

<b>Model Engineering Lab</b> 188.923 IT/ME VU, WS 2011/12	<b>Assignment 1</b>
<b>Deadline:</b> Upload (ZIP) in TUWEL until Monday, November 14, 2011, 23:55 Assignment Review: Wednesday, November 16, 2011	25 Points

## Overview

In the course of this lab, you will develop a family of object-oriented modeling languages that allows its users to specify the structure, the behavior and the execution of a software system. Starting with this lab, you will develop the *abstract syntax* of the modeling language, called SOOML, to represent such a software system with EMF. In Lab 2, your goal is to build a *concrete textual syntax* using Xtext for SOOML and, in Lab 3, you will develop a model-to-model transformation, which translates SOOML models to the so-called *Simple Object-oriented Programming Language (SOOPL)*. Finally, in Lab 4, you will develop a model-to-code transformation that generates executable Java code from SOOPL models.

## Metamodeling

The goal of this assignment is to develop the *Simple Object-Oriented Modeling Language (SOOML)*.

## Part A: Development of the Metamodel for SOOML

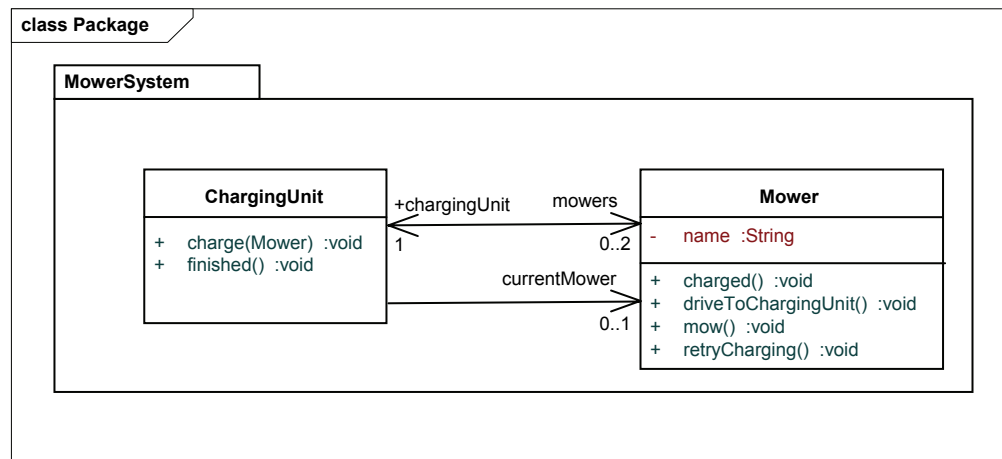
SOOML models describe the structure and the behavior of software systems. To model the structure of a system, we employ a simplified version of UML class diagrams. To specify the behavior of a system, a simplified version of UML state machines is used.

The concepts for SOOML are described with the help of an example model depicted in Figure 1, which specifies the interaction between a mower and its charging unit. Syntax and semantics of the model elements are defined in Table 1.

As depicted in Figure 1, the modeling language comprises, among others, the following concepts:

- For modeling the structure: classes (e.g., ChargingUnit, Mower), operations (e.g., mow, driveToChargingUnit, ...), references (e.g., mowers)
- For modeling the behavior: state machines (e.g., Mower), states (e.g., ready, busy), transitions between states (e.g., arrows)

## Structural model of the example system



## Behavioral model of the example system

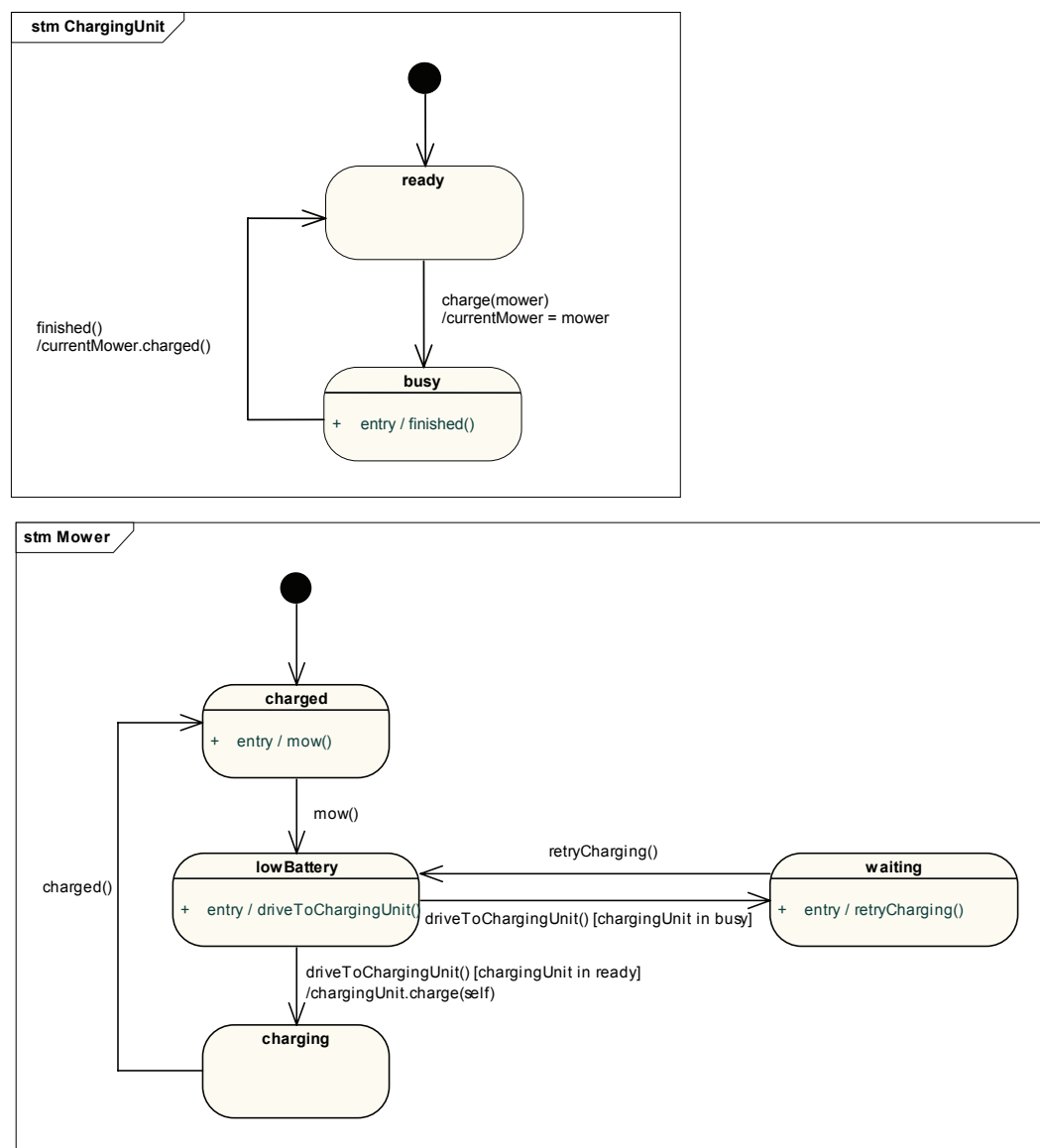
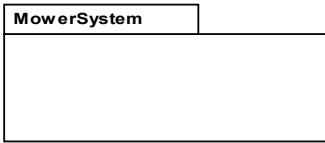
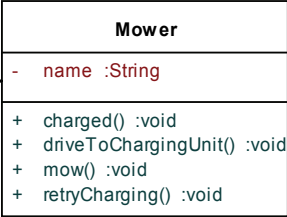
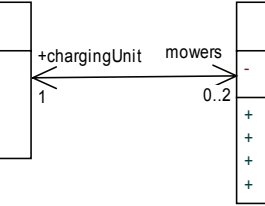
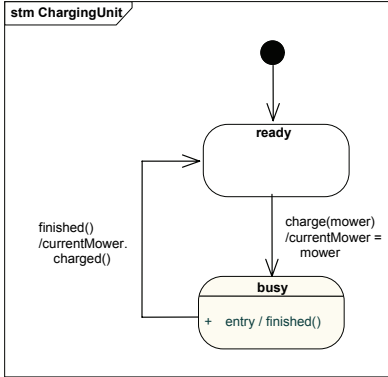
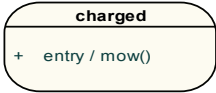



Figure 1: Example SOOML Model

Modeling the structure of a system	
Syntax	Semantics
 <pre> packageDiagram     package MowerSystem         </pre>	<p><i>Packages</i> are used to organize classes. Packages have a <i>name</i> and they can be nested, i.e., packages can contain <i>subPackages</i> and can have a <i>parentPackage</i> (the root package has no parent package).</p>
 <pre> classDiagram     class Mower {         - name :String         + charged() :void         + driveToChargingUnit() :void         + mow() :void         + retryCharging() :void     }         </pre>	<p><i>Classes</i> have a unique <i>name</i> and can contain attributes, and operations.</p> <p><i>Attributes</i> have a <i>name</i> and a <i>DataType</i>. There exist three predefined <i>DataTypes</i>: <i>String</i>, <i>Integer</i>, and <i>Boolean</i>. Further, attributes have a multiplicity defined by <i>lower bound</i> and <i>upper bound</i> (the multiplicity '*' indicates an unbounded multiplicity and should be modeled using the integer '-1').</p> <p><i>Operations</i> have a <i>name</i>. Further, operations can have <i>parameters</i> that have a <i>name</i> and a <i>type</i>, which is either a simple <i>DataType</i> (allowed are again <i>String</i>, <i>Integer</i>, and <i>Boolean</i>) or a complex type, i.e., a class.</p>
 <pre> classDiagram     class MowerSystem     class Mower     Mower --&gt; MowerSystem : mowers     Mower "1" -- "0..2" MowerSystem         </pre>	<p><i>References</i> represent unidirectional relationships pointing from a <i>source</i> to a <i>target</i> class. Bi-directional references between two classes are created with two distinct unidirectional references pointing in opposite direction, where each of the unidirectional references maintains a pointer to the corresponding <i>opposite</i> reference. Each reference has a <i>name</i>, a <i>target</i> (i.e., the referenced class), and a multiplicity defined by a <i>lower bound</i> and an <i>upper bound</i> ('*' again should be represented by the integer '-1'). Further, an <i>opposite</i> pointer identifies the opposing reference of a bi-directional reference.</p>

Modeling the behavior of a system	
Syntax	Semantics
 <pre> stateDiagram-v2     [*] --&gt; ready     ready --&gt; busy : charge(mower) / currentMower = mower     busy --&gt; ready : finished() / currentMower. charged()     state busy {         + entry / finished()     }         </pre>	<p>Each class owns exactly one <i>StateMachine</i>, which describes the behavior of the respective class. A <i>StateMachine</i> has a <i>name</i>. It contains a set of <i>states</i> and a set of <i>transitions</i> among these states.</p>
 <pre> stateDiagram     state charged {         + entry / mow()     }         </pre>	<p>The <i>StateMachine</i> defines the set of legal states an instance of the class may be in. A <i>state</i> has a <i>name</i> and is connected to other states via <i>incoming</i> and <i>outgoing</i> transitions. A state may have a so-called <i>EntryOperation</i> (e.g., <i>mow()</i>) which is executed when the state is entered.</p>
	<p>Each <i>StateMachine</i> has exactly one <i>initial state</i>, which is represented by a black dot in the graphical syntax. An initial state has no incoming transitions and exactly one outgoing transition.</p>

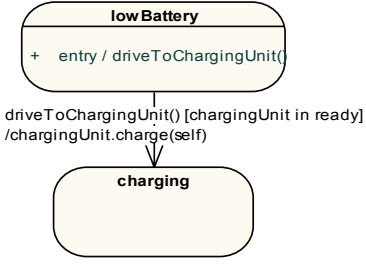
 <pre> stateDiagram-v2     state lowBattery {         + entry / driveToChargingUnit()     }     state charging     lowBattery --&gt; charging : driveToChargingUnit() [chargingUnit in ready] /chargingUnit.charge(self) </pre>	<p>A <i>Transition</i> indicates a state change, that is, the instance transitions from a <i>source</i> state to a <i>target</i> state.</p> <p>A transition can define an <i>Event</i>, which points to the <i>operation</i> that triggers the state change (e.g., <code>driveToChargingUnit()</code>).</p> <p>Additionally a <i>Guard</i> may define a <i>condition</i> which is required to be satisfied in order to execute the transition (e.g., <code>[chargingUnit in ready]</code>).</p> <p>A transition can also specify <i>Actions</i> that are executed when the transition is executed (e.g., <code>/chargingUnit.charge(self)</code>).</p>
<p><code>driveToChargingUnit()</code></p>	<p>An <i>Event</i> is always the call of an operation of the class for which the state machine is defined (e.g., <code>driveToChargingUnit()</code> is an operation of the class <i>Mower</i> and can therefore be used as an event in the state machine of <i>Mower</i>)</p>
<p><code>[chargingUnit IN ready]</code></p>	<p><i>Guards</i> can be defined using so-called <i>IsInStateConditions</i>. Such a condition defines that a particular instance has to be in a particular state, called the <i>in-state</i>.</p> <p>The instance to be checked against a particular state can either be</p> <ol style="list-style-type: none"> <li>the parameter of the event-operation of the transition (then we speak of <i>ParameterIsInStateConditions</i>)</li> <li>OR –</li> <li>a reference, which is defined in the class associated with the state machine. We call such a condition a <i>ReferenceIsInStateCondition</i>, e.g., <code>chargingUnit</code> is a reference of the class <i>Mower</i>.</li> </ol> <p>The <i>in-state</i> of the condition has to be a possible state of the class (e.g., <code>ready</code> is a possible state of the class <i>ChargingUnit</i>).</p>
<p><code>/chargingUnit.charge(self)</code></p>	<p>In the course of a transition, actions can be executed. These so-called <i>CallOperationActions</i> can call an operation of any object.</p> <p>This object can again be either</p> <ol style="list-style-type: none"> <li>the parameter of the event-operation of the transition (<i>CallParameterOperationAction</i>)</li> <li>OR –</li> <li>a reference of the class of the state machine (<i>CallReferenceOperationAction</i>). For instance, <code>chargingUnit</code> is again a reference of the class <i>Mower</i>.</li> </ol> <p>For operations with parameters, these parameters have to be set using <i>ParameterBindings</i>, which specify the values that are provided for the parameters of the called operation.</p>
<p><code>/currentMower = mower</code></p>	<p><i>ReferenceAssignmentActions</i> are used to set references which are defined for the class associated with the state machine. Only event-parameter values may be assigned to references. Thus, a <i>ReferenceAssignmentAction</i> sets the value of a <i>reference</i> according to a <i>parameterBinding</i>. For example, in the <i>ChargingUnit</i>'s state machine the reference <i>currentMower</i> is set to the value of the <i>mower</i> parameter from the <i>ChargingUnit.charge(mower)</i> operation.</p>

Table 1: **Syntax and semantics of the SOOML concepts**

## Task Description

Develop a metamodel for SOOML with EMF's metamodeling language *Ecore*.

- Note that not every single concept is represented graphically (e.g., due to inheritance).
- Use the above mentioned terms and introduce new terms only when necessary. When using own terms, provide the definitions of syntax and semantics in an own text or pdf file.
- Use EMF's "Sample Reflective Ecore Editor" or "Ecore Diagram Editor" [1] to create the model.
- **IMPORTANT:** Do NOT translate the example model (in Fig. 1) to Ecore, but develop a language (i.e., the metamodel) to describe the systems classes, their references, states, etc. as described above.

## Part B: Metamodel Testing

Generate a tree-based editor as shown in the video on TUWEL to test your metamodel. Next, model the example SOOML model shown in Figure 1 with the generated editor.

### Task Description

In part A you developed a metamodel based on Ecore for SOOML. Make sure that you have done this as precisely as possible, i.e., the metamodel has to contain all necessary language concepts: use appropriate classes, attributes, relationships, role names, etc. and constrain the relationships between the concepts. Use the functionalities of EMF to create a tree-based model editor (cf. video in TUWEL), and test your metamodel by modeling the above stated example model (shown in Figure 1) with the help of this editor.

## Submission & Assignment Review

At the assignment review you will have to present your metamodel for SOOML, the corresponding modeling editor, and the example model. The language has to contain all described concepts and it must be possible to model the example depicted in Figure 1.

### Upload the following components in TUWEL:

- Class Diagram of the metamodel as PDF
- Entire EMF project exported as archive file:
  - Ecore file of metamodel (sooml.ecore)
  - Example model (ExampleModel.xmi)
  - Model-, edit- and editor-plug-ins

**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:** For the lab part, we provide a description of how to set up an Eclipse version that contains all necessary plug-ins for all assignments (cf. TUWEL). Once you have installed Eclipse and all necessary plug-ins, you can create Ecore models

with the graphical editor. You can employ the graphical editor by, creating an *Empty EMF Project* and selecting *New → Other → Ecore Tools → Ecore Diagram*.

An Ecore Diagram provides a graphical view (.ecorediag file) on the abstract syntax of the metamodel defined in the Ecore model (.ecore file). You can also modify the Ecore-file directly with a tree-based editor, but many changes cannot be automatically updated in the graphical view. In such cases, the graphical view (*Ecore Diagram*) must be rebuilt by right clicking on the .ecore file and selecting *Initialize Ecore Diagram* from the context menu. You can decide, whether you create the metamodel with the graphical editor in the concrete syntax or with the tree-based editor in the tree-based abstract syntax.

If you decide to model using the abstract syntax, generate a graphical view after creating the metamodel. In the tree-based structure, you can open the XMI file via *Open with → Sample Reflective Ecore Model Editor*.

To test the metamodel you can right click on the created .ecore file and select *EPackages Registration → Register Metamodel*. Now, open the .ecore file in the tree-based editor, right click the package of your Ecore model and select *Create Dynamic Instance*. Specify a name for the new model (e.g., *test.xmi*) and add new model elements in the generated model.

***For more information confer to the 3 videos in TUWEL, which demonstrate the usage of the EMF tools.***

- [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.
- [3] **Literature** about EMF can be found on [www.eclipse.org/emf](http://www.eclipse.org/emf). Very supportive resources are linked in TUWEL.