5. Realization

Contents

5.1. GUI and Concept

The concept, presented in this chapter, is built on the requirements that were described in section 4.2 and 4.3. According to the requirements, a web application should be designed, which can be integrated into the Project Management Cockpit and is developed for inexperienced users. The tool should provide a rule-based generation of dashboard templates, which help to compose the dashboard. Hence, the resulting dashboard should include all essential information, that are needed to cover the user's information needs and it should help to interpret the visualization of the data.

Within this bachelor thesis a dashboard template consists of a set of *Information Need-Dashboard Item* pairs. Those pairs are arranged according to different dimensions and dimension characteristics selected by the user.

The relationships of the dimensions, the information needs and the dashboard items are defined by initialization rules. The approach presented in this bachelor thesis allows to generate those rules. In section 4.1 the dashboard template expert is mentioned, who possesses all necessary knowledge about metrics, metric visualizations, different information needs and their relationships. On the basis of this knowledge, the dashboard template expert composes the initialization rules, where one or more *Information Need-Dashboard Item* pairs are assigned to different combinations of dimensions and their characteristics.

The statechart diagram in figure 5.1 illustrates the creation process of template initialization rules. As shown in figure 5.1, the activation of the stimulus *createTemplate()* initiates the transition from the state **Start** to the state **New Template**. The stimulus corresponds to the activation of the button *Create New Template*. After a dashboard template has been created, all existing dimensions and a characteristic from the characteristics list of each dimension will be assigned to the new template automatically. This is represented by the

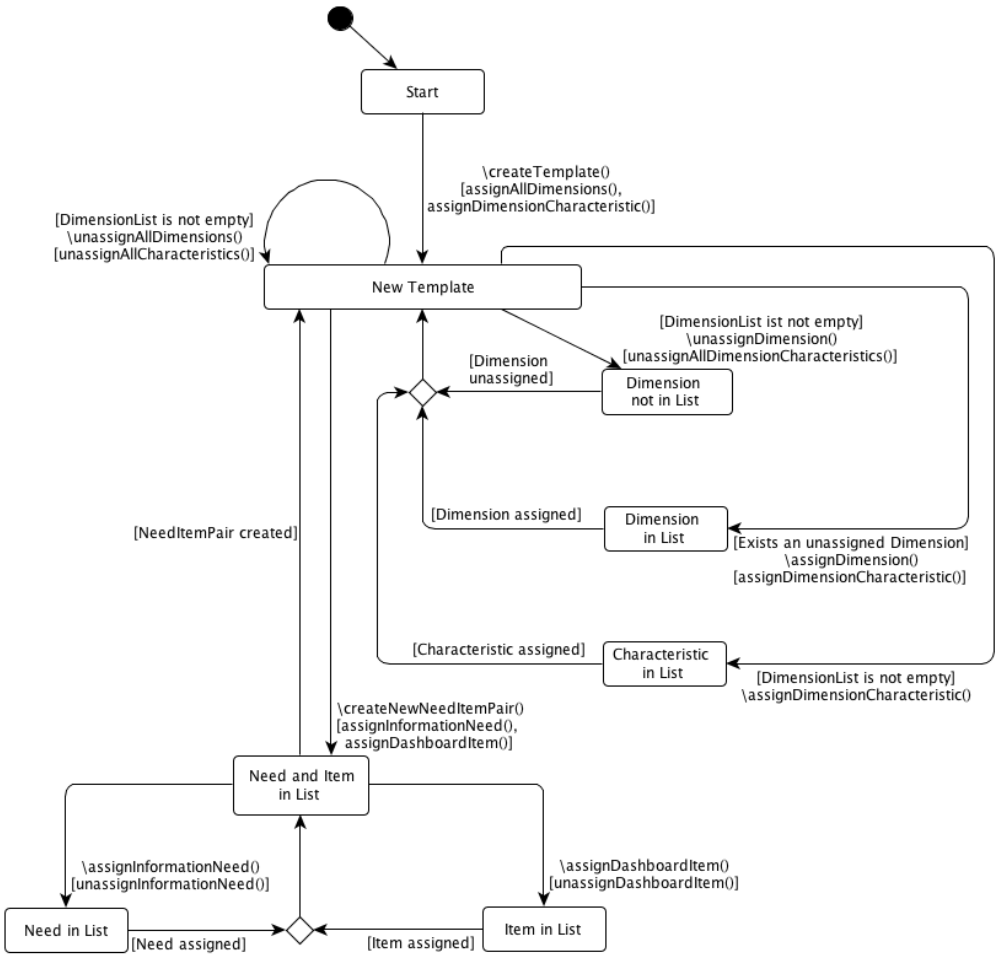

Figure 5.1.: Statechart Diagram of Creation of a Dashboard Template and Initialization Rules.

postcondition of the transition. In the state **New Template** dimensions and characteristics can be unassigned and assigned. It is possible to unassign all dimensions at once (see the transition with the stimulus *unassignAllDimensions()* in fig. 5.1) or a particular dimension (see the transition with the stimulus *unassignDimension()* in fig. 5.1) from the created template. If a dimension is unassigned, all its characteristics should also be unassigned. This is represented by the corresponding postcondition.

The creation of a new *Information Need-Dashboard Item* pair is represented by the stimulus *createNewNeedItemPair()*. This stimulus starts the transition from the **New Template** state to the **Need and Item in List** state. If a new pair is created, an arbitrary existing information need and a dashboard item will be assigned to the new template automatically (see the postcondition of the stimulus *createNewNeedItemPair()* in fig. 5.1). The user can change the value of the information need and dashboard item, which is demonstrated by the

stimuli *assignInformationNeed()* and *assignDashboardItem()*. If the new object of an element of the pair is assigned, the old object will be unassigned, which is represented by the postconditions of the corresponding transitions.

Figure 5.2 shows the graphical user interface (GUI) software prototype for the dashboard template list. Each dashboard template list includes a column **Template Name** containing template names, a column **Edit Template** containing a link to the edit page of the corresponding template and the column **Delete**, which contains the command link to delete the corresponding template. A new dashboard template can be created by activating the button *Create New Dashboard Template*. In this case, a pop-up dialog appears (see fig. 5.3), where a new template name can be inserted and a new row in the dashboard template table will be created.

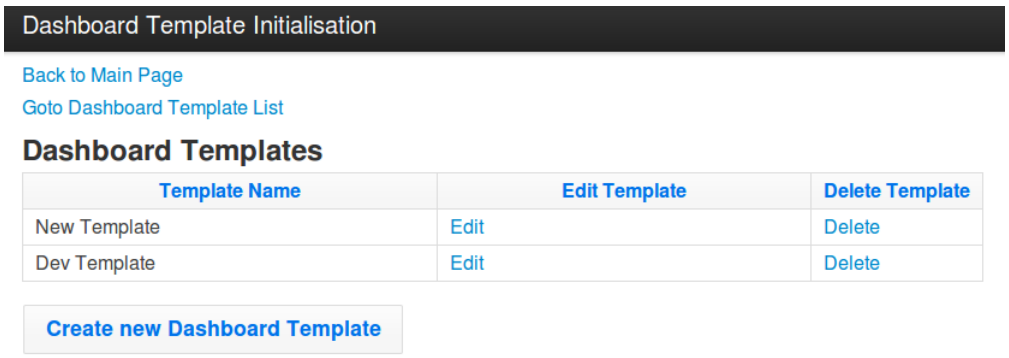

Figure 5.2.: A Dashboard Template List in the Software Prototype.

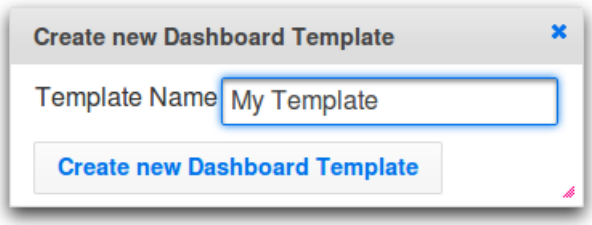

Figure 5.3.: Software Prototype: The Pop-Up Dialog for the Creation of a new Template.

The sequence diagram in figure 5.4 illustrates the method invocations after the button *Create New Dashboard Template* in the dialog from figure 5.3 has been activated. A click on the button initiates a HTTP request, which starts the life cycle of JSF. As described in section 2.2.1, the managed bean properties, that are bound to UI components values, are updated in the *Update Model Views* phase. This action is represented in figure 5.4 by the arrow denoted with the method *managedBean.setTemplateName()*. The new template name is read

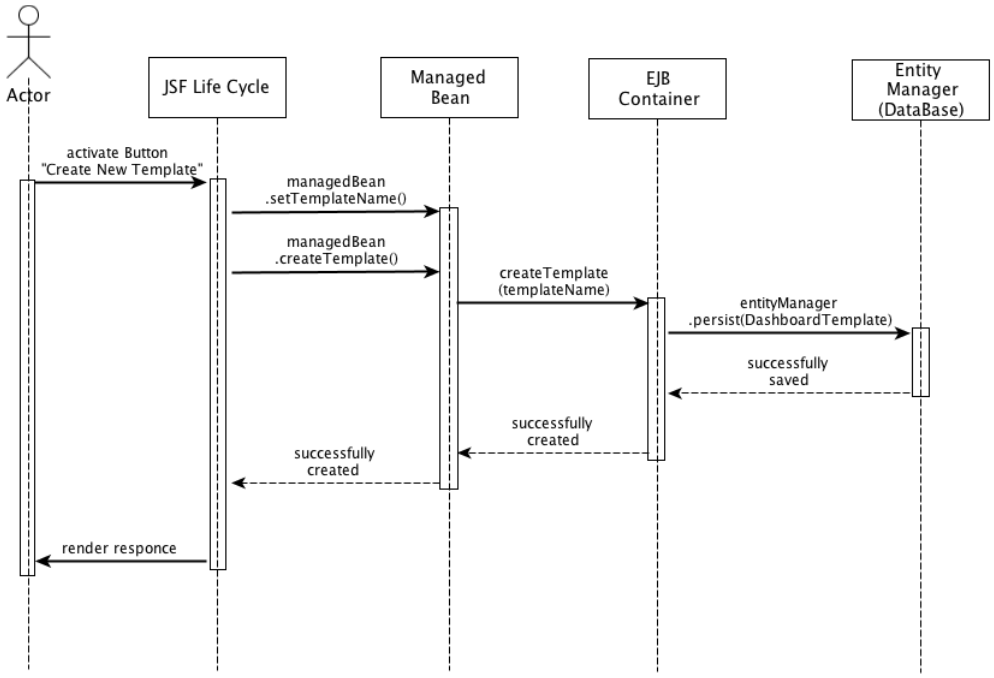

Figure 5.4.: Sequence Diagram of Method Invocations in the System during HTTP Request initiated by Activation of the Button *Create New Dashboard Template*.

from the input field of the dialog (see fig. 5.3) and written in the corresponding property *templateName* of the managed bean.

In the *Invoke Application* phase of the JSF life cycle, the business application logic is executed. As a consequence, the methods that are bound to UI components, like *action* or *actionListener*, are called. For the creation of a dashboard template, the method *createTemplate()* of the managed bean is invoked. The managed bean delegates the creation operation to the EJB container by calling the method *createTemplate(templateName)* (see fig. 5.4). The template name is read from the corresponding property of the managed bean, which was updated in the previous life cycle phase. The EJB container then delegates the creation operation to the Entity Manager, which records the new template in the data base.

The dashed arrows in the sequence diagram denote, that the creation action was successful. Otherwise, if the template creation is failed, system exceptions are invoked. As shown in figure 5.4, in the last step of the creation action, the current view state of the application is rendered to the browser and the new row in the templates table appears.

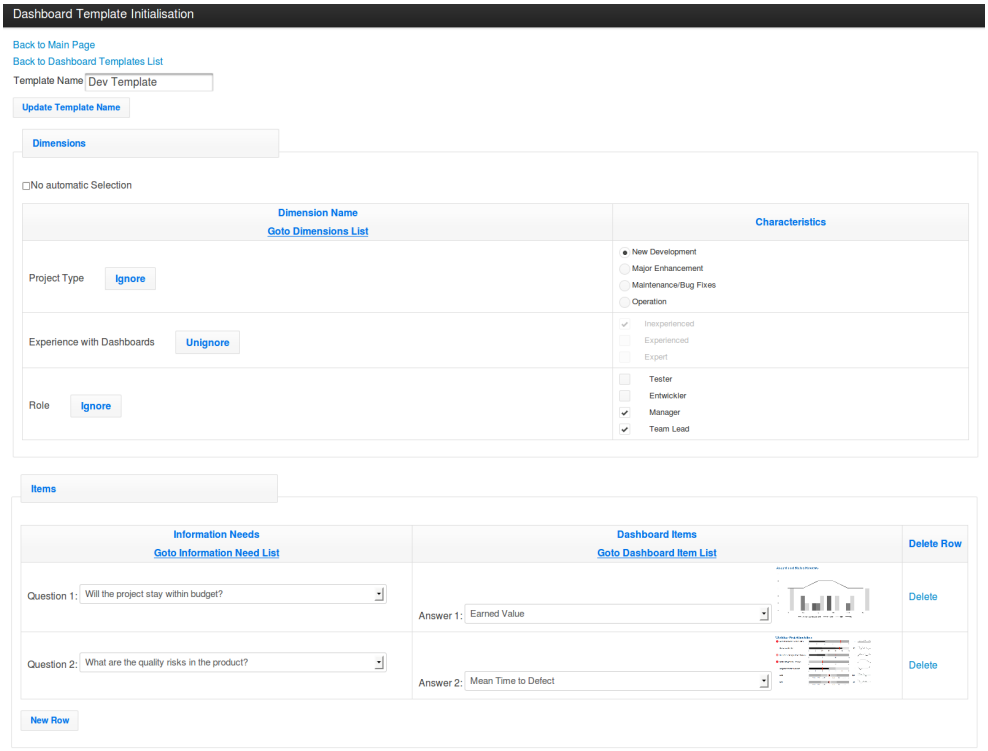

Figure 5.5.: The Dashboard Template Edit Page in the Software Prototype.

Through the activation of the link *Delete*, a row can be deleted from the templates table. In order to edit a template, the link *Edit* should be clicked on in the corresponding row. In this case a new edit page will be load. Figure 5.5 illustrates the software prototype for a dashboard template edit page. The template name can be edited by using an input box. The edited name can be saved by clicking on the button *Update Template Name*. The dimensions can be selected or disabled by activating the button *Ignore* resp. *Unignore*.

On the *Items* panel, the *Information Need-Dashboard Item* pairs are created. In order to create a new pair, the button *New Row* should be clicked on. In the combo box of the new row a question (information need) and an appropriate answer (dashboard item) can be selected. If an answer in the combo box is selected, a mini image of the corresponding dashboard item appears.

The links *Back to Dimensions List*, *Back to Information Need List* and *Back to Dashboard Item List* navigate the application to the lists of dimensions, information needs and dashboard items respectively, which are then visualized in a table, as shown in figure 5.2.

Figure 5.6 shows the software prototype of the dimension edit page. On the panel *Dimension* the dimension name and a widget for the selection form of the dimension characteristics can be chosen. After a click on the button *Update Dimension* the corresponding changes will be applied. On the panel *Character-*

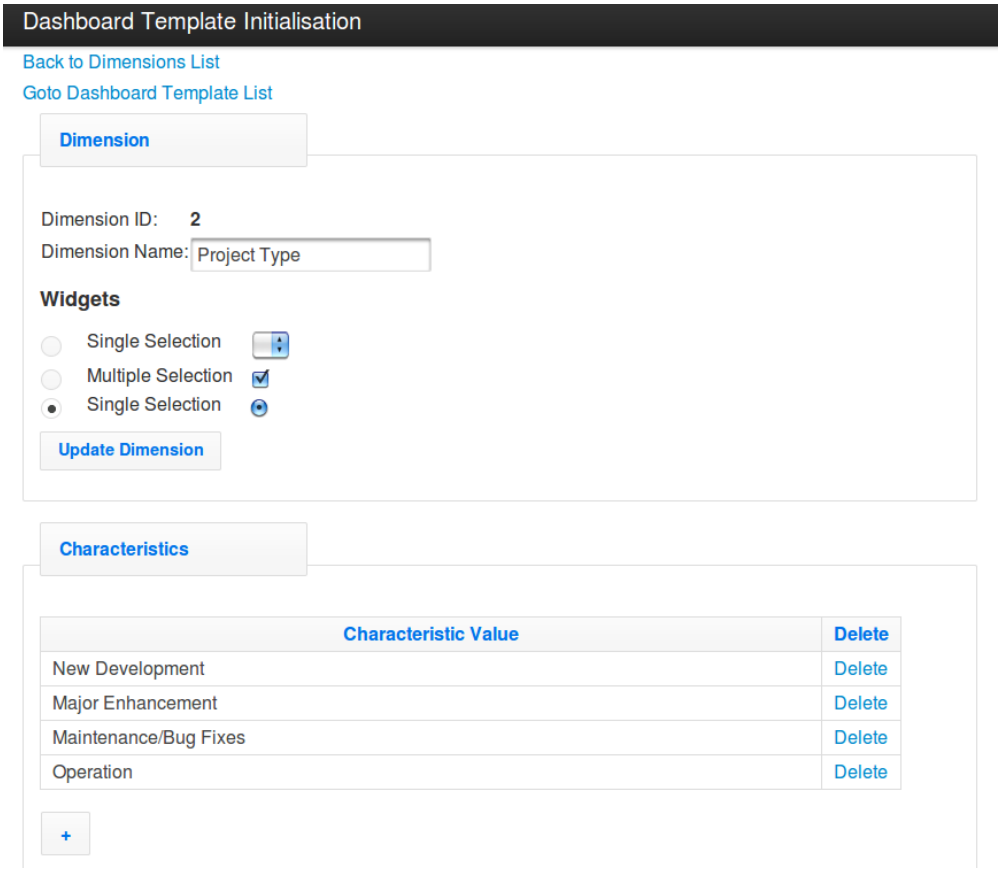

Figure 5.6.: The Dimension Edit Page in the Software Prototype.

istics, new characteristic values can be added or deleted (see fig. 5.6).

After a dashboard template and the corresponding template rules are created, this template can be used to initialize dashboards according to its rules.

For users that do not have any experience with operating with Project Management Cockpits, it may be difficult to formulate the information needs precisely and select the appropriate metrics to measure the information data. By using the dashboard setup support, the user simply needs to specify the dimensions (for instance, experience with the Project Management Cockpit) with appropriate characteristics. According to the selected dimensions, the system applies the initialization rules. If the rules for the selected dimensions and characteristics are defined, the system returns the corresponding question-answer (*Information Need-Dashboard Item*) pairs.

It can also be helpful for an experienced user to have a look at the possibilities how to compose different dashboards for her information needs.

Figure 5.7 illustrates the starting point of dashboard creation, where a dashboard template is generated by using the initialization rules. The user has two

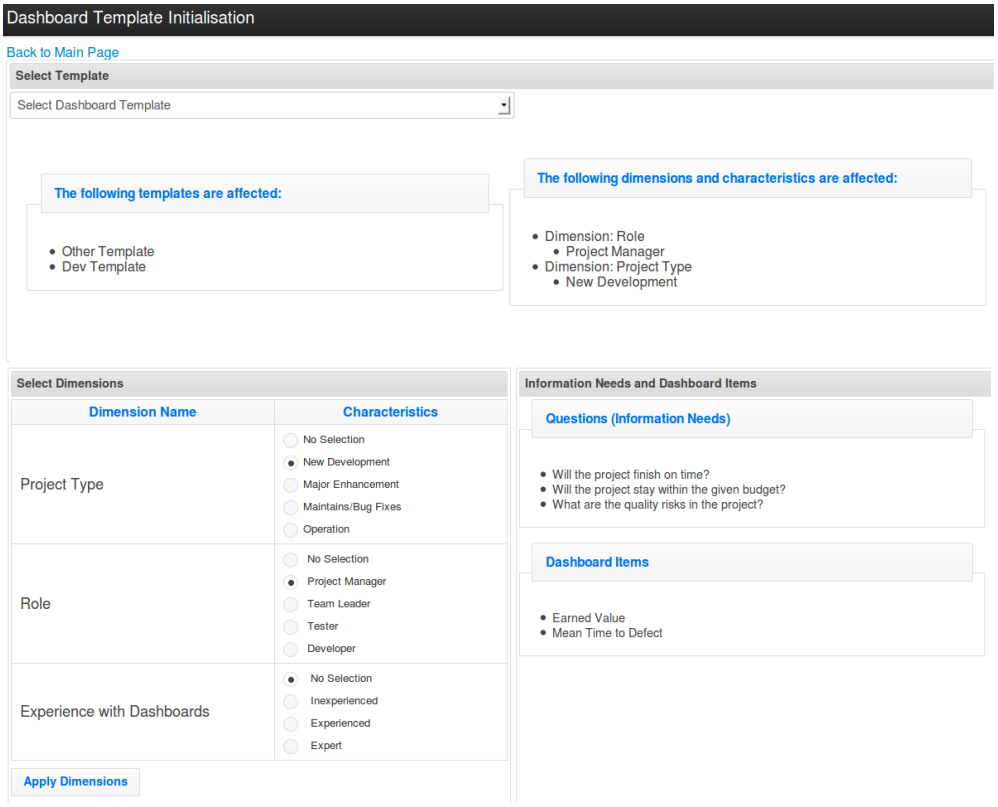

Figure 5.7.: The Setup Page of the Project Management Cockpit in the Software Prototype.

options to create a dashboard template. The first option is to select a dashboard template name from the combo box located at the top of the page. In this case the system returns the corresponding affected dimensions with characteristics at the top of the page (see fig. 5.8) and the *Information Need-Dashboard Item* pairs at the right-hand side of the setup page (see fig. 5.9)

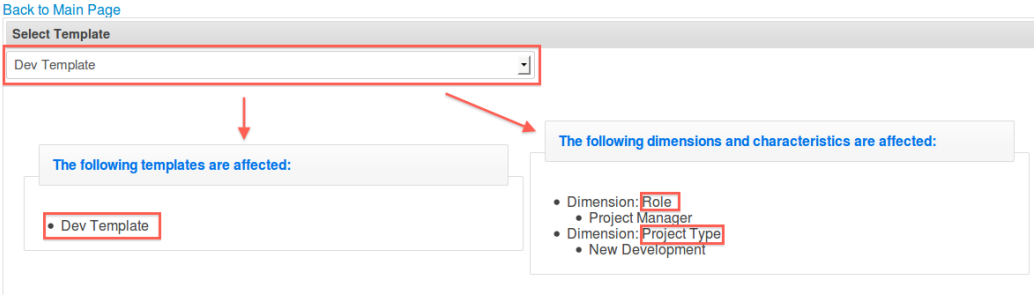

Figure 5.8.: The first Option: Selection of Dashboard Template in the Combo Box of the Setup Page.

With the second option, the user selects the appropriate characteristics of the

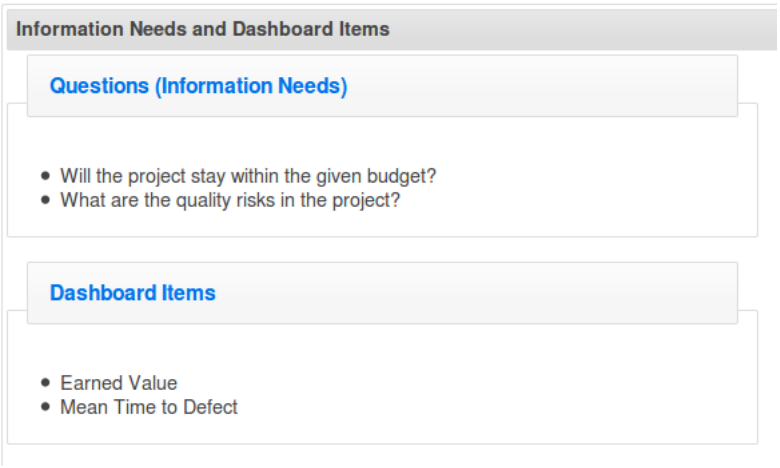

Figure 5.9.: The first Option: the returned *Information Need-Dashboard Item* pairs.

dimensions, which are visualized as a table in the left hand side of the page (see fig. 5.10). If a dimension should be ignored, the user selects the item *No Selection* of the radio button in the corresponding characteristics cell. If the button *Apply Dimensions* is activated, the system returns the affected template names(see fig. 5.11) and the *Information Need-Dashboard Item* pairs (see fig. 5.12).

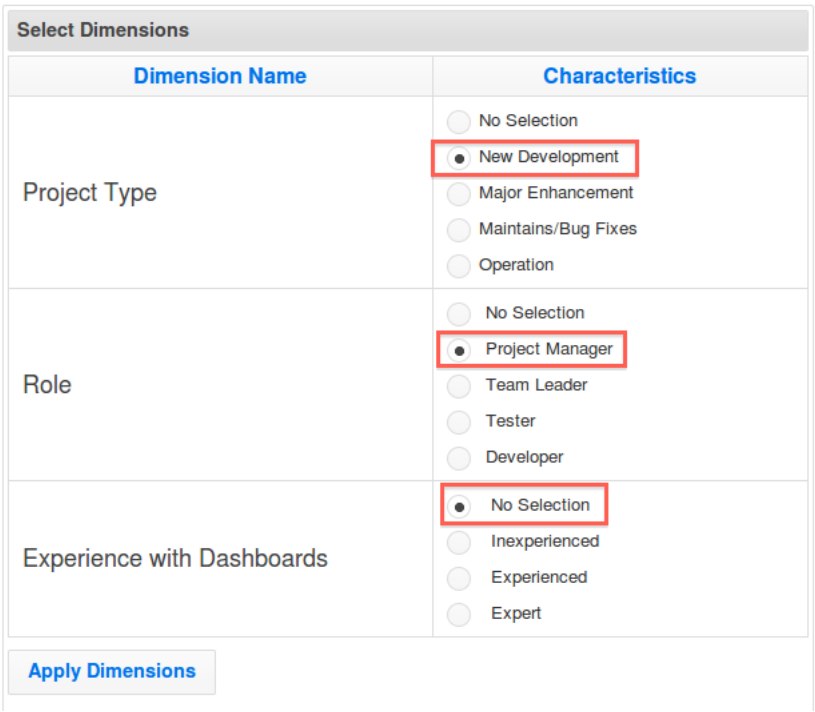

Figure 5.10.: The second Option: Selection of Dimension Characteristics.

Figure 5.11.: The second Option: the affected Templates and Dimensions with Characteristics.

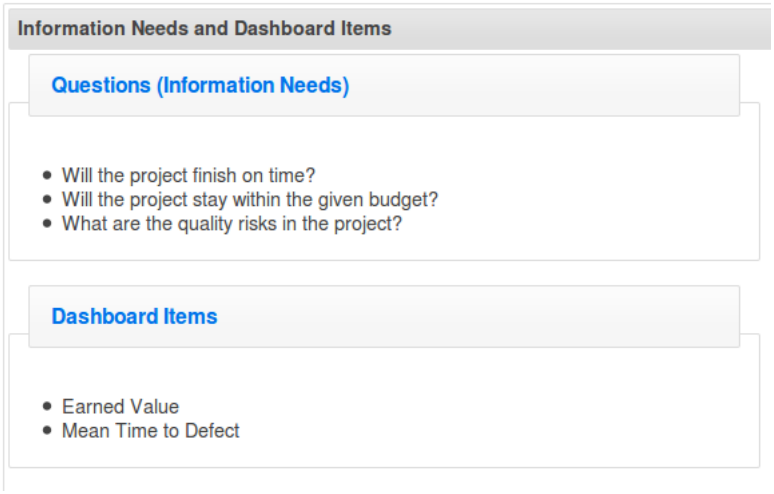

Figure 5.12.: The second Option: the returned *Information Need-Dashboard Item* pairs.

5.2. Architecture

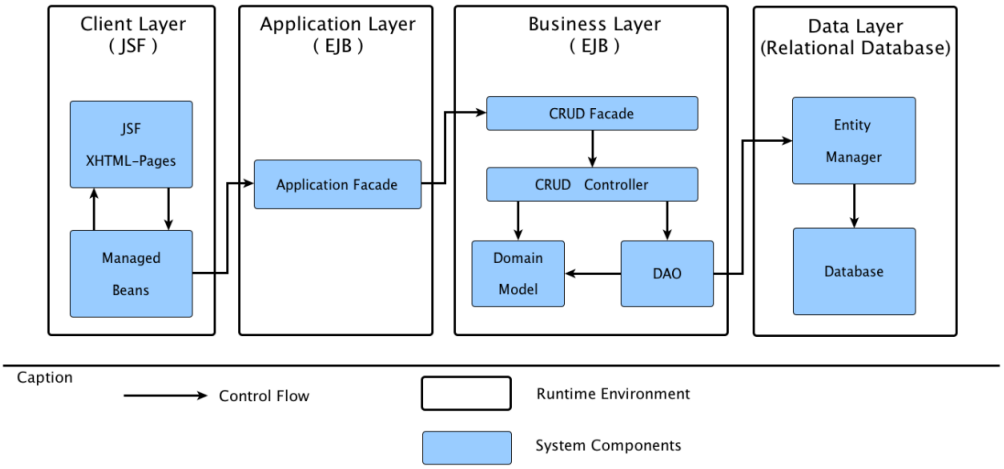

Figure 5.13.: The Architecture of the Rule-Based Dashboard Initialization Tool.

In this section the architecture of the rule-based dashboard initialization sys-

tem is described. The architecture of the system is based on the multi-layered software architecture approach, which offers reusability and flexibility of the application. The goal of the multi-layered software structure is to divide the software system into many different blocks, which can be added, modified and extended independently of the other parts of the system. The approach also allows easy error detection and error handling during the system development.

In figure 5.13 the general architecture of the dashboard initialization tool is visualized. The architecture is divided into four layers, where each layer represents a particular level of abstraction. The three main layers (Client, Business and Data Layers) are extended by the fourth layer, which is called Application Layer.

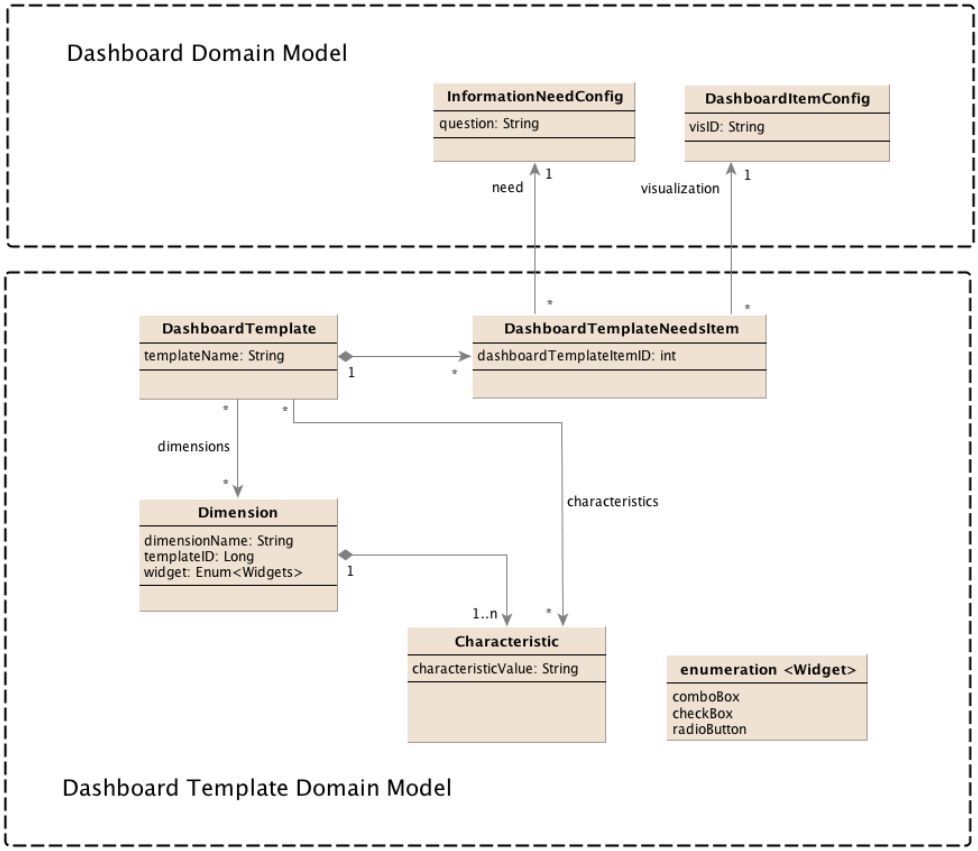

Figure 5.14.: The UML Class Diagram of the Domain Model of the Rule-Based Dashboard Initialization Tool.

As described in section 2.3, the Gargoyle code generator was used to generate the basic functions of the application, which build the Business Layer logic of the software system. This functions are create, retrieve, update, delete (CRUD) and persistence methods of the application objects. In contrast to the Business Layer, the Application Layer provides an additional application logic for a par-

ticular software tool, which specifies the functionality of the application. The Business Layer and the Application Layer are implemented by using the EJB technology. The Client Layer is implemented by using JavaServer Faces and realizes the GUI and the processing of the user input data. It then forwards the data to the Application and Business Layers (see fig. 5.13 and section 2.3) .

As shown in figure 5.13, the Business Layer is divided into four sublayers: the CRUD Facade, the CRUD Controller, the Domain Model and the DAO. The CRUD Facade offers an interface for interactions between the Application Layer and the Business layer and interactions between Managed Beans and the Business Layer. The CRUD Facade of the dashboard template consists of an interface class and a java class, which implements the interface. The CRUD Facade gives the control to the CRUD Controller, which validates the data forwarded from the CRUD Facade. If the data is not valid, the CRUD Controller throws exceptions. Otherwise, the methods of the CRUD Controller invoke the corresponding methods of the Domain Model and the DAO classes. If an entity shall be created, the CRUD Controller calls a method of the EntityFactoryLocal class of the Domain Model, which creates the Java objects of the Domain Model. This Java objects represent the POJOs, which are the Entity Beans described in section 2.1. In order to store, delete or update the data in the database, the DAO classes calls the Entity Manager.

Figure 5.14 illustrates the UML class diagram of the Domain Model. The Domain Model includes all components of the dashboard template, according to the concept described in previous chapter. Each Dimension class includes one or more Characteristics, which are represented by the Composition relationships (see 5.14). A DashboardTemplate class contains a list of dimensions and a list of characteristics. Additionally, each DashboardTemplate includes a list of DashboardItemConfig-InformationNeedConfig pairs. Each pair is represented by an object of the DashboardTemplateNeedsItem class. The DashboardTemplate and the DashboardTemplateNeedsItem are in a composition relation, such that the DashbordTemplate is the container class of the DashboardTemplateNeedsItem.

Within this bachelor thesis, the Application Layer and the Client Layer were extended. Figure 5.15 demonstrates the structural relations of the tool components between the Client, Application and Business Layer at the example of the Dimension component. As shown in figure 5.15, the JSF components of the application have a hierarchical template structure, which is illustrated by the dashed arrows. Templates offer a consistent organization of the pages structure and enable reusability of the JSF components. In the example of figure 5.15, the templates simpleDialog and simpleListTemplate of the package **resource.comonHelpers** are implemented for the creation of dialogs and tables respectively. These templates are used by different components of the application. The navigation between pages is demonstrated by the dashed arrows with the label *link*.

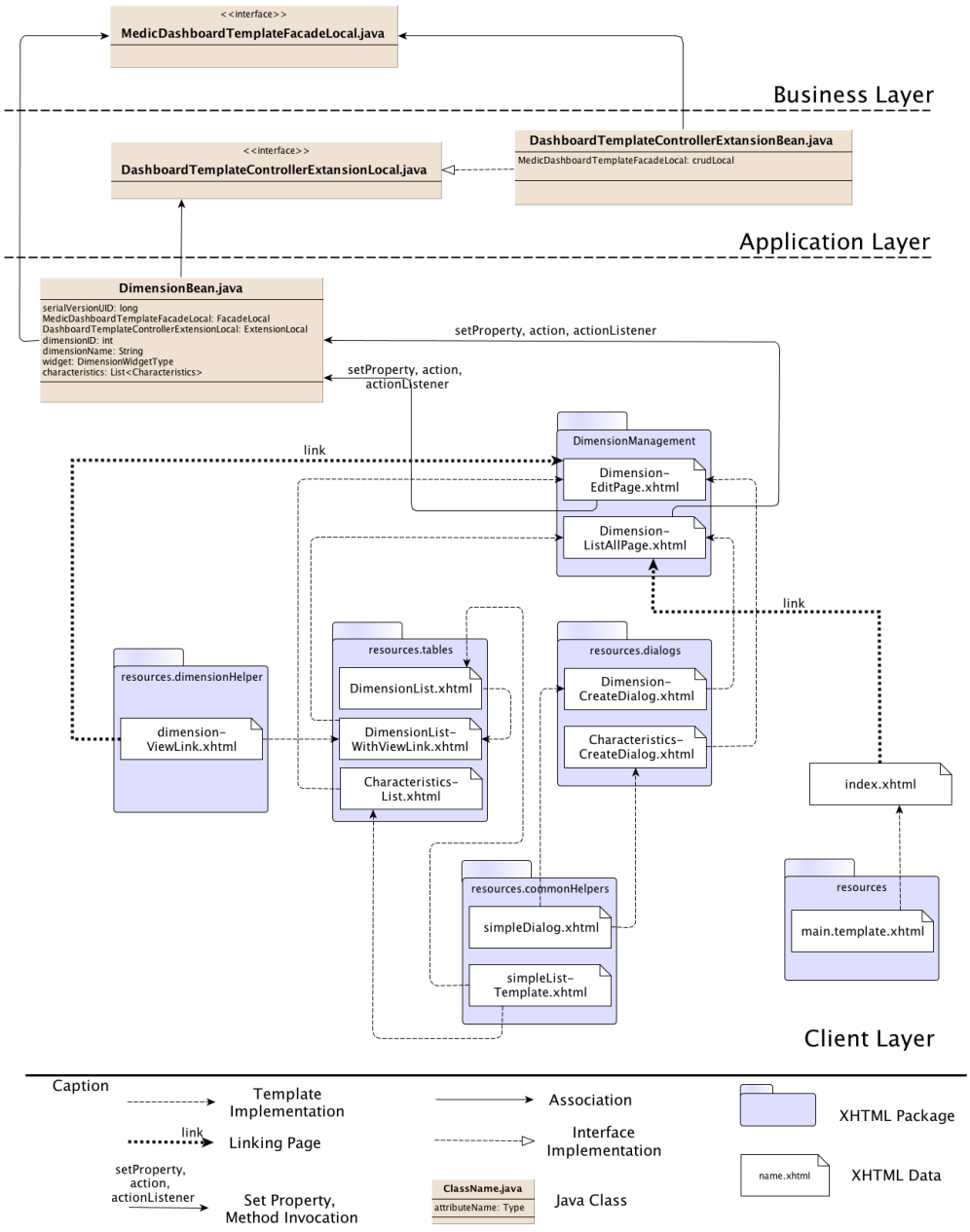

Figure 5.15.: The Structural Relation of the Tool Components.

The DimensionList template includes the columns *Dimension Name* and *Dimension ID*. DimensionListWithViewLink implements the DimensionList template and includes the two additional columns *Edit* and *Delete* for *edit* and *delete* links. In order to realize a navigation link to the DimensionEdit page, the DimensionListWithViewLink template implements the dimensionViewLink template. As shown in figure 5.15, the DimensionListAll page implements the DimensionListWithViewLink template and includes the table with the four

columns as described above.

For further input data processing and updating, the managed bean methods of the DimensionBean are invoked, which is denoted by the arrows with the label *setProperty, action, actionListener* (see fig. 5.15). The DimensionBean class calls the methods of the Application Layer for the execution of the application logic through the interface DashboardTemplateControllerExtensionLocal. The Application Layer of the Rule-Based Dashboard Initialization Tool includes two components, the interface DashboardTemplateControllerExtensionLocal and the Java class DashboardTemplateControllerExtensionBean, which implements the interface. The methods of the DashboardTemplateControllerExtensionBean class forwards the method calls to the Business Layer.

5.3. Implementation and Experiences

This section describes important implementation aspects and experiences collected during the implementation process. The main task of this bachelor thesis was the implementation of the Application Layer and the extension of the Client Layer. In the implementation of the concept described earlier, the PrimeFaces framework was used. The PrimeFaces framework extends JSF by a large number of the predefined GUI components. The further benefit of the PrimeFaces usage within this bachelor work was to achieve a homogenous design of the application pages.

```
<?xml version='1.0' encoding='UTF-8' ?>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:composite="http://java.sun.com/jsf/composite"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:prime="http://primefaces.org/ui"
    xmlns:tables="http://java.sun.com/jsf/composite/tables"
    xmlns:dialogs="http://java.sun.com/jsf/composite/dialogs"
    xmlns:dashboardItemRefHelper="http://java.sun.com/jsf/composite/dashboardItemRefHelper"
    xmlns:informationNeedRefHelper="http://java.sun.com/jsf/composite/informationNeedRefHelper"
    xmlns:dimensionHelper="http://java.sun.com/jsf/composite/dimensionHelper"
    xmlns:dashboardTemplateHelper="http://java.sun.com/jsf/composite/dashboardTemplateHelper"
    xmlns:dashboardTemplateNeedsItemHelper="http://java.sun.com/jsf/composite/
                                            dashboardTemplateNeedsItemHelper"
    xmlns:characteristicsHelper="http://java.sun.com/jsf/composite/characteristicsHelper">
```

Figure 5.16.: The Resources List.

Figure 5.15 of the previous section shows an example for the structure of the JSF resource organization. The resources are divided into different packages according to their functionality. In order to embed the necessary PrimeFaces links and script tags, the resources components of the PrimeFaces have to be listed on all pages, where the PrimeFaces are used. Figure 5.16 illustrates an example of a page of the application with the PrimeFaces resources list implemented within the *head*-tag of the page.

In order to display the available elements of the application, the list structure was used (see fig. 5.2). Here, the application objects are listed according to their IDs and Names. Additionally, each list includes the columns *Edit* and *Delete*, where the links to edit and to delete the element in the corresponding row are located. During the implementation of the component lists in the ListAll and the Edit pages the following experiences and problems occurred.

Fetching of the Application Objects

```
<f:metadata>
    <f:event type="preRenderView"
        listener="#{dimensionBean.fetchDimension}" />
</f:metadata>
```

Figure 5.17.: Fetching of a Dimension.

```
1 <?xml version='1.0' encoding='UTF-8' ?>
2 <html xmlns="http://www.w3.org/1999/xhtml"
3     xmlns:ui="http://java.sun.com/jsf/facelets"
4     xmlns:h="http://java.sun.com/jsf/html"
5     xmlns:f="http://java.sun.com/jsf/core"
6     xmlns:composite="http://java.sun.com/jsf/composite"
7     xmlns:prime="http://primefaces.org/ui">
8
9 <body>
10    <ui:composition template="../../commonHelpers/simpleListTemplate.xhtml">
11
12        <ui:param name="label" value="#{label}" />
13        <ui:param name="onEmptyText" value="#{onEmptyText}" />
14        <ui:param name="list" value="#{list}" />
15
16        <ui:define name="additionalColumns">
17
18            <prime:column style="width:60px">
19                <f:facet name="header">
20                    <h:outputText value="Characteristic Value" />
21                </f:facet>
22                <h:outputText value="#{item.characteristicValue}" />
23            </prime:column>
24
25            <ui:insert name="additionalColumns" />
26
27            <prime:column style="width:30px">
28                <f:facet name="header">
29                    <h:outputText value="Delete" />
30                </f:facet>
31                <prime:commandLink value="Delete" id="deleteChar"
32                    actionListener="#{characteristicsBean.setCharacteristics(item)}"
33                    action="#{characteristicsBean.deleteCharacteristics}" ajax="false" />
34            </prime:column>
35        </ui:define>
36    </ui:composition>
37 </body>
38 </html>
```

Figure 5.18.: The CharacteristicsList Template.

If the link *Edit* of the component table has been activated, the new page is loaded, where the properties of the corresponding application object can be added. In order to set the selected object and its properties on the edit page, the JSF *f:metadata*-tag was used. In figure 5.17 the example source code for

fetching of a dimension object is shown.

Implementation of Templates

As mentioned in section 5.2, the template components are used in order to implement the lists and the create dialogs. To implement the characteristic list in the DimensionEditPage the CharacteristicsList template was used, but the attributes *action* and *actionListener* of the *Delete* link were not invoked, while the attributes *characteristicValue* and *dimensionID* were picked up correctly. Figure 5.18 shows the CharacteristicsList template, which is implemented in the DimensionEdit page (see fig. 5.19). Figure 5.19 demonstrates the implementation, which does not work, because, the dimension object is not set and the methods of the corresponding managed bean, that are bound to the *action* and *actionListener* attributes, are not invoked.

```
<tables:CharacteristicsList label="Characteristics"
    list="#{dimensionBean.characteristics}"
    id="characteristicTable" />
```

Figure 5.19.: Not working Implementation of the CharacteristicsList Template

```
<tables:CharacteristicsList label="Characteristics"
    list="#{dimensionBean.dimension.characteristics}"
    id="characteristicTable" />
```

Figure 5.20.: Correct Implementation of the CharacteristicsList Template.

Figure 5.20 illustrates the correct implementation of the CharacteristicsList template, where the property *dimension* is set first and then the corresponding characteristic list.

Handling of the *valueChangeEvent*

The application includes UI components like the *selectOneMenu*, the *selectOneRadio* and the *selectManyCheckbox*, which include dynamic lists of items. This components are used to realize combo boxes, radio buttons and check boxes respectively. In order to register a new selected item in the managed bean, the *valueChangeListener* attribute is set, where a method to handle the value change event is defined. If a page includes a table with many different UI components, that have a *valueChangeListener*(see fig. 5.5) bound to them, the value change event is triggered for all components, regardless of a change of state of the UI components. In order to update the values of the application object properties properly, the following solution has been implemented. Each time an event is fired and the value of the event parameter is not null, the old values of all components are deleted and the new values are set.

Implementation of the *selectOneRadio* UI Component

Another problem that occurred during the implementation of the UI components was, that the input and output data of the UI components are strings. Thus, it was necessary to identify objects by an ID or another unique attribute. For instance, the table of the DashboardTemplateEdit page includes dimensions and the corresponding characteristics. According to the concept of the application tool, the characteristics can be identified by the characteristic name and the dimension id. This means, that two different dimensions could contain characteristics with equal names. If the *valueChangeListener* of an UI component returns only a characteristic name, it is impossible to identify its dimension ID.

```
<prime:selectOneRadio id="selectOneRadioChar" style="width:200;"
    layout="pageDirection"
    value="#{dashboardTemplateBean.getAssignedCharacteristicForSelectOne(item)}"
    valueChangeListener="#{dashboardTemplateBean.changedCharacteristicValueEvent}"
    onchange="submit()" rendered="#{item.widget == 'RadioButton'}"
    disabled="#{dashboardTemplateBean
                .assignedDimensionToDashboardTemplate(template,item)==false}"
    immediate="true">
    <f:selectItems value="#{item.characteristics}" var="char"
        itemLabel="#{char.characteristicValue}"
        itemValue="#{char.characteristicValue};#{item.dimensionID}" />
</prime:selectOneRadio>
```

Figure 5.21.: Implementation of the *selectOneRadio* UI Component.

Figure 5.21 illustrates the solution for the problem described above. In this case, the attribute *itemValue* of the *selectOneRadio* component is composed of the characteristic name and the corresponding dimension ID. Note, that the value of the *itemValue* attribute should be equal to the return value of the method bound to the attribute *value* of the *selectOneRadio* component. Thus, the method *getAssignedCharacteristicForSelectOne* of the DashboardTemplateBean class returns the assigned characteristic and the dimension ID as a string.

Setup Page

For the implementation of the setup page, a new managed bean, called *DashboardSetUpBean*, was created. The setup page has no domain model and therefore cannot be persisted in the database. The selected dimensions and characteristics as well as the affected templates are store in temporal lists of the *DashboardSetUpBean*.