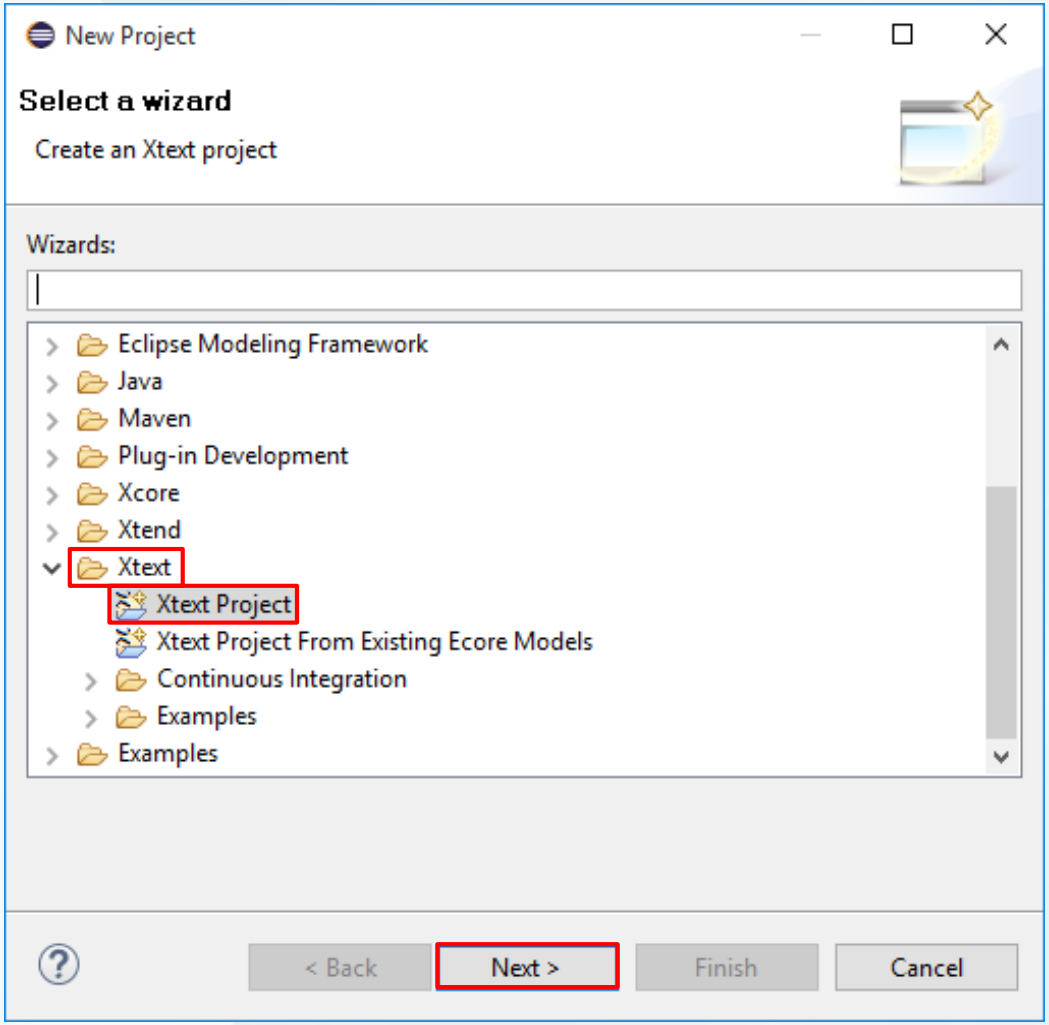
Creating a Project - First Method

1.Click on on File -> New -> Project...



Creating a Project - First Method

2.Choose XtextProject



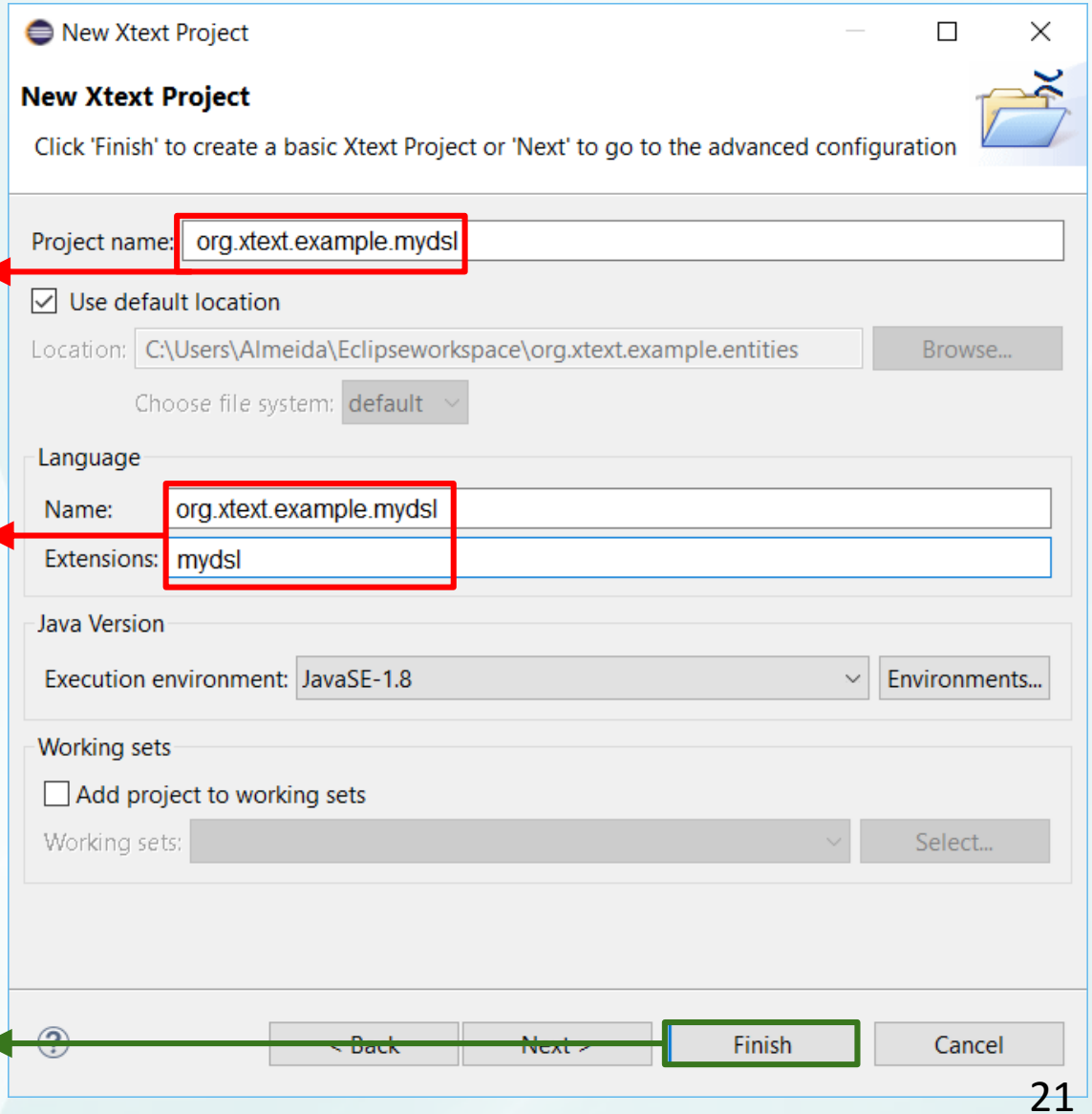
Creating a Project - First Method

3.

1º Change project name

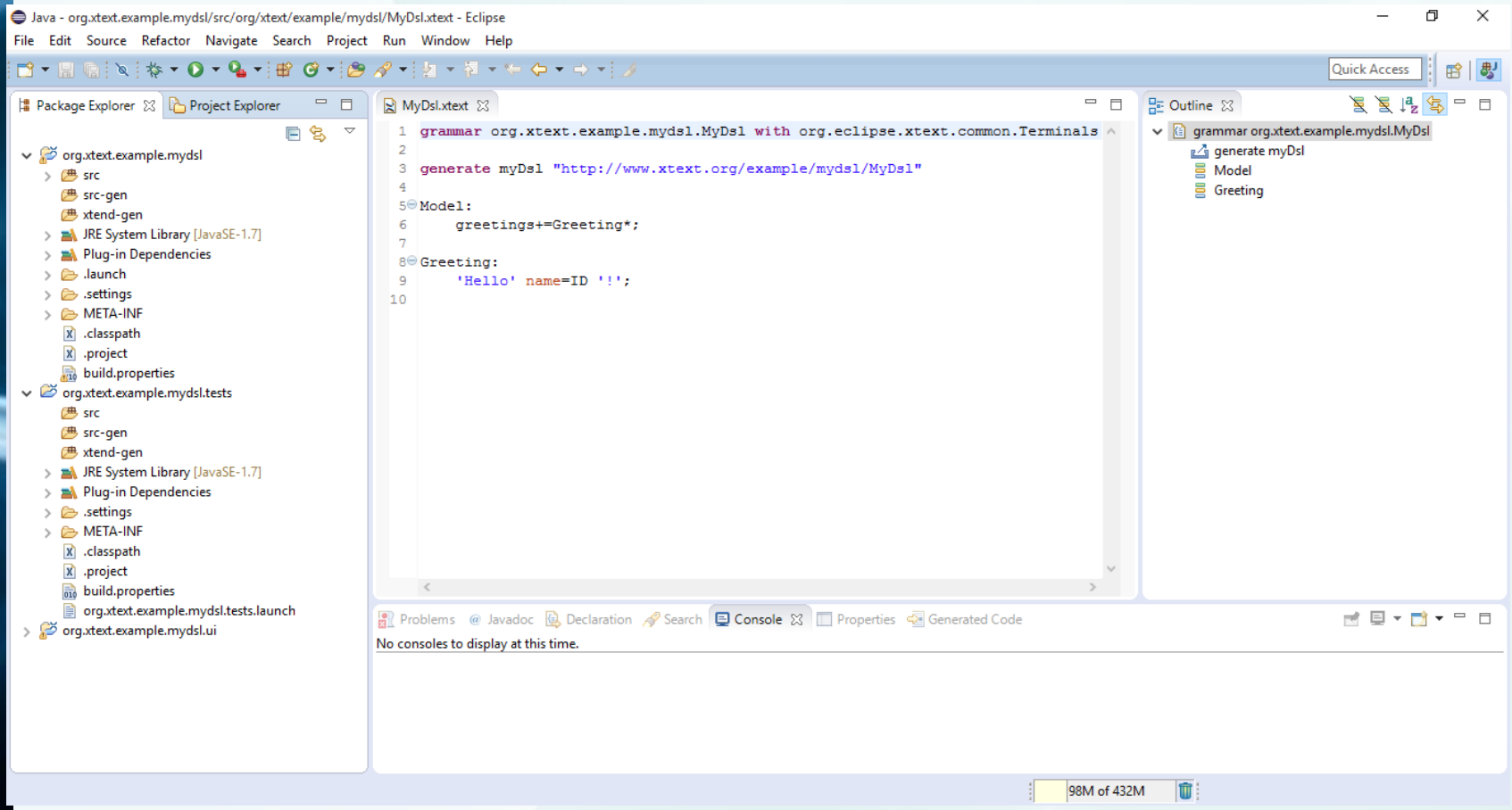
2º Change language and extensions name

3º Click button “Finish”



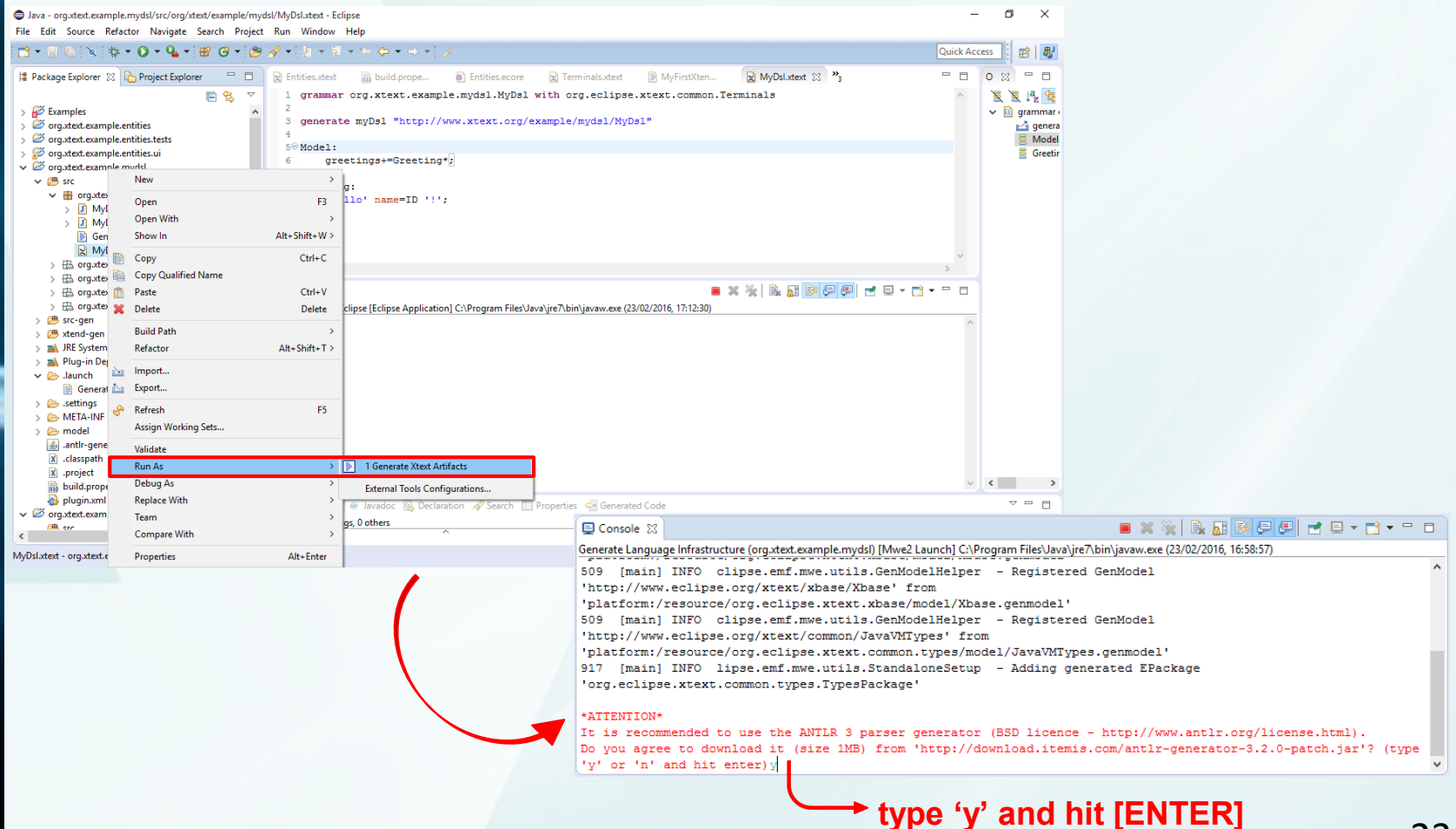
Creating a Project - First Method

4.



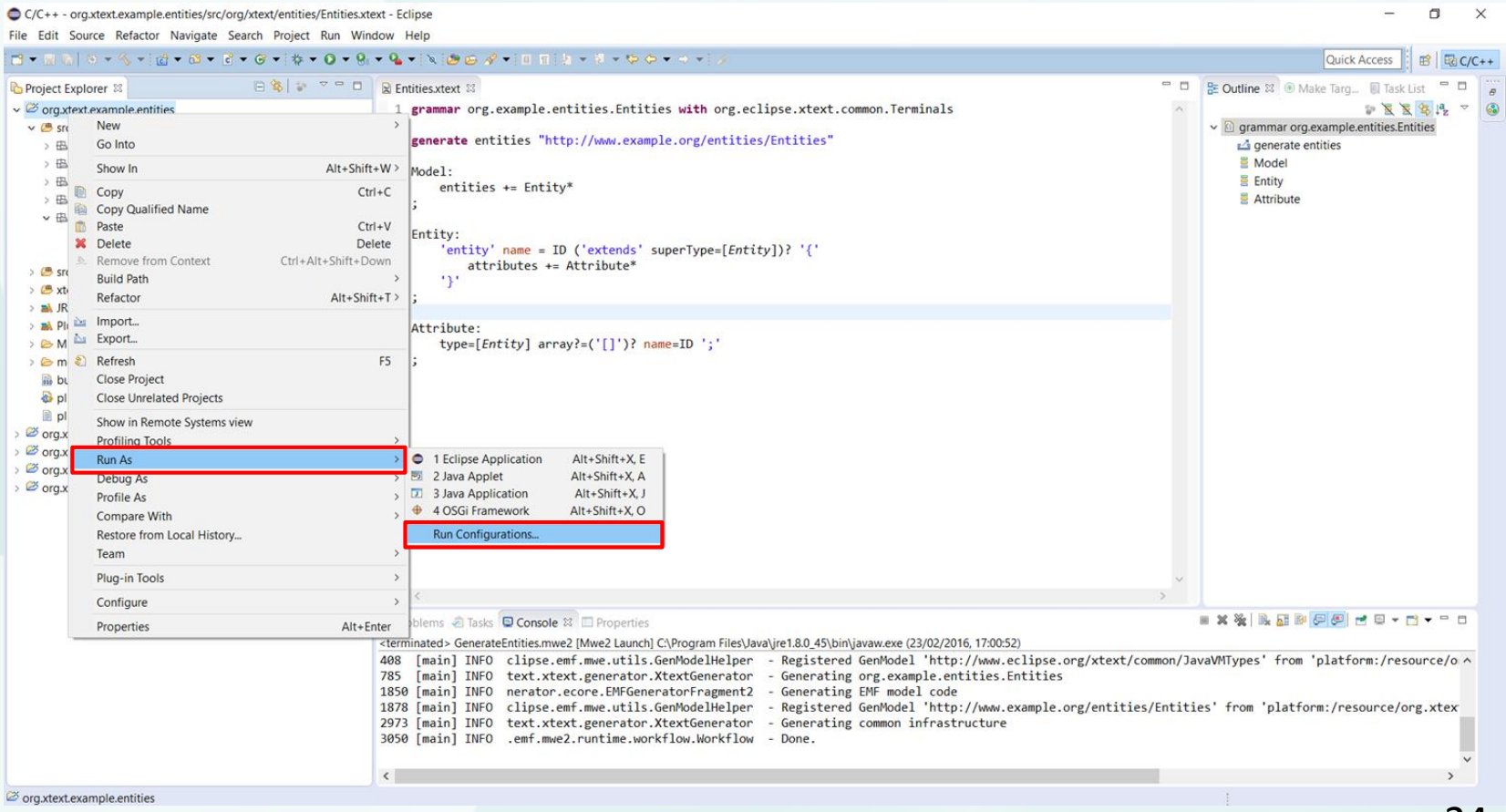
Creating a Project - First Method

5.Right-click on ‘MyDsl.xtext’



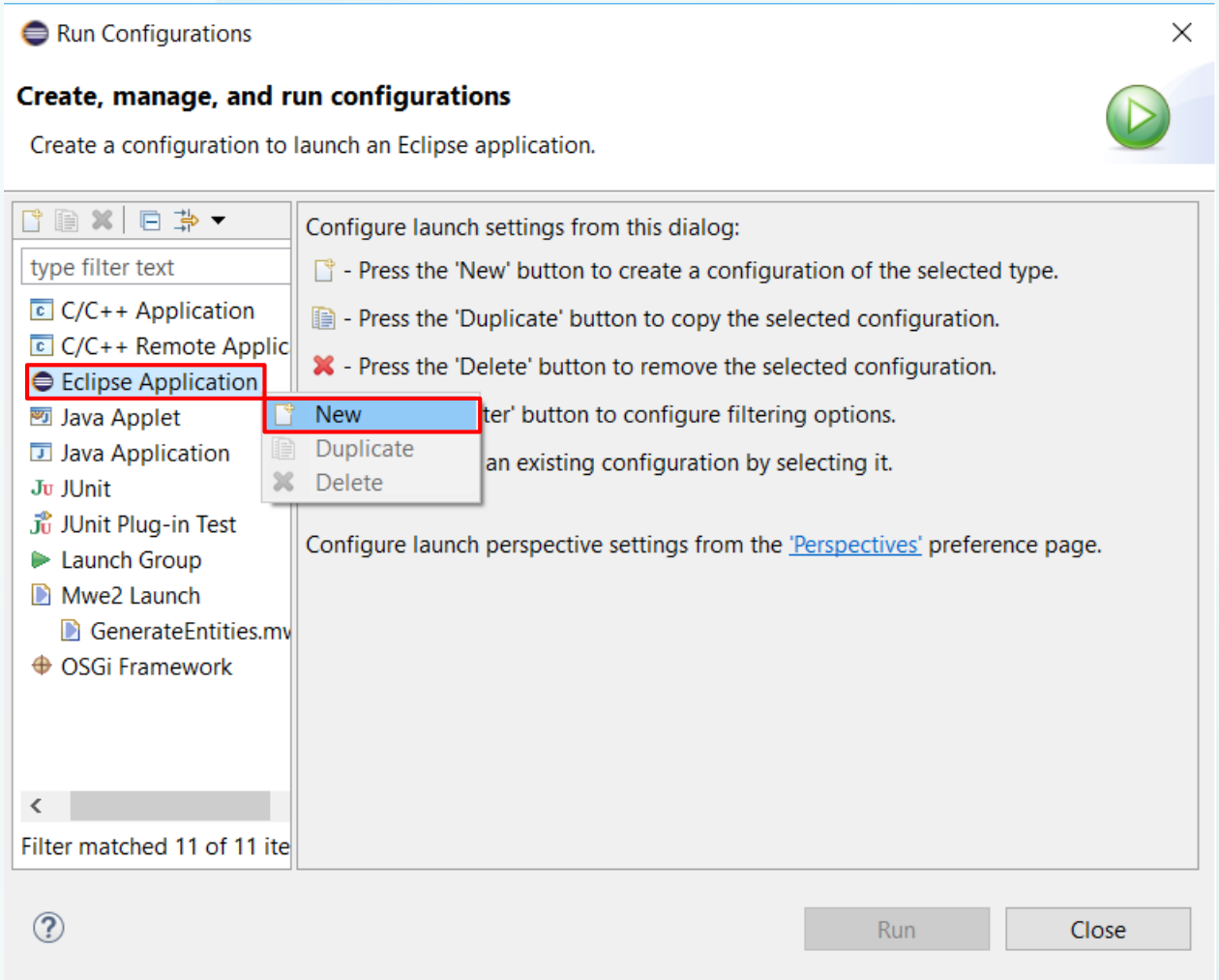
Creating a Project - First Method

6.Right-click on ‘org.xtext.example.mydsl’



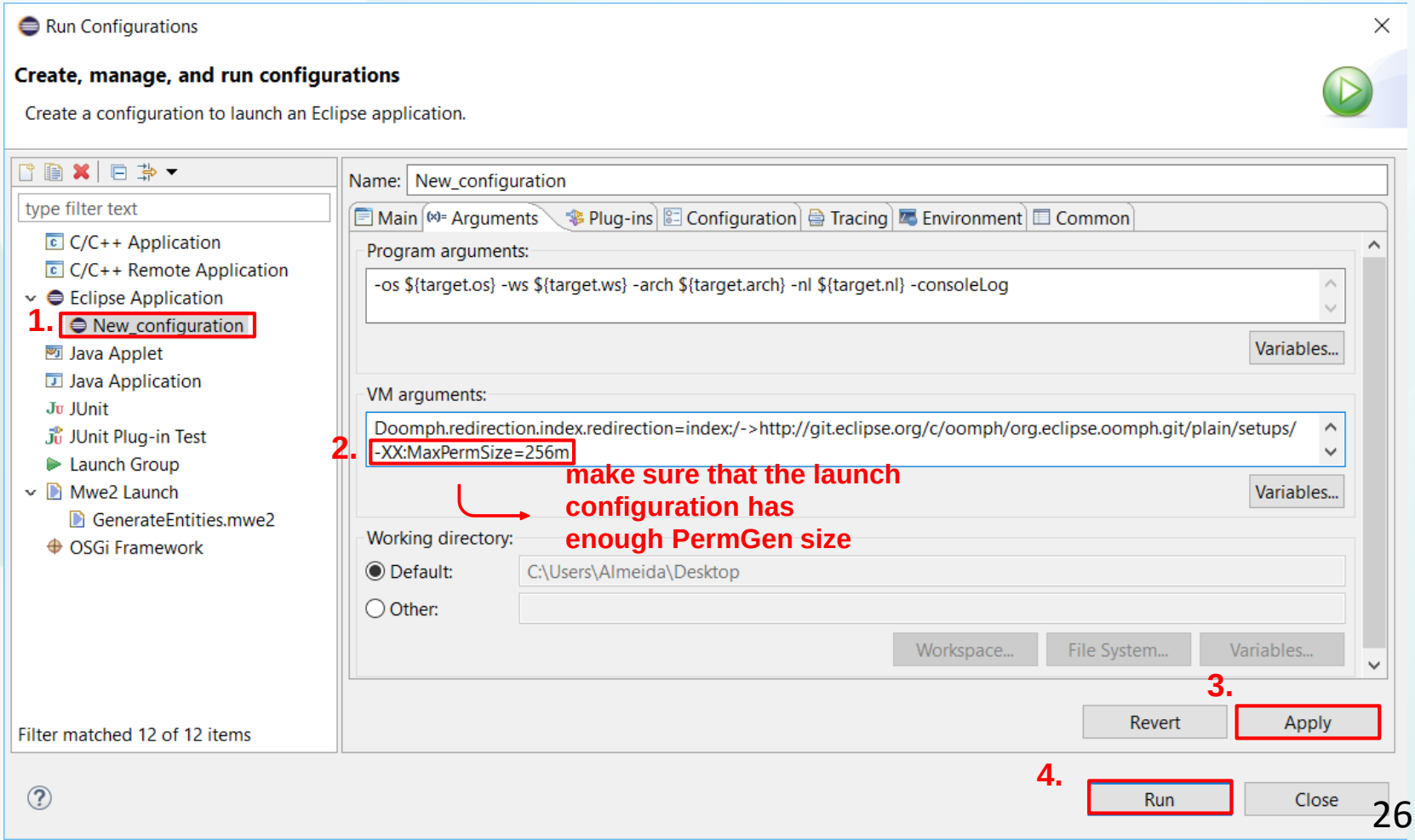
Creating a Project - First Method

7.Right-click on ‘Eclipse Application’



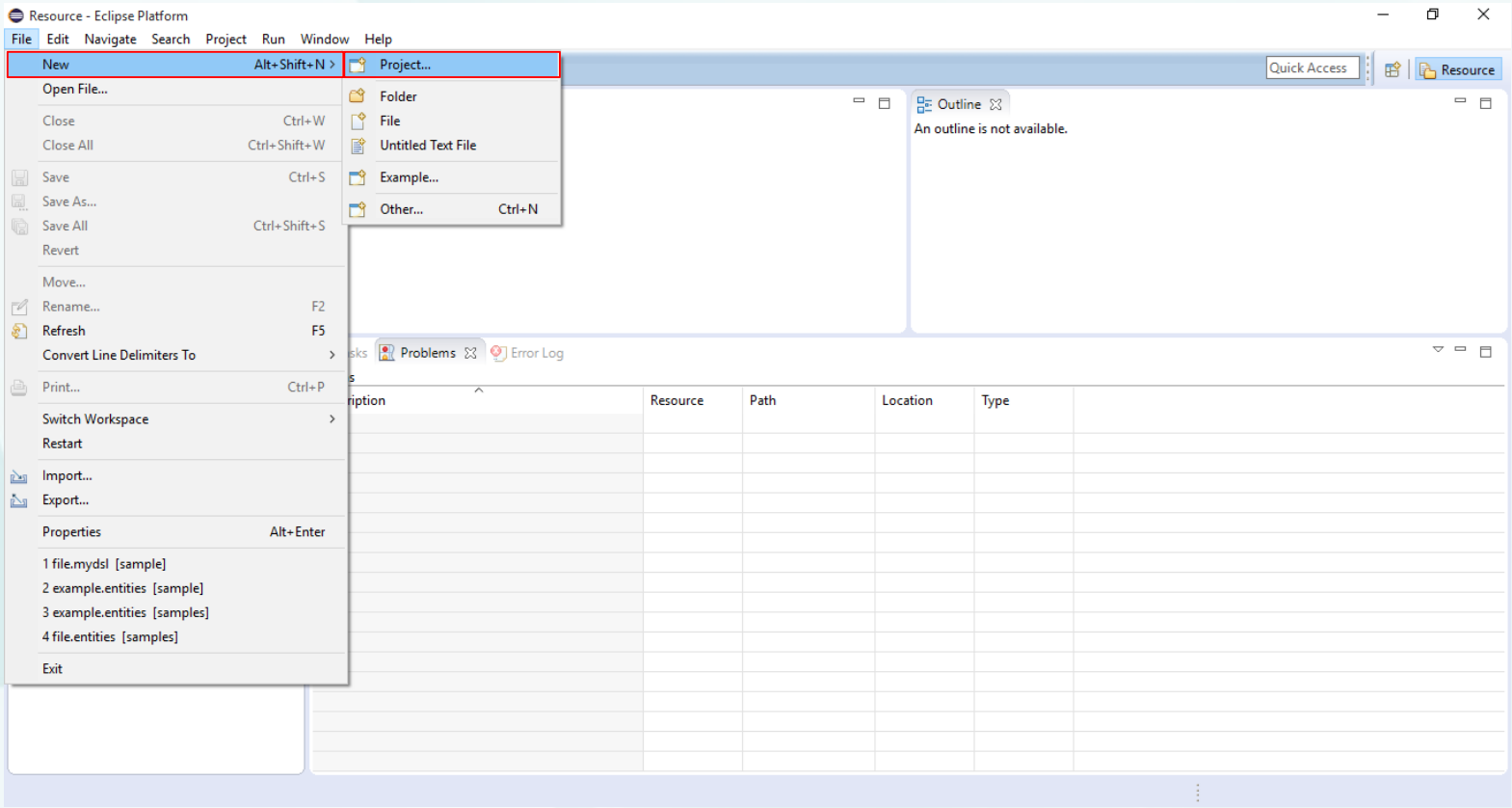
Creating a Project - First Method

8. Make



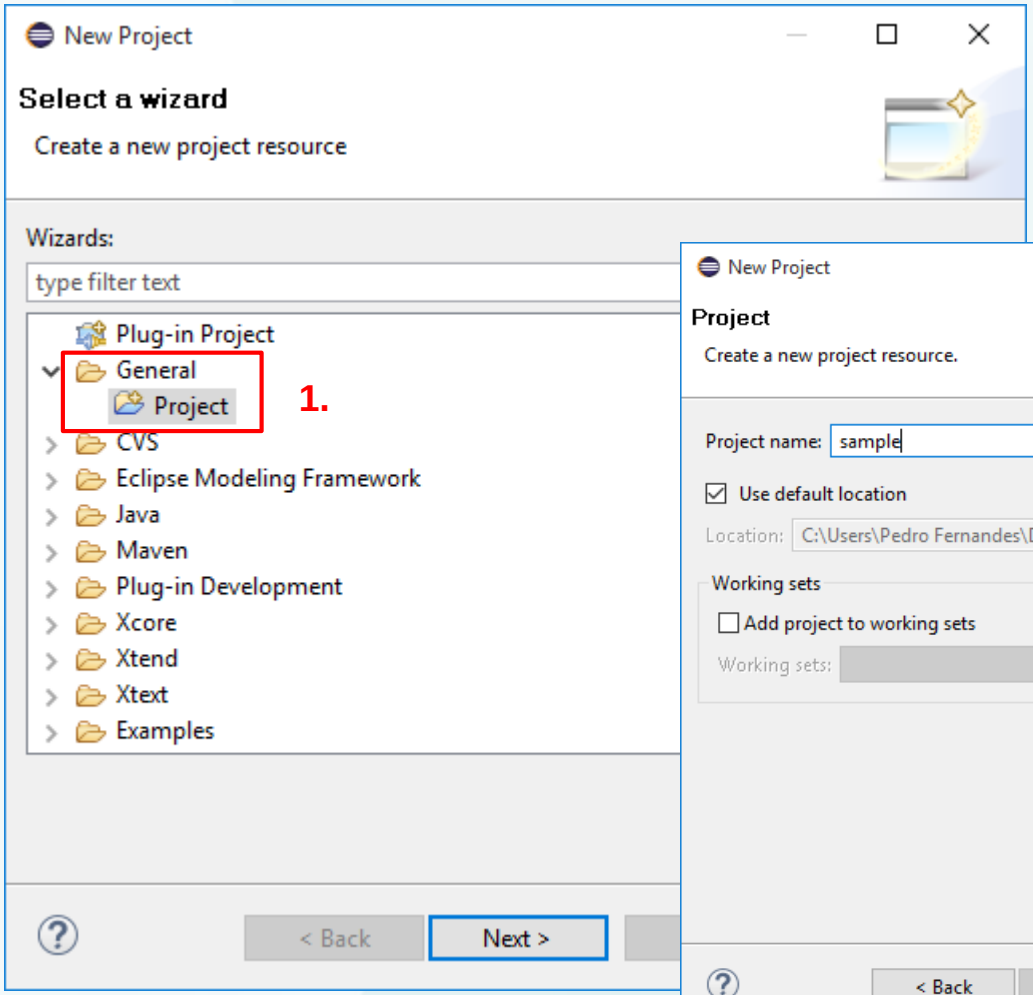
Creating a Project - First Method

9.In the new Eclipse instance:



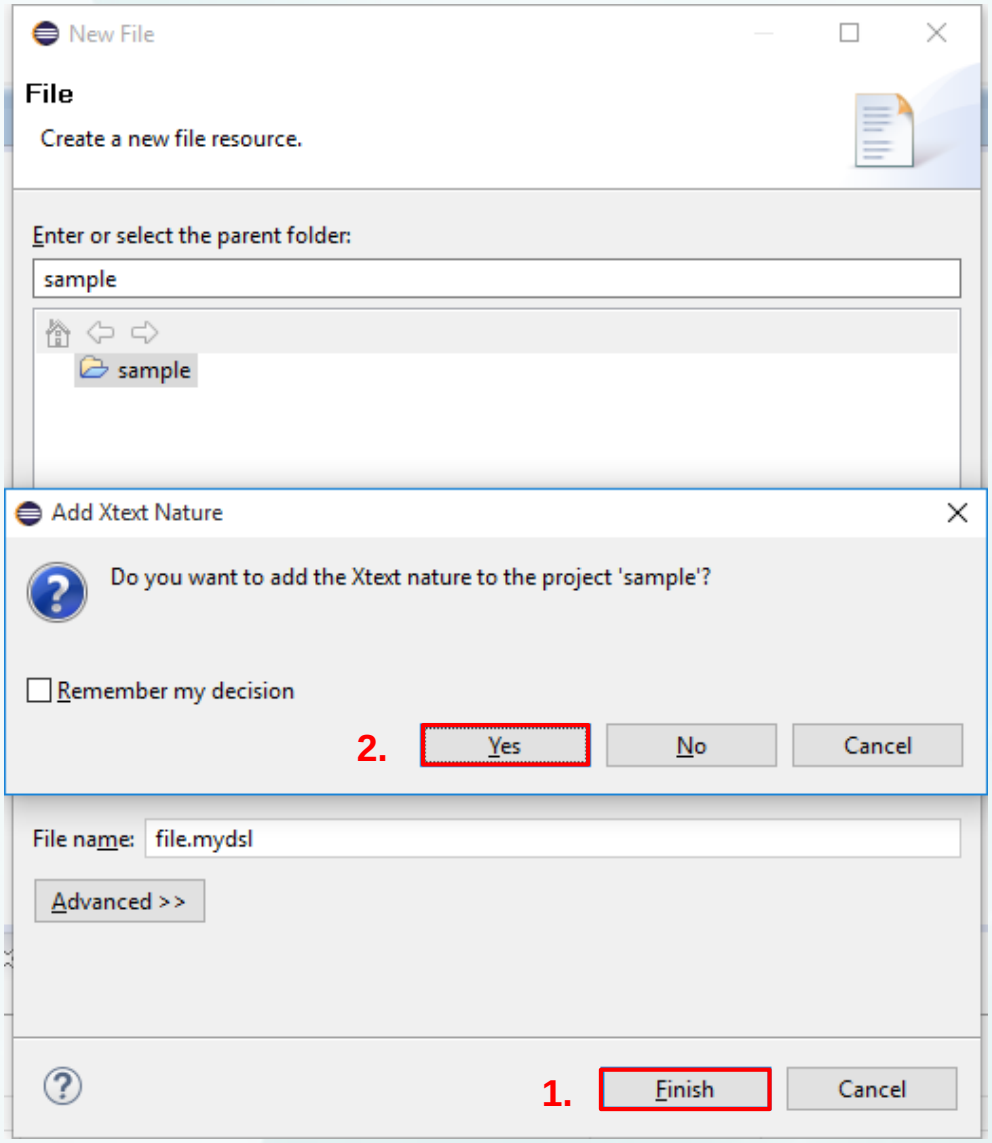
Creating a Project - First Method

10.



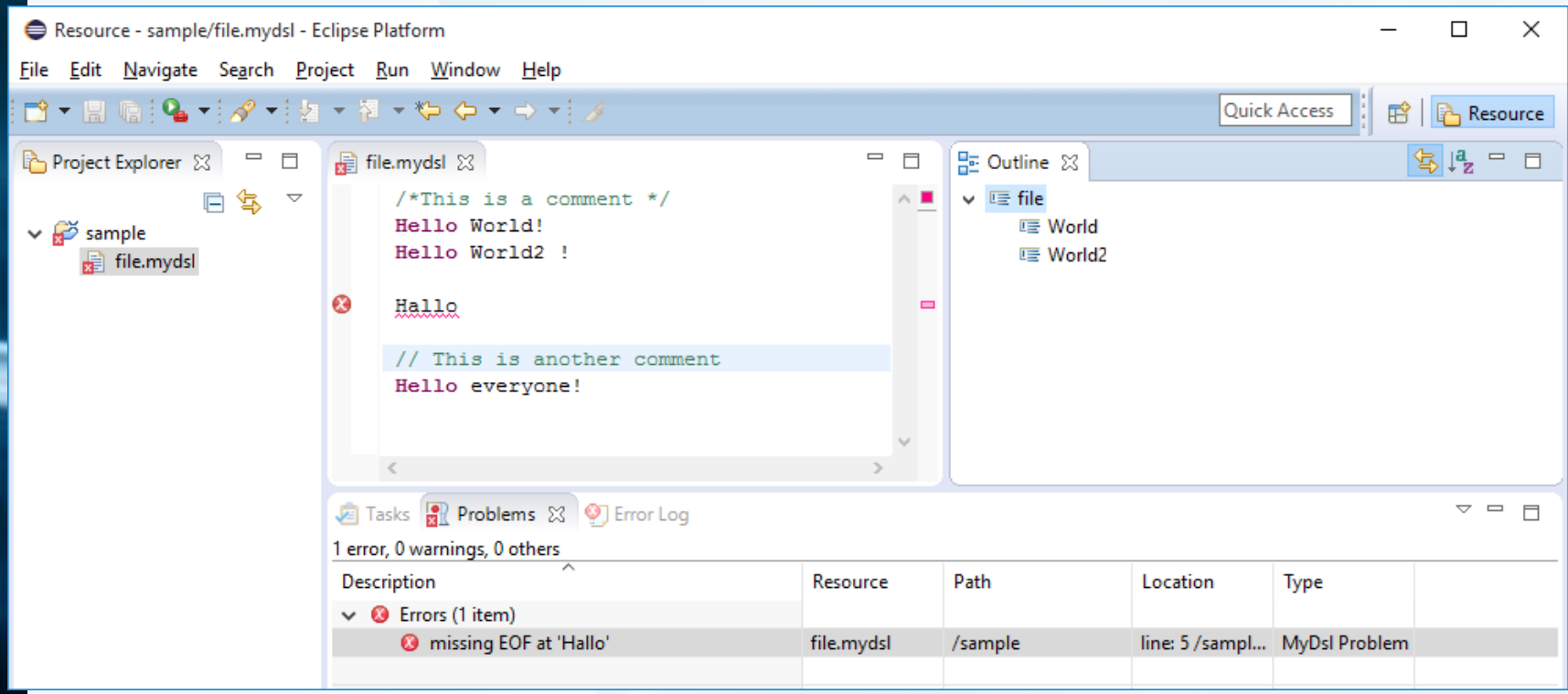
Creating a Project - First Method

11.



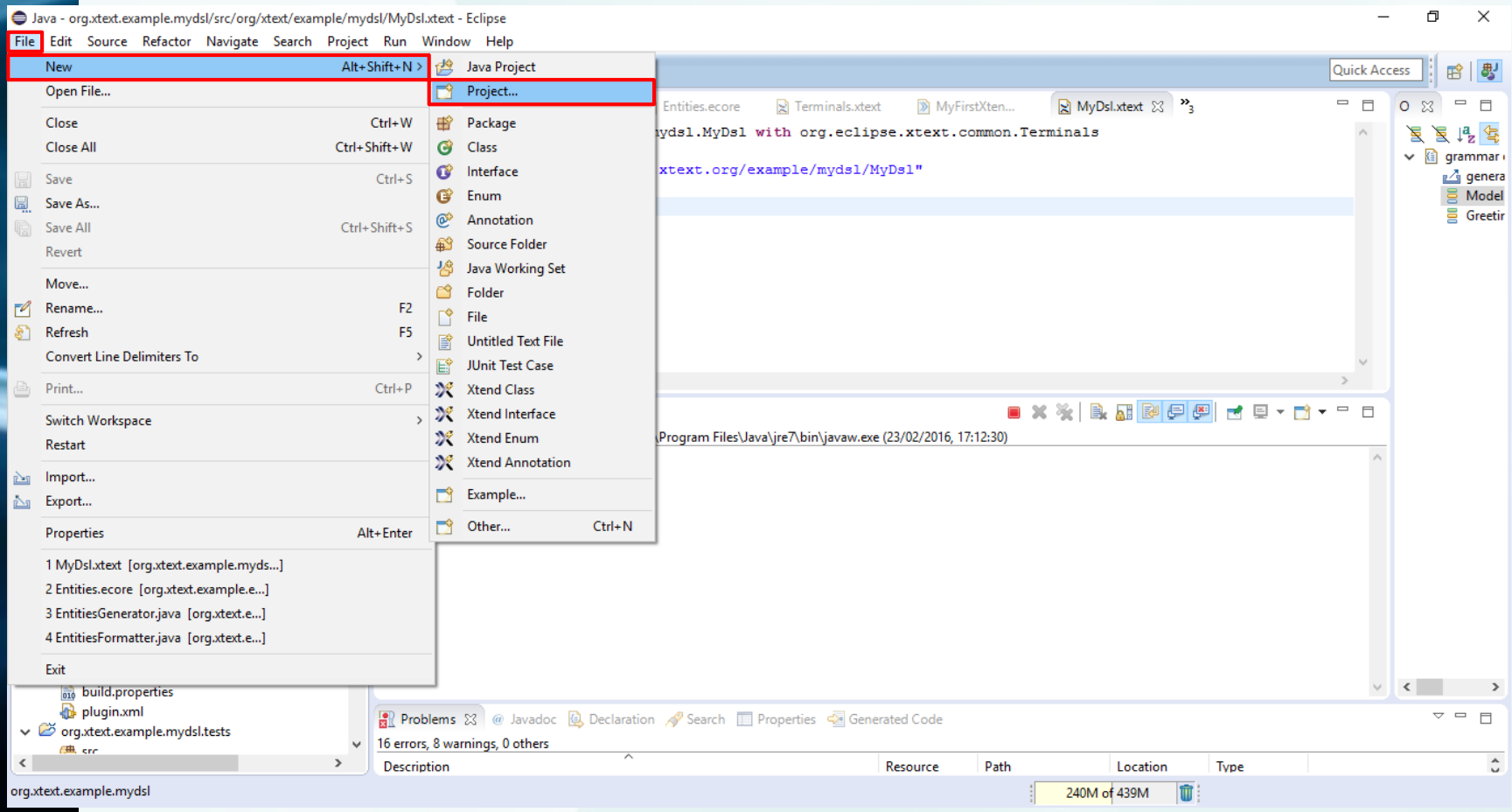
Creating a Project - First Method

12.



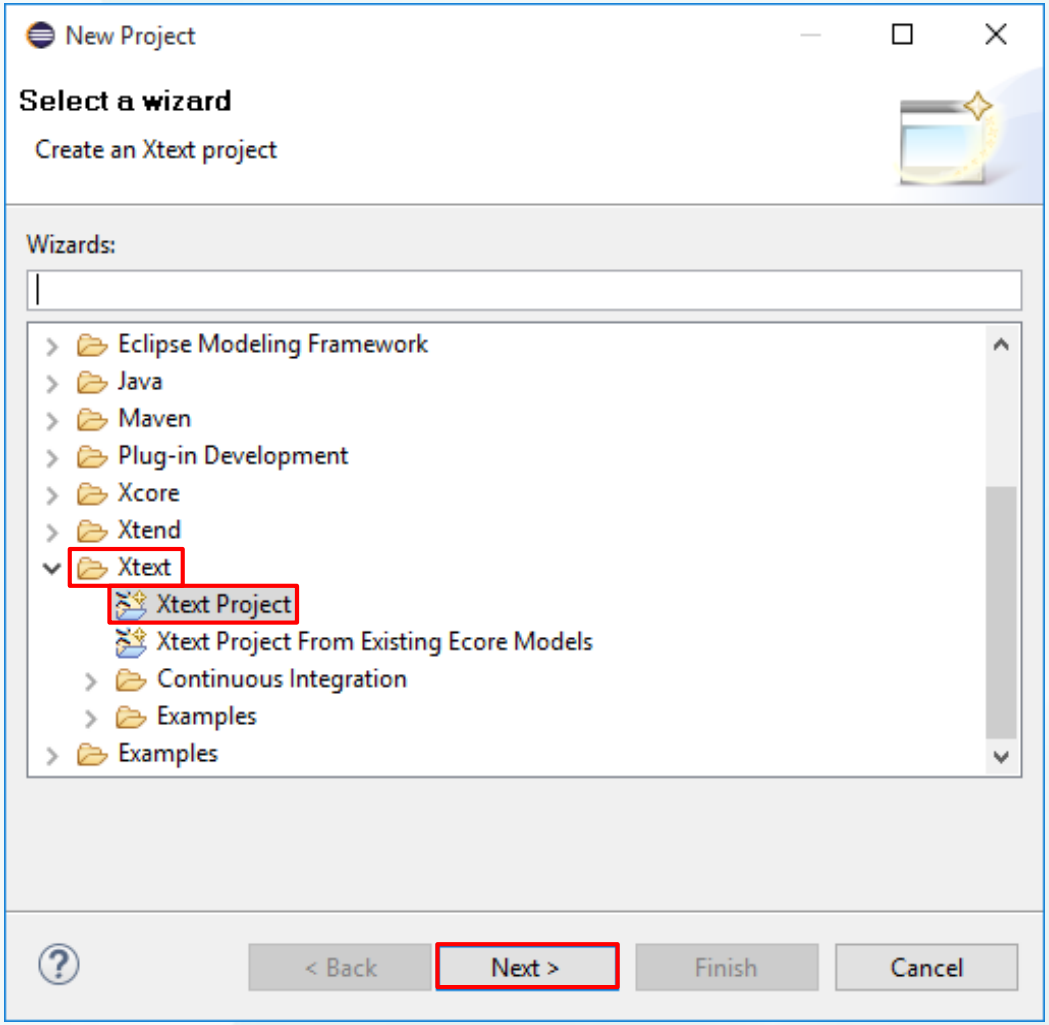
Creating a Project - Second Method

1.Click on on File -> New -> Project...



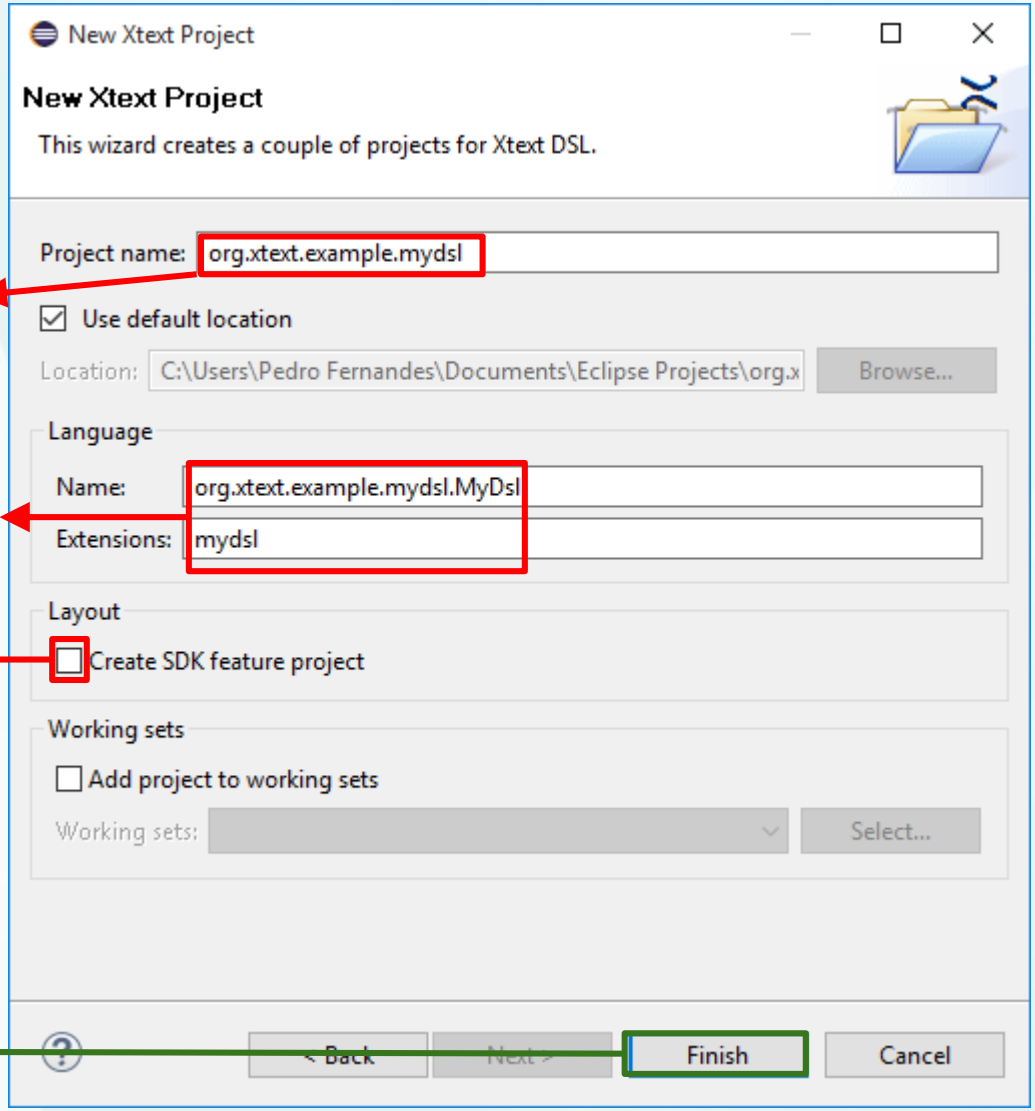
Creating a Project - Second Method

2.Choose XtextProject



Creating a Project - Second Method

3.



1º Change project name

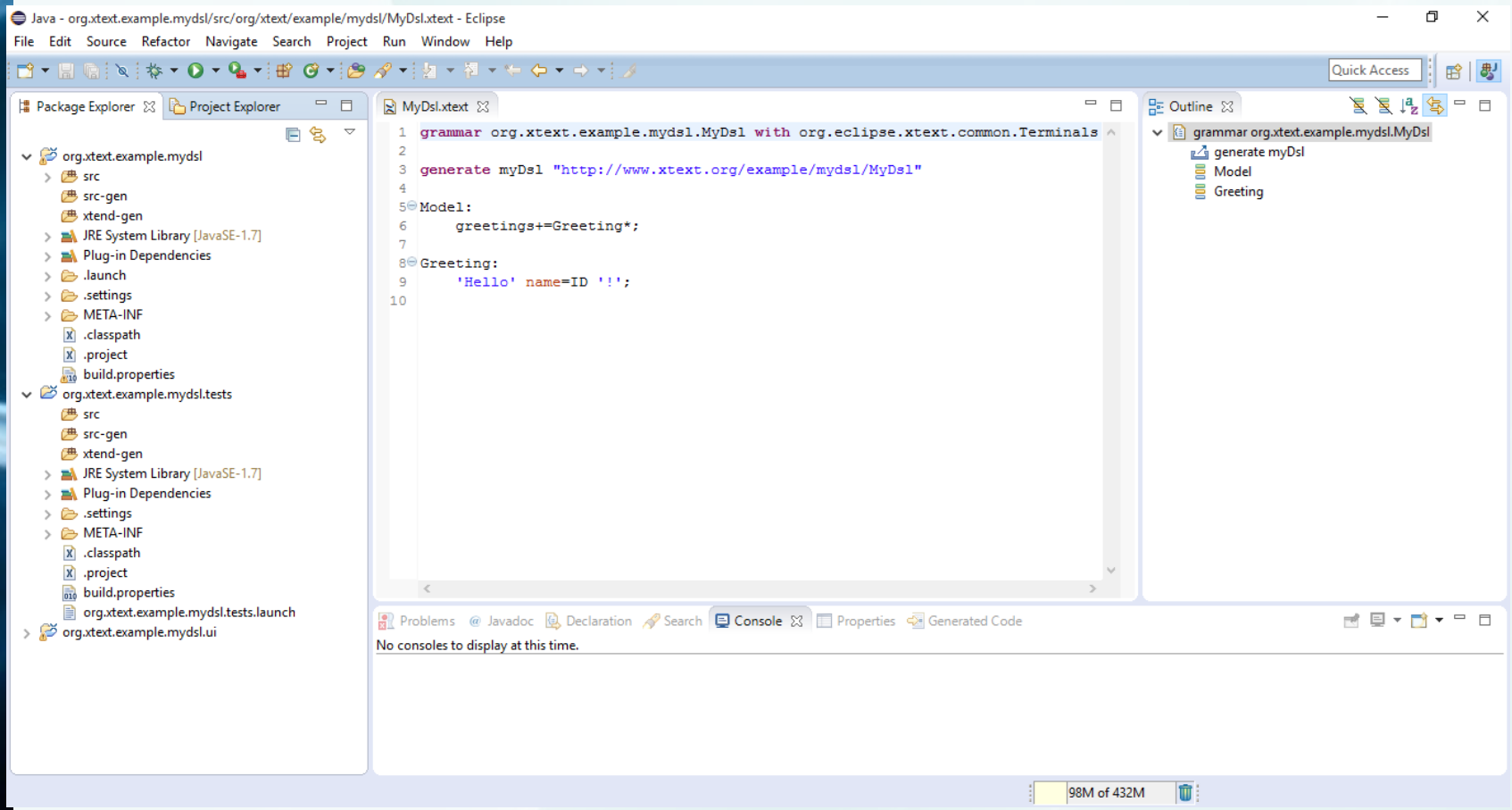
2º Change language and extensions name

Uncheck

3º Click button “Finish”

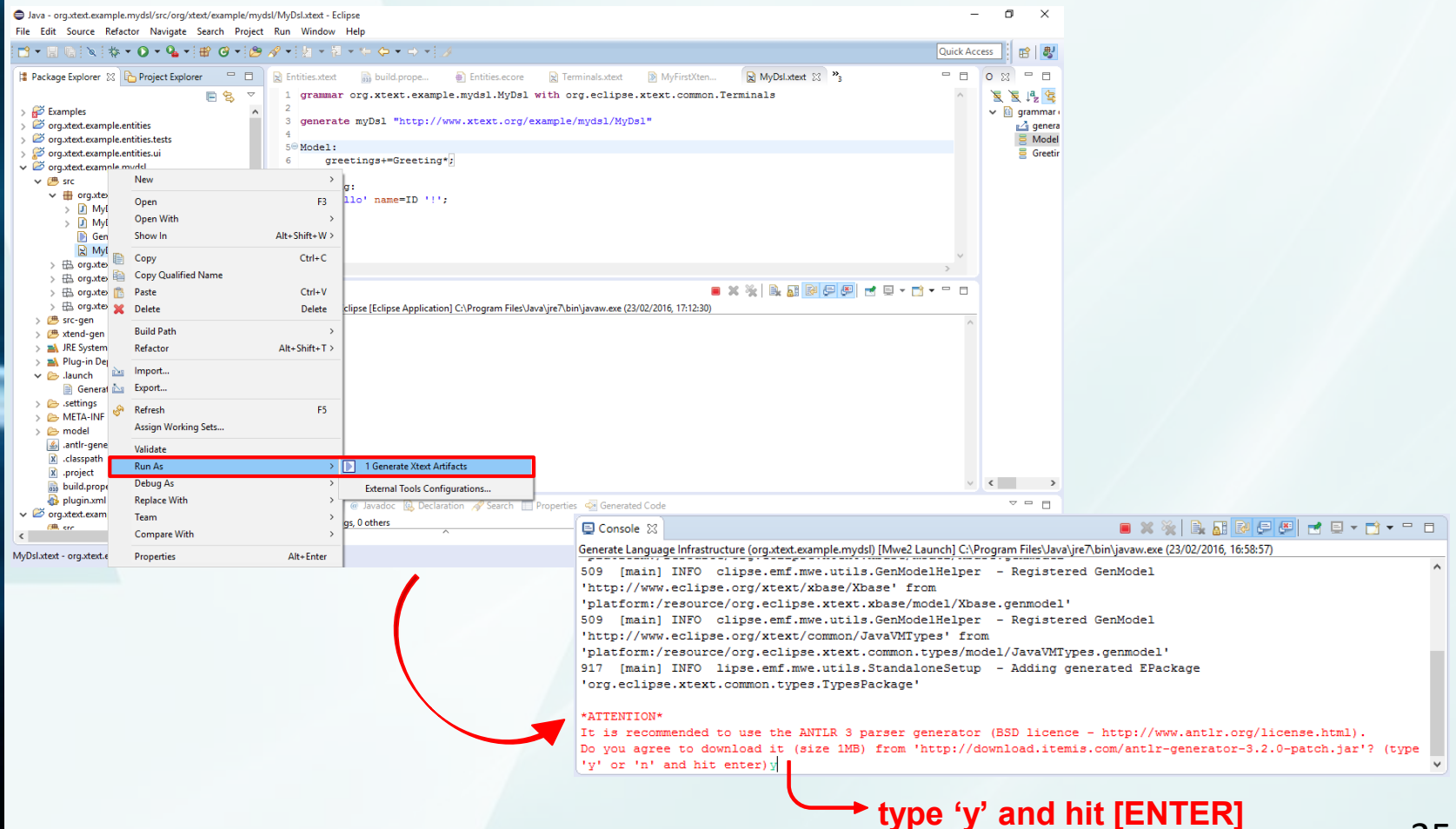
Creating a Project - Second Method

4.



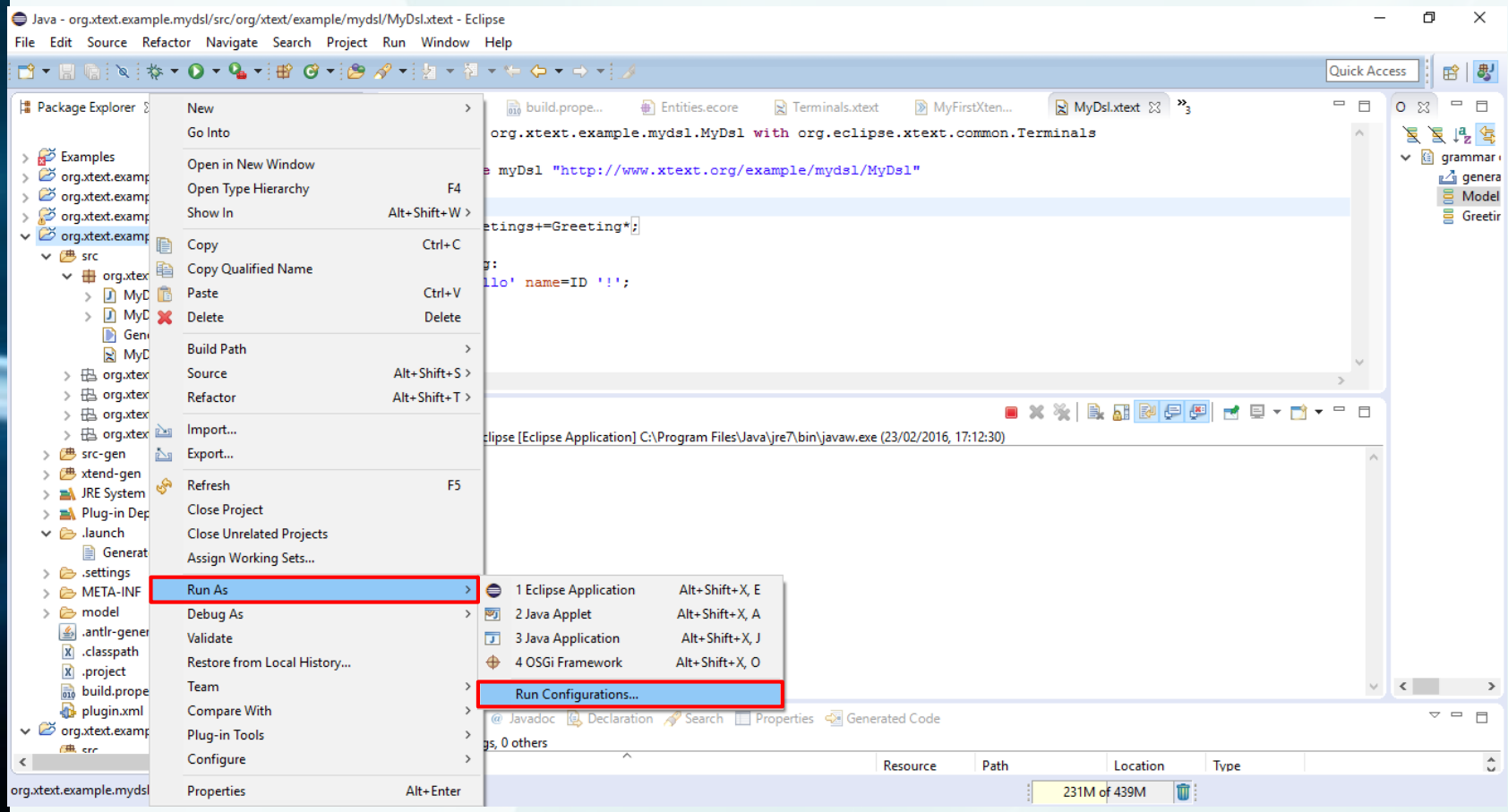
Creating a Project - Second Method

5.Right-click on ‘MyDsl.xtext’



Creating a Project - Second Method

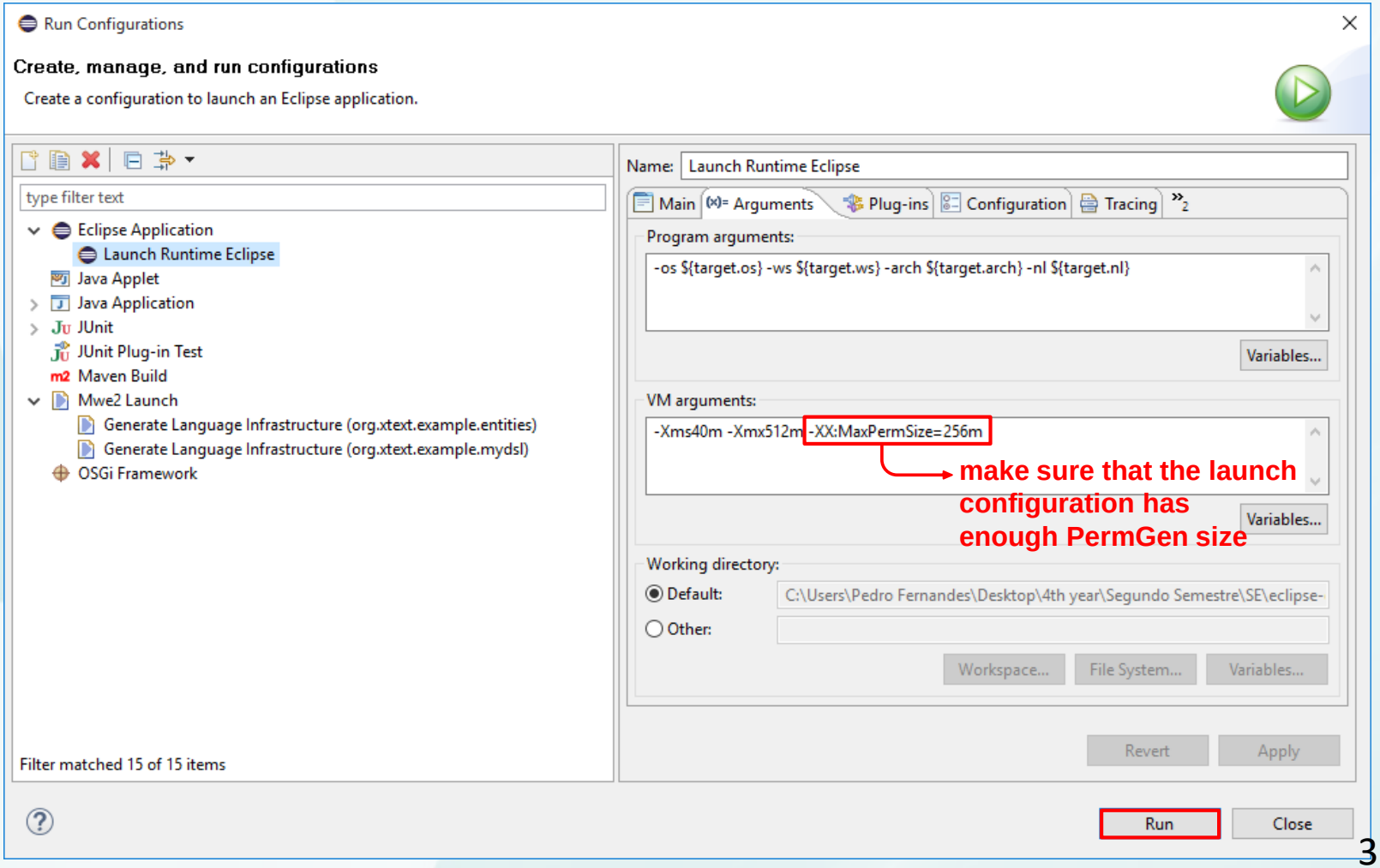
6.Right-click on ‘org.xtext.example.mydsl’



Creating a Project - Second Method

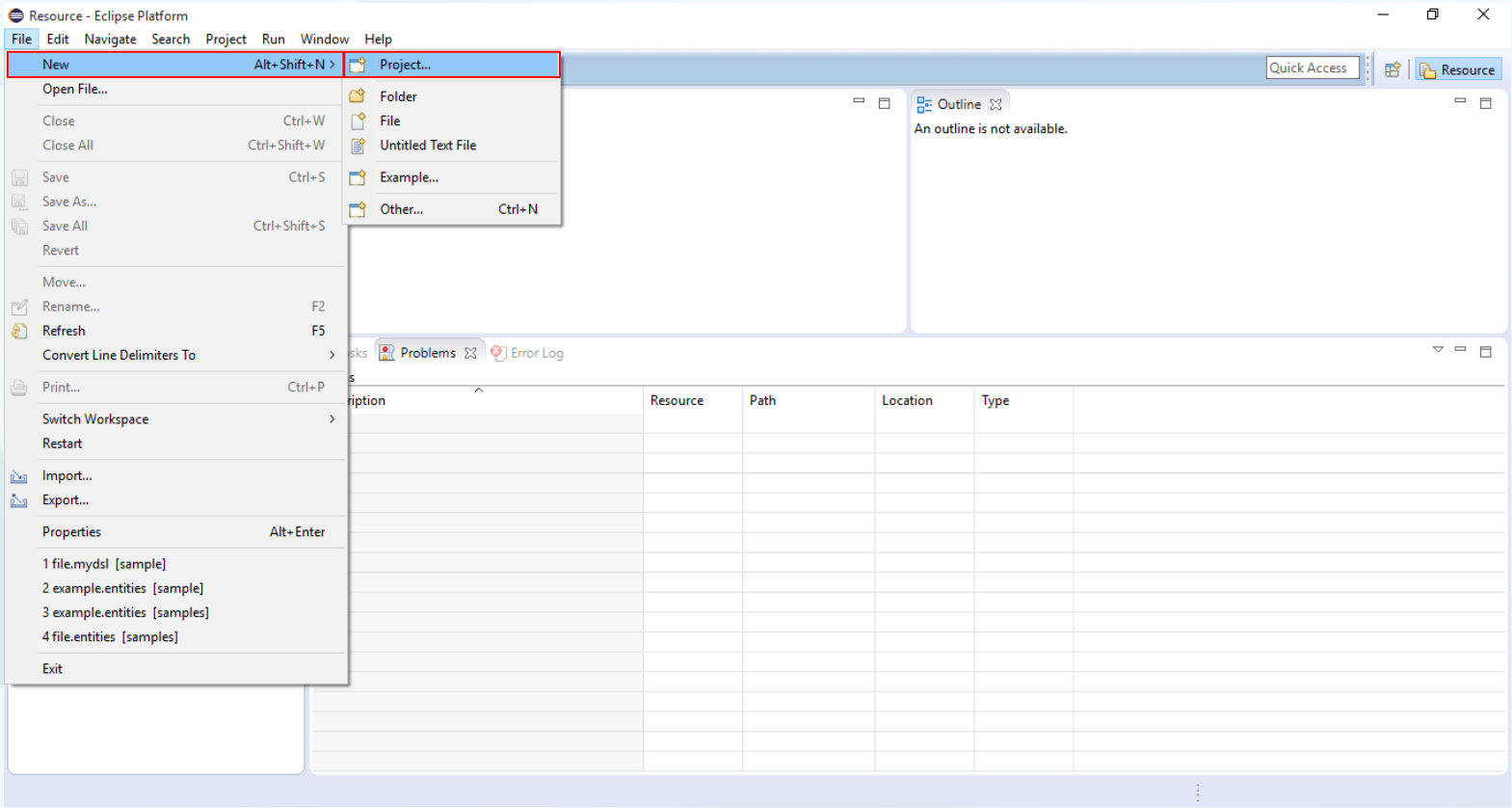
CENTROALGORITMI

7.



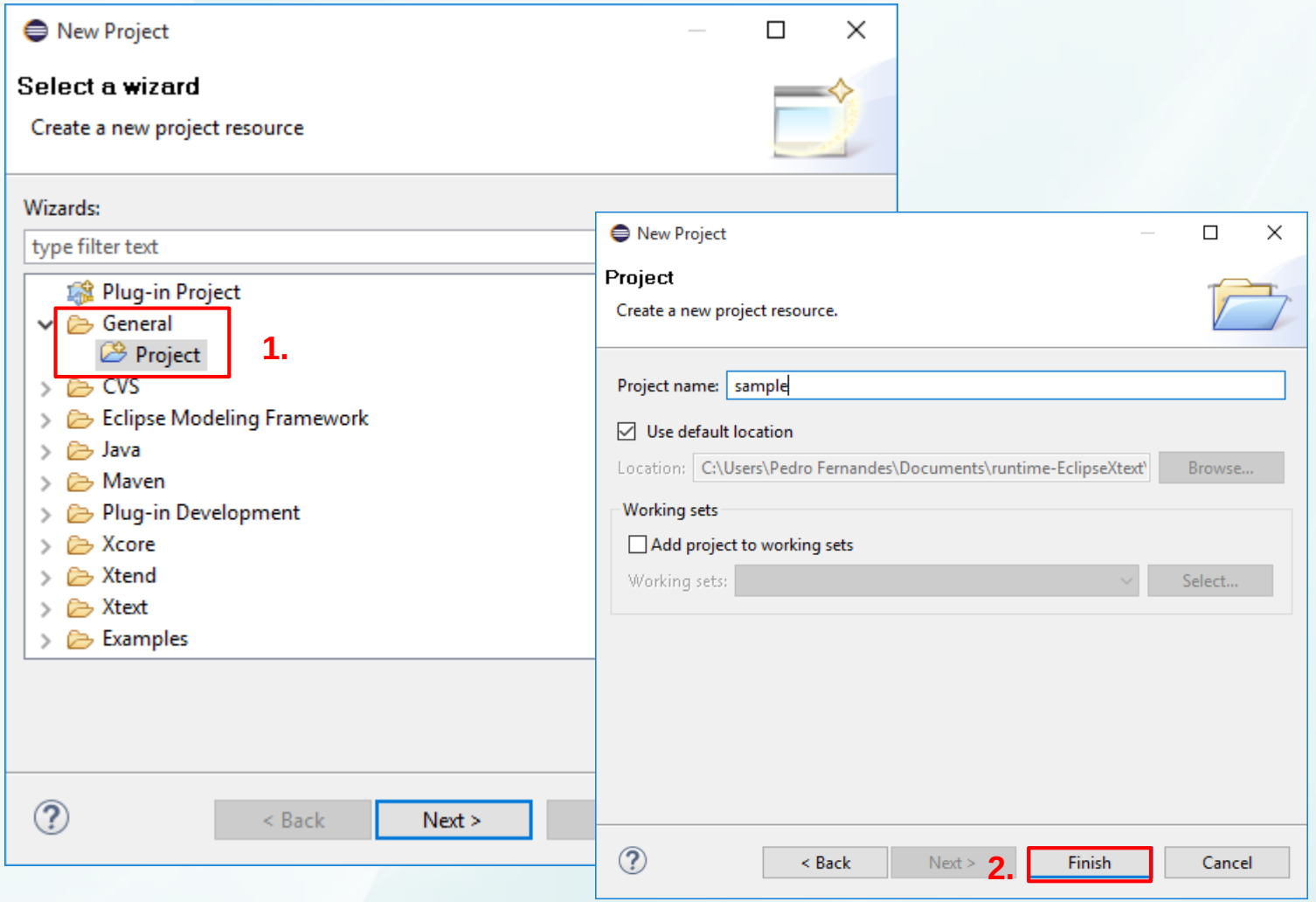
Creating a Project - Second Method

8. In the new Eclipse instance:



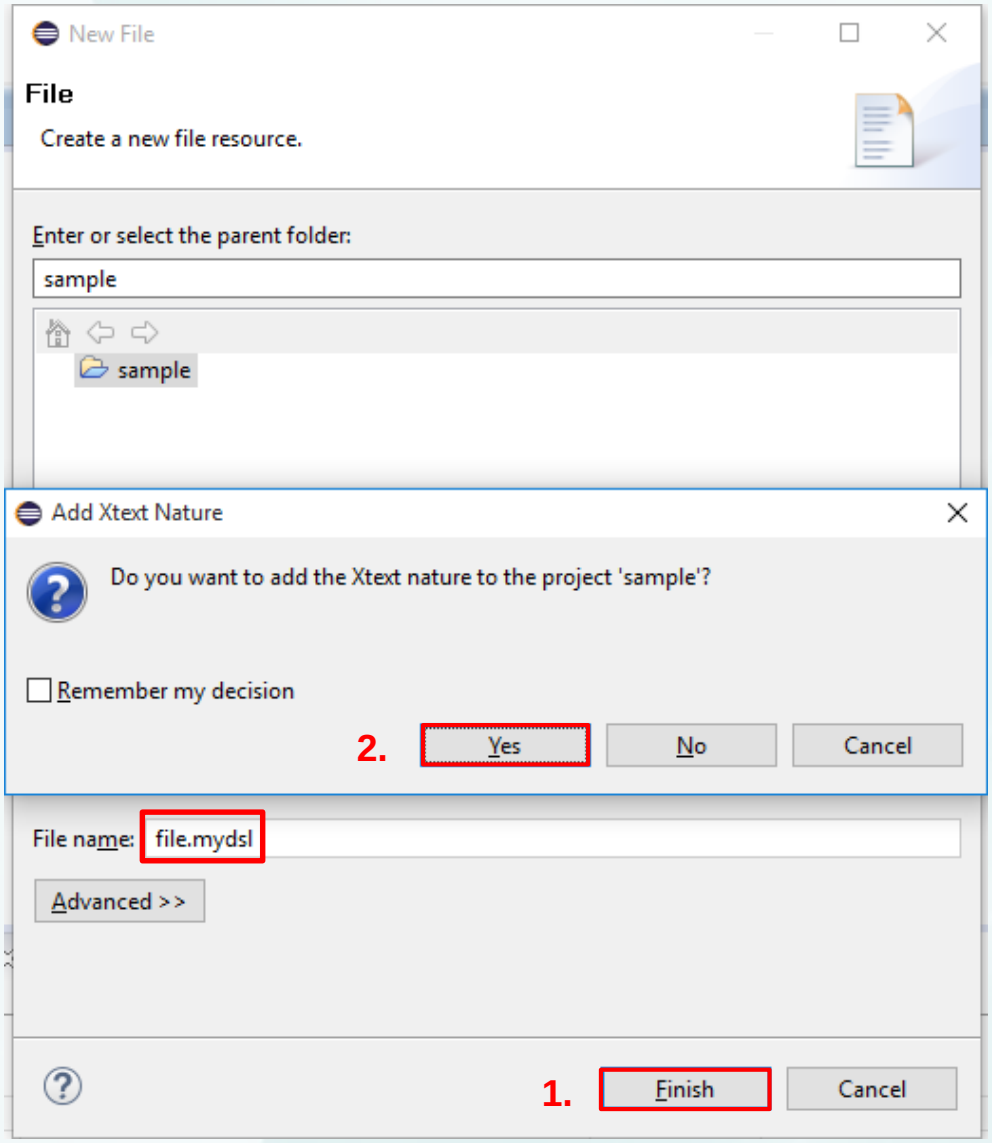
Creating a Project - Second Method

9.



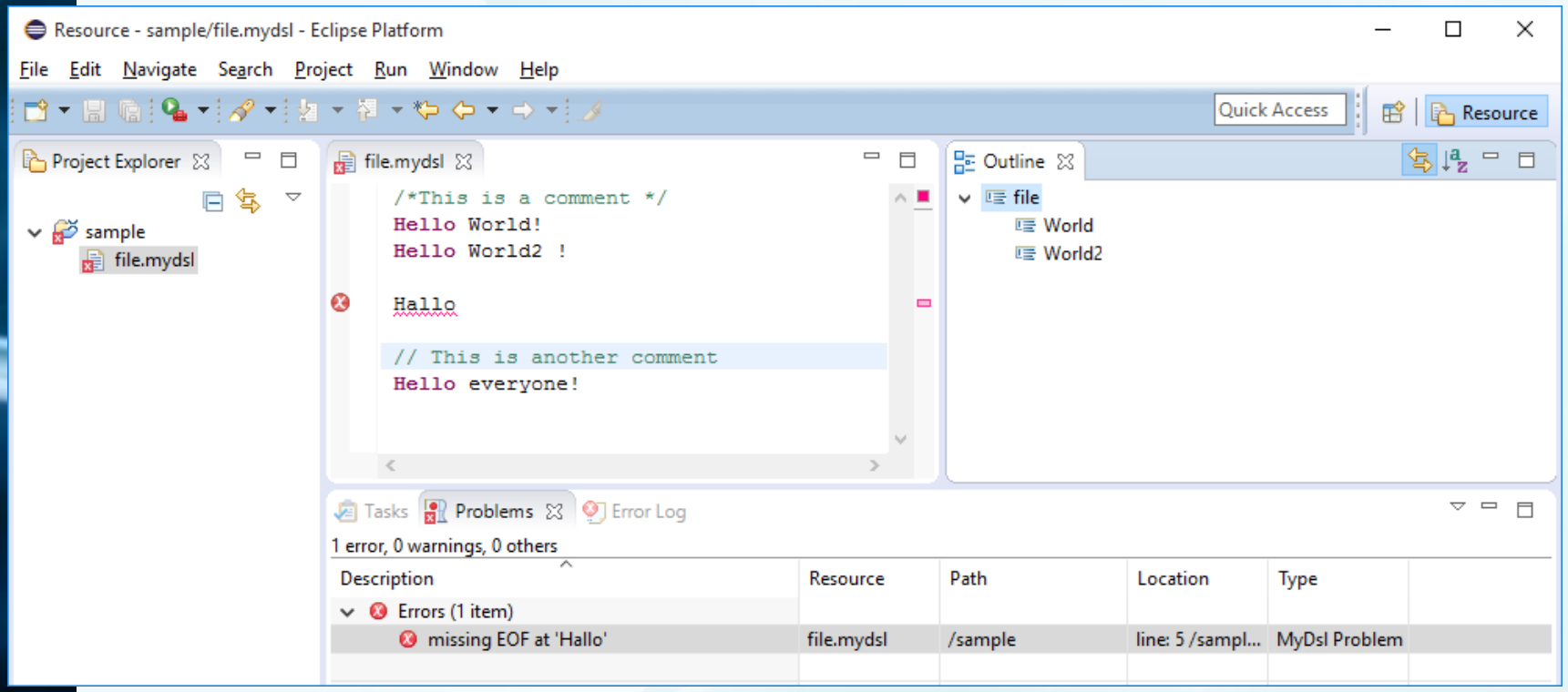
Creating a Project - Second Method

10.



Creating a Project - Second Method

11.



Creating the project

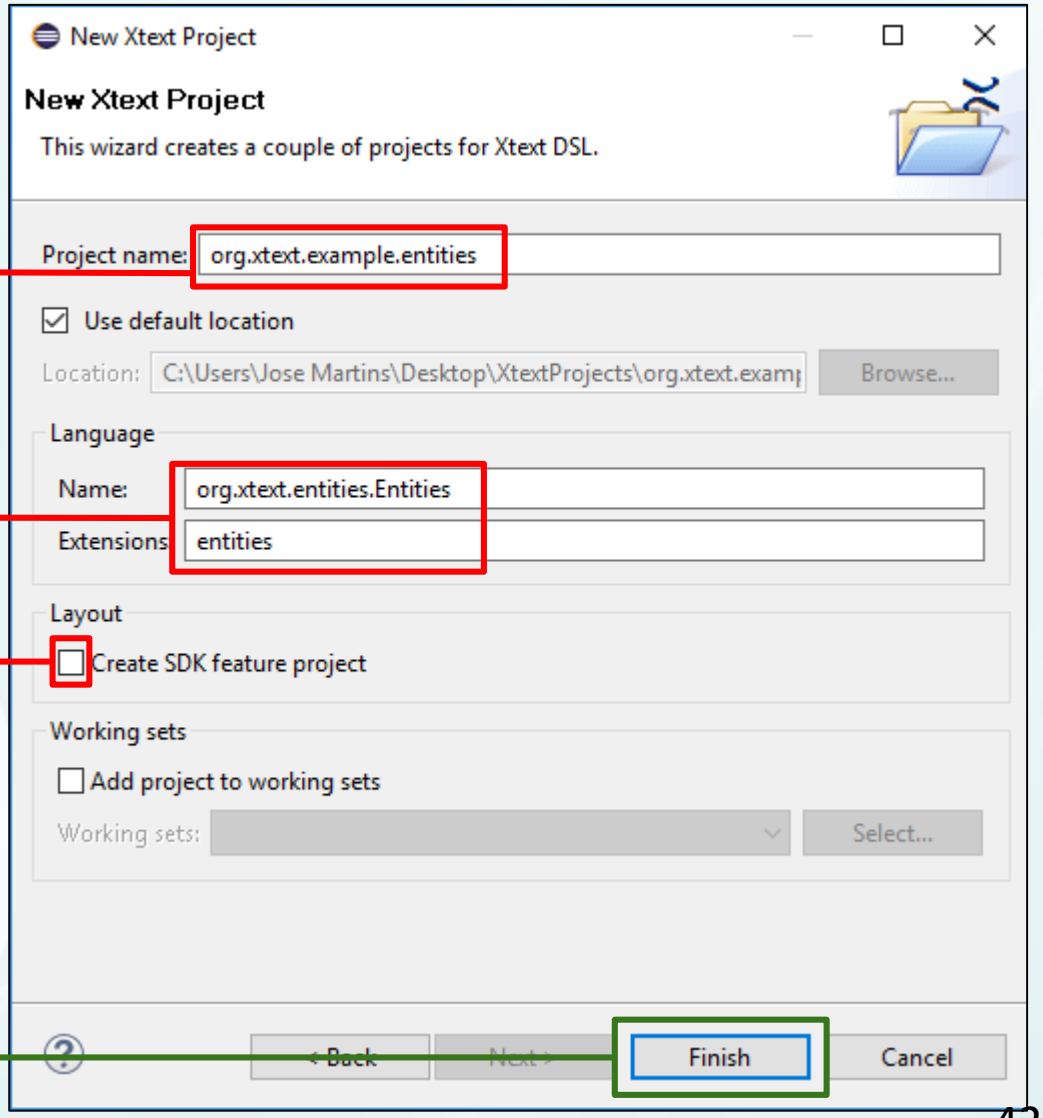
- **Creating a new Xtext project**

1º Change project name

2º Change language and extensions name

3º Uncheck SDK feature

4º Click button "Finish"



Xtext Projects

- Every DSL implemented in Xtext will have 3 projects:
  - **org.xtext.example.entities** contains the grammar definition;
  - **org.xtext.example. entities.tests** contains unit tests;
  - **org.xtext.example. entities.ui** contains features related to the UI.

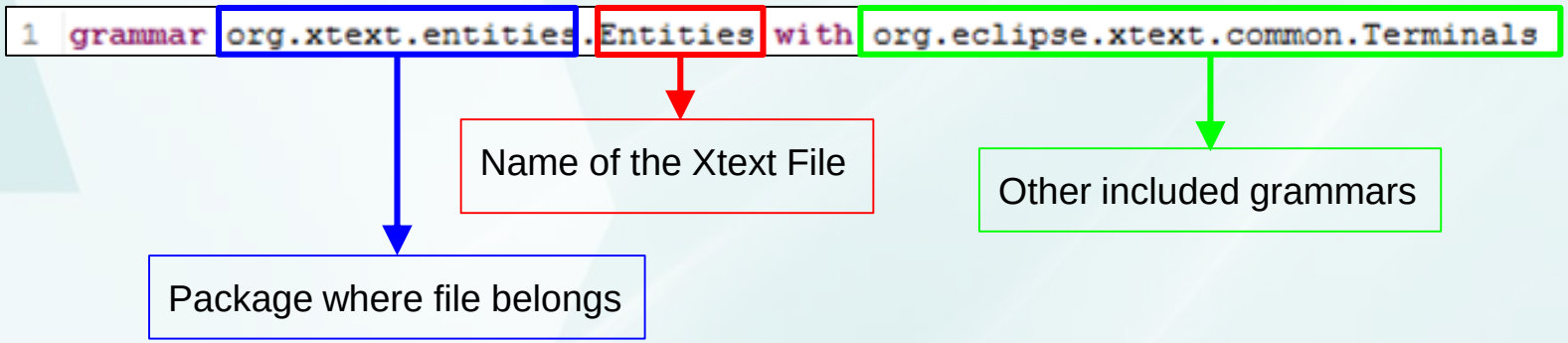
Creating a simple language

Entities.xtext

```
1  grammar org.xtext.entities.Entities with org.eclipse.xtext.common.Terminals
2
3  generate entities "http://www.xtext.org/entities/Entities"
4
5  Model:
6      entities += Entity*
7  ;
8
9  Entity:
10     'entity' name = ID ('extends' superType=[Entity])? '{'
11         attributes += Attribute*
12     '}'
13 ;
14
15 Attribute:
16     type=[Entity] array?=('[]')? name=ID ';'
17 ;
```

Creating a simple language

- Each Xtext grammar starts with a **header** that defines some properties of the grammar;
- The first part starting with the keyword "**grammar**" declares the name of the language;
- The second part starting with the keyword "**with**" declares other existing grammar to be reused.



Creating a simple language

- The second line starting with the keyword "**generate**" instructs the framework to infer the model from your grammar;

```
3 generate entities "http://www.xtext.org/entities/Entities"
```

- All the grammar rules are composed by a **name**, a **colon**, the **actual syntactic form** accepted by that rule and are terminated by a **semicolon.**

```
19 RuleName: //Rule Name
20     'keyword' feature=ID //Accepted syntactic form
21     /* ... other forms */
22 ;
```

Creating a simple language

Start Rule expression

```
5  Model:
6      entities += Entity*
7  ;
```

- Is the **first rule** in every grammar that defines where the parser starts and the type of the root element of the model of the DSL, that is, of the Abstract Syntax Tree (AST);

- This collection is stored in a feature called entities.

Creating a simple language

Entity's Rule expression

```
9  Entity:
10     'entity' name = ID ('extends' superType=[Entity])? '{'
11         attributes += Attribute*
12     '}'
13 ;
```

- The shape of each Entity element is expressed in its own rule. This rule specifies that an Entity starts with the literal entity, followed by the name of the entity (which, in turn, must be an identifier). An entity definition has a body which is surrounded by curly braces;

- The body may then contain any number (zero or more) of Attributes, in this case.

Creating a simple language

Attribute's Rule expression

```
15 Attribute:
16     type=[Entity] array?='[]'? name=ID ';'
17 ;
```

- The shape of each Attribute element is expressed in its own rule. This rule specifies that an Attribute requires:
  - An Entity name (cross-referenced) that will be stored in the type feature;
  - The array feature which is optional;
  - A name for the attribute;
  - Must be terminated with ' ; '.

Creating a simple language


```
9  Entity:
10     'entity' name = ID ('extends' superType=[Entity])? '{'
11         attributes += Attribute*
12     '}'
13 ;
14
15 Attribute:
16     type=[Entity] array?='[]'? name=ID ';'
17 ;
```

- In Xtext, string literals define **keywords** of the DSL. In this rules can be seen different keywords: ‘entity’, ‘extends’, ‘{‘ and ‘}’, ‘[]’ and ‘;’.
- The **ID** following the initial keyword is a terminal rule inherited from the grammar *Terminals*, which are normally in upper case.
- The ‘( )?’ operator declares an **optional part**. Therefore, in the Entity rule, after the ID, you can write the keyword 'extends' and the name of an existing Entity between square brackets that will be ***cross-referenced***.

Creating a simple language


```
9  Entity:
10     'entity' name = ID ('extends' superType=[Entity])? '{'
11         attributes += Attribute*
12     '}'
13 ;
14
15 Attribute:
16     type=[Entity] array?=('[]')? name=ID ';'
17 ;
```

- **Assignments** are used to assign the consumed information to a feature of the currently produced object, by the ECore;
- The **left hand side** refers to a feature name of the current object instance of the EMF class, corresponding to the rule, and its type will be inferred from the right side value;
- The **right hand side** can be a rule call, a keyword, a cross-reference or an alternative comprised by the former.

Assignments Operators	
=	features of one element
+=	list of features
?=	boolean feature

Terminal rules

- Also referred to as **token rules** or **lexer rules**, defining the regular expressions for tokens in the lexical stage;
- Have the same syntactic form as a grammar parser rule but are initiated by the keyword "**terminal**";
- The order of the terminal rules is crucial for the grammar, as they may shadow each other;
- **Terminals grammar** is part of the Xtext libraries and defines common grammar rules for tokens like string literals, numbers or white spaces.

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

EBNF

- Terminal rules are described using **Extended Backus-Naur Form-Like (EBNF)** expressions.

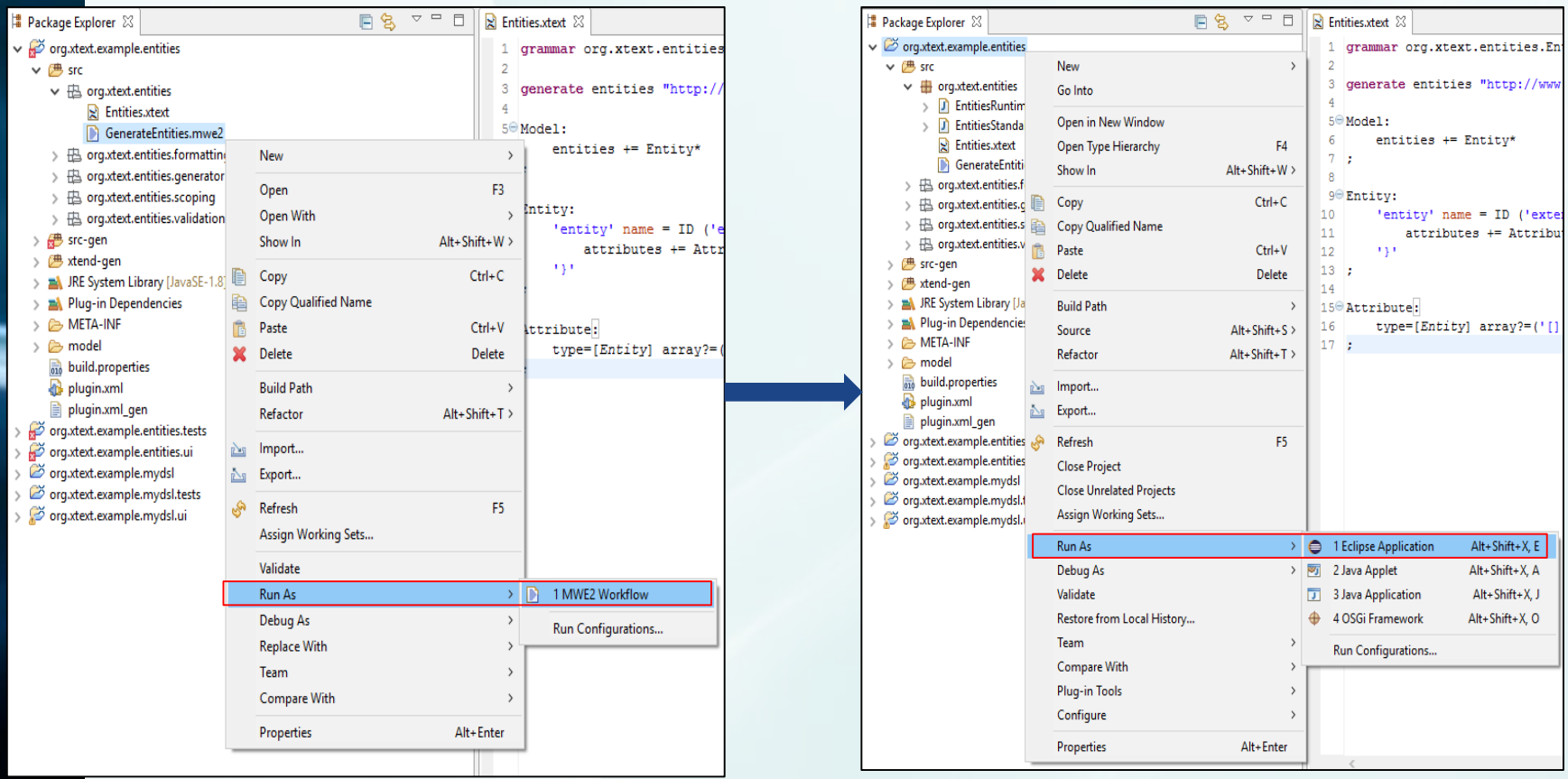
Cardinality	Operator	Examples	Results
Exactly One	the default, no operator	“Hello”	Hello
One or none	?	(ab)?	ab **or** __
Zero or more	*	(ab)*	__ **or** ab **or** (ababab…)
One or more	+	(ab)+	ab **or** ababab...

EBNF

	Definition	Expression	Results
Keywords	terminal rule literals	‘^’	^
Character Ranges	declare a character range	(‘0’...’9’)	0,1,2,3,4,5,6,7,8,9
Wildcard	allows any character	‘f’ . ‘o’	foo, f0o, f_o
Until Token	everything consumed between tokens	‘/*’ -> ‘*/’	/* int a = 0 */
Negated Token	invert the expression definition	’#’ (!’#’)* ’#’	#4#
Rule Calls	rules can refer to other rules	DOUBLE : INT ’.’ INT	2.3
Groups	group tokens	’0x’ (’0’..’7’) (’0’..’9’|’A’..’F’);	0x3F
EOF - End of File	describe the end of the input stream	EOF	”” (!””)* EOF;

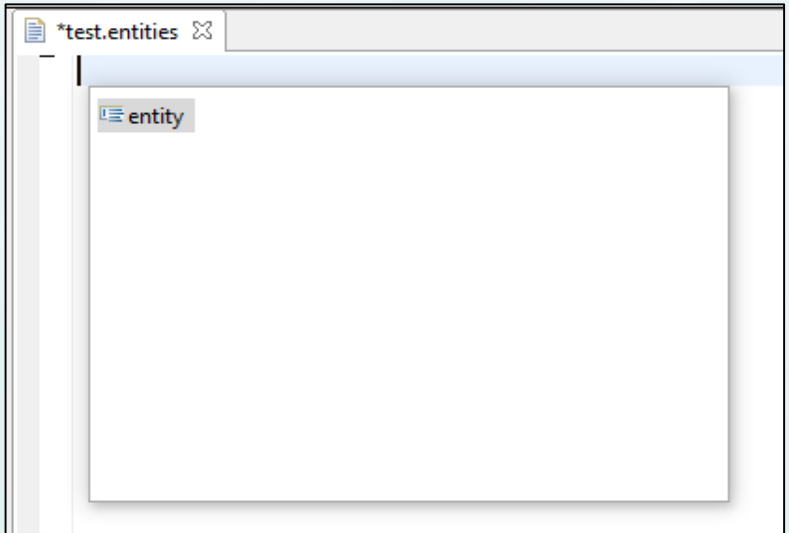
Try Editor

- Alternatively to what was explained previously, a way to **run the editor** is:

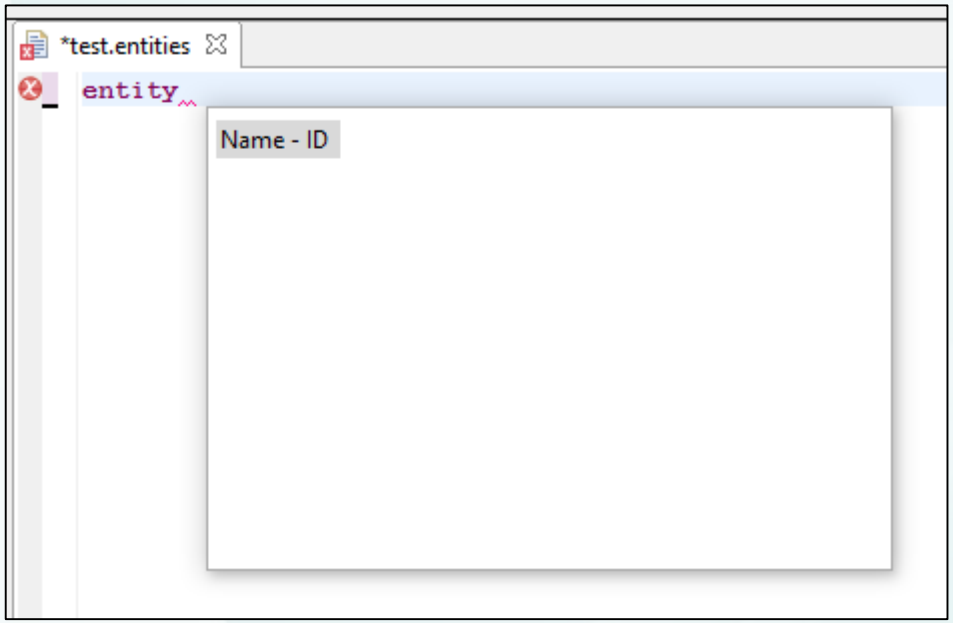


Try Editor

- Initially the editor is empty which means an empty program is a valid Entities program, because the *Model* rule was defined with the operator *;
- After, while accessing the **content assist (*Ctrl + Space bar*)** an *entity* keyword option will be provided.



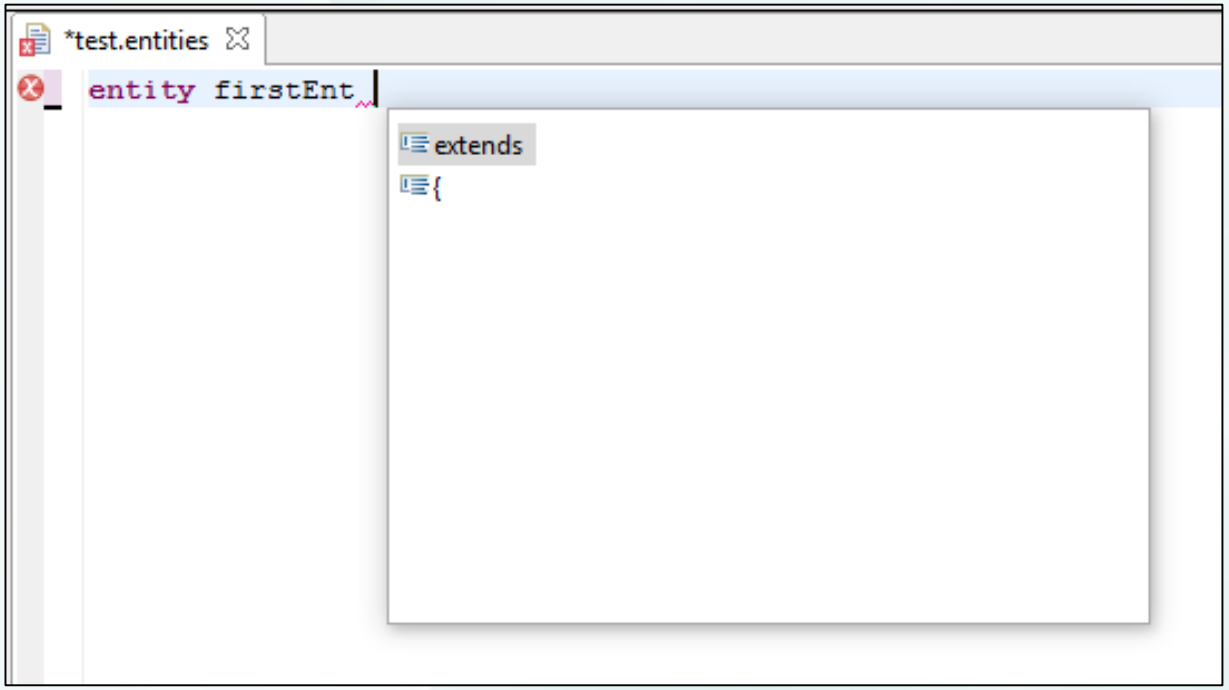
Try Editor



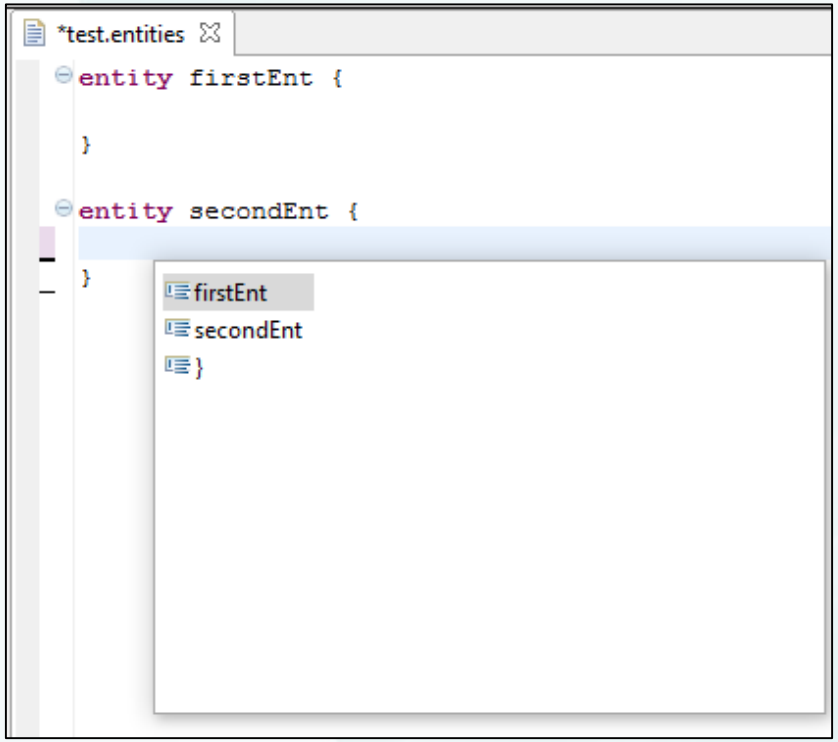
- After inserting *entity* the error occurs because the entity is incomplete, so if the content assistance is accessed it will provide the hint for the next identifier.

Try Editor

- Next the content assistance provides two options. If the curly bracket is chosen the editor closes it automatically.



Try Editor



- Finally, after the curly brackets are closed the error will disappear, so the current program is correct.

Xtext Generator

- Xtext uses **MWE2 DSL** to configure the generation of its artifacts like the UI editor; The .mwe2 file can be tweaked to support additional features;
- It will be derived an **ANTLR** specification from the Xtext grammar, so that the AST can be created during the parsing (the classes for the nodes will be generated using EMF);
- Generator **must be run after every modification** to the grammar and the Eclipse instance where the editor is being tested **must be restarted**.

Eclipse Modeling Framework

- Provides code generation facilities for building tools and applications based on structured data models;
- Xtext automatically infers the **EMF metamodel** for the target language;
- The language metamodel is defined in the **Ecore format**, which is an implementation of a subset of UML, provided as a Java API;
- The model derived from the metamodel will correspond to the **AST** from the parsed program.

Eclipse Modeling Framework

- For **each rule** in the grammar, an **EMF interface** and **class will be created** with a **field for each feature** in the rule (together with accessors), which will be stored in the project's src-gen folder;
- Regarding the Entity rule, the corresponding implementation of the Java class will be:

```
public interface Entity extends EObject
{
  String getName();
  void setName(String value);
  Entity getSuperType();
  void setSuperType(Entity value);
  EList<Attribute> getAttributes();
} // Entity
```

Eclipse Modeling Framework

EntityImpl.java | Entity.java

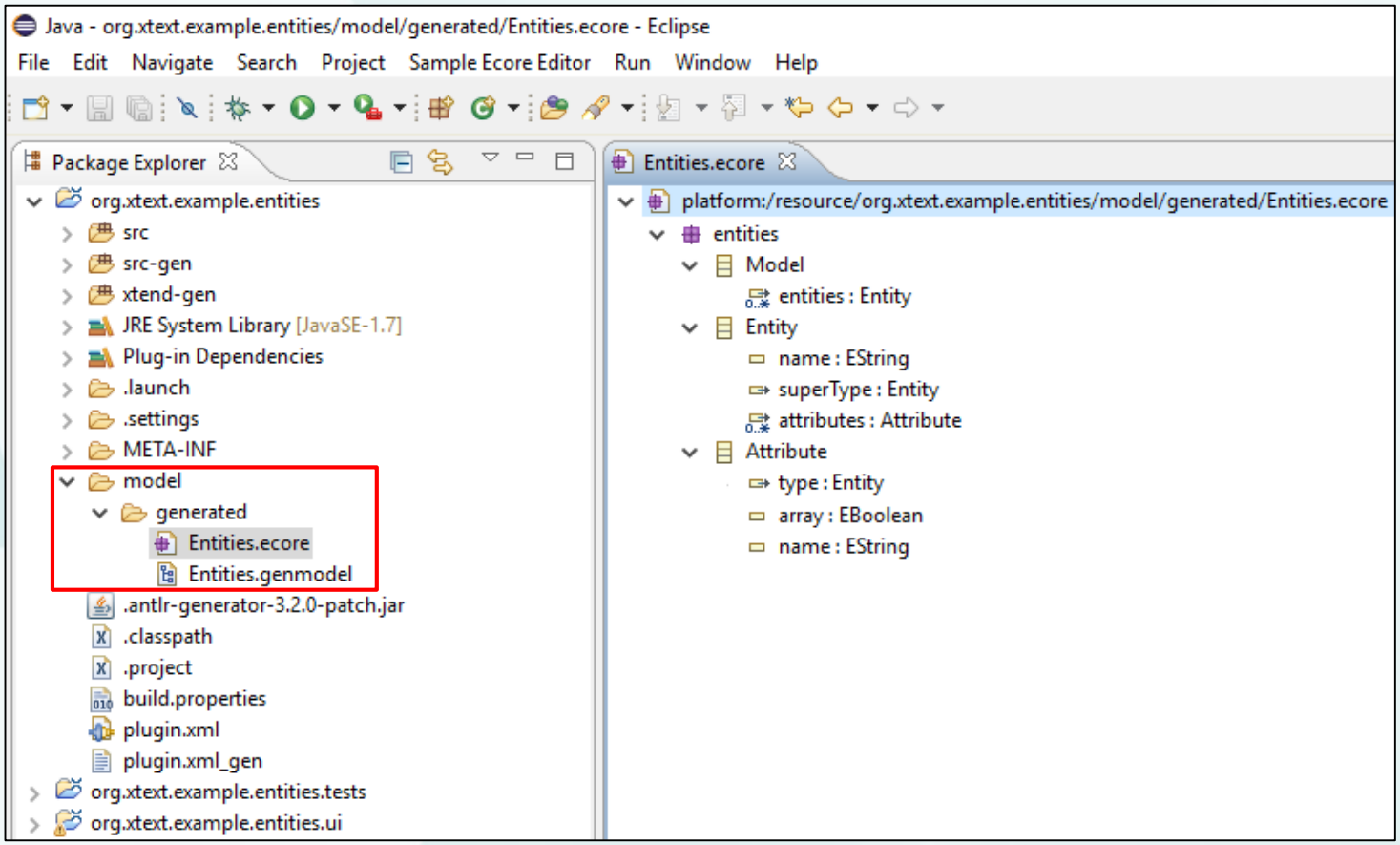
```
1  package org.xtext.entities.entities.impl;
2
3  import java.util.Collection;
22
23 public class EntityImpl extends MinimalEObjectImpl.Container implements Entity
24 {
25
26   protected static final String NAME_EDEFAULT = null;
27   protected String name = NAME_EDEFAULT;
28   protected Entity superType;
29   protected EList<Attribute> attributes;
30
31   protected EntityImpl()
32   {
33     super();
34   }
35
36   @Override
37   protected EClass eStaticClass()
38   {
39     return EntitiesPackage.Literals.ENTITY;
40   }
41
42   public String getName()
43   {
44     return name;
45   }
46
47   public void setName(String newName)
48   {
49     String oldName = name;
50     name = newName;
51     if (eNotificationRequired())
52       eNotify(new ENotificationImpl(this, Notification.SET, EntitiesPackage.ENTITY__NAME, oldName, name));
53   }
54
55   public Entity getSuperType()
56   {
57     if (superType != null && superType.eIsProxy())
58     {
```

Eclipse Modeling Framework

- Xtext creates:
  - EPackage for each ‘generate’;
  - EClass for each rule;
  - EEnum for each enum rule;
  - EDataType for each terminal or data type rule.
- While parsing for each rule feature Xtext creates:
  - EAttribute for each assignment with a value corresponding to a terminal rule;
  - EReference when on the right side of the assignment a cross-reference is present.
- All of this classes are part of ECore.

Eclipse Modeling Framework

- In the EMF Ecore editor, the file Entities.ecore is where it can be observed the generated metamodel for the Entities DSL;



Eclipse Modeling Framework

- The **model of the DSL** programs are generated as instances of these generated EMF Java classes, that follow some conventions, such as:
  - Instancing an EMF class must be done through a static factory;
  - Initialization of fields can be done through accessors.
- So it's possible to programmatically manipulate the model of a program through Java.

Eclipse Modeling Framework

- Entities Java model that corresponds to an Entities DSL program:

```
import org.example.entities.entities.Attribute;
import org.example.entities.entities.EntitiesFactory;
import org.example.entities.entities.Entity;
import org.example.entities.entities.Model;

public class EntitiesEMFExample {
    public static void main(String[] args) {
        EntitiesFactory factory = EntitiesFactory.eINSTANCE;

        Entity superEntity = factory.createEntity();
        superEntity.setName("MySuperEntity");

        Entity entity = factory.createEntity();
        entity.setName("MyEntity");
        entity.setSuperType(superEntity);

        Attribute attribute = factory.createAttribute();
        attribute.setName("myattribute");
        attribute.setArray(false);
        attribute.setType(superEntity);

        entity.getAttributes().add(attribute);

        Model model = factory.createModel();
        model.getEntities().add(superEntity);
        model.getEntities().add(entity);
    }
}
```

```
entity MySuperEntity {

}

entity MyEntity extends MySuperEntity {

    MySuperEntity myattribute;

}
```

No direct mapping in the code. It corresponds to the start symbol of the grammar.

Eclipse Modeling Framework

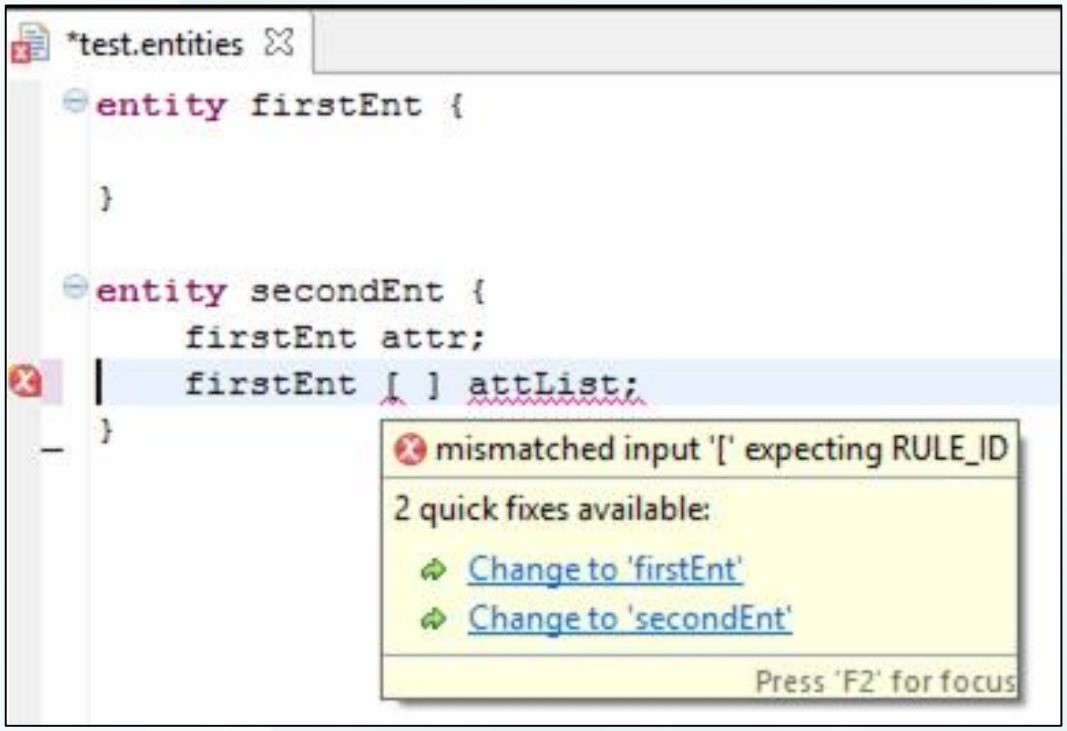

```
Model {
    cref Entity entities [
        0: Entity {
            attr EString name 'MySuperEntity'
        }
        1: Entity {
            attr EString name 'MyEntity'
            ref Entity superType ref: Entity@//@entities.0
            cref Attribute attributes [
                0: Attribute {
                    cref AttributeType type AttributeType {
                        cref ElementType elementType EntityType {
                            ref Entity entity ref: Entity@//@entities.0
                        }
                    }
                    attr EString name 'myattribute'
                }
            ]
        }
    ]
}
```

Improvements to the DSL

- In order to **avoid some usual syntax errors**, modifications to the grammar must be performed;
- It will be necessary to make some **adaptations in the Entities grammar** to give support to wanted statements;
- For example, **white spaces are not supported** by this grammar and should be irrelevant in our DSL.

Improvements to the DSL

- Before modifying the Attribute rule, the error in the editor will appear:



Improvements to the DSL

- After the Attribute rule adaptation, the error in the editor that appeared before is now solved:

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

```
entity firstEnt {

}

entity secondEnt{
    firstEnt attr;
    firstEnt [ ] attList;
}
```

Improvements to the DSL

- The array specification in our DSL can be refined in order to make possible the attribution of length:

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

- There is no rule defining INT in our grammar. This rule is inherited from the grammar *Terminals*;
- INT requires an integer literal, thus the length feature in our model will have an integer type as well, EInt in ECore.

Improvements to the DSL

- After the last Attribute rule change, we are now able to specify the attribute array size and perform the attribute definition, as illustrated below:

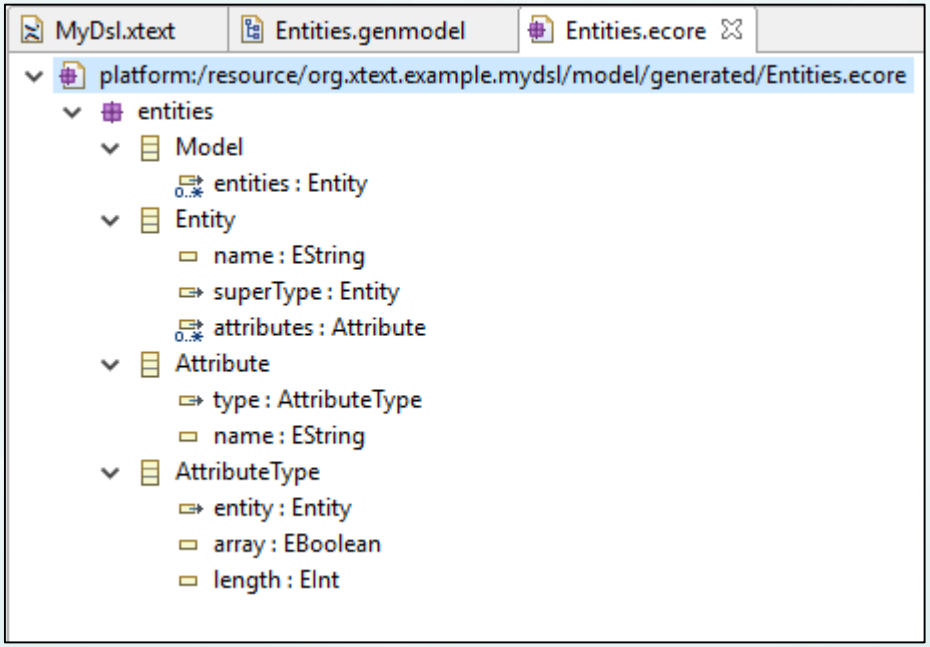
```
entity firstEnt {

}

entity secondEnt{
    firstEnt attr;
    firstEnt [23] attList;
}
```

- If the length is not specified, the length feature will hold the default integer value zero (0).

Dealing with Types

- To be conceptually correct, the **array** feature should not be a part of the **Attribute.** Instead, it should be something that concerns only the type of the Attribute**;**
- To solve this, the concept of **AttributeType** should be put in a separate rule, which will also result in a new **EMF class** in the **Ecore** model.

```
Attribute:

 type=AttributeType name=ID ';';

AttributeType:

 entity=[Entity] (array ?='[' (length=INT)? ']')?;
```



Dealing with Types

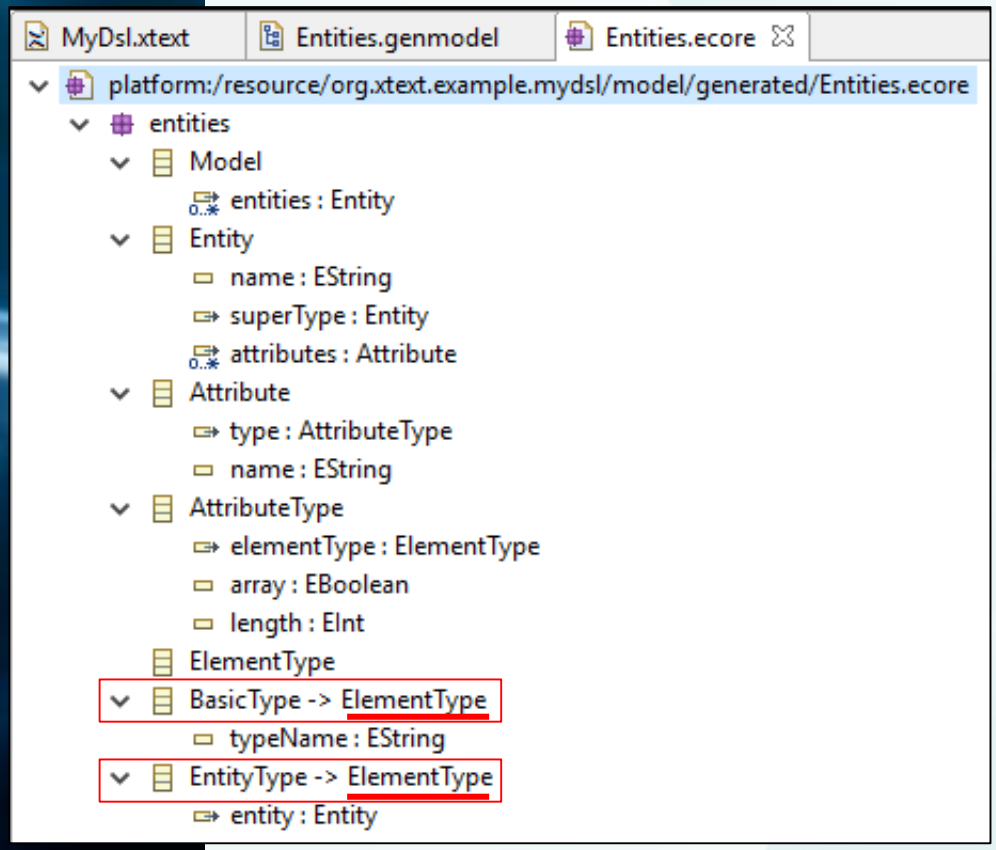
- In order to create **basic types** (string, int and boolean), syde-by-side with the existing **entity types**, it's necessary to abstract over the element types;

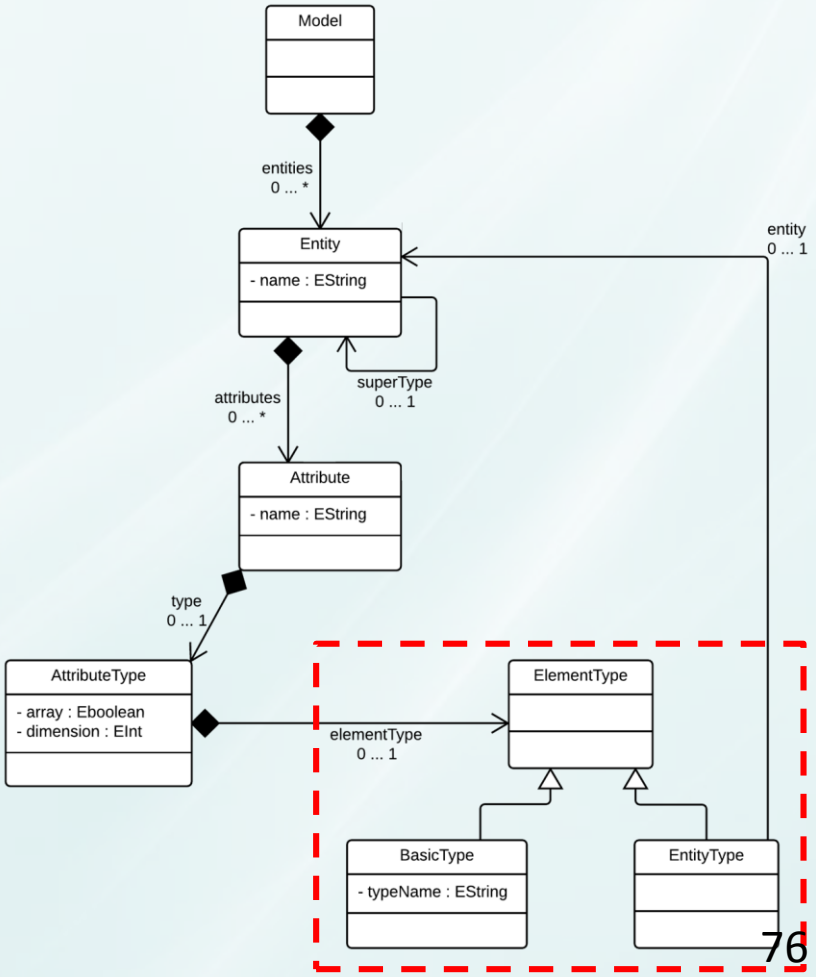
- This can be done with the establishment of a new rule, **ElementType**, which in turn relies on two alternative rules (mutually exclusive): **BasicType** and **EntityType**.

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

ElementType:
    BasicType | EntityType ;

BasicType:
    typeName = ('string'|'int'|'boolean') ;

EntityType:
    entity = [Entity] ;
```

Dealing with Types

- The **ElementType** rule(EDataType) delegates to other alternative rules, introducing an **inheritance relation** in the generated EMF classes.
- This way, both **BasicType** and **EntityType** inherit from **ElementType**.

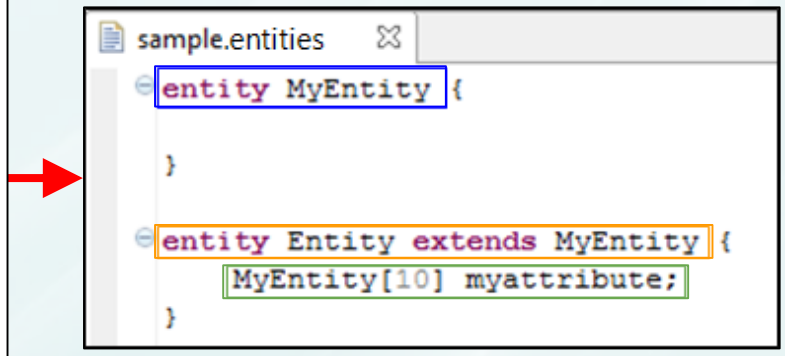




Dealing with Types

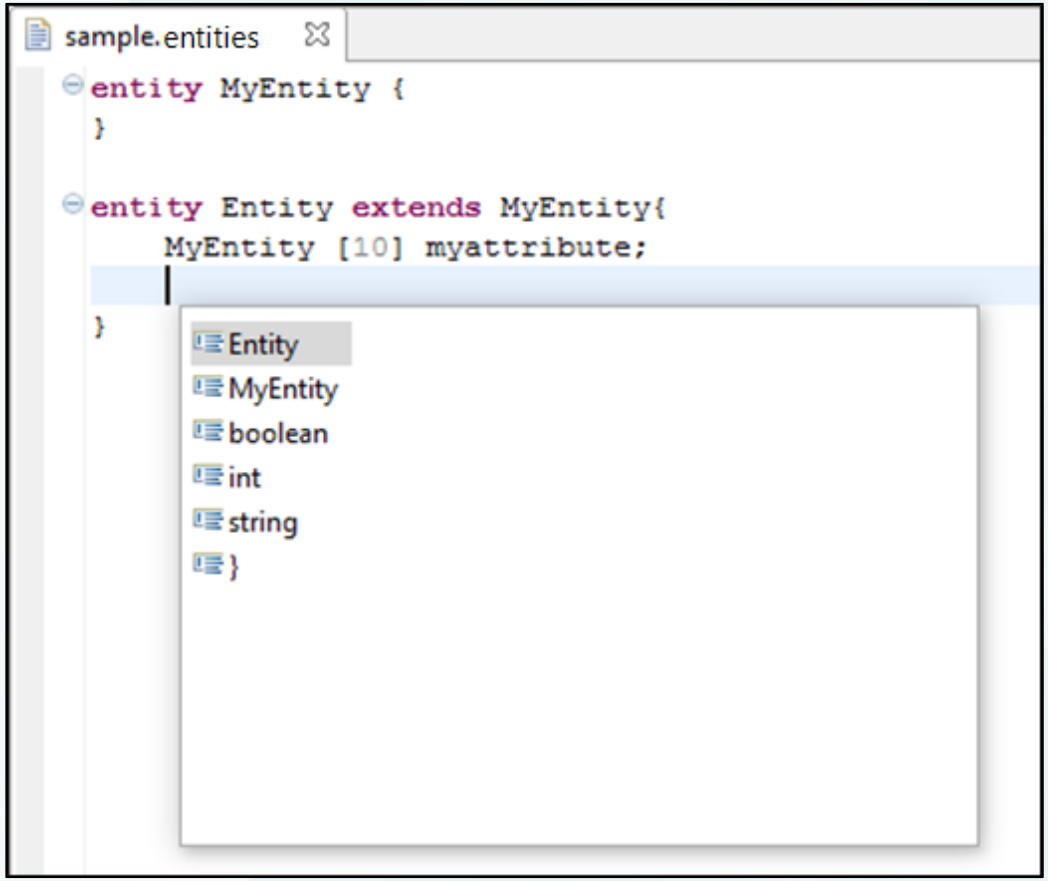
- Entities Java model that corresponds to an Entities DSL program

```
import org.example.entities.entities.Attribute;
import org.example.entities.entities.EntitiesFactory;
import org.example.entities.entities.Entity;
import org.example.entities.entities.Model;

public class EntitiesEMFExample {
    public static void main(String[] args) {
        EntitiesFactory factory = EntitiesFactory.eINSTANCE;

        Entity superEntity = factory.createEntity();
        superEntity.setName("MyEntity");

        Entity entity = factory.createEntity();
        entity.setName("Entity");
        entity.setSuperType(superEntity);

        Attribute attribute = factory.createAttribute();
        attribute.setName("myattribute");

        AttributeType attributeType = factory.createAttributeType();
        attributeType.setArray(false);
        attributeType.setLength(10);

        EntityType entityType = factory.createEntityType();
        entityType.setEntity(superEntity);
        attributeType.setElementType(entityType);
        attribute.setType(attributeType);

        entity.getAttributes().add(attribute);

        Model model = factory.createModel();
        model.getEntities().add(superEntity);
        model.getEntities().add(entity);
    }
}
```



Dealing with Types

- After running the MWE2 workflow, you can try your editor and see that now you can also use the three basic types:



References

- L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. 2013
- Xtext 2.5 Documentation. 2013