| | |
|---|---|
| **Model Engineering Lab**<br>188.923 IT/ME VU, WS 2011/12 | **Assignment 3** |
| **Deadline**:<br>Upload (ZIP) in TUWEL until Monday, January 9, 2012, 23:55<br>Assignment Review: Wednesday, January 11, 2012 | 25 Points |

# Modell transformation

Development of a model-to-model transformation between the **Simple Object Oriented Modeling Language (SOOML)** and the **Simple Object Oriented Programming Language (SOOPL)**.

The following artifacts are given:

## 1) SOOML Metamodel (SOOML.ecore) – Source Metamodel

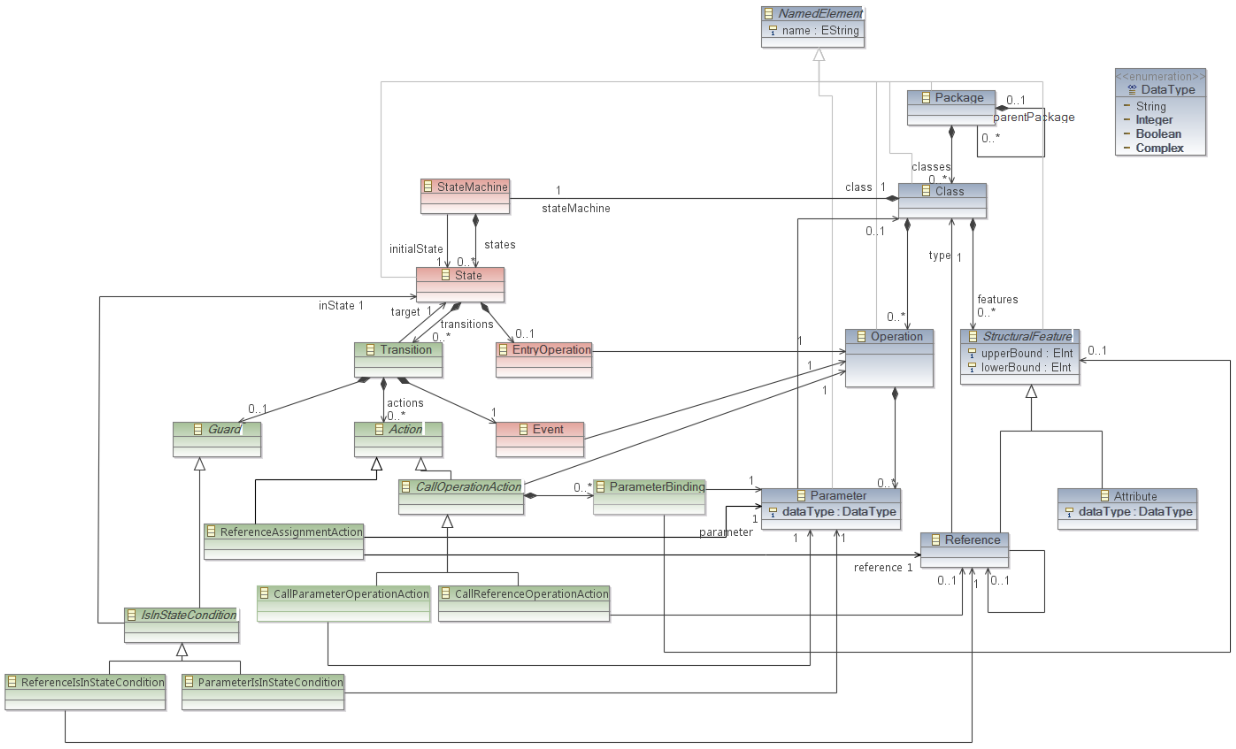


**Figure 1: SOOML Metamodel**

Figure 1 shows the **SOOML metamodel** from Assignment 1. It is provided as a valid Ecore model (**SOOML.ecore**). Furthermore, the input **SOOML model** for the ATL transformation is shown in Figure 3 (**mowersystem-sooml.xmi**). The Ecore classes which do not only get transformed into equal Ecore concepts in SOOPL are colored in red.

## 2) SOOPL Metamodel (SOOPL.ecore) – Target Metamodel

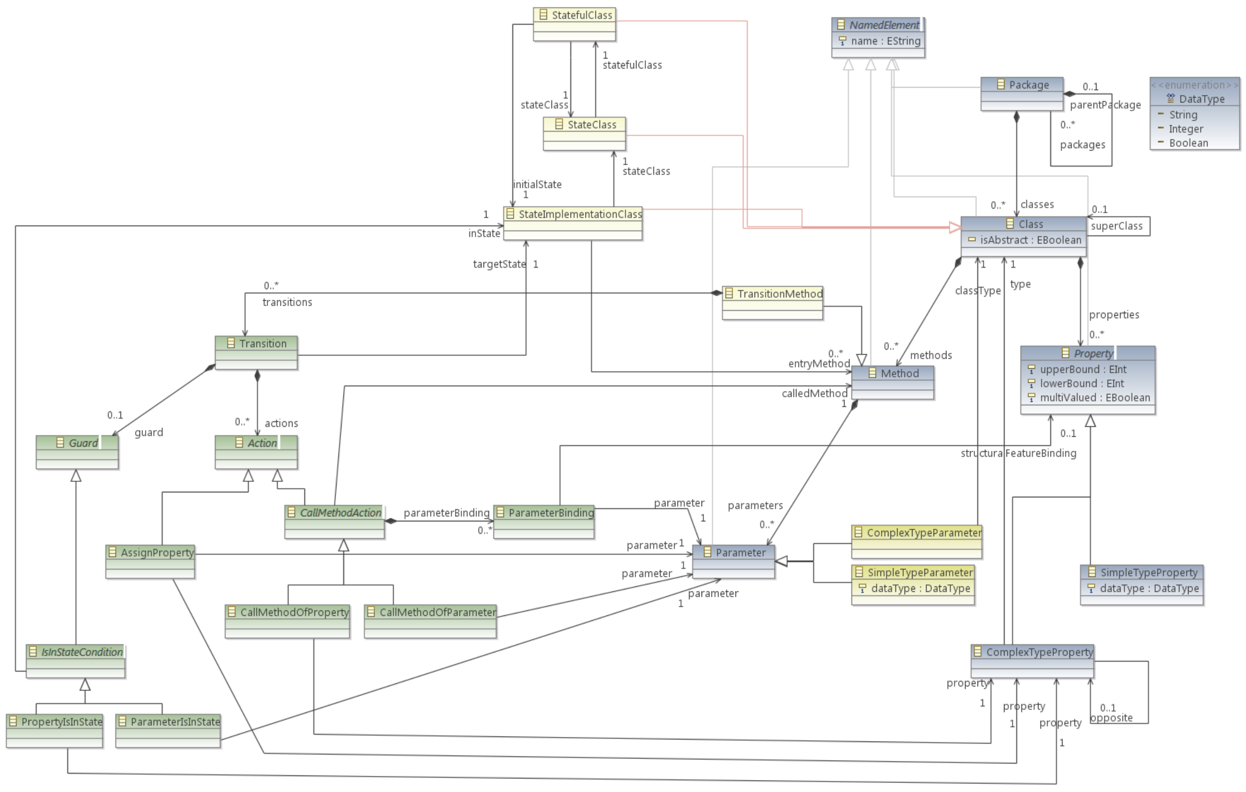


**Figure 2: SOOPL metamodel**

Figure 2 shows the target metamodel called **SOOPL (SOOPL.ecore)**. Yellow parts show the additional concepts of SOOML over SOOPL and are discussed in the following.

Class now has three specialized classes: StatefulClass, StateClass, and StateImplementationClass. In SOOPL the state machine is transformed to proper classes (**StateClass** and **StateImplementingClass**) according to the **state pattern**. Classes from the source model become **StatefulClasses**. The PDF document **statepattern.pdf** (cf. TUWEL) elaborates in more detail how a **state machine** can be mapped to the **design pattern state**.

Methods can now have a **return type**. A Parameter can now be either a **ComplexTypeParameter** or a **SimpleTypeParameter**.

**ComplexTypeProperty** and **ComplexTypeParameter** Ecore classes are now used instead of the **DataType** Complex. **SimpleTypeParameter** is now the same as **Parameter** from the source meta model.

**TransitionMethods** are specific methods, which implement the transitions of the states.

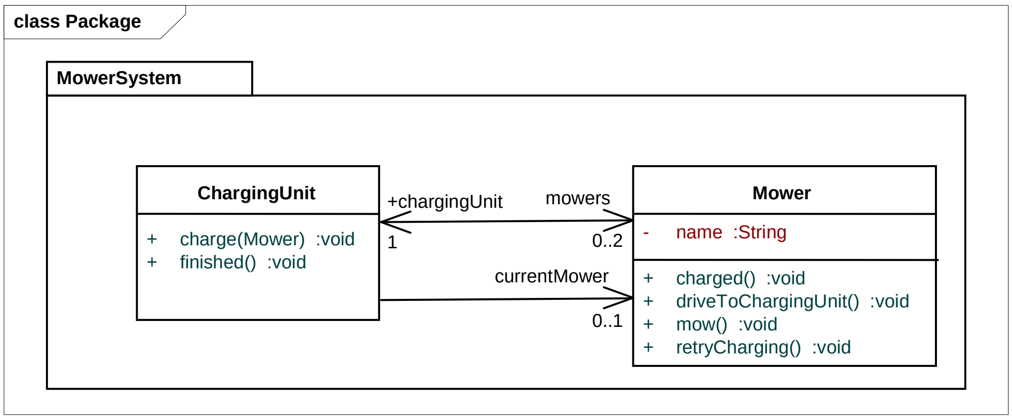
**Property** has a multi-valued attribute, which is true if the upperBound is > 1 or -1.

The green and blue Ecore classes of the meta model are – except for the name - basically the same in SOOPL and SOOML. Thus, you only have to specify simple one-to-one transformation rules for them.

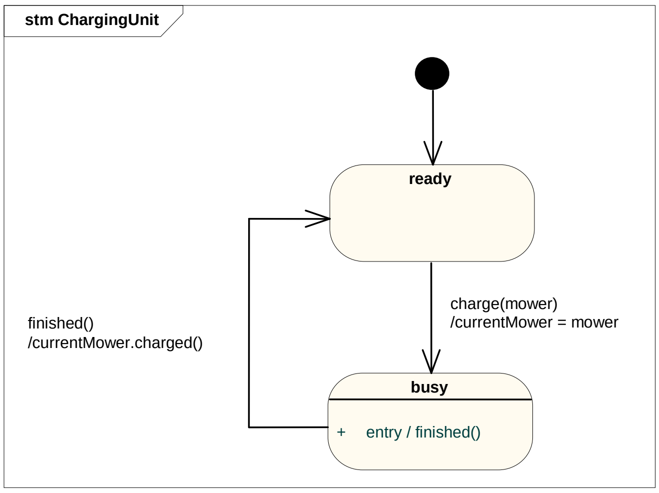
An output **SOOPL model** is shown in Figure 4 (**expected_mowersystem-soopl.xmi**). It is an example for a correct transformation from the source model **mowersystem-sooml.xmi** (Figure 3). Detailed transformation information will be given in the task section.

## 3) SOOML Model (mowersystem-sooml.xmi) – Source Model

### Structural model of the example system



### Behavioral model of the example system


stm ChargingUnit
ready
charge(mower) /currentMower = mower
busy
+ entry / finished()
finished() /currentMower.charged()

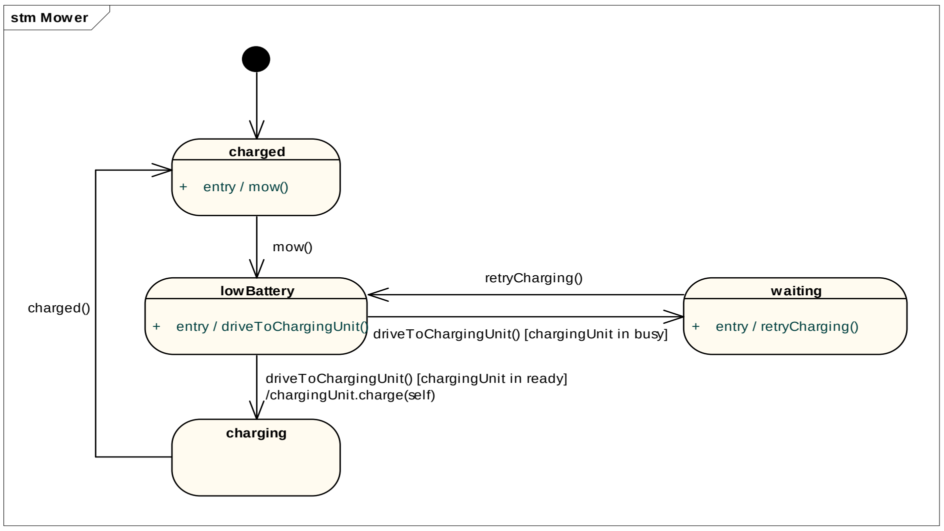

stm Mower
charged
+ entry / mow()
mow()
lowBattery
+ entry / driveToChargingUnit()
retryCharging()
waiting
+ entry / retryCharging()
driveToChargingUnit() [chargingUnit in busy]
driveToChargingUnit() [chargingUnit in ready] /chargingUnit.charge(self)
charging
charged()

**Figure 3: Example SOOML model**

Figure 3 shows the class diagram known from Assignment 1. In the current Assignment 3 it will be used as an example of the source model for the transformation.

## 4) SOOPL Model (expected_mowersystem-soopl.xmi)–Target Model

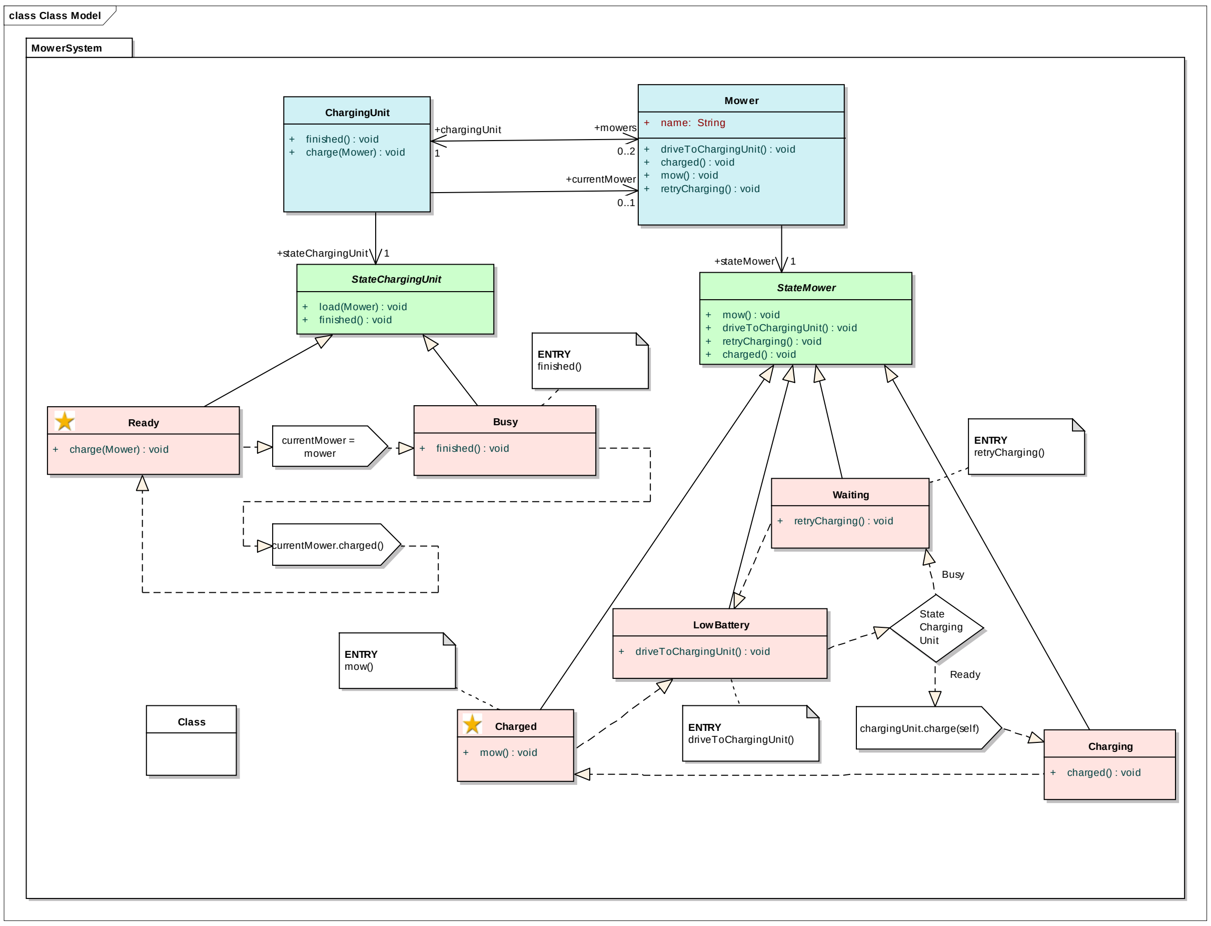


**Figure 4: Example SOOPL model**

In Figure 4 a schematic SOOPL model transformed from the SOOML model (shown in Figure 3) is depicted. The graphical syntax was created with Enterprise Architect and should only help to understand the transformation process. Therefore, the output looks different from the tree structure of the output.xmi file of your transformation.

| | |
|---|---|
| Class | Blue: StatefulClass – classes which refer to StateClasses<br>Green: StateClass – abstract class representing the statemachine<br>Red: StateImplementationClass – classes representing one state |
| ★ | The initial StateImplementationClass. |
| ENTRY<br>mow() | The method of the class, which will be called on entry into this StateImplementationClass. |
| - - - -▷ | Transition started by the method to the new StateImplementationClass. On the transition there might be guards or actions explained next. |
| currentMower.charged() | Actions performed during the transition. |
| State Charging Unit | Guards directing transitions according to conditions |

## Task Description

Create an ATL transformation between the source SOOML and target SOOPL metamodels which transforms SOOML models to SOOPL models.

Figure 5 shows an example input SOOML model which is transformed to a SOOPL output model. The dashed lines indicate, which input part is mapped conceptually to which output part.

SOOML packages are translated into SOOPL packages and SOOML classes get transformed to **StatefulClasses**.

As for the *behavioral models* (according to **statepattern.pdf**), each of the state machines is transformed into a set of classes: (i) one abstract **StateClass** for the **state machine** itself and (ii) one derived **StateImplementationClass** for each of the **states** in the state machine. The **StatefulClasses** (ChargingUnit, Mower) reference their related **StateClass** (StateChargingUnit, StateMower). All methods of the **StatefulClass** also get copied to the referenced **StateClass**.

**StateImplementationClasses** only have the methods which lead to a state change and are of type **TransitionMethods**.

The **StateImplementationClass** can represent the initial state (★), if it was derived from the initial state of the source model.

**StateImplementationClasses** can have an entry method according to the entry operation defined in the state of the source model.

Transitions contained in the source states become transitions contained in the **TansitionMethod**. The transitions themselves (including guards and actions) get transformed from the source to the target model without any changes (only the graphical representation of this example SOOML and SOOPL changed, which does not affect the transformation you need to create).

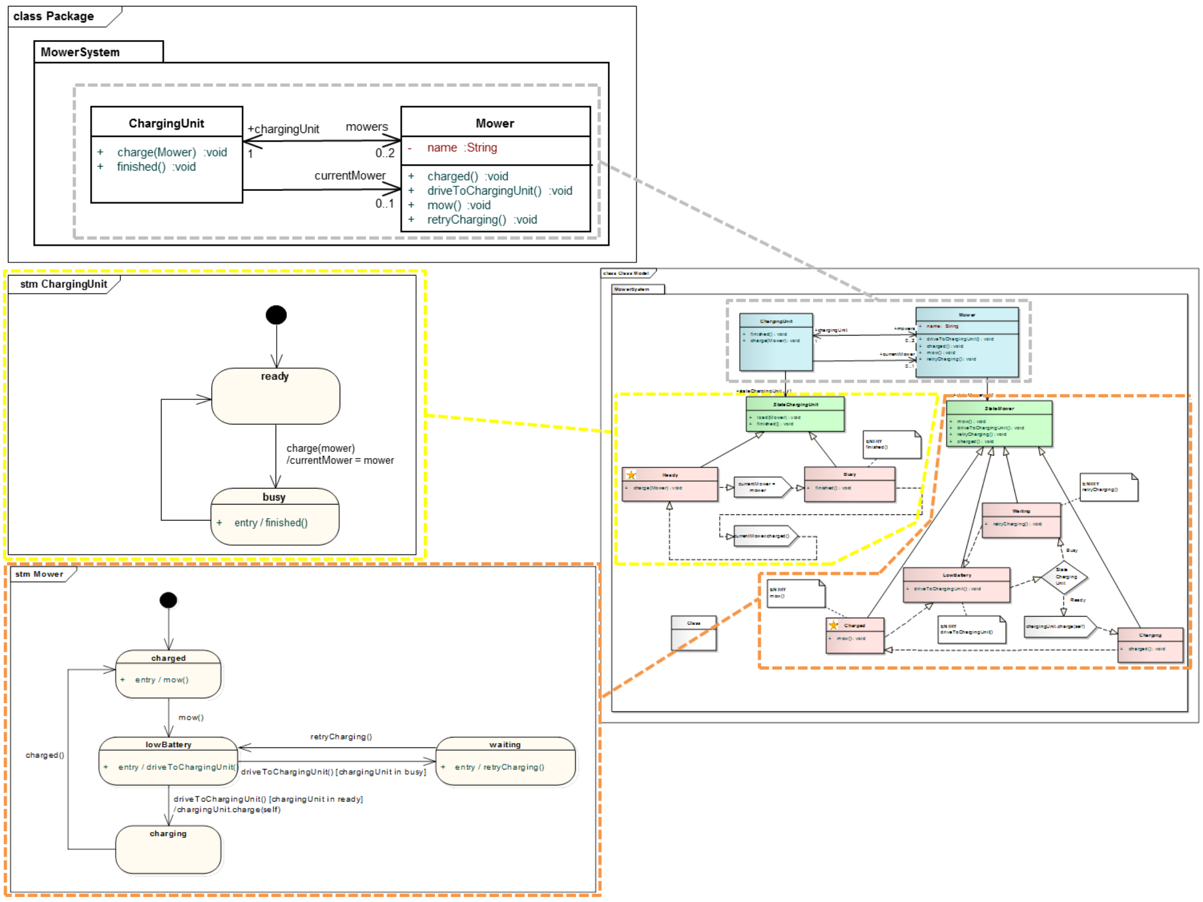


**Figure 5: SOOML model to SOOPL model mapping**

Table 1 shows the relation between the concepts of the two metamodels in detail:

| SOOML Concept | SOOPL Concept | Comments |
|---|---|---|
| **Named Element** | Named Element | |
| **Package** | Package | |
| **Data Type** | Data Type | Complex data type got removed. For Complex data type there are now ComplexTypeProperty and ComplexTypeParameter |
| **Class** | StatefulClass | References the related StateClass<br>Specifies the initial StateImplementationClass according to the source initial state |
| **StructuralFeature** | Property | Multivalued is true, if upperBound > 1 or -1 |
| **Attribute** | SimpleTypeProperty | |
| **Reference** | ComlexTypeProperty | |
| **Operation** | Method | |
| **Parameter** | ComplexTypeParameter<br>SimpleTypeParameter | If source DataType is Complex<br>If source DataTYpe is not Complex |
| **State Machine** | StateClass | References its StatefulClass<br>isAbstract = true |
| **State** | StateImplementationClass | Transitions with the same event will be combined into a single **transitionMethod** in the StateImplementationClass. All these transitions will be part of the **transitionMethod**.<br>The StateClass becomes the SuperClass of this StateImplementationClass.<br>entryOperation becomes entryMethod |
| **Transition** | Transition | Target becomes StateImplementationClass |
| **Event** | - | Obsolete (however, it is used for grouping - cf. State) |
| **Guard** | Guard | |
| **IsInStateCondition** | IsInStateCondition | References StateImplementationClass of related State |
| **ReferenceIsIn StateCondition** | PropertyIsInState | |
| **ParameterIsIn StateCondition** | ParameterIsInState | |

| | |
|---|---|
| **Action** | Action |
| **Reference AssignmentAction** | AssignProperty |
| **CallOperationAction** | CallMethodAction |
| **CallParameter OperationAction** | CallMethodOfProperty |
| **CallReference OperationAction** | CallMethodOfParameter |
| **ParameterBinding** | ParameterBinding |

**Table 1: Relation between SOOML and SOOPL concepts**

## Assignment Resources

Alongside this document, we also provide an Eclipse project archive file (*ME_WS11_Lab3.zip*) in TUWEL, which contains the stub for the ATL transformation to be developed by you (*src/sooml2soopl.atl*), the exemplary input SOOML model (*src/mowersystem-sooml.xmi*), as well as the expected output SOOPL model (*src/expected_mowersystem-soopl.xmi*). Furthermore, in the folder model, we included the SOOML and the SOOPL metamodel. Import this project into your Eclipse workspace and start the ATL transformation stub (select *Run As* → *ATL Transformation* in the context menu of the *sooml2soopl.atl* file). When executing the transformation for the first time, you'll have to configure the run configuration. Therefore, in the run configuration dialog, choose the input metamodel (*sooml.ecore*), the output metamodel (*soopl.ecore*), the input model (*mowersystem-sooml.xmi*) and the output model (*mowersystem-soopl.xmi*). Now, you may run the ATL transformation and check whether your output (*mowersystem-soopl.xmi*) corresponds to the expected output (*expected_mowersystem-soopl.xmi*). All you need to change in this assignment is the ATL file (*src/sooml2soopl.atl*).

## Important ATL Information

- Create at least one helper function.
- Avoid imperative code (do section) as much as possible.
- Use at least one rule inheritance construct.

## Additional Information

- The xmi files can be viewed without the generation of EMF editor code. This is achieved by registering the corresponding Ecore metamodel (context menu of model -> Register EPackages).
- Using EMF Compare, two (meta) models can be compared (select two models in the navigator and choose *Compare With Each Other*). This can be used to compare/prove your transformation output with the provided target model output.
- Literature to solve this Assignment is provided in the TUWEL course. Furthermore, you can find additional information, examples, and use cases on the ATL project homepage http://www.eclipse.org/atl/. All relevant ATL concepts for this assignment are found in the ATL user manual.
- To execute the ATL-transformation in Eclipse [1] you just need the source and target meta models as well as the input model in EMF-XMI format. A small example and introduction videos for ATL as well as the ATL user manual is provided in TUWEL.
- Literature on state patterns can be found in Erich Gamma et al.: *Design Patterns*. Addison-Wesley, 1994, or in Head First Design Patterns. However, the information in the statepattern.pdf provided in TUWEL should be sufficient.

## Submission & Assignment Review

At the assignment review, you will have to present your ATL-transformation from SOOML models to SOOPL models.

**Upload the following components in TUWEL:**

- The ATL file following this pattern: WS11_LAB3_<Groupnumber>.atl

**All group members have to be present at the assignment review.** Application for the assignment review can be done in TUWEL. The assignment review is divided into two parts:

- Submission and **group evaluation:** 20 out of 25 points can be reached.
- **Individual evaluation:** every group member is interviewed and evaluated separately. Remaining 5 points can be reached [2].

## Note

[1] **Tool Support:** Same as in previous assignments. Make sure you have the ATL package installed.

[2] **Evaluation:** The evaluation of your submission includes the **joint** development of this assignment. If a group member did not participate, (s)he can’t reach any points.