User's Handbook

map.apps with MD2

Project Seminar

Model-driven Mobile Development

(University of Münster)

24th February 2015

Contents

1 Introduction

2 Modeler's Handbook

2.1 Installation

The following software is required prior to the next steps:

- map.apps 3.1.0
- Eclipse IDE for Java and DSL Developers (e.g. Version Kepler)
- NetBeans EE (e.g. Version 8.0)
- Apache Tomcat 7.0 with running map.apps runtime
- GlassFish 4.+

2.2 Getting Started

2.2.1 Developing a Single App

In this section it is described how an application can be developed based on the current (March 2015) state of the MD2 DSL. To this end, this section is structured according to the different layers which constitute MD^2 model and are represented in Figure 2.1. Aside from the conventional MVC layers – model, view and controller – this includes an additional layer, which enables and models the generation of workflows within and across apps and was established during the project seminar.

All components in MD^2 are organized in a package structure that represents the aforementioned structure. All documents have to be placed in corresponding packages (views, models, controllers or workflow). For example, all view files are expected to be in the package any.project.package.views. The package has to be defined in each MD^2 file as follows:

```
package PACKAGE_NAME
```

The package name has to be a fully qualified name that reflects the actual folder structure.

Workflow

The workflow layer is an additional abstraction on top of the controller layer. It thereby allows to specify the general course of action of one or more apps with few simple and well understandable model

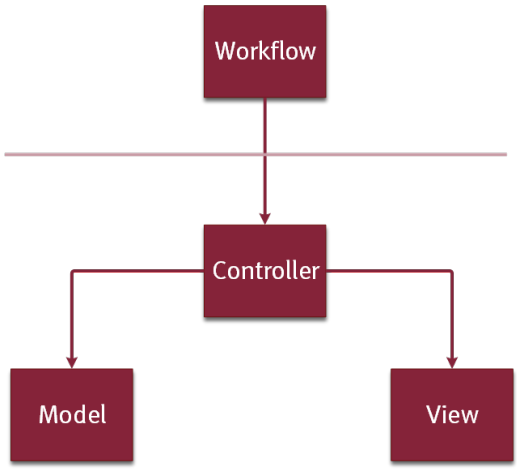

Figure 2.1: Architecture of MD^2 Models

language constructs. Furthermore, this abstract workflow representation is intended to serve as a basis for communication with customers, e.g. for requirements engineering and collaborative app development.

In the workflow file of an MD2 model a workflow can be specified as a (possibly cyclic) directed graph of workflow elements. Workflow elements represent encapsulated functionality which is specified in detail in the controller layer. The workflow layer references the workflow elements from the controller layer to define their interaction.

For this purpose, workflow elements are linked via events. For each workflow element one or more events can be specified that can be fired. However, at runtime a workflow element can fire only one of these events, i.e. a parallel processing of the workflow is not intended. Similar to the workflow elements, workflow events in the workflow layer are references to workflow events that are created in the respective controller.

In addition to the events that can be fired, the workflow element also specifies which workflow element is to be started in response to a fired event using the keyword `start`. Moreover, when an event is fired, not only workflow elements can be started but the workflow can be terminated by using the `end workflow` keyword. A workflow element in the workflow layer typically looks as shown in Listing 2.1.

Listing 2.1: Workflow Elements in the Workflow Layer

```
WorkflowElement <NameOfWorkflowElement>
      fires <NameOfEventOne> {
            start <NameOfSubsequentWfeOne>
      }
      fires <NameOfEventTwo> {
            start <NameOfSubsequentWfe>
      }
```

After defining the sequence of workflow elements, the workflow also requires the specification of an app. As shown in Listing 2.2, an app consists of its ID, a list of workflow elements that are used in the app and a name that is to be used as app titel. For the scenario where only a single app is modeled, all workflow elements can be listed in the app. However, it is also possible to have unused workflow elements.

Listing 2.2: App Definition in MD2

```
App <AppID> {
      WorkflowElements {
            <WorkflowElementOne>,
            <WorkflowElementTwo> (startable: STRING),
            <WorkflowElementThree>
      }
      appName STRING
}
```

A workflow has one or more entry points, i.e. startable workflow elements. These are marked as `startable` in the app specification. During code generation this will result in a button on the app's start screen that starts the corresponding workflow element. In addition, aa alias needs to be provided which is used as label or description for the button.

Finally, the complete workflow specification for one app will be structured as shown in Listing 2.3. Note that MD2 does not differentiate between different workflows. However, it is possible to implicitly create multiple workflows by using two or more startable workflow elements that start independent, disjunct sequences of workflow elements.

Listing 2.3: Workflow Definition in MD2

```
package <ProjectName>.workflows

WorkflowElement <WorkflowElementOne>
[...]
WorkflowElement <WorkflowElementTwo>
[...]
WorkflowElement <WorkflowElementThree>
[...]

App <AppID> {
      [...]
}
```

Model

In the model layer the structure of data objects is being described. As model elements Entities and Enums are supported.

Entity An entity is indicated by the keyword `entity` followed by an arbitrary name that identifies it.

```
entity NAME {
<attribute1 ... attribute n>
}
```

Each entity may contain an arbitrary number of attributes of the form

```
ATTRIBUTE_NAME: <datatype>[] (<parameters>) {
name STRING
description STRING
}
```

The optional square brackets [] indicate a one-to-many relationship. That means that the corresponding object may hold an arbitrary number of values of the given datatype. Supported complex data types are:

- Entity
- Enum

Supported simple data types are:

- `integer` – integer
- `float` – float of the form #.#
- `boolean` – boolean
- `string` – a string that is embraced by single quotes (') or double quotes (")
- `date` – a date is a string that conforms the following format: `YYYY-MM-DD`
- `time` – a time is a string that conforms the following format: `hh:mm:ss[(+|-)hh[:mm]]`
- `datetime` – a date time is a string that conforms the following format: `YYYY-MM-DDThh:mm:ss [(+|-)hh[:mm]]`

Parameters are optional and will be transformed into implicit validators during the generation process. They have to be specified as a comma-separated list. On default each specified attribute is mandatory. To allow null values the parameter optional can be set. Further supported parameters depend on the used data type and are explained as follows:

- `integer` supports
  - `max INTEGER` – maximum allowed value of the attribute
  - `min INTEGER` – minimum allowed value of the attribute
- `float` supports
  - `max FLOAT` – maximum allowed value of the attribute

`min` `FLOAT` – minimum allowed value of the attribute

- `string` supports

  `maxLength` `INTEGER` – maximal length of the string value

  `minLingth` `INTEGER` – minimal length of the string value

Optionally, attributes can be annotated with a name and a description which are used for the labels and the tooltips in the auto-generation of views. If a tooltip is annotated a question mark will be shown next to the generated input field. If no name is annotated, a standard text for the label will be derived from the attribute's name by transforming the camel case name to natural language. E.g. the implicit label text of the attribute firstName is "First name".

Exemplary entity that represents a person:

```
entity Person {
      name: string
      birthdate: date {
            name: "Date of Birth"
            description: "The exact day of birth of this person."
      }
      salary: float (optional, min 8.50, max 1000)
      addresses: Address[]
}
```

Enum An enumeration is indicated by the keyword `enum` followed by an arbitrary name that identifies it. Each enum may contain an arbitrary number of comma-separated strings. Other data types are not supported. Exemplary enum element to specify weekdays:

```
enum Weekday {
"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"
}
```

View

View elements are either ContentElements or ContainerElements that can contain other content or container elements. Furthermore, basic styles for some content elements can be defined.

Container Elements *Grid layouts* align all containing elements in a grid. Elements can either be containers or content elements. The grid is populated row-by-row beginning in the top-leftmost cell.

```
GridLayoutPane NAME (<parameters>) {
      <Container | Content | [Container] | [Content]>
      <Container | Content | [Container] | [Content]><...>
```

```
}
```

For each grid layout at least the number of rows or the number of columns has to be specified. If only one of these parameters is given, the other is calculated by MD^2 on generation time. In case that both parameters are specified and there are too few cells, all elements that do not fit in the layout will be discarded. The following comma-separated parameters are supported:

- `columns INTEGER` – the number of columns of the grid
- `rows INTEGER` – the number of rows of the grid
- `tabIcon PATH` – if the layout is a direct child of a TabbedPane, an icon can be specified that is displayed on the corresponding tab. See section on TabbedPanes for more details.
- `tabTitle STRING` – if the layout is a direct child of a TabbedPane, a text can be specified that is displayed on the corresponding tab. See section on TabbedPanes for more details.

A *flow layout* arranges elements (containers or content elements) either horizontally or vertically. By default all elements are arranged in a left-to-right flow.

```
FlowLayoutPane NAME (<parameters>) {
      <Container | Content | [Container] | [Content]>
      <Container | Content | [Container] | [Content]>
      <...>
}
```

The following comma-separated parameters are supported:

- `vertical` or `horizontal` (default) – flow direction
- `tabIcon PATH` – if the layout is a direct child of a TabbedPane (which are described subsequently), an icon can be specified that is displayed on the corresponding tab.
- `tabTitle STRING` – if the layout is a direct child of a TabbedPane, a text can be specified that is displayed on the corresponding tab.

A *tabbed pane* is a special container element that can only contain container elements. Each contained container will be generated as a separate tab. Due to restrictions on the target platforms, tabbed panes can only be root panes, but not a child of another container element. By default the title of each tab equals the name of the contained containers. By using the tabTitle and tabIcon parameters the appearance of the tabs can be customized.

```
TabbedPane NAME {
      <Container | [Container]>
      <Container | [Container]>
      <...>
}
```

Content Elements *Input elements* can be used to manipulate model data via mappings (see controller section). At the moment text inputs, dropdown fields and checkboxes are supported. All input elements support the optional attributes label and tooltip that can be used to create compound input fields with a label and a help text button added.

Text input fields can be used for freetext as well as date and time inputs.

```
TextInput NAME {
      label STRING
      tooltip STRING
      type <textfield_type>
}
```

Besides the tooltip and label attribute, a text field type can be specified to influence the appearance of the actual input field. The following text field types are supported:

- `default` - display a standard input field (this is the default)
- `date` - display a date picker
- `time` - display a time picker
- `timestamp` - display a combined date and time picker

Option inputs are used to represent enumeration fields in the model.

```
OptionInput NAME {
      label STRING
      tooltip STRING
      options [Enum]
}
```

Besides the tooltip and label attribute, option inputs support the optional options attribute. This can be used to populate the input with the string values of the specified Enum. If options is not given, the displayed options depend on the Enum type of the attribute that has been mapped on the input field (see Section 2.2.1).

Check boxes are used as a representation for boolean model attributes.

```
CheckBox NAME {
      label STRING
      tooltip STRING
      checked BOOLEAN
}
```

Besides the tooltip and label attribute, check boxes provide the optional attribute checked that allows to specify whether the checkbox is checked by default or not. This setting will be overruled by actual values loaded from the model.

Labels allow the modeler to present text to the user. Often they are used to denote input elements. For the label definition there exists the following default definition

```
Label NAME {
      text STRING
      style <[style] | style>
}
```

as well as this shorthand definition

```
Label NAME (STRING){
      style <[style] | style>
}.
```

The text can either be annotated as an explicit text attribute or the label text to display can be noted in parentheses directly after the label definition. The optional style can either be noted directly or an existing style definition can be referenced (styles are described later in this section).

Tooltips allow the modeler to provide the user with additional information. For the tooltip definition there exists the following default definition

```
Tooltip NAME {
      text STRING
}
```

as well as this shorthand definition that allows to note the help text in parentheses directly after the label definition

```
Tooltip NAME (STRING).
```

Buttons provide the user the possibility to call actions that have been bound on events of the Button. For the button definition there exists the following default definition

```
Button NAME {
      text STRING
}
```

as well as this shorthand definition that allows to specify the text in parentheses directly after the button definition

```
Button NAME (STRING).
```

For the *image* exists the following default definition

```
Image NAME {
      src PATH
      height INT
```

```
      width INT
}
```

as well as this shorthand definition that allows to specify the image path in parentheses directly after the image name

```
Image NAME (PATH) {
      height INT
      width INT
}.
```

Images support the following attributes:

- `src` – Specifies the source path where the image is located. The path has to be relative to the directory /resources/images in the folder of the MD^2 project
- `height` (optional) – Height of the image in pixels
- `width` (optional) – Width of the image in pixels

A *Spacer* is used in a `GridLayoutPane` to mark an empty cell or in a `FlowLayoutPane` to occupy some space. Using an optional additional parameter the actual number of spacers can be specified.

```
Spacer (INT)
```

The *AutoGenerator* is used to automatically generate view elements to display all attributes of a related entity and the according mappings of the view elements to a Content Provider. It is possible to either exclude attributes using the `exclude` keyword or to provide a positive list of attributes using the keyword `only`.

```
AutoGenerator NAME {
      contentProvider [ContentProvider] (exclude|only [Attribute])
}
```

In case of one-to-many relationships for attributes (annotated with []) or a content provider it has to be defined which of the elements should be displayed in the auto-generated fields. The *EntitySelector* allows the user to select an element from a list of elements. The attribute `textProposition` defines which ContentProvider stores the list and which attribute of the elements shall be displayed to the user to allow him to find the desired element.

```
EntitySelector NAME {
      textProposition [ContentProvider.Attribute]
}
```

Styles can be annotated to several view elements such as labels and buttons to influence their design. They can either be defined globally as a root element in the view and then be referenced or annotated directly to the appropriate elements.

```
style NAME {
      color <color>
      fontSize INT
      textStyle <textstyle>
}
```

The following optional style attributes are supported. If a attribute is not set, the standard setting is used for each platform.

- `color <color>` (optional) – specifies the color of the element as a named color or a six or eight digit hex color (with alpha channel)
- `fontSize INT` (optional) – specifies the font size
- `textStyle <textstyle>` (optional) – the text style can be normal or italic, bold or a combination of both.

As named colors the 16 default web colors are supported: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, yellow.

Elements can not only be defined where they should be used, but there is also a mechanism of defining an element once and *reusing* it several times. Instead of defining a new element, another element can be referenced – internally this leads to a copy of the actual element. However, names have to be unique so that each element could only be referenced once. To avoid those name clashes a renaming mechanism had been implemented that allows to set new names for the actual copied element.

```
Element -> NAME
```

Controller

Main The `main` object contains all basic information about a project. Each project must contain exactly one `main` object that can be in an arbitrary controller.

```
main {
      appVersion STRING
      modelVersion STRING
      workflowManager [WorkflowBackendConnection]\todo{what exactly needs to be
         written inside the brackets?}
      defaultConnection [RemoteConnection]
}
```

The attributes are explained as follows:

- `appVersion` – a string representation of the current app version, e.g. “RC1”
- `modelVersion` (optional) – a string representation of the current model version that has to be in accordance with the model version of the backend

- `defaultConnection` (optional) – a default remote connection can be specified here, so that it is not necessary to specify the same connection in each content provider
- `workflowManager` –

describe workflow-Manager

Furthermore, the controller layer is subdivided into one or more workflow elements. While the workflow describes the interaction of workflow elements, their internal functionality needs to be specified in the controller layer. Each workflow element can be seen as an independent controller which is responsible for the successful execution of its functionality. In general, workflow elements are structured as follows

```
WorkflowElement <workflowElementName>{
      defaultProcessChain <ProcessChain>
      onInit {
            <...>
      }
      <...>
}.
```

Workflow elements work with `ProcessChains` that define a sequence of possible steps inside the workflow element and are further described in Section 2.2.1. Since several process chains can be defined in a single workflow element, the `defaultProcessChain` keyword is used to set a default process chain. This process chain will then be used as starting point for the workflow element and the first view to be shown will also be derived from the default process chain.

The `onInit` block is used to define everything that is supposed to happen on initialization of the workflow element, e.g. binding actions to buttons or mapping content to view fields. In the `onInit` block, no workflow events may be fired, as this means handing off control from the workflow element directly at the initialization. This is enforced by raising a validator error in case the modeler tries to do this.

Typically, workflow elements should be modeled in a way that they fire a workflow event after termination of their functionality to start a new workflow element or end the whole workflow. This can be done using the SimpleAction `FireEventAction` as will be described in the following.

Actions An action provides the user the possibility to declare a set of tasks. An action can be either a `CustomAction` or a `CombinedAction`.

A *CustomAction* contains a list of CustomCodeFragments where each CustomCodeFragment contains one task. For each type of task there exist a specific CustomCodeFragment that is distinguished by the keyword that introduces it. The main tasks are binding actions to events, binding validators to view elements and mapping view elements to model elements. For every task there is a counterpart for unbinding and unmapping. Furthermore there are CallTasks that can call other actions.

```
CustomAction NAME {
      <CustomCodeFragment>
```

```
        <...>
}
```

Actions are bound to events. There are several types of actions and events available. CustomActions and CombinedActions are referenced externally whereas SimpleActions are declared directly. For events, there are local event types that listen to the state of a certain view element as well as global event types. The most powerful event type is the OnConditionEvent.

```
bind|unbind action
        <[CustomAction] | [CombinedAction] | SimpleAction> <...>
        on|from
        <[Container] | [Content]> . <elementEventType> |
        <GlobalEventType> | <[OnConditionEvent]> <...>
```

SimpleActions provide a quick way to change the state of the project:

- ProcessChainProceedAction: `ProcessChainProceed`
  proceed to the next ProcessChain step
- ProcessChainReverseAction: `ProcessChainReverse`
  go back to the last ProcessChain step
- ProcessChainGotoAction: `ProcessChainGoto <processChainStep>`
  change to the given ProcessChain step
- SetProcessChainAction: `SetProcessChain [ProcessChain]`
  changes the current ProcessChain
- GotoViewAction: `GotoView (<ViewElement>)`
  change to the given view element
- DisableAction: `Disable (<ViewElement>)`
  disables a view element
- EnableAction: `Enable (<ViewElement>)`
  enables a view element
- DisplayMessageAction: `DisplayMessage (<SimpleExpression>)`
  displays a message
- ContentProviderOperationAction: `ContentProviderOperation (<AllowedOperation> <ContentProvi`
  perform a CRUD action (save, load, remove) on the given ContentProvider
- ContentProviderResetAction: `ContentProviderReset (<ContentProvider>)`
  resets the given ContentProvider
- FireEventAction: `FireEvent (<WorkflowEvent>)`
  fires a workflow event to the backend. In response a new workflow element will be started or the workflow terminated

In addition, the LocationAction allows to extract an address from content provider fields and to generate a punctual location (i.e. longitude and latitude) for this address. For this purpose, all input and output fields have to be defined. The calculation result for longitude and latitude will be written in their respective output fields. The LocationAction is structured as follows:

```
Location (
      inputs (
            cityInput <ContentProviderPath>
            streetInput <ContentProviderPath>
            streetNumberInput <ContentProviderPath>
            postalInput <ContentProviderPath>
            countryInput <ContentProviderPath> )
      outputs (
            latitudeOutput <ContentProviderPath>
            longitudeOutput <ContentProviderPath>).
```

There are different event types available:

- `ElementEventType` – onTouch, onLeftSwipe, onRightSwipe, onWrongValidation; preceded by a dot and a reference to a ContainerElement or ContentElement
- `GlobalEventType` – onConnectionLost
- `OnConditionEvent`

The *OnConditionEvent* provides the user the possibility to define own events via Conditions. The event is fired when the conditional expression evaluates to true.

```
event NAME {
      <Condition>
}
```

A condition can be defined recursively in one of the following ways. This is a simplified version of the grammar. For a comprehensive overview the grammar in the appendix can be consulted.

```
Boolean |
<[Container] | [Content]> equals not? <[Container] | [Content]> |
<[Container] | [Content]> equals not? <STRING | INT | FLOAT> |
is not? <valid|empty|checked|filled> <[Container] | [Content]> |
not? <Condition> and|or not? <Condition>
```

Validators are bound to view elements. The validator can be a referenced element or a shorthand definition can be used in place.

```
bind|unbind validator
      <[Validator]> <...>
      on|from
```

```
      <[ContainerElement] | [ContentElement]> <...>
```

The shorthand definition has the same options but does not allow reuse.

```
bind|unbind validator
      <IsIntValidator | NotNullValidator | IsNumberValidator | IsDateValidator
         | RegExValidator |NumberRangeValidator | StringRangeValidator
         (<params>)
      on|from
      <[ContainerElement] | [ContentElement]> <...>
```

A detailed description for validator type can be found in the validator description in the following. The available parameters at params are identical to those of the validator element.

View elements are *mapped* to model elements that are in turn accessed through a ContentProvider.

```
map|unmap
      <[ContainerElement] | [ContentElement]>
      to|from
      <[ContentProvider.Attribute]>
```

CallTasks call a different Action.

```
call
      <[CustomAction] | [CombinedAction] | [SimpleAction]>
```

CombinedActions allow the composition of Actions.

```
CombinedAction NAME {
      actions <Action> <...>
}
```

Validators are used to validate user input. For each validator type corresponding parameters can be assigned. The `message` parameter is valid for every type and will be shown to the user if the validation fails.

The RegExValidator allows the definition of a regular expression that is used to validate the user input.

```
validator RegExValidator NAME (message STRING regEx STRING)
```

The IsIntValidator checks whether the user input is a valid integer.

```
validator IsIntValidator NAME (message STRING)
```

The IsNumberValidator checks whether the user input is a valid integer or float value.

```
validator IsNumberValidator NAME (message STRING)
```

The IsDateValidator allows to define a format that the date at hand shall conform to.

```
validator IsDateValidator NAME (message STRING format STRING)
```

The NumberRangeValidator allows the definition of a numeric range that shall contain the user input.

```
validator NumberRangeValidator NAME (message STRING min FLOAT max FLOAT )
```

The StringRange allows the definition of a string length range. The length of the `STRING` input by the user will be checked against this range.

```
validator StringRangeValidator NAME (message STRING minLength INT maxLength INT)
```

The NotNullValidator makes the input field required.

```
validator NotNullValidator NAME (message STRING)
```

The *RemoteValidator* allows to use a validator offered by the backend server. By default only the content and id of the field on which the RemoteValidator has been assigned are transmitted to the backend server. However, additional information can be provided using the `provideModel` or `provideAttributes` keyword.

```
validator RemoteValidator NAME (message STRING connection <RemoteConnection>
    model <ContentProvider>) |
validator RemoteValidator NAME (message STRING connection <RemoteConnection>
    attributes <ContentProvider.Attribute> <...>)
```

ProcessChain A ProcessChain is used to define several steps in which the WorkflowElement can currently be. It is possible to define several ProcessChains. ProcessChains can be nested and there is at most one ProcessChain active.

```
ProcessChain NAME {
      <ProcessChainStep> <...>
}
```

Each ProcessChainStep defines one view that is related to it and will be displayed if the ProcessChainStep becomes the current ProcessChainStep of the active ProcessChain. Additionally conditions can be defined, that restrict switching to the next or previous ProcessChainStep. Also events can be specified that trigger the change to the next or previous ProcessChainStep.

Instead of the aforementioned settings, a ProcessChain that will become active while this ProcessChainStep is the current one, can be referenced.

```
step NAME:
      view <[ContainerElement] | [ContentElement]>
      forwardCondition { <Condition> }
      forwardMessage STRING
```

```
      backwardCondition { <Condition> }
      backwardMessage STRING
      forwardOnEvent forwardEvents <EventDef>
      backwardOnEvent <EventDef>
```

ProcessChains can be refined using SubProcessChains.

```
step NAME:
      subProcessChain <ProcessChain>
```

The event definition for `EventDef` is the same as for event bindings:

```
<Container | Content> . <elementEventType> |
<GlobalEventType> |
<OnConditionEvent> <...>
```

Each *content provider* manages one instance of an entity. View fields are not mapped directly to a model element, but only content providers can be mapped to view elements. Data instances of the content providers can be updated or persisted using DataActions.

It allows to `CREATE_OR_UPDATE` (save), `READ` (load) and `DELETE` (remove) the stored instance. Which of those operations is possible is specified in `allowedOperations`. By default all operations are allowed. A filter enables to query a subset of all saved instances. The `providerType` defines whether the instances shall be stored locally or remotely.

The *remote connection* allows to specify a URI for the backend communication. The backend must comply with the MD2 web service interface as specified in the appendix.

```
remoteConnection NAME {
      uri URI
}
```

2.2.2 Deploying a Single App

Backend

map.apps

2.3 Development and Deployment of Multiple Apps

2.3.1 Workflow across multiple Apps

2.3.2 Deployment on map.apps

2.4 Additional Features

2.4.1 Uploading, Saving and Displaying RESTful Web Services

2.4.2 Calling RESTful Web Service from the App

2.4.3 Control of Workflow by calling a RESTful Web Service

3 Developer’s Handbook

A Sample Workflow