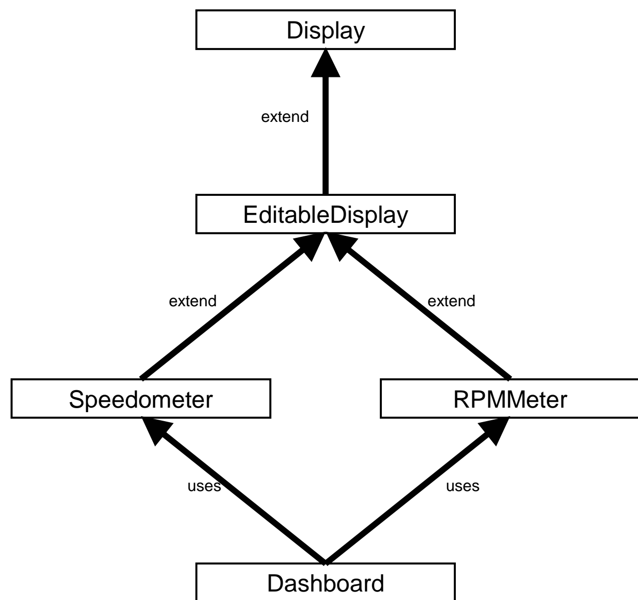
Towards Object Orientation: ODISL

We tried to make an Object oriented Version of DISL with the aim to have re-usable interfaces, where not only structure and style can be re-used but also the behaviour can be inherited. Since our dialog model was based on DSN, we looked at the DSN extension ODSN which modelled dialog objects with local states and events. As a result, DISL has been modified in several ways, which we show in following examples. A side effect is that now templates seem to be obsolete, since interface parts specified in templates can be replaced with base classes.

As an example, we want to model a car dashboard in an OO fashion. This dashboard consists of two displays: A speedometer and RPM-meter. Since both display have in common, that they are used for displaying a value, both inherit from a class “EditableDisplay”, which in turn inherits from a generic base class “Display”. The class hierarchy is shown in the following picture:



To allow this subclassing and instantiating, DISL had to be changed in some parts, which we show, by presenting each interface class and explaining the changes with respect to the original DISL specification.

First of all, we changed the name of the root element to indicate that we are now dealing with an OO-Version of DISL and the second major change is, that DISL now consists of interface classes instead of interfaces. Furthermore, we added in addition to structure, style and behavior a methods-section, which allows the specification of methods that are used to operate on the elements of the object, they where defined in. As we want to be flexible with subclassing and determine which part of the interface need to be overridden or which parts are inherited, we now attach an id to any Element that can be reused. With the Element definition, ODISL also allows to set public or private access rights. This can be seen in the code for the Display class:

```
<!-- Changeed root element to reflect object orientation -->
<odisl>
  <!--interfaces are replaced by interfaceclass to provide inheritence.
      They still have structure, style and behavior but also methods
      Associated with this class.
  -->
  <interfaceclass id="Display">
    <!-- Not only interface (classes) have an ID but also structure, style,
         behavior and methods need an ID, so that they can referenced from
         interfaces that include several interface classes (see
         "Car-Dashboard.xml").
    -->
    <structure id="displayStructure">
      <widget id="showValue" generic-widget="variablefield"/>
    </structure>

    <style id="displayStyle">
      <part generic-widget="showValue">
        <property id="type">string</property>
        <property id="value">0</property>
        <property id="title">display</property>
        <property id="description">
          This is a simple Display for extension purposes
        </property>
        <property id="help">
          Extend this ODISL class for creating displays
        </property>
        <property id="visible">no</property>
      </part>
    </style>

    <behavior id="displayBehavior">
      <!-- As we are in OO anyway, we differenciate between public and private
           access to support encapsulation
      -->
      <variable id="isvisible" access="private" type="boolean">
        false
      </variable>
      <!-- this rule can be publically accessed -->
      <rule id="newVisible" access="public">
        <condition>
          <equal negation="yes"><!-- meaning: not(variable == property)-->
            <variable-content id="isVisible"/>
            <property-content generic-widget="showvalue" id="visible"/>
          </equal>
        </condition>
      </rule>
      <!-- But setting the visibility of the widget is private to this class
      -->
      <transition id="setNewVisible" access="private">
        <if-true rule-id="newVisible"/>
        <action>
          <statement>
            <property-content generic-widget="showvalue" id="visible"/>
            <variable-content id="isVisible"/>
          </statement>
        </action>
      </transition>
```

```
    </behavior>

    <!-- The methods for this class -->
    <methods id="displayMethods">
      <!-- A public method that receives one boolen parameter and returns a
           boolean parameter. Purpose of this method is to set the visibility
           of this interface.
      -->
      <method id="setVisible" access="public">
        <params>
          <param id="visible" type="boolean">
        </params>
        <returns id="isVisible" type="boolean"/>
        <!-- in the action-part, the variable "isvisible" is set to the value
             from the parameter
        -->
        <action>
          <statement>
            <variable-content id="isVisible"/>
            <param-content id="visible"/>
          </statement>
        </action>
        <!-- We need to defne what is to happen, after the behavior of this
             interface class is resolved, because only then, the new state of
             the interface is known and the method can return its result.
        -->
        <afterresolving>
          <statement>
            <return-content id="isVisible"/>
            <property-content generic-widget="showValue" id="visible"/>
          </statement>
        </afterresolving>
      </method>
      <!-- A method with empty parameter list -->
      <method name="getVisible" access="public">
        <params/>
        <returns name="isVisible" model="boolean">
        <action/>
        <afterresolving>
          <statement>
            <return-content id="isVisible"/>
             <property-content generic-widget="showValue" id="visible"/>
          </statement>
        </afterresolving>
      </method>
    </methods>
  </interfaceclass>
</odisl>
```

To establish inheritance in DISL, we have to include the source code from the class from which we want to inherit and within the interface class, we use the “extends” attribute to specify from which class we inherit. However, inheritance from a class does not imply that we inherit from all parts, but that we have the possibility to extend them. For this reason, structure, style etc. have to be extended separately. This is particulary useful, if we want to reuse a structure but need a completely different behaviour. If however an Element extends the one of the parent class (for example Methods extends the

displayMethods in the following example code.), all sub parts are included in the new class and new specified elements are added to the inherited ones:

```
<odisl>
  <!-- Include-mechanism as in C/C++ to make sure, the proper odisl-files are
       referenced
  -->
  <include source="Display.xml">

  <!-- This interface class inherits from the display class -->
  <interfaceclass id="EditableDisplay" extends="Display">
    <!-- Since structure, style, and behavior are treated seperately, we
         extend them as well, if necessary. In this case, structure and style
         are adopted from the display class without modifications
    -->
    <structure id="editableDisplayStructure" extends="displayStructure"/>
    <style id="editableDisplayStyle" extends="displayStyle"/>

    <!-- The behavior section adds to the inherited elements a new variable
         with rules and transitions for showing a string value
    -->
    <behavior id="editableDisplayBehavior" extends="displayBehavior">
      <variable id="showValue" access="private" type="string">0</variable>
      <rule id="newValue" access="public">
        <condition>
          <equal negation="yes">
            <variable-content id="showValue"/>
            <property-content generic-widget="showValue" id="value"/>
          </equal>
        </condition>
      </rule>
      <transition id="setNewValue" access="private">
        <if-true rule-id="newValue"/>
        <action>
          <statement>
            <property-content generic-widget="showValue" id="value"/>
            <variable-content id="showValue"/>
          </statement>
        </action>
      </transition>
    </behavior>

    <!-- Methods for getting and setting values are added in addition to the
         inherited methods to get and set the visibility state of the widget.
    -->
    <methods id="editableDisplayMethods" extends="displayMethods">
      <method id="setValue" access="public">
        <params>
          <param id="value" type="string">
        </params>
        <returns id="newValue" type="string">
        <action>
          <statement>
            <variable-content id="showValue"/>
            <param-content id="value"/>
          </statement>
        </action>
```

```
        <afterresolving>
          <statement>
            <return-content id="newValue"/>
            <property-content generic-widget="showValue" id="value"/>
          </statement>
        </afterresolving>
      </method>
      <method id="getValue" access="public">
        <params/>
        <returns id="value" type="string">
        <action/>
        <afterresolving>
          <statement>
            <return-content id="value"/>
            <property-content generic-widget="showValue" id="value"/>
          </statement>
        </afterresolving>
      </method>
    </methods>
  </interfaceclass>
</odisl>
```

The next example shows a concrete instance of the editable display from the previous example. As within the last example, structure and style where adopted from the base class by inheritance, here some properties of the “showValue” widget in the style section are overridden. This can be done, as we exactly know from which class we extended and all properties had a unique ID.

```
<odisl>
  <!-- A concrete instance from editable display, that overrides some
       properties from the base class
  -->
  <include source="EditableDisplay.xml"/>
  <interfaceclass id="Speedometer" extends="EditableDisplay">
    <structure id="speedometerStructure" extends="editableDisplayStructure"/>
    <!-- Overrides the title, description and help properties but adopts the
         rest
    -->
    <style id="speedometerStyle" extends="editableDisplayStyle">
      <part generic-widget="showValue">
        <property id="title">Car Speedo</property>
        <property id="description">Cars current speed</property>
        <property id="help">
          This widget shows the current speed of your vehicle
        </property>
      </part>
    </style>

    <behavior id="speedometerBehavior" extends="editableDisplayBehavior">
      <variable id="maxSpeed" access="private" type="integer">-1</variable>
      <rule id="speedTooLow" access="private">
        <condition>
          <op>
            <op-expression expression="lessthan">
              <property-content generic-widget="showValue" id="value"/>
              <constant>0</constant>
            </op-expression>
```

```
      </op>
    </condition>
  </rule>
  <rule id="speedTooHigh" access="private">
    <condition>
      <op>
        <op-expression expression="greaterthan">
          <property-content generic-widget="showValue" id="value"/>
          <variable-content id="maxSpeed"/>
        </op-expression>
      </op>
    </condition>
  </rule>
  <transition id="setLowValue" access="private">
    <if-true rule-id="speedTooLow"/>
    <action>
      <statement>
        <property-content generic-widget="showValue" id="value"/>
        <constant>0</constant>
      </statement>
    </action>
  </transition>
  <transition id="setHighValue" access="private">
    <if-true rule-id="speedTooHigh"/>
    <action>
      <statement>
        <property-content generic-widget="showValue" id="value"/>
        <variable-content id="maxSpeed"/>
      </statement>
    </action>
  </transition>
</behavior>

<methods id="speedometerMethods" extends="editableDisplayMethods">
  <method id="setMaxSpeed" access="public">
    <params>
      <param id="newMaxSpeed" type="integer">
    </params>
    <returns id="maxSpeed" type="integer">
    <action>
      <statement>
        <variable-content id="maxSpeed"/>
        <param-content id="newMaxSpeed"/>
      </statement>
    </action>
    <afterresolving>
      <statement>
        <return-content id="maxSpeed"/>
        <variable-content id="maxSpeed"/>
      </statement>
    </afterresolving>
  </method>
  <method id="getMaxSpeed" access="public">
    <params/>
    <returns id="maxSpeed" type="integer">
    <action/>
    <afterresolving>
      <statement>
```

```
            <return-content id="maxSpeed"/>
            <variable-content id="maxSpeed"/>
          </statement>
        </afterresolving>
      </method>
    </methods>
  </interfaceclass>
</odisl>
```

The next short example shows, how we inherit and extend the behaviour. Since in behaviour specifications temporal relations are of importance, we are able to specify, when an action has to be taken e.g. how the behaviour has to be resolved. For the RPM-Display, it would for example make no sense, if the division though 1000 is applied after the new value has been set. In that case the display would show for example 3000 instead of 3.

```
<odisl>
  <include source="EditableDisplay.xml"/>
  <interfaceclass id="RPMMeter" extends="EditableDisplay">
    <structure id="RPMMeterstructure" extends="editableDisplayStructure"/>

    <style id="RPMMeterStyle" name="editableDisplayStyle">
      <part generic-widget="showValue">
        <property id="title">RPM Meter</property>
        <property id="description">
          Motor revolutions in 1000 rounds per minute
        </property>
        <property id="help">
          This widget shows the rate of revolutions in 1000 rounds
          per minute
        </property>
      </part>
    </style>

    <behavior id="RPMMeterBehavior" extends="editableDisplayBehavior">
      <!-- To control the sequence of operations, transitions can be aligned
           with respect to a fixpoint, just like widgets can be aligned in the
           structure section.
      -->
      <transition id="translateValue" access="private" align="before"
                  fixpoint="setNewValue">
        <if-true rule-id="newValue"/>
        <action>
          <statement assignment="div">
            <variable-content id="showValue"/>
            <constant>1000</constant>
          </statement>
        </action>
      </transition>
    </behavior>

    <!-- If only elements are inherited but nothing added, than an empty
         element is sufficient
    -->
    <methods id="RPMMeterMethods" extends="editableDisplayMethods"/>
  </interfaceclass>
```

```
</odisl>
```

The final example shows, how an interface-class can instance other classes. In order to differentiate between parts of the current class and instances of other classes, the element names have the appendix ”-instance” e.g. “part-instance”.

```
<odisl>
  <include source="Speedometer.xml"/>
  <include source="RPMMeter.xml"/>
  <!-- This interface class shows the instancing of other pre defined
       classes
  -->
  <interfaceclass id="Dashboard" state="start">
    <structure name="dashboardStructure">
      <widget id="title" generic-widget="variablefield"/>
      <widgetclass id="speedometer" classname="Speedometer"/>
      <widgetclass id="RPMMeter" classname="RPMMeter"/>
    </structure>

    <style id="dashboardStyle">
      <part generic-widget="title">
        <property id="type">string</property>
        <property id="value">Generic Car Dashboard</property>
        <property id="title">Dashboard</property>
        <property id="description">
          This is a dashboard showing speed and rotations per minute
        </property>
        <property id="help">
          The first widget shows your current speed. Second shows current
          rotations per minute of your motor.
        </property>
        <property id="visible">yes</property>
      </part>
      <!-- instancing parts of the speedometer and RPM interfaces -->
      <part-instance id="speedometer" generic-widget="showValue">
        <property id="visible">yes</property>
      </part-instance>
      <part-instance id="RPMMeter" generic-widget="showValue">
        <property id="visible">yes</property>
      </part-instance>
    </style>

    <behavior id="dashboardBehavior">
      <event id="pollSpeed" activated="yes" timer="500ms" repeat="yes"
             access="private">
        <action>
          <call source="osgi://systemcalls/getCurrentSpeed()" id="getSpeed"
                synchronized="yes" timeout="200ms" maxsize="4"/>
          <method-call instance-id="speedometer" method-id="setValue">
            <parameter id="value>
              <call-content id="getSpeed"/>
            </parameter>
          </method-call>
        </action>
      </event>

      <event id="pollRotationalSensor" activated="yes" timer="500ms"
```

```
              repeat="yes" access="private">
        <action>
          <call source="osgi://systemcalls/getRotationalSensor()" id="getRPM"
                synchronized="yes" timeout="200ms" maxsize="5"/>
          <method-call instance-id="RPMMeter" method-id="setValue">
            <parameter id="value">
              <call-content id="getRPM"/>
            </parameter>
          </method-call>
        </action>
      </event>
    </behavior>

    <!-- no methods available, could be made optional -->
    <methods id="dashboardMethods"/>
  </interfaceclass>
</odisl>
```