

# The Lunifera UI DSL

Licenced under the Eclipse Public Licence v1.0

## 1 Purpose

The Lunifera UI DSL facilitates the creation of model-based user interfaces (UIs) and their integration into complex applications. It is not intended for styling and design specifications.

Using the Lunifera UI DSL, layouts and fields can easily be defined along with the appropriate validation and binding mechanisms. The underlying ECView model uses Eclipse databinding to create an efficient databinding between UI elements and the Java objects they represent. The UI model is kept independent of rendering implementations so that it can be easily re-used.

By referencing DTO models (as defined by the Lunifera DTO DSL) in the UI DSL, UI model and internal application logic can be tightly interwoven. The ECView UI model is extensible and allows for a straightforward implementation of new UI elements.

## 2 Architecture

The Lunifera UI DSL is used to define UI models along with the desired validators and databindings. Valid UI DSL files are compiled to ECView model definitions (Java classes) that are interpreted by the UI renderer in real time. Thus, the Lunifera UI DSL acts as a simplified frontend for the ECView model.

In the reference implementation, Vaadin has been chosen as the UI framework that is used to render the UI model. However, since the model-view-controller distinction has been observed, it is possible to change the rendering service without having to change the underlying UI model. For example, JavaFX might be used to render the exact same UI model as Vaadin.

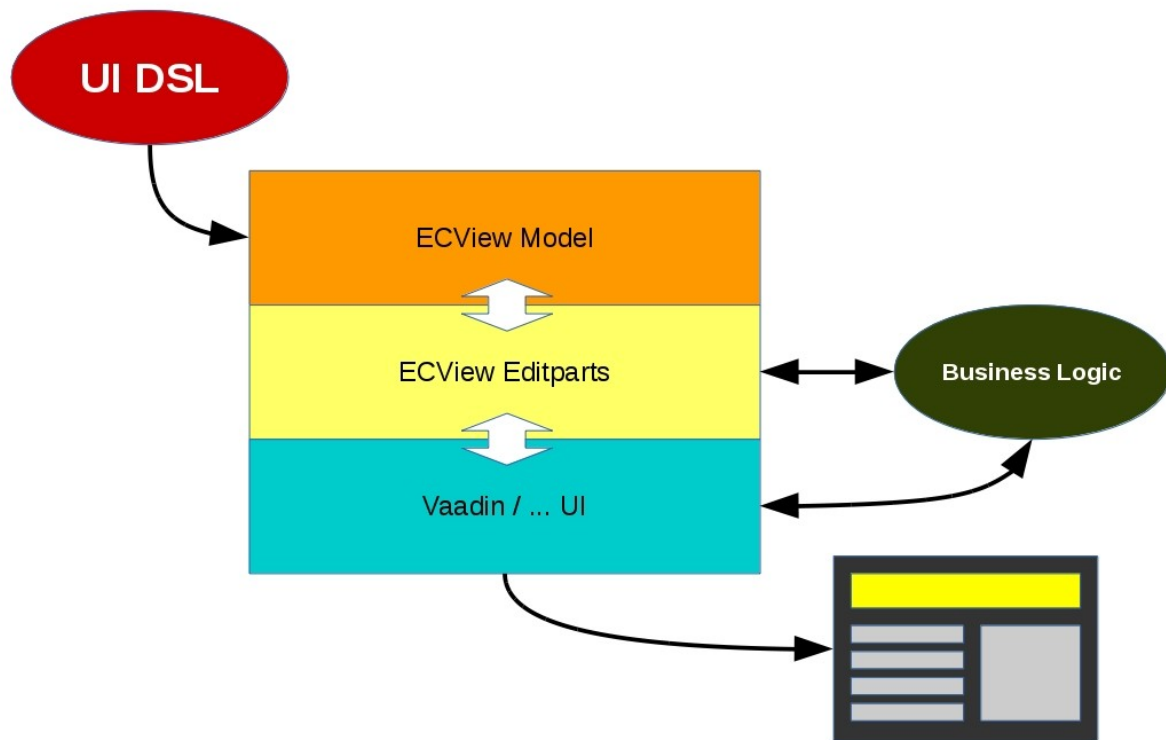
ECView is a UI model based on the Eclipse Modeling Framework (EMF). It defines various UI elements and their properties and possible relations. UI elements are thus represented as EObjects and can be manipulated accordingly. Furthermore, databinding can be done efficiently by using the Eclipse databinding mechanism.

ECView attaches so-called editparts to its model elements. Acting as controllers, these editparts tie the application's model and its visual representation together. Editparts are responsible for making changes to the model.

ECView provides support for various validation mechanisms (min-length, max-length, regex ...). The Lunifera UI DSL can make use of such validations easily and allows even the definition of custom validators using Xexpressions.

Application logic can interface with the UI model using the editparts of the respective model elements (reading values, changing visibility etc.). Furthermore, the rendered UI may be addressed

by application logic directly, if necessary.



*Architecture: The Lunifera UI DSL helps to define an ECView UI model that is propagated to the rendering layer (Vaadin) by so-called editparts.*

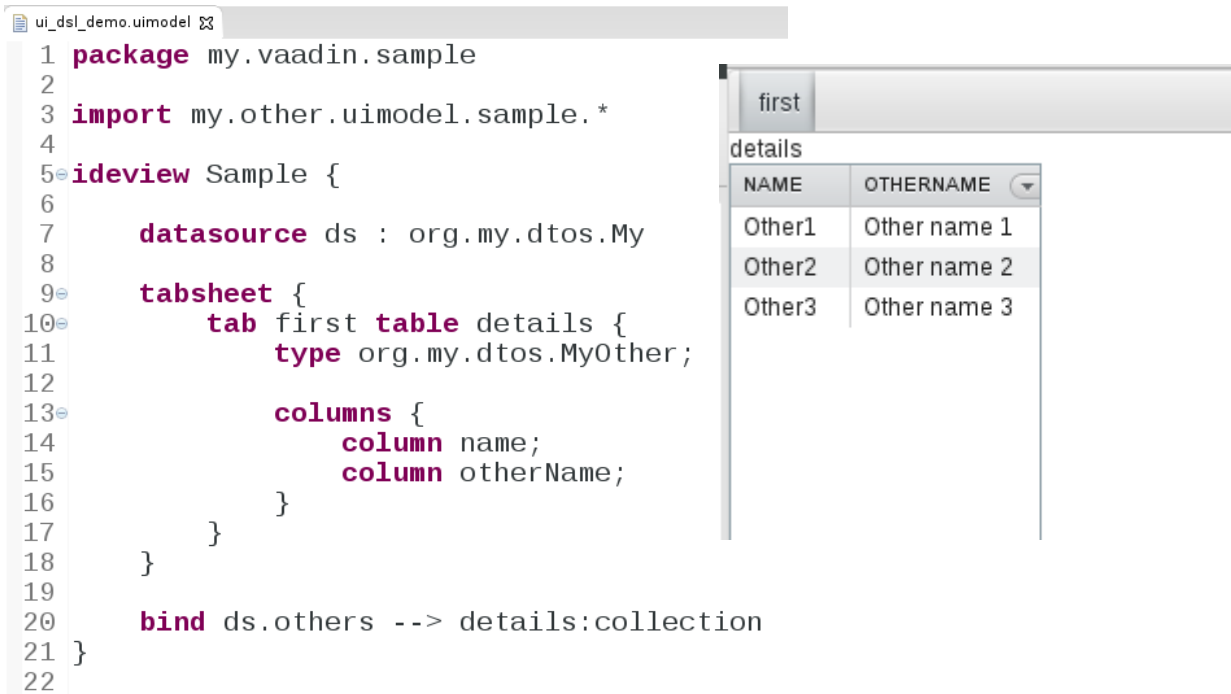
### 3 UI model files

UIs are defined in text files with the file extension “.uimodel”. These UI model files carry the information necessary to render a user interface and are made up of the following sections:

- Package declaration: The UI model file must have a unique fully qualified name that can be specified with the **package** keyword at the beginning of the file.
- Import section: UI model files can address definitions from other models (e.g. datasource or validator specifications). The model files that contain these definitions can be specified with the **import** keyword followed by the fully qualified name of the file and the “\*” wildcard.
- Validator definitions: The Lunifera UI DSL supports the use of validators that can be attached to UI elements directly in the UI model file. If validators are to be used for several UI elements, they may be defined once in this section and given an alias that can then be used to attach them to several UI elements. A validator can be defined for later use by the **validatorAlias** keyword.
- Viewset definitions: The keyword **viewset** allows for the definition of common

datasources and aliases that can be used in several views.

- The UI model: The UI is described as a tree-like containment structure, starting with the root element. This can either be a viewset (containing several views) or directly a view. The keywords for these root elements are **ideview** (for a desktop view) and **mobile** (for a view optimised for smartphones).



The screenshot shows a code editor window titled 'ui\_dsl\_demo.uimodel' with the following code:

```
1 package my.vaadin.sample
2
3 import my.other.uimodel.sample.*
4
5 ideview Sample {
6
7     datasource ds : org.my.dtos.My
8
9     tabsheet {
10         tab first table details {
11             type org.my.dtos.MyOther;
12
13             columns {
14                 column name;
15                 column otherName;
16             }
17         }
18     }
19
20     bind ds.others --> details:collection
21 }
22
```

To the right of the code editor is a preview of the rendered UI. It features a tabbed interface with a tab labeled 'first'. Below the tab is a table titled 'details' with two columns: 'NAME' and 'OTHERNAME'. The table contains three rows of data:

NAME	OTHERNAME
Other1	Other name 1
Other2	Other name 2
Other3	Other name 3

*UI model file: After the package name specification and an import declaration, a view is described. This file is automatically translated to an ECView model that is rendered by the Vaadin UI.*

## 4 UI elements

The Lunifera UI DSL keywords to define UI elements. The UI elements represent a containment tree starting with either a viewset, a view or a mobile view as topmost element. In order to reflect the containment structure, keywords for UI elements may be followed by { } braces that surround the keywords for contained elements.

The keywords for the various UI elements that are supported by the Lunifera UI DSL are documented in the following sections.

### 4.1 Top level elements

- **viewset** – This keyword introduces data sources that are shared between several views. A viewset has a name. Within the viewset definition, objects (for example, DTOs generated by the Lunifera DTO DSL) can be declared as data sources using the **datasource** keyword. Fields within data sources can be given an alias in order to facilitate addressing them. The keyword for this is **dataAlias**.

If a subsequent view definition contains the keyword **viewset** and the name of a given viewset, the data sources defined in the viewset can be addressed by elements within the view.

```
5 viewset asdf {  
6     datasource ds : org.my.dtos.My  
7     dataAlias ds.name as myname  
8 }  
9  
10 ideview Sample {  
11     viewset asdf
```

*Viewset: The viewset defines data sources that can be shared among several views. By specifying the same viewset inside a view definition, the data sources of the viewset are made accessible within the view.*

- **ideview** – This keyword defines a view. A view is the topmost UI element and can contain further UI elements. It is used as a root element for rendering the UI. In addition to UI elements, an ideview definition can contain:
  - datasource and data alias definitions (see viewset above – only that datasources declared within a view are exclusive to this view)
  - the declaration of a viewset the view belongs to (for sharing datasources)
  - validation assignments (see chapter 5)
  - databinding assignments (see chapter 6)
- **mobile** – This keyword defines a view optimised for smartphones. This “mobile” view serves as topmost UI element and can contain further UI elements. In addition to the elements supported by the ideview keyword, mobile views can manage transitions between pages. See chapter 8 for details.

```

1 package org.my.demo
2
3 viewset asdf {
4     datasource ds : org.my.dtos.My
5     dataAlias ds.name as myname
6 }
7
8 ideview Sample {
9     viewset asdf
10
11     horizontalLayout {
12         textfield Name
13         textfield Address
14     }
15
16     bind Name:value <--> myname
17     fieldValidation Name += RegexValidator("[A-Z][a-z]*")
18
19 }

```

*Ideview: The keyword ideview defines a view that serves as root element for the UI. The Model is rendered in real time by the Vaadin presentation layer. Databinding and validation mechanisms are available.*

## 4.2 Layouts

Layouts define how elements are arranged in a UI. The Lunifera UI DSL supports four basic types of UI layouts: horizontal, vertical, grid and form layouts. Layouts can be nested in order to achieve more complicated arrangements than the four basic types.

- A horizontal layout can be generated with the **horizontalLayout** keyword and can optionally be given a name. In a horizontal layout, elements are arranged next to each other with labels above the element. If the option fill-h is specified in parentheses after the keyword, the layout is expanded horizontally to fill the available space.

Syntax<sup>1</sup>:

```

horizontalLayout (fill-h) layoutname {
    // content
}

```

```

1 package my.uidsl.demo
2
3 ideview sample {
4
5     horizontalLayout {
6         textfield Name
7         textfield Street
8         decimalField Number
9     }
10
11     bind ds.others --> details:collection
12 }
13

```

*Horizontal layout: The horizontalLayout keywords arranges the fields next to each other.*

<sup>1</sup> Optional elements are shown in gray; variables in *italics*, datatypes in <angle brackets>.

- A vertical layout can be generated with the **verticalLayout** keyword and can optionally be given a name. In a vertical layout, elements are arranged above each other with labels above the element. If the option fill-v is specified in parentheses after the keyword, the layout is expanded vertically to fill the available space.

Syntax:

```
verticalLayout (fill-v) layoutname {
    // content
}
```



*Vertical layout: The verticalLayout keywords arranges the fields above each other.*

- A gridlayout can be generated with the **gridlayout** keyword and can optionally be given a name. In a gridlayout, elements are arranged in columns, the rows being filled from left to right. Element labels are displayed above the element. In parentheses after the keyword, the desired number of columns may be specified, and the options fill-h and fill-v layout to make the layout fill the available space may be given.

Syntax:

```
gridlayout (columns=<int> fill-h fill-v) layoutname {
    // content
}
```

```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     gridlayout (columns=2) {
6         textfield Name
7         textfield Street
8         decimalField Number
9     }
10 }
11

```

*Gridlayout: A gridlayout arranges its elements in rows and columns. The rows are filled from left to right, then the columns from top to bottom.*

- A form layout can be generated with the **form** keyword and can optionally be given a name. In a form layout, elements are arranged in a column. Element labels are displayed on the left side of the element.

Syntax:

```

form layoutname {
    // content
}

```

```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     form MyFormLayout {
6         textfield Name
7         textfield Street
8         numericField Number
9     }
10 }

```

*Form layout: A form layout lists the contained elements below each other, setting the label to the left of the element.*

## 4.3 Tabs

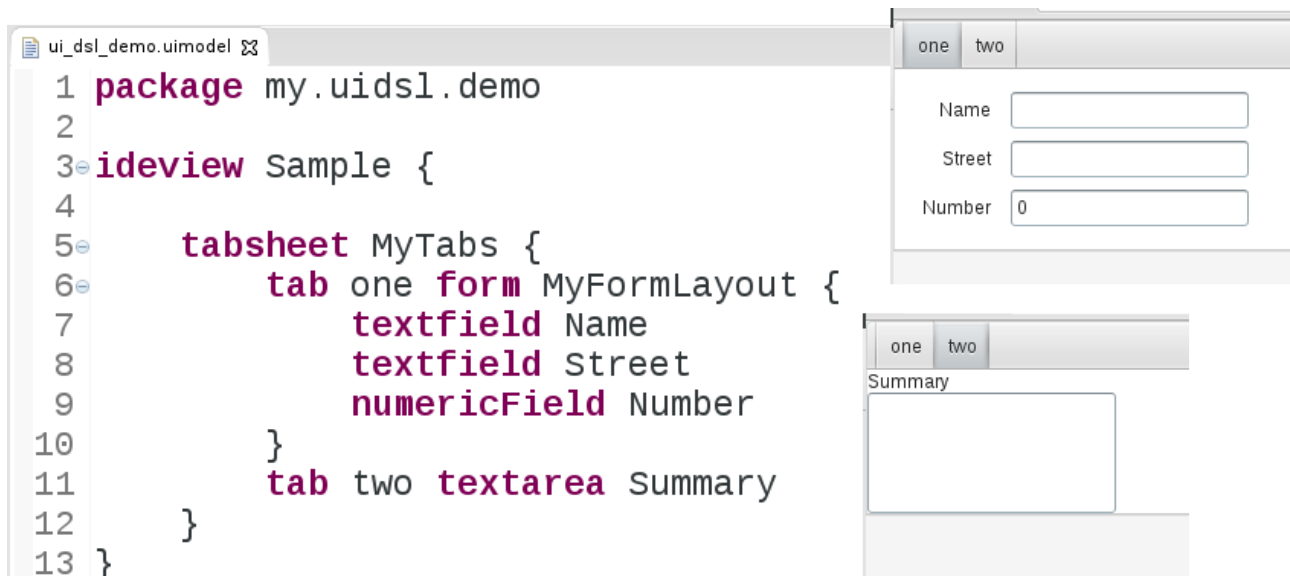
Tabs are similar to layouts insofar as they can contain other layouts. Tabs themselves have to be contained inside a tabsheet, however. The tabsheet in its turn can reside inside a layout or be directly attached to a view. An illustration of these containment possibilities can be found in Appendix 11.1.

A tabsheet is created by using the **tabsheet** keyword (followed by an optional name). The tabs contained in the tabsheet are described within the braces following the tabsheet statement: Each tab is introduced with the **tab** keyword followed by a name and the tab contents: either a single UI element or a layout with several UI elements.

Similarly to layouts, tabsheets can be nested (a tab can contain an “inner” tabsheet).

Syntax:

```
tabsheet tabsheetname {  
    tab tabname containedUiElement  
}
```



*Tabsheet: A tabsheet contains tabs that can contain either single UI elements or layouts.*

## 4.4 Simple UI elements

The Lunifera UI DSL supports a wide range of “simple” UI elements. This section documents these elements along with their appropriate options.



- Label – a label is generated with the **label** keyword. Labels can receive a name that is then displayed.

Syntax:

**label** *name*



```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         label MyLabel
7     }
8
9 }
10

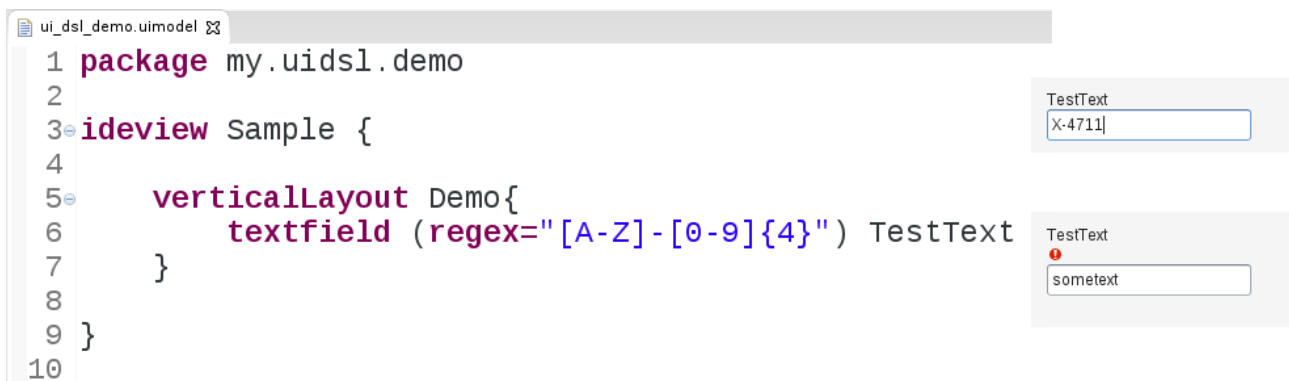
```

*A label is the most basic UI element.*

- Text field – a text field is generated with the **textfield** keyword. Textfields can receive a name that is then displayed. Furthermore, the following may be defined in parentheses between the keyword and the name:
  - maxLength: creates a maximum-length validator for this field
  - minLength: creates a minimum-length validator for this field
  - regex: creates a regular expression validator for this field

Syntax:

**textfield** (maxLength=<int>  
minLength=<int>  
regex="<String>") *name*



```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         textfield (regex="[A-Z]-[0-9]{4}") TestText
7     }
8
9 }
10

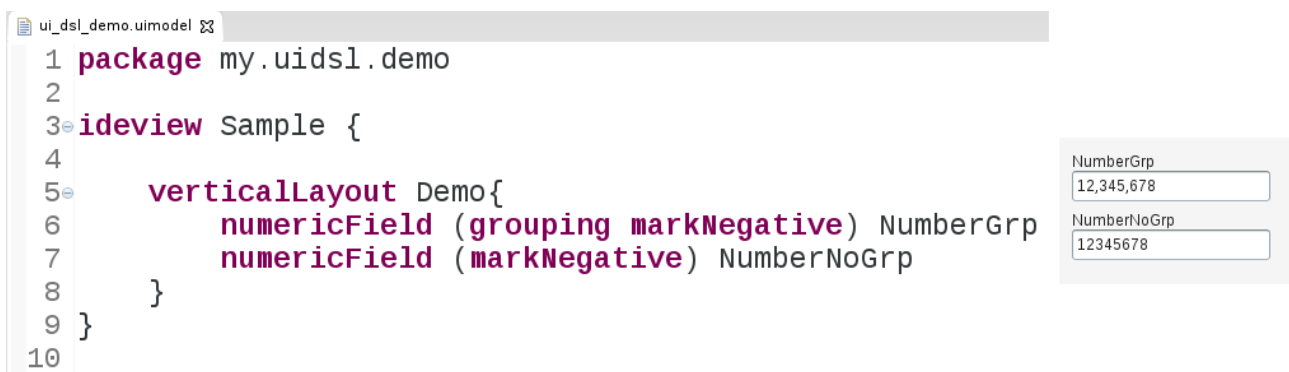
```

*Textfields can be created along with validators that check the input in the UI field for conformity to certain conditions. If the requirements are not met, an error mark is displayed.*

- Numeric field – a numeric field contains integers and is generated with the **numericField** keyword. Numeric fields can receive a name that is then displayed. Furthermore, the following may be defined in parentheses between the keyword and the name:
  - grouping: if this option is given, then the number in the field will be grouped according to the current locale.
  - markNegative: if this option is given, then the CSS class “lun-negative-value” will be added to the field if the value is negative. This can be picked up by the CSS styling and rendered appropriately later.

Syntax:

**numericField** (grouping markNegative) *name*



*Numeric fields are generated with the numericField keyword. There are options to control the number grouping and a CSS class for negative numbers.*

- Decimal field – a decimal field contains floats and is generated with the **decimalField** keyword. Decimal fields can receive a name that is then displayed. Furthermore, the following may be defined in parentheses between the keyword and the name:
  - grouping: if this option is given, then the number in the field will be grouped according to the current locale.
  - markNegative: if this option is given, then the CSS class “lun-negative-value” will be added to the field if the value is negative. This can be picked up by the CSS styling and rendered appropriately later.
  - precision: this option controls the number of digits after the decimal point to be displayed.

Syntax:

**decimalField** (grouping markNegative precision=<int>) *name*

ui\_dsl\_demo.uimodel

```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         decimalField (grouping precision=2) DecimalGrp
7         decimalField (precision=5) DecimalNoGrp
8     }
9 }
10

```

DecimalGrp

1,234,567.89

DecimalNoGrp

1234567.89000

Decimal fields are created by the `decimalField` keyword. Mechanisms for controlling the number grouping and for adding a CSS class for negative numbers are available. The desired precision can be set optionally.

- Text area – a text area is generated with the `textarea` keyword. Text areas can receive a name that is then displayed.

Syntax:

**textarea** *name*

ui\_dsl\_demo.uimodel

```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         textarea MyText
7     }
8 }

```

MyText

Text areas are created by the `textarea` keyword.

- Checkbox – a checkbox is generated with the `checkbox` keyword. Checkboxes can receive a name that is then displayed.

Syntax:

**checkbox** *name*

ui\_dsl\_demo.uimodel

```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         checkbox klickMe
7     }
8 }

```

klickMe

☐

Checkboxes are created by the `checkbox` keyword.

- Button – a push button is generated with the **button** keyword. Buttons can receive a name that is then displayed as the button text.

Syntax:

**button** *name*

```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         button klickMe
7     }
8 }

```



A button can be created with the button keyword. The name is then displayed as the button label.

- Date/time field – a date/time field is generated with the **datefield** keyword. Date/time fields can receive a name that is then displayed.

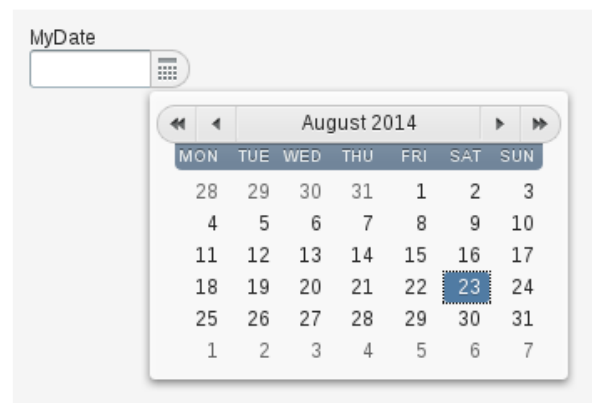
Syntax:

**datefield** *name*

```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         datefield MyDate
7     }
8 }
9

```



A date field can be created with the datefield keyword. The Vaadin UI provides a date picker along with the field.

- Progress bar – a progress bar is generated with the **progressbar** keyword. Progress bars can receive a name that is then displayed. The value that is displayed by the progressbar can be determined by databinding (see chapter 6).

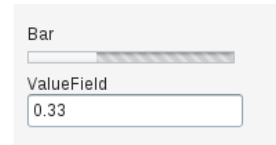
Syntax:

**progressbar** *name*

```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         progressBar Bar
7         decimalField (precision=2) ValueField
8     }
9     bind [ValueField].value --> [Bar].value
10 }

```



A progress bar can be created with the `progressbar` keyword. It is used to display values between 0 and 1. In this case, the value is retrieved via databinding from a `decimalField`.

- Table – a table is generated with the `table` keyword. Tables can receive a name that is then displayed. Tables **require** a datatype (e.g. a DTO) that can be set with the `type` keyword. The selection type (single, multi or none) can optionally be controlled with the `selectionType` keyword. If images are to be displayed in the table, the image source path within the datatype can be set with the `imageField` keyword. The values that are to be displayed can be fed into the table by databinding (see chapter 6). The columns displayed and their order are determined by the `column` keywords in the `columns` section of the table.

Syntax:

```




table name {
    type datatype
    selectionType <multi|single|none>
    imageField pathToImageFieldWithinDataType
    columns {
        column columnName
        // order determines order in UI
    }
}

```

```

1 package my.uidsl.demo
2
3 ideview Sample {
4     datasource ds : org.my.dtos.My
5
6     verticalLayout Demo{
7         table Table {
8             type org.my.dtos.MyOther
9             selectionType multi
10            imageField imagePath
11            columns {
12                column name
13                column otherName
14            }
15        }
16    }
17    bind list ds.others --> [this.Demo.Table].collection
18 }

```

	NAME	OTHERNAME
	Other1	Other name 1
	Other2	Other name 2
	Other3	Other name 3

A table is created with the `table` keyword. A datatype has to be assigned to the table. Columns and their order are specified in the `columns` section. The values are entered into the table by databinding. Images can be embedded with the `imageField` keyword.

- Options group – an options group is generated with the **optionsgroup** keyword. Options groups can receive a name that is then displayed. They **require** a datatype (e.g. a DTO) that can be set with the **type** keyword. The selection type (single, multi or none) can optionally be controlled with the **selectionType** keyword. Type “single” creates a radio button group; type “multi” creates a group of checkboxes. The expressions that are displayed next to the radio buttons respectively checkboxes can be specified by the **captionField** keyword that points to the path within the datatype. If images are to be displayed along with the options, the image source path within the datatype can be set with the **imageField** keyword. The values that are to be displayed can be fed into the options group by databinding (see chapter 6).

Syntax:

```

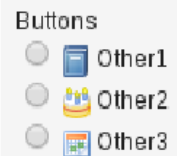
optionsgroup name {
    type datatype
    selectionType <multi|single|none>
    captionField pathToCaptionFieldWithinDataType
    imageField pathToImageFieldWithinDataType
}

```

```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     datasource ds : org.my.dtos.My
6
7     verticalLayout Demo{
8
9         optionsgroup Buttons {
10             type org.my.dtos.MyOther
11             captionField name
12             imageField imagePath
13         }
14     }
15
16     bind list ds.others --> [this.Demo.Buttons].collection
17 }

```



Options groups (respectively radio button groups) are created with the **optionsgroup** keyword. A datatype has to be assigned to the group. The values are entered into the optionsgroup by databinding. The expression displayed for an object can be controlled by the **captionField** keyword, and images are embedded with the **imageField** keyword.

- Combo boxes – a combo box is generated with the **combo** keyword. Combo boxes can receive a name that is then displayed. They **require** a datatype (e.g. a DTO) that can be set with the **type** keyword.

The expressions that are displayed in the dropdown can be specified by the **captionField** keyword that points to the path within the datatype. If images are to be displayed along with the options, the image source path within the datatype can be set with the **imageField** keyword.

The values that are to be displayed can be fed into the combo box by databinding (see chapter 6).

Syntax:

```

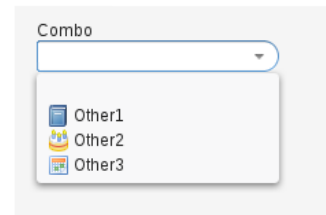
combo name {
    type datatype
    captionField pathToCaptionFieldWithinDataType
    imageField pathToImageFieldWithinDataType
}

```

```

1 package my.uidsl.demo
2
3 ideview Sample {
4     datasource ds : org.my.dtos.My
5
6     verticalLayout Demo{
7
8         combo Combo {
9             type org.my.dtos.MyOther
10            captionField name
11            imageField imagePath
12        }
13    }
14    bind list ds.others --> [this.Demo.Combo].collection
15 }

```



Combo boxes are created with the `combo` keyword. A datatype has to be assigned to the combo box. The values are entered into the combo box by databinding. The expression displayed for an object can be controlled by the `captionField` keyword, and images are embedded with the `imageField` keyword.

## 5 Validation

The Lunifera UI DSL supports four types of validators: minimum and maximum length validators, regular expression validators and Xbase expression validators. The first three (simple) types of validators can be assigned to textfields as options in a short-hand fashion (see section 4.4).

For the fourth type of validator and for validators on fields other than textfields, the generic way of implementing validators has to be introduced: Validators can be declared at the end of a view using the `fieldValidation` keyword and the following syntax:

```
fieldValidation fieldname += validator
```

where `validator` can be either a validator type with the appropriate options or a validator alias.

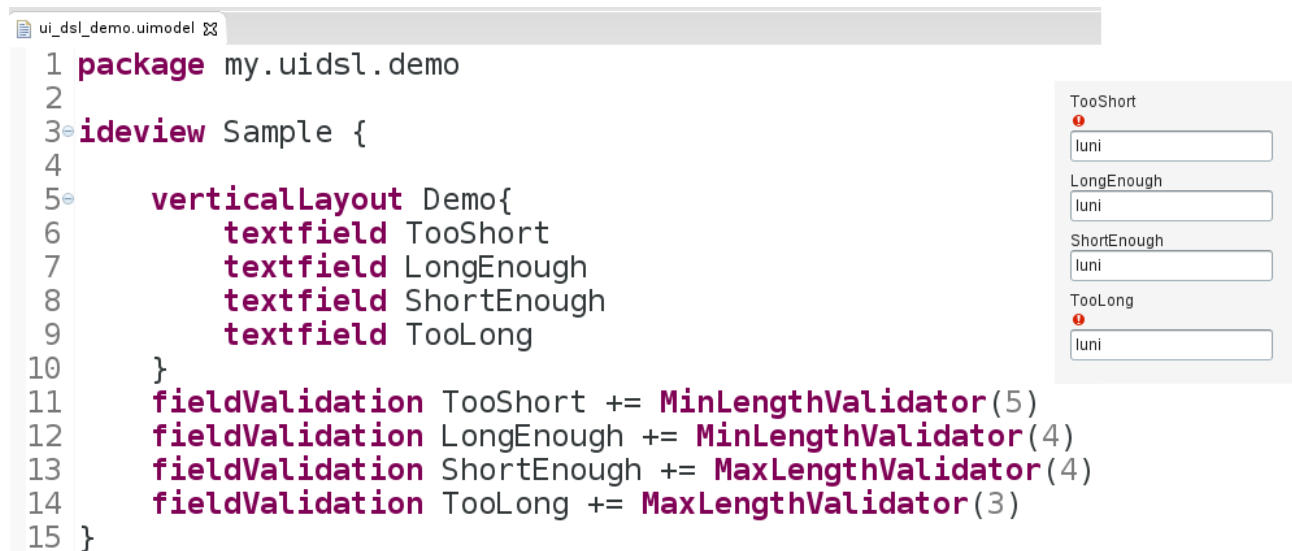
The possible options are:

- `MinLengthValidator(<int>)` – the minimum number of characters the value in the field must have
- `MaxLengthValidator(<int>)` – the maximum number of characters the value in the field may have
- `RegexValidator("<String>")` – a regular expression that the value must match
- `Expression(inputDatatype) { expression }` – a freely programmable expression that determines the behaviour of the validator. In the expression, the keyword `input` may be used to reference the value in the validated field. Typically, the expression



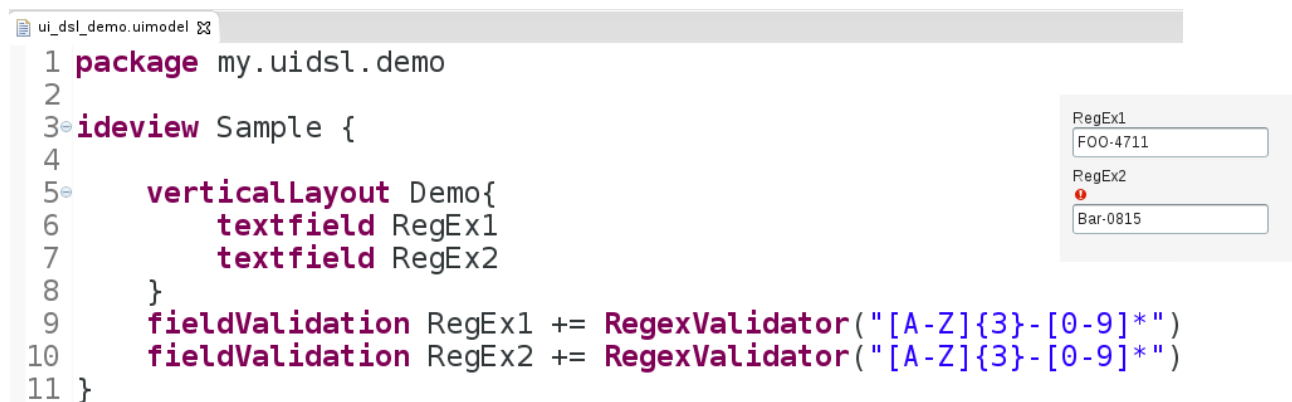
contains an if-clause that matches invalid values. In this case, the function `error("<errorCode>", "<errorMessage>")` is used to mark the value as invalid and display an error. Returning `org.eclipse.emf.ecp.ecview.common.validation.Status.OK` breaks the validation process off and marks the field as valid.

Examples:



```
1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         textfield TooShort
7         textfield LongEnough
8         textfield ShortEnough
9         textfield TooLong
10    }
11    fieldValidation TooShort += MinLengthValidator(5)
12    fieldValidation LongEnough += MinLengthValidator(4)
13    fieldValidation ShortEnough += MaxLengthValidator(4)
14    fieldValidation TooLong += MaxLengthValidator(3)
15 }
```

*MinLength and MaxLength validators take integer arguments. If they are attached to fields, the field is marked as invalid if the validation condition is not met.*



```
1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         textfield Regex1
7         textfield Regex2
8     }
9     fieldValidation Regex1 += RegexValidator("[A-Z]{3}-[0-9]*")
10    fieldValidation Regex2 += RegexValidator("[A-Z]{3}-[0-9]*")
11 }
```

*A regex validator takes a String as an argument. It marks a field as invalid if the field's content does not match the specified regex.*

```

1 package my.uidsl.demo
2
3 import org.eclipse.emf.ecp.ecview.common.validation.*
4
5 ideview Sample {
6
7     verticalLayout Demo{
8         numericField OddNumber
9         numericField EvenNumber
10    }
11
12    fieldValidation OddNumber += Expression IsEven (Double) {
13        if (input == null) return Status.OK
14        if (input % 2 == 0)
15            error("ERR-1234", "Error message")
16    }
17    fieldValidation EvenNumber += Expression IsOdd (Double) {
18        if (input == null) return Status.OK
19        if (input % 2 != 0)
20            error("ERR-1234", "Error message")
21    }
22 }

```

*Expression validators need a name and a source datatype. The value in the UI field can be addressed with the input keyword. Negative validation is done by calling the error method. Null pointer exceptions that would occur with empty fields can be avoided by returning Status.OK.*

## 6 Databinding

The Lunifera UI DSL possesses very powerful databinding mechanisms. Values can not only be bound from one UI element to another, but also from a UI element to a general object (datasource); even binding values to commands is possible.

The keyword for databinding is **bind** followed by the two binding endpoints between which an arrow determines the direction of the binding. For lists and list-like objects, the keyword **bind list** can be used. The binding endpoints are defined by a dot notation following the field in brackets. Binding statements can be placed either directly with the UI element that is bound or at the end of the view that contains it.

Datasources for the binding can be defined at the beginning of the view description; aliases for fields within the datasource objects can be defined either along the datasource definition or with the binding statement.

There are a few pre-defined binding endpoints that can instantly be used:

- **value** – provides a binding endpoint for the value of a field
- **editable** – provides a binding endpoint for the “editable” state of a field

- visible – provides a binding endpoint for the “visible” state of a field
- collection – provides a binding endpoint for a list
- selection – provides a binding endpoint for a selected item within a list
- multiselection – provides a binding endpoint for a list of items selected from another list

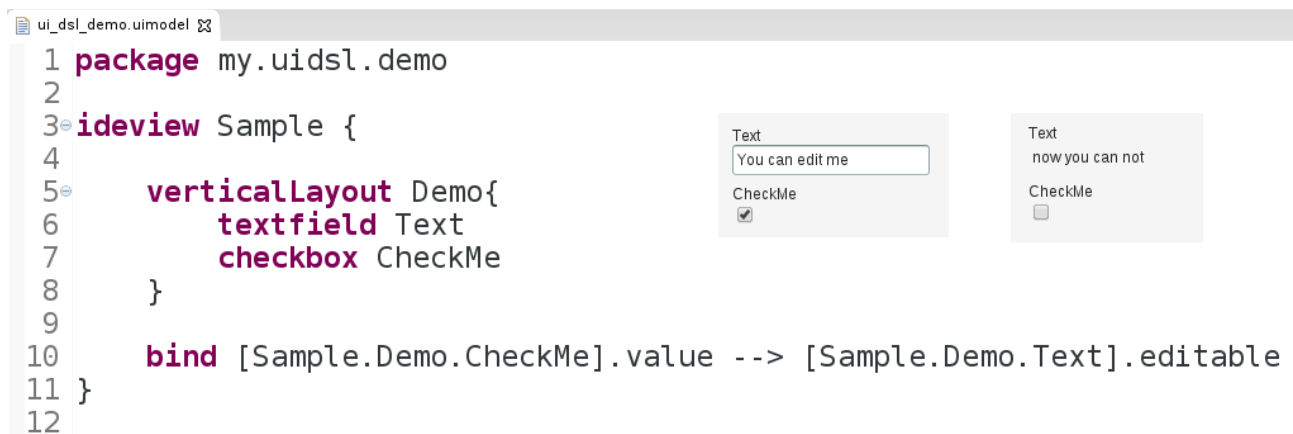
Syntax:

```
bind list [path.to.field].bindingEndpoint
        <Direction>
        [path.to.field].bindingEndpoint
```

where <Direction> is one of <--, <--> or -->.

If the binding statement is given directly with a UI element, the keyword **this** can be used as a shorthand for the path to the field.

Examples:



The screenshot shows a code editor window titled 'ui\_dsl\_demo.uimodel'. The code defines a package 'my.uidsl.demo', an ideview 'Sample', and a vertical layout 'Demo' containing a 'textfield' 'Text' and a 'checkbox' 'CheckMe'. A binding statement is shown: 'bind [Sample.Demo.CheckMe].value --> [Sample.Demo.Text].editable'. To the right of the code, two UI mockups are displayed. The left mockup shows the 'Text' field with the value 'You can edit me' and the 'CheckMe' checkbox checked. The right mockup shows the 'Text' field with the value 'now you can not' and the 'CheckMe' checkbox unchecked, demonstrating the one-way binding where the checkbox's state controls the text field's editability.




```
1 package my.uidsl.demo
2
3 ideview Sample {
4
5     verticalLayout Demo{
6         textfield Text
7         checkbox CheckMe
8     }
9
10    bind [Sample.Demo.CheckMe].value --> [Sample.Demo.Text].editable
11 }
12
```

*Binding between UI elements: The value of the checkbox determines whether the textfield is editable. This is a one-way binding.*

```

1 package my.uidsl.demo
2
3 ideview Sample {
4
5     datasource ds : org.my.dtos.My
6
7     verticalLayout Demo{
8         table MyTable {
9             type org.my.dtos.MyOther
10            imageField imagePath
11            columns {
12                column name
13                column othername
14            }
15            bind ds.others --> [this].collection
16        }
17    }
18 }

```




	NAME	OTHERNAME
	Other1	Other name 1
	Other2	Other name 2
	Other3	Other name 3




Binding between DTOs and UI: The values of the DTO fields are taken into the collection that makes up the table values. Since the binding statement is located within the table declaration, the keyword “this” can be used. This is a one-way binding.




```




17     }
18     table MyTable2 {
19         type org.my.dtos.MyOther
20         imageField imagePath
21         selectionType multi
22         columns {
23             column name
24             column othername
25         }
26         bind ds.others --> [this].collection
27     }
28 }
29 bind list [Sample.Demo.MyTable].multiSelection
30 <--> [Sample.Demo.MyTable2].multiSelection
31 }

```

	NAME	OTHERNAME
	Other1	Other name 1
	Other2	Other name 2
	Other3	Other name 3

	NAME	OTHERNAME
	Other1	Other name 1
	Other2	Other name 2
	Other3	Other name 3

	NAME	OTHERNAME
	Other1	Other name 1
	Other2	Other name 2
	Other3	Other name 3

	NAME	OTHERNAME
	Other1	Other name 1
	Other2	Other name 2
	Other3	Other name 3

Expanding the previous example: A second (identical) table is created, and a list binding between the multi-selection endpoints of the tables is defined. This is a two-way binding.

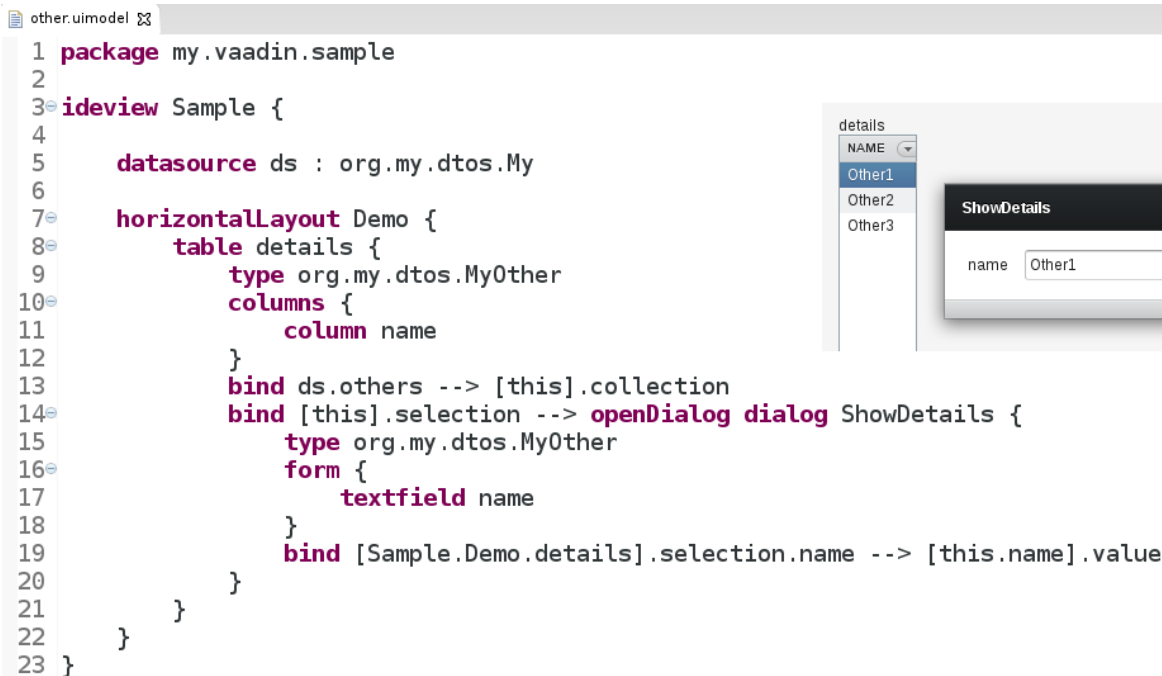
## 7 Commands

In its present version, the Lunifera UI DSL has three commands for UI actions: **openDialog** for opening a user dialog, **searchWith** for opening a search dialog, and **navigateTo** for moving to a subsequent navPage (for the last, see chapter 8, mobile UI).

Commands are embedded into the UI model by bindings. The syntax for this is

```
bind [path.to.field].endpoint --> command options
```

Dialogs are UI elements that are laid over the UI. They can contain layouts with other UI elements that interact with the UI using databinding.



Binding the command "openDialog" to the selection in a table makes a dialog window pop up when an entry is selected (Line 14). The data that triggered the dialog can be referenced with the `[this].value` keyword from within the dialog. Thus, it would be possible to replace the expression in line 19 by `bind [this].value.name --> [this.name].value`.

## 8 Mobile UI

The Lunifera UI DSL supports a variety of elements that are optimised for display on mobile devices. Mobile UIs are described in .uimodel files similar to regular UI models. The main difference is that the top level element in a mobile UI is a “mobile view”, created by the keyword **mobile**.

### 8.1 Mobile layouts

There are a few layouts specifically for mobile UIs:

- Navigation page – a navigation page is a very important top level layout within a mobile view. It is created by the **navPage** keyword. Navigation pages can be linked together hierarchically via databinding, which allows the creation of a tree of pages that can be used for browsing details on a screen of reduced size. The keyword for navigation buttons is **navButton** (see mobile navigation 8.3 below).

Syntax:

```
navPage name {  
    // content  
}
```

- 

- Vertical component group – fields can be grouped together under a common heading with the **verticalGroup** keyword.

Syntax:

```
verticalGroup name {  
    // content  
}
```

```
ui_dsl_demo.uimodel x  
1 package my.uidsl.demo  
2  
3 mobile Sample {  
4     navPage Main {  
5         verticalGroup Person {  
6             form Demo {  
7                 textfield FirstName  
8                 textfield LastName  
9             }  
10        }  
11        verticalGroup Details {  
12            form Demo {  
13                textfield Address  
14                textfield City  
15            }  
16        }  
17    }  
18 }  
19 }
```

Main
Person
FirstName
LastName
Details
Address
City

*A vertical group puts fields under a common header. In this example, it is located within a single navigation page.*

- Mobile tabsheet – a mobile tabsheet is similar to the regular tabsheet with the big difference that the tab selection is at the bottom. It is created by the **mobileTab** keyword. Tabs containing either single elements or layout with several elements can be inserted into the mobile tabsheet by describing them within the braces.

Syntax:

```
mobileTab name {  
    tab tabName containedUiElement  
}
```

```

1 package my.uidsl.demo
2
3 mobile Sample {
4     mobileTab Main {
5         tab first verticalGroup Person {
6             form Demo {
7                 textfield FirstName
8                 textfield LastName
9             }
10        }
11       tab second verticalGroup Details {
12           form Demo {
13               textfield Address
14               textfield City
15           }
16       }
17   }
18 }

```

A mobile tabsheet is created with the keyword `mobileTab`. Each tab can contain a UI element or a layout (in this case, a vertical group).

## 8.2 Mobile-only element

A mobile layout can contain all elements that a regular layout can contain plus (at the moment) one specific “mobile-only” element:

- Mobile switch – a mobile switch is generated with the `switchIt` keyword. Mobile switches are equivalent to checkboxes, but look much better in mobile UIs.

Syntax:

**switchIt** *name*

```

1 package my.uidsl.demo
2
3 mobile Sample {
4     navPage Main {
5         form {
6             textarea Message
7             switchIt RequestReceipt
8         }
9     }
10 }

```

A “sliding style” switch can be created with the “switchIt” keyword.

## 8.3 Mobile navigation

One way of navigation between pages in a mobile UI has already been shown: the navigation between tabs in a `mobileTab` environment (see above).

Another method of navigation in a mobile UI can be achieved by `navPages` that are linked by databinding. The Lunifera UI DSL automatically adds a “Back” button to children pages.

Syntax for `navPage` navigation:

```
navButton buttonName navPage pageName {  
    // navigation page content  
}
```

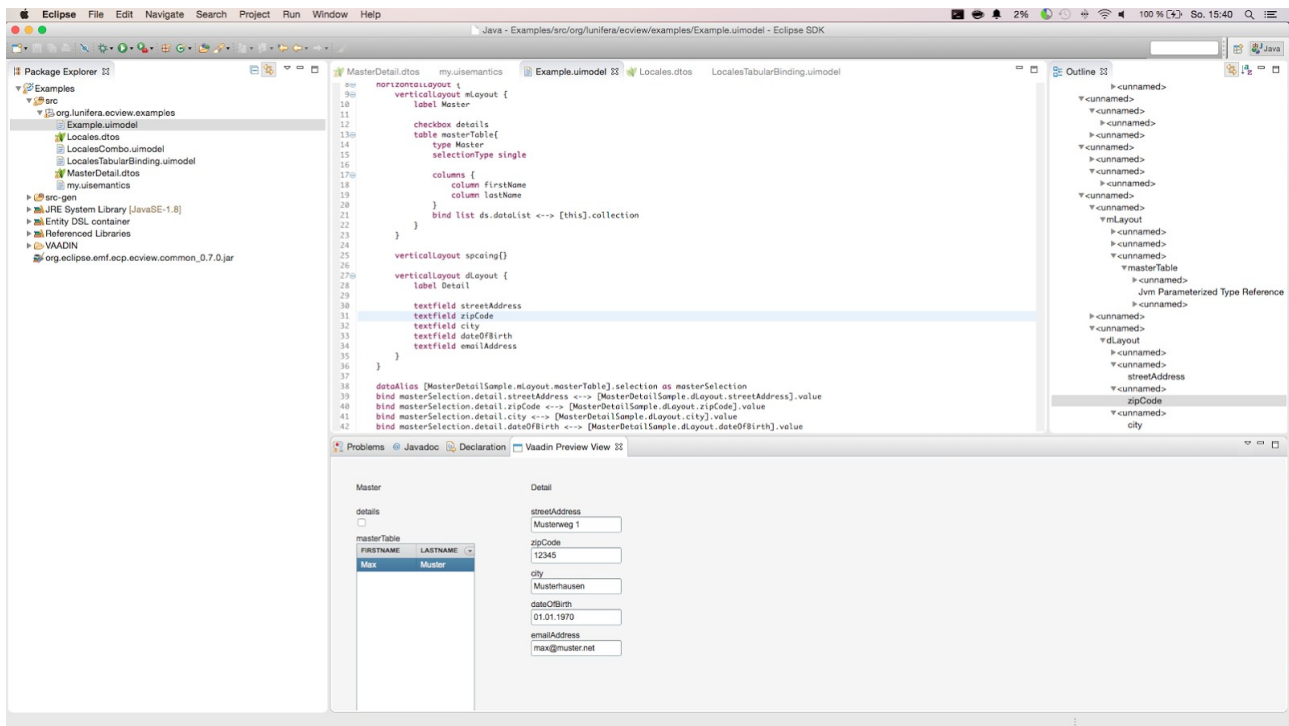
The screenshot shows the Lunifera UI DSL editor with a code snippet on the left and a corresponding UI preview on the right. The code defines a mobile application structure with a `mobileTab` containing two tabs: `first` and `second`. The `first` tab contains a `verticalGroup` with `Information`, `navButton` `Person`, and a `navPage` `Person` containing `verticalGroup` `Details` with `textfield` `Name` and `textfield` `Position`. The `second` tab contains a `navButton` `Company` and a `navPage` `Company` containing `verticalGroup` `Details` with `textfield` `Founder`. The UI preview on the right shows a mobile interface with a `mobileTab` containing two tabs: `first` and `second`. The `first` tab shows a `verticalGroup` with `Information`, `Person` (with a right arrow), and `Company` (with a right arrow). The `second` tab shows a `navPage` `Person` with a `verticalGroup` `Details` containing `textfield` `Name` and `textfield` `Position`. A `Back` button is visible at the bottom of the `Person` page.

Navigation pages can be reached by navigation buttons. The page has to be defined immediately after the button. A "back" button is automatically generated and inserted into the child page for navigating back to the parent page.

## 9 Preview UI

The Lunifera UI DSL comes with a preview UI. It relies on the appropriate Vaadin widgetsets to be present in order to render the UI model in real time. The preview can either be displayed in an Eclipse view or in an external browser. The port that is used by the preview UI can be determined with the `-Dorg.osgi.service.http.port=<int>` command line option.





*The preview UI can be displayed within the Eclipse IDE and renders the UI model file in real time.*

## 10 Installation

The Lunifera UI DSL can be installed into Eclipse – along with all necessary runtime features – by adding the following site to software sites and installing the all-in-one feature:

<http://lun.lunifera.org/downloads/p2/lunifera/allinonesdk/>

This will pull in all dependencies for running the Lunifera DSLs in the Eclipse IDE.

## 11 Appendix

### 11.1 Containment graph

The following illustration shows how the various elements of the Lunifera UI DSL can be contained in each other:

