Model Engineering Lab 188.923 IT/ME VU, WS 2017/18	**Assignment 2**
Deadline: Upload (ZIP) in TUWEL until Sunday, November 19th, 2017, 23:55 Assignment Review: Wednesday, November 22nd, 2017	25 Points

Concrete Syntax

The goal of this assignment is to develop the concrete syntax of the *Simple Transportation Line Modeling Language (STL)* using Xtext and Sirius. In particular, you will develop a textual concrete syntax and textual editor with Xtext in Part A of this assignment, and a graphical concrete syntax and graphical editor with Sirius in Part B of this assignment.

Assignment Resources

```
ME_WS17_Lab2_Resources.zip
    at.ac.tuwien.big.stl (Modeling project with solution of Assignment 1)
    at.ac.tuwien.big.stl.edit (Edit project with solution of Assignment 1)
    at.ac.tuwien.big.stl.editor (Editor project with solution of Assignment 1)
    at.ac.tuwien.big.stl.xtext (Xtext project for developing the textual syntax of STL)
    at.ac.tuwien.big.stl.xtext.ide (Xtext project for the textual editor of STL)
    at.ac.tuwien.big.stl.xtext.ui (Xtext project for the textual editor of STL)
    at.ac.tuwien.big.stl.design (Sirius project for developing the graphical syntax of STL)
    at.ac.tuwien.big.stl.examples.xtext (Project with STL example models defined with the
            textual editor of STL)
    at.ac.tuwien.big.stl.examples.sirius (Project with STL example models defined with the
            graphical editor of STL)
    at.ac.tuwien.big.stl.examples.xtext.serializer.xmi (Project that can be used to
            serialize a textual STL model as XMI resource)
    lab2.pdf (this document)
```

Before starting this assignment, make sure that you have all necessary components installed in your Eclipse. A detailed installation guide can be found in the TUWEL course[1].

It is recommended that you read the complete assignment specification at least once. If there are any parts of the assignment specification or the provided resources that are ambiguous to you, don't hesitate to ask in the forum for clarification.

[1] Eclipse Setup Guide: https://tuwel.tuwien.ac.at/mod/page/view.php?id=388945

Part A: Textual Concrete Syntax

In the first part of this assignment, you have to develop the textual concrete syntax (grammar) for STL with Xtext. Additionally, you have to implement scoping support for the textual editor that is generated from the developed grammar.

The grammar has to be built upon the sample solution of the STL metamodel provided in the assignment resources (`at.ac.tuwien.big.stl/model/stl.ecore`).

Do not use the metamodel you have developed in Assignment 1 but the sample solution provided in the assignment resources!

A.1 Xtext Grammar for STL

Develop an Xtext grammar that covers all modeling concepts provided by STL.

The Xtext grammar has to follow the example given in Figure 1. This textual STL example models is also available in full length in the examples project: `at.ac.tuwien.big.stl.examples.xtext/Example2-ShelfSawingProductionLine.stltxt`. A corresponding model serialized with XMI is also provided in the file `Example2-ShelfSawingProductionLine.stl` located in the same project.

Figure 1: Example STL model in textual concrete syntax

```
system ShelfSawingProductionLine{
      item RawShelf="Raw piece of wood"
      item Shelf="Finished shelf"
      item WoodWaste="Wood that goes to waste"

      area ProductionArea {

            generator RawShelfProducer generates RawShelf(cost=1000){
                  output RawShelfProducer_Output:RawShelf
            }

            conveyor ToBufferConveyor(cost=200){
                  input ToBufferConveyor_Input:RawShelf
                  output ToBufferConveyor_Output:RawShelf
            }

            buffer Buffer(cost=200){
                  input Buffer_Input:RawShelf
                  output Buffer_Output:RawShelf
            }

            machine SawingMachine(cost=1000){
                  input SawingMachine_Input:RawShelf
                  output SawingMachine_ProductOutput:Shelf
                  output SawingMachine_WasteOutput:WoodWaste

                  service Saw(cost=5,reliability=0.99,processingTime=30){
                        input x
                        input y
                        input z
                  }
            }

            conveyor ToShelfStoreConveyor(cost=200){
                  input ToShelfStoreConveyor_Input:Shelf
                  output ToShelfStoreConveyor_Output:Shelf
            }

            conveyor ToWasteStoreConveyor(cost=200){
                  input ToWasteStoreConveyor_Input:WoodWaste
                  output ToWasteStoreConveyor_Output:WoodWaste
            }

            turntable TurnTable(cost=200){
                  input TurnTable_Input:WoodWaste
                  output TurnTable_Output:WoodWaste
            }

            RawShelfProducer_Output > ToBufferConveyor_Input
            Buffer_Output > SawingMachine_Input
            ToBufferConveyor_Output > Buffer_Input
            SawingMachine_ProductOutput > ToShelfStoreConveyor_Input
            SawingMachine_WasteOutput > TurnTable_Input
            TurnTable_Output > ToWasteStoreConveyor_Input
            ToWasteStoreConveyor_Output > WoodWasteStore_Input
            ToShelfStoreConveyor_Output > ShelfStore_Input
      }

      area StoreArea {
            productStore ShelfStore(cost=1000,capacity=500){
                  input ShelfStore_Input:Shelf
            }

            wasteStore WoodWasteStore(cost=800,capacity=5000){
                  input WoodWasteStore_Input:WoodWaste
            }
      }
}
```

To implement the Xtext grammar for STL, perform the following steps:

1. Setting up your workspace

As mentioned in the beginning, the Xtext grammar has to be based on the sample solution of the STL metamodel. We provide this sample solution as well as skeleton projects for the Xtext grammar as importable Eclipse projects.

Import projects

To import the provided project in Eclipse select *File* → *Import* → *General/Existing Projects into Workspace* → *Select archive file* → *Browse*. Choose the downloaded archive *ME_WS17_Lab2_Resources.zip* and import the following projects:

1. `at.ac.tuwien.big.stl`
2. `at.ac.tuwien.big.stl.edit`
3. `at.ac.tuwien.big.stl.editor`
4. `at.ac.tuwien.big.stl.xtext`
5. `at.ac.tuwien.big.stl.xtext.ide`
6. `at.ac.tuwien.big.stl.xtext.ui`

Register the metamodel

Should the Xtext grammar file `Stl.xtext` (project `at.ac.tuwien.big.stl.xtext`) show errors, then register the STL metamodel in your Eclipse instance, such that the Xtext editor with which you develop your grammar knows which metamodel you want to use. You can register your metamodel by clicking right on `at.ac.tuwien.big.stl/model/stl.ecore` → *EPackages registration* → *Register EPackages into repository*. Clean your projects afterwards by selecting *Project* → *Clean… Clean all projects*.

After performing these steps, you should have six projects in your workspace without any errors (there may be warnings that you can ignore).

2. Developing your Xtext grammar

Define the Xtext grammar in the file `Stl.xtext` located in the project `at.ac.tuwien.big.stl.xtext` in the package `src/at.ac.tuwien.big`. Documentation about how to use Xtext can be found in the lecture slides and in the Xtext documentation[2].

Hint: Make use of the `ID` rule defined in the base grammar `org.eclipse.xtext.common.Terminals` to specify the names of different model elements, e.g., the *name* of components.

Hint: When defining cross-references to other model elements, make sure to use the rule `QualifiedName` already defined in the Xtext grammar `Stl.xtext`, e.g., `referenceToClass=[Class|QualifiedName]`.

3. Testing your editors

Generate Xtext artifacts

After you have finished developing your Xtext grammar in `Stl.xtext`, you can automatically generate the textual editor. The Xtext grammar project `at.ac.tuwien.big.stl.xtext` contains a so called model workflow file `GenerateStl.mwe2`, which orchestrates the generation of the textual editor. By default, the workflow will generate all the necessary classes and stub classes in the grammar project and in the related ide-project and ui-project.

To run the generation, right-click on the workflow file `GenerateStl.mwe2` and select *Run As → MWE2 Workflow*. This will start the editor code generation. Check the output in the Console to see if the generation was successful. Please note that after each change in your grammar, you have to re-run the workflow to update the generated editor code.

If you start the editor code generator for the first time, the following message may appear in the console:

```
*ATTENTION*
It is recommended to use the ANTLR 3 parser generator (BSD licence - http://www.antlr.org/license.html). Do you agree to
download it (size 1MB) from 'http://download.itemis.com/antlr-generator-3.2.0.jar'? (type 'y' or 'n' and hit enter)
```

Type *y* and download the jar file. It will be automatically integrated into your project.

Start a new Eclipse instance with your plugins

For starting the generated editor, we have already provided a launch configuration located in the project `at.ac.tuwien.big.stl.xtext`, which is called "`ME Lab 2 Runtime.launch`". Run it by right-clicking on it and selecting *Run As → ME Lab 2 Runtime.* You can have a look at the configuration by right-clicking on that file and selecting *Run As → Run Configurations…*.

Start modeling

In the newly started Eclipse instance, you can create a new empty project (*File → New → Project → General/Project*) and create a new file (*File → New → File*) with the extension *.stltxt* in this project. If you are asked to add the Xtext nature to the project ('Do you want to convert 'projectname' to an Xtext project?') hit *Yes.* Right-click on the created **.stltxt* file and select *Open With → Stl Editor*. Now you can start modeling an STL model to test the generated textual STL editor.

[2] Xtext documentation: http://www.eclipse.org/Xtext/documentation/

Hint: By pressing Ctrl+Space you activate content assist/auto completion, which provides you a list of keywords or elements that can be input at the next position.

Hint: When you have cross-references to other elements in your grammar, you can check which element is actually referenced in a model by using the linking feature. For this just hold Ctrl and click with the mouse on the cross-reference.

Hint: Note that after each change in your grammar, you have to re-run the workflow to update the generated editor code. Furthermore, you will also have to restart the Eclipse instance that you use for testing the generated editor. We also advise you to always clean your example project (*Project → Clean… Clean all projects*) and re-open your example model (`*.stltxt`).

Check examples
In the newly started Eclipse instance, import the example models included in the project `at.ac.tuwien.big.stl.examples.xtext` provided in the assignment resources. Import the project in the workspace (see "Setting up your workspace") and make sure that your Xtext editor for STL can process the examples without problems, i.e., shows no errors or warnings when you open the models.

Check the following examples:

- `Example1-IAFProductLine.stltxt`
- `Example2-ShelfSawingProductionLine.stltxt`

Also make sure that values for attributes and references are actually assigned to the model elements. For checking this, serialize your textual `*.stltxt` model as XMI resource `*.stl`. To achieve this, you can use the following project provided in the assignment resource: `at.ac.tuwien.big.stl.examples.xtext.serializer.xmi`
Import this project into the Eclipse instance that you use for testing your editors, and run the launch configuration `XMISerializer.launch` located in the folder `launcher` (right-click on `XMISerializer.launch` and select *Run As → XMISerializer*. This will produce the following two STL models in the folder `model-gen` (you may have to refresh your workspace for seeing the models; for this, right-klick on the folder `model-gen` and select *Refresh*):

- `Example1-IAFProductLine.stl` (corresponds to `Example1-IAFProductLine.stltxt`)
- `Example2-ShelfSawingProductionLine.stl` (corresponds to `Example2-ShelfSawingProductionLine.stltxt`)

Investigate the elements of these models in the tree editor of STL (right-click on the `*.stl` model file and select *Open With → STL Model Editor*) and make sure that their attribute values and references are correctly set via the *Properties* view (if the *Properties* view is not shown, right click on any model element and select *Show Properties View*).

A.2 Scoping Support for STL

In the grammar that you have developed, you should have defined some cross-references, e.g., a slot refers to an item type and a connector refers to one entry slot and an exit slot. When testing your editor, you will notice that whenever there is a cross-reference, the content assist (CTRL+Space) will provide all elements of the respective type that the editor can find. Which elements the editor provides is defined in the scoping[3] of the respective reference. By default, the editor searches through the whole class-path of the project and you will notice that if you have many models in your project, a lot of elements will show up as possible reference.

To restrict this behavior, Xtext provides a stub where developers can define their own scope. The stub for the STL language can be found in the grammar project `at.ac.tuwien.big.stl.xtext` in the package `src/at.ac.tuwien.big.scoping` (`StlScopeProvider.java`). Your task is to implement the following scoping behavior in addition to the already existing ones:

1) Scoping for the output slot of an item generator

The required item type defined by an output slot of an item generator (reference `Slot.requiredType`) has to be the item type defined by the item generator as generated type (reference `ItemGenerator.generatedType`)

Example: The output slot "RawShelfProducer_Output" of the item generator "RawShelfProducer" defined in the example model `Example2-ShelfSawingProductionLine.stltxt` in line 9 may only refer to the item type "RawShelf", because the item generator defines this item type as generated item type (line 8, "**generates** RawShelf"). It is not allowed to refer to the other two item types "Shelf" and "WoodWaste".

2) Scoping for the exit slot of a connector

The exit slot of a connector has to be an input slot of a component defined in the same STL system. The slot may not be contained by the component that contains the entry slot of the same connector. Furthermore, the exit slot of a connector has to define the same required type (reference `Slot.requiredType`) as the entry slot of the connector.

Example: In the model `Example2-ShelfSawingProductionLine.stltxt`, a connector from the output slot "ToBufferConveyor_Output" of the component "ToBufferConveyor" (this is the entry slot of the connector) can only reference an input slot as exit slot. The referenced input slot must be the input slot of any component in the system other than the component "ToBufferConveyor". Furthermore, it must have set the type "RawShelf" as required type, because the entry slot of the connector ("ToBufferConveyor_Output", see line 14) also defines this item type as required type. It is not allowed to connect the input slot of the conveyor "ToBufferConveyor", any output slot, or any input slot that has a required type other than "RawShelf". Thus, only the input slot of the "Buffer" ("Buffer_Input"), and the input slot of the "SawingMachine" ("SawingMachine_Input") are possible exit slots of such a connector.

Hint: Xtext uses a so called polymorphic dispatcher to handle declarative scoping. This dispatcher uses the Reflection API to search for methods with a specific signature in the `ScopeProvider` class. How you can use this is explained in the lecture slides and in the Java documentation of the class `AbstractDeclarativeScopeProvider`[4], which is the super class of your scoping stub `StlScopeProvider.java`.

After you have implemented the scoping as defined above, start again a new instance of your Eclipse (use the provided launch configuration *"ME Lab2 Runtime.launch"*) and test your scoping with the content assist (Ctrl+Space) and linking feature (Ctrl+Left Click).

3 http://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#scoping
4 http://download.eclipse.org/modeling/tmf/xtext/javadoc/2.3/org/eclipse/xtext/scoping/impl/AbstractDeclarativeScopeProvider.html

Check examples

In the newly started Eclipse instance you can check the remaining models provided by the examples project `at.ac.tuwien.big.stl.examples.xtext` and see whether your scoping covers these cases. However, it should be noted that these tests do not cover all possible cases and additional testing of the implemented scoping from your side is required.

scoping-test1.stltxt (folder *scoping*)	This is a variation of the example Example2-ShelfSawingProductionLine.stl, in which the wrong item type is set for the output slot "RawShelfProducer_Output" (see line 9). In particular, the item type "Shelf" is wrongly set as required type instead of the correct item type "RawShelf". Your STL editor should show an error marker for this line. When you use the content assist in line 9 after "`output RawShelfProducer_Output:`", only the item type "RawShelf" should be offered. After choosing "RawShelf", the errors should disappear.
scoping-test2.stltxt (folder *scoping*)	This is a variation of the example Example2-ShelfSawingProductionLine.stl, in which wrong connectors are defined in the lines 49-51. In line 49, a connector is defined from the output slot "RawShelfProducer_Output" (entry slot) to the input slot "Buffer_Output" (exit slot). Because "Buffer_Output" is an output slot and not an input slot, this connector is incorrect. When you use the content assist in line 49 after "`RawShelfProducer_Output >`", only the slots "Buffer_Input", "SawingMachine_Input" and "ToBufferConveyor_Input" should be offered. Once you select "ToBufferConveyor_Input", the error shown for this line should disappear. In line 50, a connector is defined from the output slot "Buffer_Output" (entry slot) to the input slot "ToShelfStoreConveyor_Input". Because "Buffer_Output" defines the required type "RawShelf" and "ToShelfStoreConveyor_Input" defines the required type "Shelf", this connector is incorrect. When you use the content assist in line 50 after "`Buffer_Output >`", only the slots "SawingMachine_Input" and "ToBufferConveyor_Input" should be offered. Once you select "SawingMachine_Input", the error shown for this line should disappear. In line 51, a connector is defined from the output slot "ToBufferConveyor_Output" (entry slot) to the input slot "ToBufferConveyor_Input" (exit slot). Because both slots are defined by the same components, this connector is incorrect. When you use the content assist in line 51 after "`ToBufferConveyor_Output >`", only the slots "Buffer_Input" and "SawingMachine_Input" should be offered. Once you select "Buffer_Input", the error shown for this line should disappear.

Part B: Graphical Concrete Syntax

In the second part of this assignment, you have to develop a graphical concrete syntax and graphical editor for STL with *Sirius*. Like the textual concrete syntax, also the graphical concrete syntax has to be built upon the sample solution of the STL metamodel provided in the assignment resources.

Sirius Diagram Editor for STL

Develop a graphical concrete syntax and graphical (diagram) editor that allows to display and edit STL model. For this, develop *diagram mappings* and *element creation tools* as described in the following.

Mappings

For displaying STL models on diagrams, develop diagram node mappings and diagram edge mappings for the following STL concepts:

- Components: Machine, ProductStore, WasteStore, Conveyor, TurnTable, Buffer, ItemGenerator
- Service
- Slot
- ItemType
- Connector

The diagram editor has to be able to display these elements following the *graphical concrete syntax* that is illustrated in Figure 2. Figure 2 shows the same example model as the one defined textually in Figure 1. The example model is available in the project `at.ac.tuwien.big.stl.examples.sirius` in the model file `Example2-ShelfSawingProductionLine.stl`.

The images for the mappings to be defined for the concepts `Machine`, `ProductStore`, `WasteStore`, `Conveyor`, `TurnTable`, `Buffer`, and `ItemGenerator` are provided in `at.ac.tuwien.big.stl.design/images`.

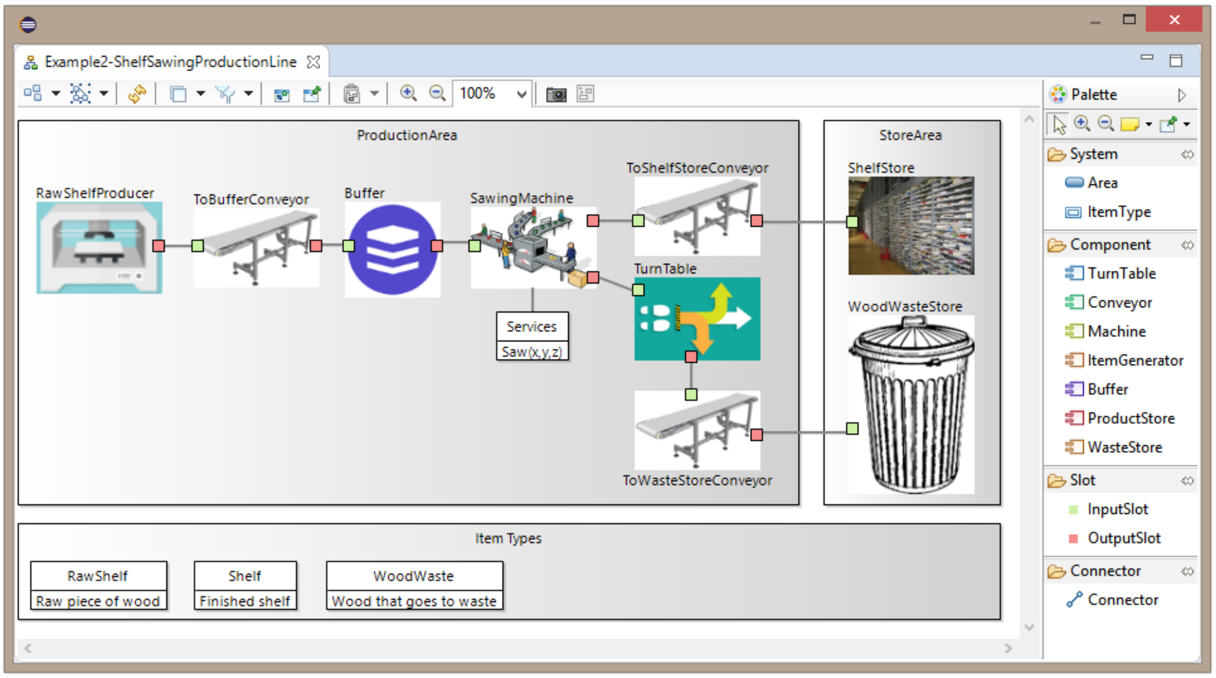
Creation Tools

For editing STL models, develop element creation tools for the following STL concepts (see also the tool palette shown in Figure 2):

- Components: Machine, ProductStore, WasteStore, Conveyor, TurnTable, Buffer, ItemGenerator
- Input Slot
- Output Slot
- ItemType
- Connector

The icons for the element creation tools are provided in `at.ac.tuwien.big.stl.design/icons`.

Figure 2: Example STL model shown in the graphical STL editor



To implement the graphical concrete syntax and graphical (diagram) editor for STL, perform the following steps:

1. Setting up your workspace

Follow the instruction of Part A to setup your workspace and launch a new Eclipse instance. In summary, the following steps have to be performed for this:

- Your workspace has to contain the following projects (imported from the assignment resources):
    1. `at.ac.tuwien.big.stl`
    2. `at.ac.tuwien.big.stl.edit`
    3. `at.ac.tuwien.big.stl.editor`
    4. `at.ac.tuwien.big.stl.xtext`
    5. `at.ac.tuwien.big.xtext.ide`
    6. `at.ac.tuwien.big.xtext.ui`
- Launch a new Eclipse instance using the launch configuration "`ME Lab 2 Runtime.launch`" located in the project `at.ac.tuwien.big.stl.xtext`. In the remainder of this document, the newly launched Eclipse instance is called "Runtime Eclipse Instance".

Next, you have to import the following projects into the Runtime Eclipse Instance:

1. `at.ac.tuwien.big.stl.design`
   You will use this project, to develop the graphical concrete syntax and graphical editor of STL.
2. `at.ac.tuwien.big.stl.examples.sirius`
   You will use this project to test your graphical editor.

Lastly, switch into the *Sirius* perspective by selecting *Window* → *Perspective* → *Open Perspective* → *Other…* → *Sirius*. This perspective provides the *Model Explorer* and *Interpreter* views, which are useful for developing Sirius viewpoint specification models.

2. Developing your graphical editor

Define the graphical concrete syntax of STL with Sirius by extending the viewpoint specification model `stl.odesign`, which is located in the project `at.ac.tuwien.big.stl.design` in the folder `description`. Documentation about how to use Sirius can be found in the lecture slides and in the Sirius documentation[5].

3. Testing your graphical editor

To test your graphical editor, open the diagram "Example2-ShelfSawingProductionLine" defined in the file `representations.aird` located in the project `at.ac.tuwien.big.stl.examples.sirius` (you need to be in the *Sirius* perspective to be able to unfold the content of `representations.aird` in the *Model Explorer*). This diagram is mapped to the STL model defined in the file `Example2-ShelfSawingProductionLine.stl`. Thus, the diagram will be automatically updated with representations of model elements for which you have correctly defined node/edge mappings. Furthermore, the tool palette will show the element creation tools that you have correctly defined. Updates on your Sirius view point specification model `stl.odesign` will be automatically reflected in the opened diagram.

For testing your element creation tools, you can use the diagram "Creation Tool Test STL Diagram" also defined in `representations.aird`. This diagram is mapped to the STL model defined in `CreationToolTestModel.stl`. If you add new elements to the diagram canvas by using the tool palette, this STL model has to be correctly updated. For instance, if you create a new TurnTable with the tool palette, a new TurnTable element must be added to the model.

Hint: You can create additional STL models using the STL tree editor (*File → New → Other... → Example EMF Model Creation Wizards / STL Model*). To create a diagram for a new STL model, expand the model in the *Model Explorer*, right-click on the System element, select *New Representation → new stl*, enter a diagram name, and hit *OK*. Your graphical editor will be automatically opened for the example model.

[5] Sirius documentation: http://www.eclipse.org/sirius/doc/
Sirius tutorials: https://eclipse.org/sirius/getstarted.html

Submission & Assignment Review

Upload the following components in TUWEL:

You have to upload one archive file, which contains the following projects:

```
Submission.zip
    at.ac.tuwien.big.stl.xtext (Xtext project with your Xtext grammar)
    at.ac.tuwien.big.stl.xtext.ide (Xtext project for your textual editor)
    at.ac.tuwien.big.stl.xtext.ui (Xtext project for your textual editor)
    at.ac.tuwien.big.stl.design (Sirius project with your viewpoint specification
           model)
```

For exporting these projects, select *File → Export → General/Archive File* and select the projects.

Make sure to include all resources that are necessary to start your editors!

Thus, if you make changes on any other project that was provided with the assignment resources or implement additional projects, make sure to include them in your submission.

At the assignment review you will have to present your solution, in particular, your Xtext grammar, the implemented scoping support, as well as your Sirius viewpoint specification model. You also have to show that you understand the theoretical concepts underlying the assignment.

All group members have to be present at the assignment review. The registration for the assignment review can be done in TUWEL. The assignment review consists of two parts:

- Submission and **group evaluation:** 20 out of 25 points can be reached.
- **Individual evaluation:** Every group member is interviewed and evaluated separately. The remaining 5 points can be reached. If a group member does not succeed in the individual evaluation, the points reached in the group evaluation are also revoked for this student, which results in a negative grade for the entire course.