University of Applied Sciences

Georg-Simon-Ohm Fachhochschule Nürnberg
Fachbereich Informatik
André Duffeck

Master Thesis

XML-Schema based generation of Graphical User Interfaces with a Qt-based prototype

Author:	André Duffeck
Matriculation Number:	936176
Address:	Kirschgartenstr. 49 90419 Nürnberg Germany
Email Address:	andre@duffeck.de
Supervisors:	Dipl. Phys. Cornelius Schumacher Prof. Dr. R. Kern
Begin:	01. March 2007
End:	31. July 2007

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Masterarbeit mit dem Thema „XML-Schema based generation of graphical User Interfaces with a Qt-based prototype“ selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

_____

Andre Duffeck

Abstract

The creation of graphical user interfaces (GUIs) is a recurring task for application developers. This task is both error-prone and time-consuming and often leads to duplicate code. Furthermore the split of data definition and user interface can lead to problems when the underlying data changes.

This work examines possibilities of describing and creating GUIs with a description language and generating such descriptions based on a schema of the underlying data structure. The goal is to provide a mechanism to automate the generation of graphical user interfaces in such a manner, that the developer ideally doesn't have to care about the interface at all. The interfaces are generated from the information available and possibly be extended or polished with "hints". Those hints describe details of how an element should be placed or what it should look like. Therefore, the interface is described using a description language based on XForms, generated from the definition of the handled data structure and the hints. This GUI description document can be used to create the concrete user interface.

Furthermore, a prototype application was created using the Qt application framework, which is capable of both creating a GUI from a schema and editing this GUI.

Contents

Chapter 1

Introduction

The human-machine interface of choice these days is the Graphical User Interface (GUI). It excels as an easy and intuitive method of running a machine and is becoming ubiquitous as computers, mobiles and PDAs become more and more popular.

For the software developers however, designing GUIs is a annoying task. Creating good GUIs is time-consuming and for non-trivial interfaces only usability experts will achieve satisfactory results. This work tries to evaluate a solution for automating the generation of the GUI.

Therefore it exploits the fact that a high percentage of GUIs are used for displaying or editing data, of which the format is specified in some way. For these cases, it should be possible to use this information to create a description of the required elements and their relations to the according data fields. This GUI description document could then be used to generate an actual GUI from it. The format for describing the GUI should be abstract enough to allow rendering on different systems.

It may be expected however, that the generated GUI is not always the perfect way of presenting the data. Thus, an additional format (hints) should be specified, which can be used to give the rendering application instructions of how to change the appearance of specific parts of the GUI.

As a second part of this work, a prototype application was created, which should be able to create non-trivial GUI descriptions and render them into a usable GUI. For this prototype the Kode project of KDE was chosen, which already provided a basic framework for this purpose. The format of the underlying data in this work is restricted to XML and XML schemas, which are a widespread method of exchanging and verifying data between systems or applications and are both human- and machine-readable and thus easy to handle.

Furthermore, this prototype should implement an edit mode, which allows the modification of the GUI. The changes should then be exportable as a hints file, so that they can be reused later.

Chapter 2

Theory

This chapter describes the basic technologies and backgrounds of this work.

2.1 Evolution Of User Interfaces

In order to control a computer, there has to be an interface, over which the input and output is given. This human-machine interface has evolved rapidly since the beginnings of computing.

With the first computers in the 1940's, input was given as punched cards (Figure 2.1). A punched card is a piece of paper that contains binary information in form of the presence or absence of holes. The position of the holes is predefined, so that the machine can determine if the hole is missing or not.

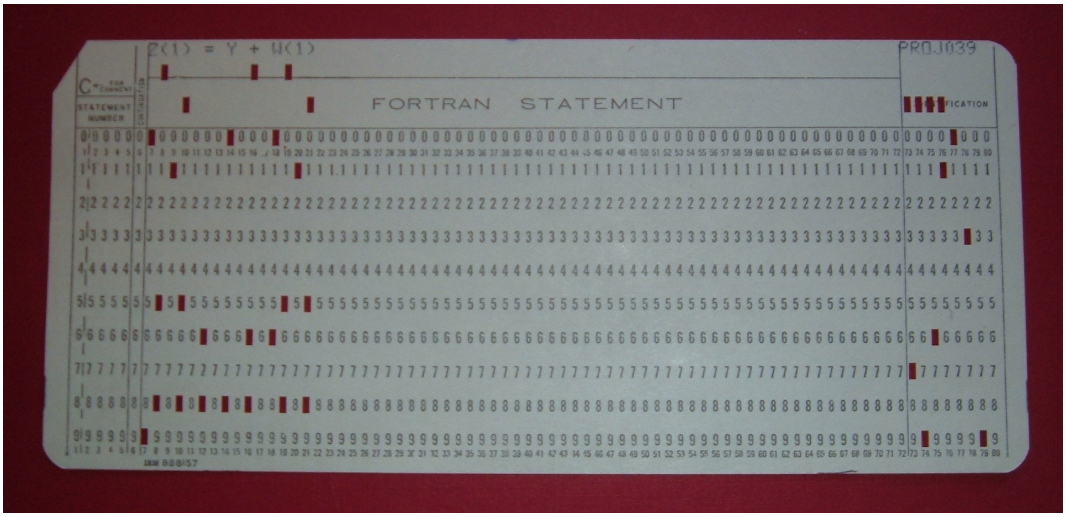

Figure 2.1: Punched Card

Punched cards were used for both the programs and the data and were created with key punch machines. These machines were manual machines in the beginning and were mecha-

nized later. The computer in return gave the output as long print-outs. This method of data input and output was slow and only specialists were able to use the computers at that time.

This changed in the 1950's, when the industry discovered the potential of computers. The new systems had monitors and keyboards, allowing fast data input and immediate output of results. This kind of interface was improved further in the 1970's with the CUI (Character-based User Interface). The interface was a commandline, where the user could send human-readable commands to the computer and immediately get the result presented on the monitor. The CUI needed only short training and made it possible to let office workers operate the computers.

The next evolutionary step in user interfaces was the introduction of the GUI, the Graphical User Interface. Here, the interaction with the machine is done with a keyboard and a pointing device such as a mouse, which feels more natural for the users. In other words, a GUI is a "hierarchical, graphical front-end to a software that accepts as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output. A GUI contains graphical objects and each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI" [10]. Those "graphical objects" are called widgets. A widget is a single graphical element such as a button, an icon or a progressbar, which, sanely assembled together, form an application.

Although there are new UI trends coming up with 3-dimensional interfaces and voice recognition the GUI is not showing any indication of going away. Thus, the generation of GUIs is an important task, because with the GUI being the direct interface between the user and the machine, having a high quality GUI is crucial for the whole application.

2.2 GUI Generation

2.2.1 Principles Of Good GUI Design

A lot of work was done on GUI design [6] [8] [9] [21]. They all describe the requirements and principles that a GUI designer has to take into account.

The most basic principles that are mentioned are listed below:

- **Simplicity**

  First of all, GUIs have to be simple. The worst thing that can happen is that the user of the application gets flooded with information and options and choices he or she has to

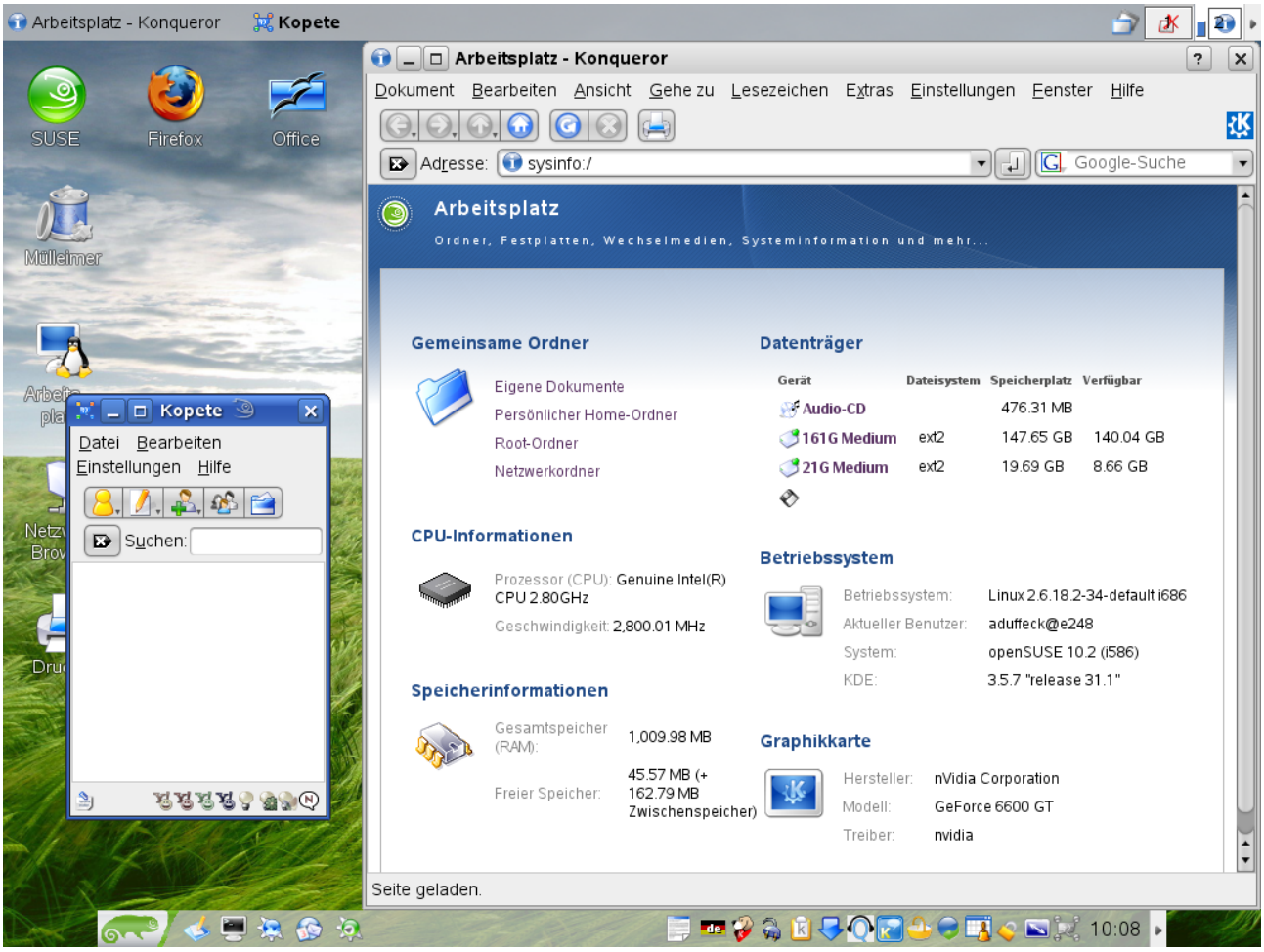

Figure 2.2: Modern GUI showing two applications (KDE 3.5)

make. Users will usually react frightened and negatively, preventing concentration. The GUI should rather try to focus the user's attention on the important things and hide the options of less importance in configuration dialogs, for example.

- **Task Orientation**

  Software Developers tend to design the GUI from the developer's point of view, e.g. the main attention is on the implementation and feasibility. Here, the user's concerns, namely performing some task efficiently, is often neglected.

- **Consistency**

  The application should be consistent in appearance and behaviour. If there are similar elements or behavioural structures, they should be consistent. This allows the user to get used to the GUI and learn and apply patterns.

2.2.2 Ordinary Approach

In ordinary programming techniques the need for Graphical User Interfaces is always handled the same way. The software developer has to identify the data that should be shown to or edited by the user at compile time. According to these data fields, he then has to create a GUI manually or using a graphical GUI Editor.

This approach has some big disadvantages:

- **High Effort**

  Creating GUIs manually is very time-consuming. Creating and integrating a GUI into an application involves two steps:

  - **Design**

    The software developer has to place one control per data field onto the widget. Therefore he has to think of a descriptive label and choose the right widget type for the corresponding data type. Additionally, the widgets should be grouped and placed reasonably, that means that logically related elements should be placed next to each other instead of spread over the form.

  - **Integration**

    In the next step the connection from the created controls to the underlying data has to be made. The developer has to create data structures to hold the data in memory and add logic to the application to dispatch changes of the GUI to the data and if it is needed, the other way around. Dependent on the data this can be a both very fiddly and error prone task, which often needs a lot of testing.

    Additionally, there are often dependencies and interactions between data elements which have to be mapped to the GUI. As an example, there are often data fields that are only relevant if another element fulfills a condition, e.g. if a certain checkbox is checked. These types of connections also has to be implemented manually which is - depending to the used framework - sometimes hard or cumbersome.

- **Non-uniform appearance**

  Every software developer has a different way of designing interfaces. Although there are guidelines for bigger projects such as the KDE HIG (Human Interface Guideline) [3] for applications that are supposed to be shipped with KDE, many developers still don't know of it.

  This results in fundamentally different GUIs, even if there are common patterns. The different look-and-feel for each application results in bad usability and might confuse the users. It also is bad from an aesthetic point of view.

- **Low-Quality interfaces**

  Software developers usually are not usability experts. That means that they have the knowledge and skills to create an application that presents some data to the user, but the quality of the presentation might be not the best possible. This might result in GUIs that are cumbersome to use and require more clicks to perform an action than it would be necessary with a optimized GUI.

- **Missing capability to adapt to data changes**

  The GUI is static. If the underlying data of the application changes, the GUI has to be adapted manually. This in return results in a recompile of the whole application which then needs to be shipped to the users again.

2.2.3 Automatic Approach

The problems described in the above section lead to the question, whether it is possible to automate the process of GUI generation. Often, the data that is processed with the application is specified in domain and type.

Taking these information into account, it becomes obvious that these descriptions are intimately connected with the GUIs that are needed to show or edit this kind of data. While it might not be possible to extract details such a sensible positioning of each element, it should still be sufficient to generate a description of the GUI that tells which elements are required and how they are related to the underlying data structure. This would be good enough to present the whole data and provide facilities to edit it.

2.2.4 Scenarios For A New Approach

- **Fast and automatic generation of a GUI**

  It is often necessary to create interfaces to simple data such as configuration files or something similar. These interfaces are usually pretty simple because there are only basic data types that need to be edited and there are only few dependencies if any. In these cases the automatic approach would allow the developer to automatically create the GUI and therefore require much less time than a manual approach.

  The GUI can simply be created from an XML Schema or any other complete data description language.

- **Creating a GUI for data that is likely to change**

  Another scenario is an application that works with data that is likely to change its format. This means that during the lifecycle of the application the underlying data might get additional fields or fields might be removed or renamed.

- **Prototyping**

  When creating applications, it is sometimes useful to create a first prototype in order to test the basic functionality or the interaction with other systems. For this purpose, the automatic approach would let the developer test the application over the whole development cycle without actually having to create and adapt a GUI first. This would again lead to less expenditure of time.

2.3 XML and XML Schema

XML (Extensible Markup Language) is a general-purpose markup language [16][17][18]. It is used in the IT world as an easy way to store and exchange arbitrary information between applications or systems. Another reason for the success of XML as a data container probably is the fact, that it is both human- and machine-readable, which allows fast and uncomplicated application development.

With XML, the data is arranged in a tree which exists of nodes. Each node can contain data, subnodes, attributes or a combination of those. This model is sufficient to map most of the data structures that are used in computer science.

It is often helpful to validate XML documents. Therefor, one can specify the structure of the XML with XML Schema. Only documents that conform to this Schema are then considered as valid.

```
<employees>
  <employee role="manager">
    <firstname>Dave</firstname>
    <surname>Smith</surname>
    <salary>300000</salary>
  </employee>
  <employee role="developer">
    <firstname>Mark</firstname>
    <surname>Rogers</surname>
    <salary>40000</salary>
  </employee>
</employees>
```

Listing 2.1: Example XML Document

XML Schema allows to define both data types and the document structure. Data types can be either SimpleTypes or ComplexTypes. SimpleTypes are basic types such as strings, numbers, time or date etc. XML Schema defines about 45 of them. ComplexTypes are either combinations of these primitive types or restrictions of them. Thus, it is possible to define a ComplexType `Employee` that consists of two strings and a number, describing the name, surname and salary.

An example for a restriction is a ComplexType that is a string, but that can only be one of a specific set of strings. Those restrictions (facets) allow to define a minimum and/or a maximum value or the amount of decimal places for numbers or the structure of a string. This is achieved with a regular expression that the string has to match.

The structural component of XML Schema describes the relation between elements, attributes and types. This allows the definition of lists of elements or in which order the elements have to appear in the document, for example.

The XML document above could be described with the following XML Schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:element name="employees">
   <xs:complexType>
     <xs:sequence>
       <xs:element ref="employee" maxOccurs="unbounded"/>
     </xs:sequence>
   </xs:complexType>
 </xs:element>

```

```
11  <xs:element name="employee">
12    <xs:complexType>
13      <xs:sequence>
14        <xs:element name="firstname" type="xs:string"/>
15        <xs:element name="surname" type="xs:string"/>
16        <xs:element name="salary" type="xs:integer"/>
17      </xs:sequence>
18      <xs:attribute name="role" type="xs:string"/>
19    </xs:complexType>
20  </xs:element>
21
22 </xs:schema>
```

Listing 2.2: Example XML Schema

2.4 XPath

XPath is a language that can be used to address elements of an XML document. In the simplest form, it just lists the steps that need to be taken to reach an element in the tree that represents the document. If one would for example want to select the first name of the manager in the XML example in the previous section, the XPath would look as follows:

```
/employees/employee[1]/firstname
```

Listing 2.3: XPath Example 1

If one would rather need all ”employee” elements that belong to a manager, it could be done that way:

```
/employees/employee[@role="manager"]
```

Listing 2.4: XPath Example 2

The ’@’ in the last example denotes that ”role” is not an element but an attribute of the ”employee” element. The expression in the square brackets is evaluated for all elements to be considered. An element is only selected if the expression is fulfilled.

There are other possibilities with XPath, such as computing values based on the content, but they are not used in this thesis and thus are not described further.

Chapter 3

Existing GUI Generation Projects

For automatic GUI generation a description format is required, which will be used to hold the information about the GUI. It should on the one hand be as simple as possible in order to ease the automatic creation, but it should also be flexible enough to allow the fulfillment of the design principles in 2.2.1. The level of abstraction should be chosen in such a manner, that the elements can be adequately described but that it at the same time is not depend on a specific system or application.

While there are only few and very limited applications that try to create GUIs out of schemas, there are some specifications of formats that describe UIs. This chapter will present the most important ones and evaluate if they are qualified for a basis for this work.

3.1 UIML

UIML is a meta-language which is designed to describe UIs that can be rendered on several devices and platforms [11]. That means, that it does not describe the elements of a UI specifically to a certain device or platform but follows a more abstract approach, so that it can even be presented on non-graphical systems, for example.

3.1.1 Concept

UIML separates the interface into three major parts: the presentation, the interface and the logic (see Figure 3.1 on page 19).

The presentation is responsible for the rendering on the respective system that is used. In the interface part the dialogue between the user and the system is specified. The logic part can be used to define interaction between the interface and the underlying system.

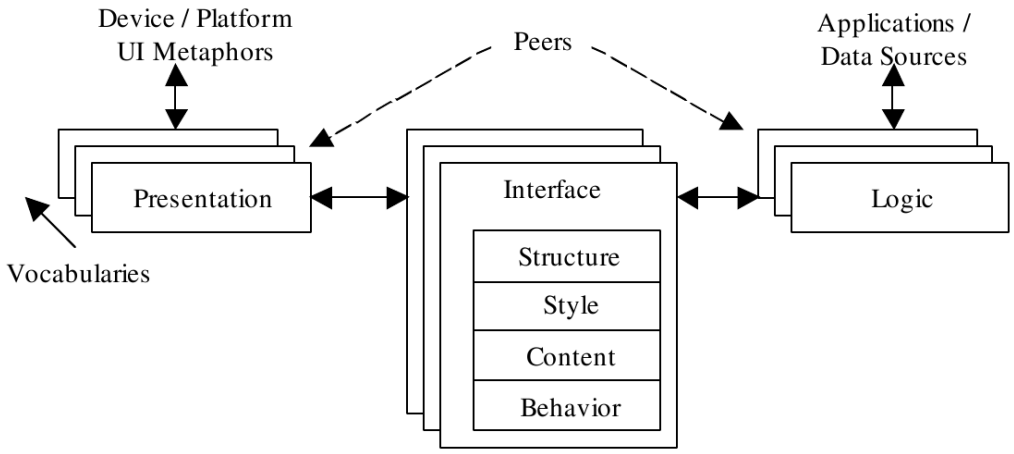

Figure 3.1: UIML Structure
Source: User Interface Markup Language (UIML) Specification [11]

3.1.2 Document Structure

A UIML document consists of a `uiml` root element, which may have 4 types of child elements. Each of these child elements is optional. The following sections describe them in detail.

The `head` Element

This element can be used to encapsulate metadata about the document. It consists of `meta` elements, which have a `name` and a `content` attribute and thus build a key-value-map which can hold arbitrary information.

It is often used to store information such as the author, date or version.

The `template` Element

With the `template` Element, one can define GUIs or parts of GUIs that can be reused by other GUIs. It might for example be reasonable to create a template for a person's contact information, which asks for name, age and address information. This template can then be used at different places in one GUI or even in other GUIs. That reduces redundant code and will make the interfaces for the same kind of information look the same, which will make the user feel more comfortable.

Most of the elements are applicable as templates. To do so, they need to be encapsulated inside a `template` element. This element has an `id` attribute which is used to instantiate this GUI snippet. Therefore, a placeholder element is put where the template should be shown at

and the `source` attribute is pointed to the according template. The bodies of the placeholder and the template element will then be merged.

The merge process can happen in three modes:

- **Replace Mode**

  In this mode, the elements in the body of the templates are appended to the original element. If there are conflicts, that means an element that should be appended already exists, the original child element is deleted and the template child element is then appended, thus the original child element was replaced.

- **Union Mode**

  In union mode no elements are deleted ever. If an element that should be appended already exists, the second instance of this element simply gets a different id before being merged.

- **Cascade Mode**

  Cascade mode means, that only child elements, that are not part of the original element already are added during the merge process. Thus, the information is not overridden by the template but only complemented, if the template has different child elements than the instantiating element.

The `interface` Element

This element contains the actual description of the interface between the application and the user, its structure, style and content.

- **UI Structure**

  In the first part of the `interface` element, the structure of the GUI is defined using the `<structure>` element. This element describes the organization of the GUI elements, e.g. if they are to be arranged on the same level or hierarchically. There can be more than one `<structure>` element if the organization should be different on different devices, for example. One might think of a voice-driven system which would probably better use a different organization than an application running on a computer would. If there are multiple structures defined, the respective rendering engine has to define the desired structure using its `id` attribute.

  Instead of describing a label showing some text as `<label>` and a button that can be clicked as `<button>` all elements are `<part>` elements. Each of them is further classi-

fied by a `<class>` attribute. The properties of the classes can be specified in the style-section of the document. Additionally, each `<part>` element has got an `id` attribute which is unique and can be used to identify the elements.

If one wants to express a hierarchy between two elements, they simply can be nested. This will result in the nested element being a part of the outer element, if the platform or toolkit supports this kind of presentation.

- **Styles**

  Another part of the document is used to describe the properties of the classes that are referenced by the `<part>` elements. Therefore, these `style` elements contain a list of property elements. Each property is related to one of the used classes and specifies the property that is defined and its value.

  The list of properties that can be defined is not part of the UIML specification, which results in a more extensible language. The valid properties are rather defined as a part of the vocabularies, which are used to map an UIML document to a specific device/platform. Properties that are often set are font, size or color of GUI elements.

  With styles, the appearance on specific devices can be optimized, so it is common practice to define one style per device that this GUI is targeted on.

- **Content**

  The `content` element introduces an abstraction layer for the content of the GUI. GUIs contain different types of content, for example text, audio or graphics. Instead of applying a fixed value to each of the elements, one can also create a link to a element specified in the content section.

  As an example, a label, asking the user if he/she really wants to proceed could be linked via the id ”confirmation_label”. Then there could be one `content` element per language. In the English content section the label value would be set to ”Do you really want to proceed?”, while in the German section it would be ”Wollen Sie wirklich fortfahren?” and so on. It is also possible to define different icons e.g. on a device that is targeted for visually impaired people. Every piece of content in the GUI can be specified per device using this technique.

- **Behaviour**

  The `behaviour` element is used to describe which actions are triggered when the user interferes with the GUI. Therefor, a list of conditions is defined, each of which is always

checked when the user performs an action. If one ore more of the conditions are met, the corresponding actions are triggered.

UIML distinguishes between two types of conditions. The first one is event-based, that means that a condition describes which event has to occur in order to meet the condition. An event is for example when a button is pressed.

The second type of condition specifies a logic expression that has to be fulfilled. An example for that is a condition that is met when the text of a lineedit element is changed to the value ”dog”.

The actions can be either local, that means that a specific part of the UIML document is changed, e.g. a property. The second possibility is to call a function in a script or execute an application.

The `peers` Element

The preceding structure gives us an abstract description of a GUI, which elements it contains, how they look like and how the application should behave when certain events occur. With the `peers` element it is possible to map this abstract description to a specific toolkit.

This can be achieved with two child elements:

- **The `presentation` Element**

  This element maps part and event classes and property and event names to the corresponding classes of the toolkit. Such a mapping is called a vocabulary. For the most common toolkits the software developer does not have to create a vocabulary but can use the already existing ones, for example for CSS (Cascading Style Sheets) , HTML (Hypertext Markup Language) or Java.

  If such a predefined vocabulary does not exist for a use case or is not sufficient, it is possible to either extend an existing vocabulary or to create a custom one, allowing UIML to be used with any toolkit.

- **The `logic` Element**

  In a section above, the `behaviour` element was presented which allows to specify how the GUI should interact with the user. What is still missing is the definition how the GUI should interact with the underlying application. This can be achieved with the `logic` element.

It therefore allows the definition of objects and functions which are mapped to external scripts, methods or objects. In the next step, the calling conventions are specified, namely the name of the functions, the return type and the number and type of the arguments. These objects and functions can then be used in the UIML document.

```
1  <interface name= "Simple␣Example">
2   <window name="Main" content="UIML␣Example">
3    <menubar name="Selections">
4     <menu name="FileSelection" caption="File">
5      <menuitem name="item0" caption ="New"/>
6      <menuitem name="item1" caption ="Close"/>
7      <menuitem name="item2" caption ="Quit" onClick="Main.hide"/>
8     </menu>
9    </menubar>
10  </window>
11 </interface>
```

Listing 3.1: UIML Example

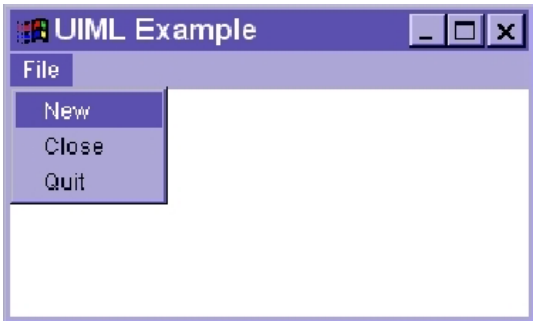

Figure 3.2: Generated GUI From An UIML Description

3.2 XUL

XUL (XML User Interface Language) is another user interface markup language which is developed by the Mozilla Project [13]. In contrast to UIML (3.1) its purpose is to define cross-platform computer applications, not abstract interfaces suitable for several different devices. Thus it is a widget-based description on a far lower level. The described GUI can be specified in detail and can be rendered on different platforms by different applications. One application

that is capable of XUL is the Mozilla Gecko rendering engine, for which XUL was originally designed for.

XUL strictly separates the presentation layer from the logic layer, which results in code that can be understood and maintained more easily and simplyfies the customization or translation of the application. The following sections will describe these concepts in more detail.

3.2.1 GUI Layout

Given the fact, that XUL was designed for a web-browser application, it is not remarkable that it uses technologies that are known from the World Wide Web. It uses CSS for describing the appearance of the elements and Javascript for the logic, for example. This section describes the layout facilities that are used.

As stated above, XUL is a widget-based description language. As a first step, XUL introduces the concept of boxes. Boxes contain child elements and layout them according to the layout directive.

The following list shows the most important properties that can be defined for a box or an element:

- **Orientation**

  The orientation of a box defines on which axis the elements are aligned. One can choose between horizontal and vertical orientation, placing the elements next to each other or on top of each other respectively.

- **Direction**

  Normally, the elements are placed from left to right. This setting can however be set to right-to-left using the direction property. It's also possible to inherit that setting from a parent element.

- **Alignment**

  With the alignment property it is possible to specify how the elements should be positioned inside of the box. By default, it is set to "stretch", which means, that all elements are of the same height, when they are put in a horizontal box, and of the same width, when they are put in a vertical box. Other options are "start", "end", "center" or "baseline" which will position the elements accordingly.

- **Element Sizes**

  Elements can be either of fixed size or flexible. Flexible means, that the element will use unused space, while a fixed size element just uses the space it needs for its size and leaves unused space empty. The flex property is automatically set to false, if an element has a size value set. In case of a flexible element, it is also possible to specify a minimum size of the element that the size will never go below, a maximum size and a preferred size.

  If there should be no flexible elements in a box, or all elements have already reached their maximum size and the box still grows further, the behaviour can be defined by the packing property. This allows to define where the conglomeration of elements should be shown inside the surrounding box.

  This allows the definition of complex layouts that resize properly with regard to the content or the box size.

Besides these box items, XUL also allows to use 3 other containers for the elements:

- **Stacked Elements**

  If elements are stacked, they are simply put on top of each other. That means, that only one of them can be visible at a time.

- **Grids**

  Elements also can be put into a grid. That allows to specify the row and column. Elements in a grid can also be nested, opening new possibilities for a GUI.

- **Popups**

  Finally, it is possible to create Popup elements, which are not part of the GUI directly but appear on top of it. Examples for popups would be a tooltip or a menu.

3.2.2 GUI Elements

All widgets and controls in XUL are derived from XULElement. This interface defines a name and id for the elements and specifies several properties, for example for the layout, the size and position, tooltip and status information and some more.

- **Description and Labels**

  These two elements are used to display string information to the user and thus are the most basic widgets available. The description element is capable of HTML markup and

is completely stylable using CSS. It additionally has a crop property which defines at which position the content should be cropped if the available space is not sufficient to display it completely.

A label is a specialised description. It is associated with another element and is capable of shortcuts, which will activate the corresponding control when actioned.

- **Images**

  It is also possible to embed images into the GUI. This can be achieved using the `image` tag which is derived from the HTML `image` tag.

- **Controls**

  A control is a widget that allows the user to interact with the application. There are two types of controls available in XUL, which are described in the next section in more detail.

3.2.3 Controls

All controls share common properties for being disabled and the tabindex describing the order in which the elements are activated when the user presses the "TAB" key. Additionally, controls have methods that are called, when the control gets the focus or is blurred.

- **Labeled Controls**

  A labeled control is a control that is connected to a specific event. That means that whenever the control is activated, the corresponding event is generated. The most common form of a labeled control is a button. Buttons can appear in different forms, being regular buttons, buttons with a menu, toolbar buttons, radio buttons and checkboxes.

- **Select Controls**

  Select controls offer some options to the user, who can then make a choice. There are also different ways of presentation possible, such as a listbox, a dropdown list of a group of radiobuttons.

3.2.4 Overlays

XUL supports overlays, which is a mechanism for extending existing applications dynamically. This can happen in two directions, namely with the application including a specific overlay or the other way around, thus an overlay plunging itself in an existing application.

That allows the software developer to create an application that uses an external XUL file as overlay, which can be customized without touching the main application, and makes it possible to provide an easy way of creating extensions to an application. The latter can for example be seen at Firefox, which provides such an interface. There are already many extensions available for almost every need one might think of [12].

3.2.5 XBL - Extensible Bindings Language

With XBL the software developer gets a tool that enables him to create custom widgets or event handlers. It allows to extend the given functionality of XUL very flexibly. The developer can create customized progressbars with special behaviour, new popup menus, toolbars and so on. Almost every piece of GUI can be created and used in the application.

3.2.6 XPInstall

XUL ships with a install tool, called XPInstall, which makes the installation of XUL applications on the supported platforms as easy as possible. The user basically only has to click on a hyperlink or open the location of the XPI file, which will open the Mozilla installer. After another click to accept the installation, the application gets automatically copied onto the hard drive and installed. No manually downloading or copying of files is required.

```
1  <?xml version="1.0"?>
2  <?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
3
4  <window id="example-window" title="Example␣2.2.1"
5     xmlns:html="http://www.w3.org/1999/xhtml"
6     xmlns="http://www.mozilla.org/keymaster/"
7           "gatekeeper/there.is.only.xul">
8   <button label="Normal"/>
9   <button label="Disabled" disabled="true"/>
10 </window>
```

Listing 3.2: XUL Example [22]

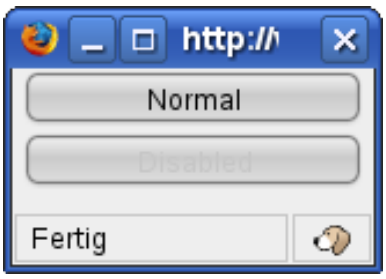

Figure 3.3: Generated GUI From A XUL Description

3.3 XForms

The XForms 1.0 specification was published as an official recommendation by the W3C in March 2006 [15]. In November 2006, version 1.1 was released as a Working Draft.

XForms uses the Model-View-Controller concept. There is a **model**, which describes the data that can be used with the form and what is to be done with the data and the **view** which defines how the model is presented. The **controller** part is given as an complex event framework.

3.3.1 The XForms Model

The model of a XForms form is specified in the `model` tag of the XForms document. The document can also contain more than just one model which can then be referenced in the view.

Below is an example XForms model taken from the XForms specification [19]:

```
1  <xforms:model>
2   <xforms:instance>
3    <ecommerce xmlns="">
4     <method/>
5     <number/>
6     <expiry/>
7    </ecommerce>
8   </xforms:instance>
9   <xforms:submission action="http://example.com/submit"
10       method="post" id="submit"
11       includenamespaceprefixes=""/>
12 </xforms:model>
```

Listing 3.3: XForms model

The model description is used to

- **Specify the structure of the data**

  As shown in the example above, the model defines the structure of the data using the `instance` tag. In this case, there is a tag called “ecommerce” which has an attribute “xmlns” and three child elements “method”, “number” and “expiry”. Nothing is said about the type of the data elements yet, however.

- **Preload the model with data**

  The instance element can also be used to set initial data that will be shown in the form. Therefor, the XML data skeleton can simply be filled with the desired data.

- **Define the action that is to be performed**

  The example also shows an `submission` tag. It is used to specify the target, where the data is transmitted to and parameters such as the method, the encoding or the id of the action.

- **Declare properties for the model elements**

  Finally, the elements of the model can be specified in more detail using the `bind` tag. Therefore, a subset of the model is selected using the `nodeset` argument and one or more properties that are to be applied to these elements.

  The most important attributes are:

  – **The `type` property**

    This property defines the type of an element of the model. The default is `xs:string` making all data strings. If one would want to make the credit card number of the above example a number, it could be achieved with this bind expression:

```
1 <bind type="xs:integer" nodeset="/ecommerce/number" />
```

    where the desired data type is given as the `type` attribute and the node is selected with a XPath expression in the `nodeset` attribute.

  – **The `readonly` property**

    With this property, elements can be made readonly, preventing the user to make changes to it.

- **The `required` property**

  If an element is marked as required, the XForms processor will ensure that the element is non-empty before it allows the transmission of the data.

- **The `constraint` property**

  The `constraint` property allows to limit the values that are considered valid for the element. That means that the transmission is only allowed if the given expression if fulfilled.

3.3.2 The XForms View

Once the model definition is in place, it can be instantiated using XForms controls. There are controls available for the most common use cases. The most important ones are explained below.

Each control is related to one XML element in the model. That means, that it will display the data of that element in the initial XML document (the instance of the model) if there is one, and that the data of the control will be put into the element when the data is written back.

These are the most important controls of XForms 1.0:

- **The `input` element**

  The `input` element is used for free-form data entry, e.g. arbitrary text. In a web browser for example, it is usually presented to the user as a line edit.

- **The `secret` element**

  If the user needs to enter secret information - a password for example - it is desirable to not show the data in cleartext. This can be achieved by using the `secret` element.

- **The `textarea` element**

  This control enables the user to input longer texts with linebreaks in it.

- **The `select` element**

  With the `select` element, a list of possible values can be presented to the user, who can then select one or more of them.

  If the number of selected items should be limited to only one, the `select1` element has to be used.

Besides these data controls, there are also some interaction controls available:

- **The `upload` element**

  This element allows the user to upload files onto the system. This needs not necessarily be a file on the local hard disk, but can also be a sound recorded from a microphone or an image from the attached scanner. The application has to provide the facilities to select the appropriate data.

- **The `trigger` element**

  The `trigger` element enables the user to trigger actions, the element is bound to. The usual way of presentation in a web browser would be a push button.

- **The `submit` element**

  With this element, the user can submit the data that was entered in the form. The submit element can also be bound to only parts of the XML document, so a part of the document may be submitted.

Figure 3.4: input, textarea and select control rendered in a browser
Source: XForms 1.0 (Second Edition) [19]

The controls can be specified further with some properties.

- **The `appearance` property**

  Many controls can be rendered in different ways, occupying more or less space. With the `appearance` element, the author of the XForms document can tell the application how the data should be presented. The possible values for the appearance are “full”, “compact” and “minimal”.

  With a “full” appearance it is ensured that every bit of information is always rendered. With a list for example, this means that every entry of the list is rendered. It might however still be necessary to scroll or browse in order to see all the elements, if they don’t fit on the screen at once.

A “compact” appearance will show a bigger part of the data at once, but not necessarily all of it.

“minimal” means, that the control should use as little space as possible, with a list resulting in a control that shows only one entry at once.

- **The `hint` property**

  The `hint` property can be used to define a explanatory description of the according element, e.g. describing the element in more detail. The application then has to provide a way for the user to find this information. A web browser would show the hint as a tooltip, for example.

3.3.3 Events and Action

The behaviour of the controls on a form can be customized using XML events and XForms actions. Events are generated and emitted at various stages of the processing of a model and the interaction with the user. These events can be bound to actions which will alter the state of the form.

The events can be separated into four groups:

- **Initialization events**

  These events are only emitted on the initial processing of the form. One event is `xforms-model-construct-done`, for example, which indicates, that the model was successfully parsed and processed.

- **Interaction events**

  These events are the result of the user interaction with the form. These include `xforms-previous` and `xforms-next` events, that indicate that the previous or next control should be focused, the `xforms-hint` event that is emitted when additional information about a control should be shown and many more.

  This allows to determine every action, the user performs on the form.

- **Notification events**

  Notification events indicate state changes of controls. Examples are: a changed value of a control (`xforms-value-change`), a control getting focused (`DOMFocusIn`), a control becoming invalid (`xforms-invalid`).

- **Error indication events**

  In order to react to errors, one can use the error indication events. An example is the `xforms-binding-exception` which indicates, that a binding expression is invalid or that the target could not be located.

3.4 Other languages

There are some more languages such as Microsoft's XAML [7] [14], WML [20] or Macromedia's MXML [1], but those are either proprietary formats, no longer developed further or designed for specific platforms or applications and hence not applicable for the purpose of this work.

3.5 Conclusion

UIML, XUL and XForms are all powerful description languages. Each of them is capable of describing GUIs system or platform independently, but they differ in some properties, which gives each of them the right to exist.

UIML is the most generic solution. With the high level of abstraction it can be used to describe interfaces for arbitrary systems. With its concepts of styles and vocabularies it is also possible to customize the look or behaviour as detailed as desired. Furthermore it provides facilities to specify the behaviour of single elements in many ways. However, this approach is rather complex and many steps need to be taken before a usable GUI emerges. Mapping this to an automatic generation would be hard and make the implementation difficult.

XUL in contrast takes a lower level approach. It is more focused on computer applications what fits the requirements of this thesis better. The low level description is however too detailed and specific to be generated automatically. A higher level of abstractness is needed there.

This right degree of abstractness is given with XForms. XForms strictly separates the model definition from the view definition. The view is instantiated using controls, but the actual rendering of the form is left to the application, which makes it possible to display the form on different systems.

Although some parts of XForms, such as the events and actions, are not required for an automatic GUI generation targetted for computers, the high-level control and properties definitions make a good base for this project. Thus, XForms is the chosen underlying format used in this work.

Chapter 4

Architecture

This chapter gives an overview of the reasons and the design of this project.

4.1 Basic Conditions

The implementation of the prototype should be released and developed as open source software (OSS). Furthermore, the chosen platform is Linux and the desktop environment that is used for the application is KDE. KDE was chosen, because it allows rapid development of high quality applications with its clean and consistent API and the Qt application framework (see 5.1.1 and 5.1.2).

4.2 Use Cases

The following use cases should be handled by the result of this thesis:

- **Create a GUI in order to show or modify some data**

  The most common use case is, that a user wants to have a GUI that allows to edit some specific data. The user has some data and a corresponding description of its format and wants to show or modify this data without creating a dedicated application for it first. This should be done by only reading the schema of the data. The rendering application then shows the generated GUI and provides a way to read the data instance for further processing.

- **Modify a generated GUI**

  If a generated GUI does not fulfill all of the requirements, it should be possible to edit the GUI and generate a description of the changes that can be used to apply these

changes to the automatically generated GUI. These changes could for example be done by an usability expert, that knows how to create good applications that are easy to use and intuitive. This could be useful, if the data description is fetched from a server on application start up, for example. Then, the description of the GUI changes could be fetched too and merged in, before the GUI is shown.

- **Embed a dynamically generated GUI into another application**

  Often, the main GUI of an application is static, but some data, such as configuration data, is not. In these cases it would be desirable to generate a part of the application's GUI dynamically. This could be done by providing this functionality in a library which could be used by other applications.

4.3 Data Layers

There are several layers involved to get from a schema to a GUI, which are shown in Figure 4.2 on page 37. At the lowest layer, there is some data, which is to be handled by the application. It can be either only shown, or also be editable by the user. This data layer is encapsulated by a schema layer, which is a description of the format, in which the data is given. This could be an XML Schema for an XML document (2.3), for example.

In the next layer, this data description is abstracted in an abstract schema layer. This approach allows to handle different data backends. For each supported data format a parser has to exist that transforms the concrete description for a specific data format instance into an abstract description.

On the same level there is also a hint layer, which allows to enrich the description with additional information on parts of the data format. This is useful if the data description is not sufficient in some parts, or a usability expert wants to influence the GUI generation process at some point, e.g. change the label of a specific element.

With the abstract schema a GUI description is generated, which together with the hints describes the elements of the GUI, their structure and data types.

Finally, a rendering application reads this GUI description and transforms it into a usable interface. It is remarkable, that there are no restrictions on the type of the application or the platform. So, the same GUI description could be rendered both in a native application and in a web application, for example.

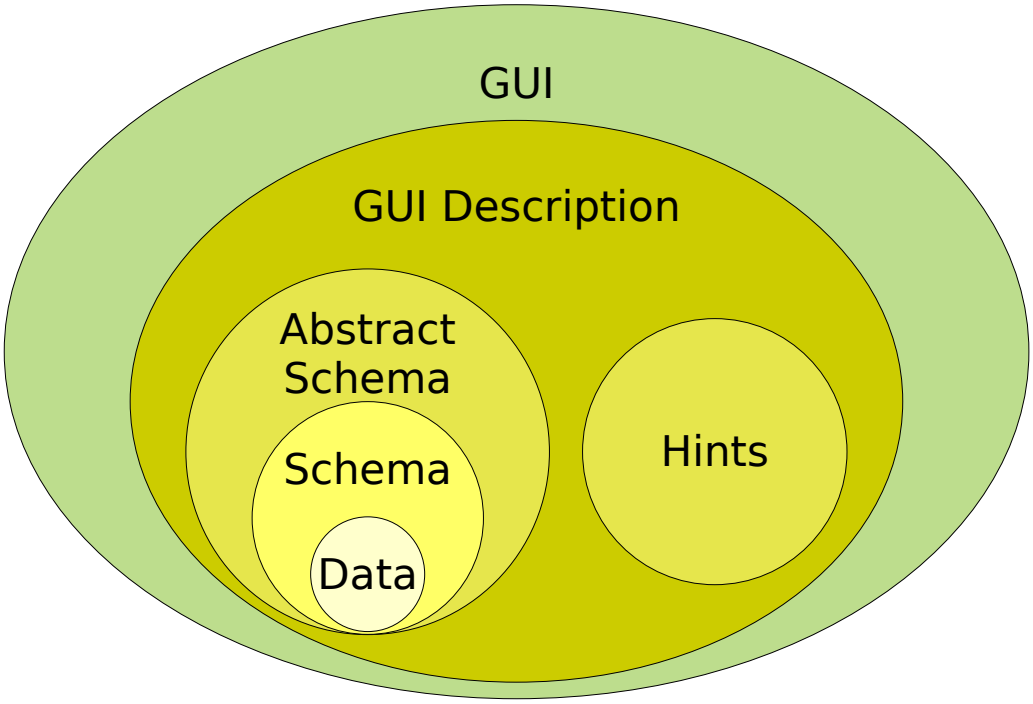

Figure 4.1: Layers in GUI generation

4.4 Components And Interaction

Figure 4.2 on page 37 shows the components that are needed to cope with the tasks described above. There is a “Storage” component, which is used to hold both the data that is to be processed and the schema of this data. It does not matter where the data is located physically. Both local files, remote data storage ore a combination of both is possible.

The schema is parsed in the “Schema Parser” component, which extracts potential hints and transforms the data description into an abstract schema.

The abstract schema is processed further in the “GUI Description Generator”. The result is a GUI description document which is referred to the “GUI Generator” part of the rendering application. This component also retrieves hints from the “Schema Parser” or hints from the “Storage“ if there are any.

With this data, the final GUI is generated. The application can then be used to load, create or modify data from the ”Storage“.

If the GUI has to be edited this can be done with the ”GUI Editor“ component of the application. It will generate hints describing the changes, which can be either stored for later usage

or directly passed on to the ”GUI Generator“. The ”GUI Generator“ then recreates the GUI with the new hints.

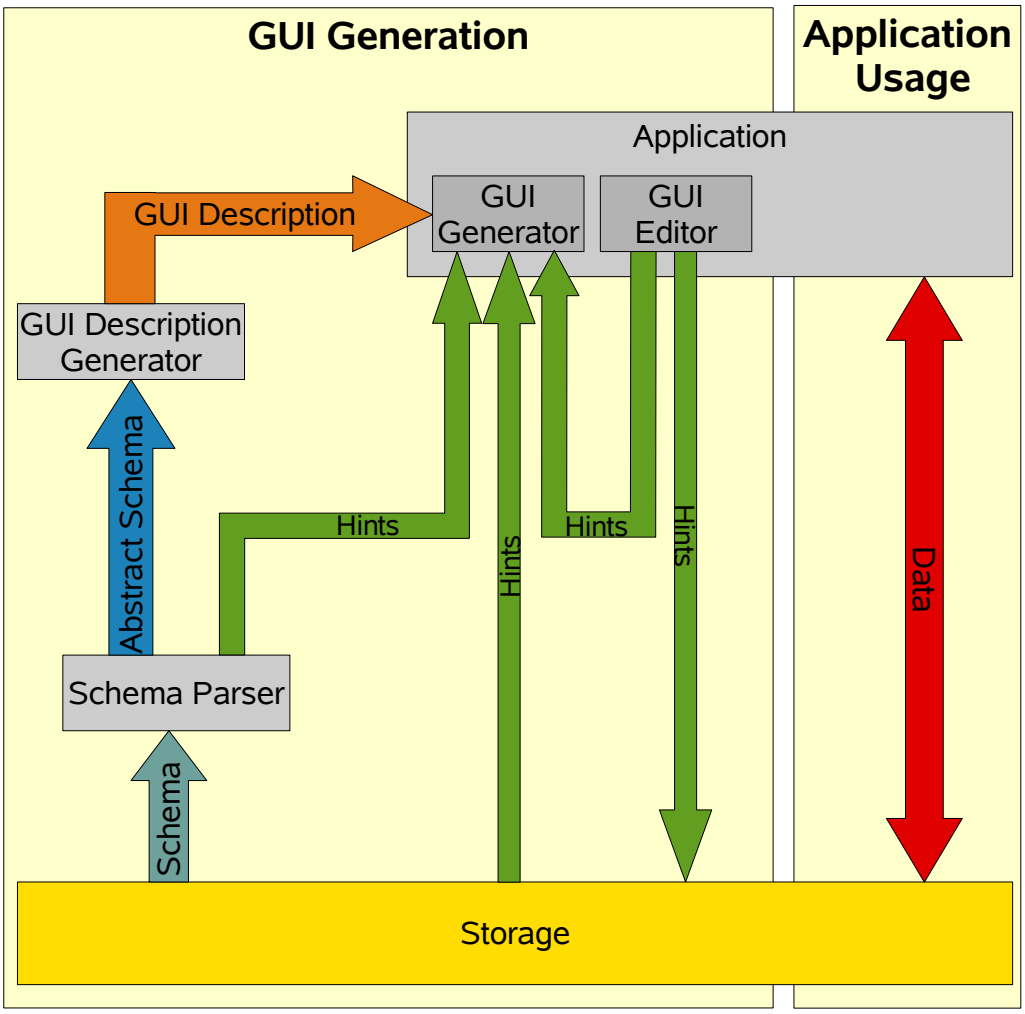

Figure 4.2: Components and Interaction

Chapter 5

The Kode Project

The prototype of this thesis was done by improving and extending the Kode project of KDE [4]. It already provided some working parts for parsing schemas and generating GUIs when this work was started.

5.1 Used Tools And Technologies

This section describes the tools and technologies that are used in this thesis.

5.1.1 The Qt Application Framework

Linux and Mac OSX are constantly gathering market share from Microsoft operating systems. For software developers, that means that they either lose potential customers if they only provide their software for one platform, or that they have to put extra work in porting their applications. This issue is addressed by the C++ cross-platform application framework Qt4. It rapidly decreases the porting effort. In fact, targeting a new platform just requires to compile the sourcecode again.

Qt is developed by the Norwegian company Trolltech and released with two different licences, a commercial one for companies that do not want to release the source code of their applications and a free open source edition, which can be used by open source developers.

Besides the ability to create cross-platform applications Qt has many other advantages:

- **Native Look-And-Feel**

  Qt applications keep the Look-And-Feel of the operating system. This makes the application integrate into the desktop and lets the user feel comfortable (see Figure 5.1 on page 39).

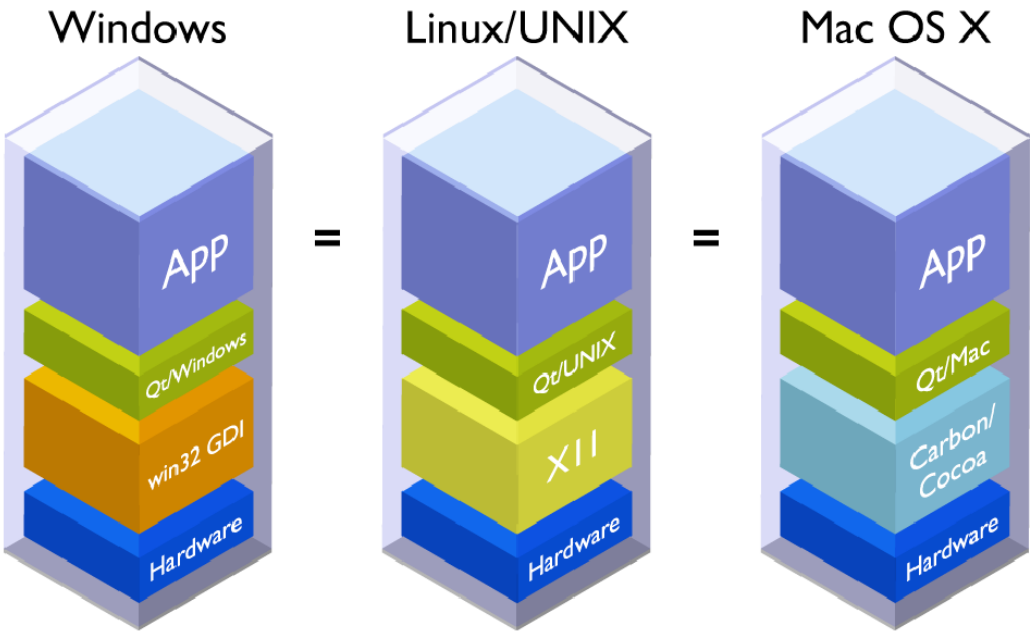

Figure 5.1: Qt Architecture
Source: Introducing Qt (http://trolltech.com/products/qt)

- **Wide variety of convenient classes and development tools**

  Qt offers classes for many tasks a software developer has to deal with. This includes the core and GUI classes for the framework and graphical user interface components, networking, XML, SQL, OpenGL and many more. It also ships with some tools that simplifies the creation of high quality applications, such as an graphical GUI designer and applications for translation and documentation.

- **High performance**

  The Qt classes are deeply tested and optimized and all aspects of the framework perform and scale very well. With Qt4 the speed was improved even further and the memory footprint was decreased. Thus the developer gets a lean but fast application.

5.1.2 The KDE Desktop Environment

KDE is a desktop environment for Linux/Unix which is based on the Qt application framework (5.1.1). Parts of the next version, KDE4, will also be available for Windows systems, which is possible because Qt is also available as an open source edition for Windows since Qt 4.0.

KDE provides a complete working environment with applications for the most common tasks, such as office applications, web browser, email client or games.

It also offers good documentation for software developers and the Qt basis, in combination with very good API designs, makes it very easy to develop new applications. The Kode project is also a part of KDE. In parallel however, a Qt-only version is developed. With the use of compatibility classes the code does not have to be duplicated but both versions can be built from the same code.

The idea behind the Qt only version is, that the functionality of Kode is also interesting for Qt developers that do not want to depend on the KDE libraries.

5.1.3 Unit Testing

Handling and working with XML is a very fiddly task and changes even to small parts of the implementation might have severe side-effects which are hard to spot and track. Thus, the whole development phase was coupled with unit testing. So, after changes were made to the implementation, it could be easily verified that all other parts were still functional and working as expected. Therefore, some testing data and the expected result were defined. The comparison of the actual results with the expected ones could then be easily and automatically done within the QTestLib framework of Qt4.

5.2 Prior Work

This chapter lists the parts, that were already there when this work was started and describes what they are used for.

5.2.1 Schema parser

Tobias Koenig has created a schema parser based on wsdlpull parser by Vivek Krishna. This parser reads an XML file describing the schema of the data structure that should be handled by the application and creates an object tree representing it. It is capable of

- parsing “elements”, “attributes” and “attributegroups”
- processing the complexity models “simple”, “complex” and “mixed”
- identifying the XSD types such as “string”, “integer”, “date”, etc.
- identifying the compositors “invalid”, “choice”, “sequence” and “all”

All further processing of the schema is based on the output of this parser, making it an essential part of Kode.

5.2.2 kxml_compiler

kxml_compiler is a little tool developed by Cornelius Schumacher and Tobias Koenig, which allows to create source code for editing data according to a schema description. It therefor includes a wrapper library around the parser described in 5.2.1. This library consists of some helper classes and two parsers, namely ParserRelaxng and ParserXsd. While ParserXsd is used to represent XML-Schemas, ParserRelaxng represents RelaxNG schemas.

For this thesis, only the library was used in order to create an abstract representation of the schema. Thereby, only XML-Schemas are considered, because they are far more widespread than RelaxNG. kxml_compiler defines an own set of classes describing the elements of an XML-Schema. Those elements are encapsulated in a document class which can then be processed further.

5.2.3 KXForms

KXForms is a KDE application which uses the kxml_compiler library to fulfill two main tasks. The first is to create a kxforms document from the schema document created by the wrappers (5.2.2). Secondly it should create a graphical user interface from this kxforms document.

Basically, kxforms was able to create simple kxform descriptions and create simple GUIs, but it was also very buggy from the transition from Qt3 to Qt4 and often failed to create even trivial schemas. Many parts, that are required for more complex schemas were missing.

5.3 Formats

Figure 5.2 shows the formats that are used in the implementation of the system described in 4.3. As can be seen, all of the formats are based on XML. The input is usually given as an XML Schema (2.3). The kxml_compiler and KXForms applications transfer it into an kxforms document. Additionally, hints can be generated with an Editor mode, influencing the appearance of the generated GUI.

The hints and the kxforms document together form the description of the GUI, which will be instantiated by KXForms.

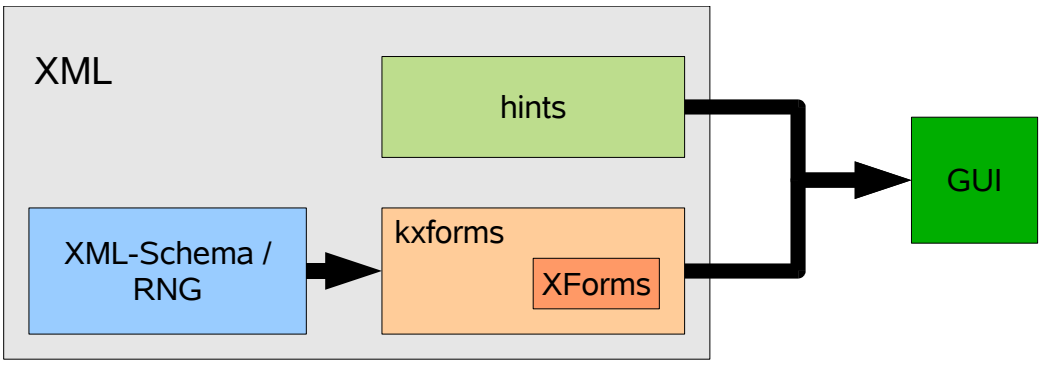

Figure 5.2: Formats used in KXForms

Chapter 6

The kxforms Specification

In this chapter, the concepts and structure of the kxforms specification are presented. The purpose of this specification is to define a format that can be used to describe GUIs in an abstract matter, so that it can be rendered on different systems. It is based on the XForms specification (see 3.3 for more information), but extends it with some elements and attributes in order to allow more appealing applications on computer systems.

Elements that are derived from XForms are put into the "xf" namespace, which can be identified with the "xf:" in front of the actual element name.

The kxforms specification was released with this thesis and can be downloaded from the KDE subversion repository [5].

6.1 Structure

The kxforms description language is a dialect of XML, thus it can be used with existing XML tools. The root element of an kxforms document is the `kxforms` element. This root element is just an encapsulation element for the forms that exist in the GUI and an optional definition of the defaults that are to be applied to them and their controls.

The properties of the kxforms controls are sanely chosen and should work for most use cases. There are however some properties that need to be adapted if the underlying data structure has an unusual form, for example. This can be done globally with the `defaults` element. Therefore, the desired properties are simply put inside this element and the application will predefine the properties of all forms and controls with these values. Applicable are all properties of kxforms, as they are described in 6.1.3.

6.1.1 Forms

A form in kxforms is a piece of GUI that represents a set of data that is needed for one working stage. It is not required that it shows all of the data on the screen at once. In an application for the management of a CD collection, there could be 2 forms, for example. The first one would be used to show a list of all the CDs in the collection. The second form would appear if a CD was selected and show all the information about that CD, such as the artist, title, year and so on.

kxforms has a group concept, which means that elements can be assigned to groups. All elements of one group logically belong together. The rendering application can take this information into account when it positions the elements on the forms. If there are more than one group and they don't fit on the screen at once, the application can put them on different tabs, several pages or similar, according to the type of application. The groups are defined inside of the `form` elements as a `groups` element. There, the different groups are listed with an unique id and a title, which can be used to identify the groups.

```
1 <form ref="feature">
2  <xf:label>Feature</xf:label>
3  <groups>
4   <group id="">Feature</group>
5   <group id="other">Other</group>
6   <group id="documentation">Documentation</group>
7  </groups>
8  ...
9 </form>
```

Listing 6.1: Example group definition

If an element should belong to a certain group, it references the group id as described in 6.1.3.

After this list of groups, which is optional, the GUI elements on that form are listed. If elements are not assigned to a group, they are ordered after the position in the XML document, e.g. the second element will be positioned below the first one, the third below the second one and so on.

If there are groups, it is ensured that all elements of one group are positioned contiguously. In this case, the group which the first element in the XML file belongs to, will be the first group shown and so on.

kxforms also provides facilities for positioning the elements inside of one group. The positioning concepts are described in more detail in 6.2.

6.1.2 Controls

A control or GUI element is represented by an XML node in the kxforms document. There are many controls available for the most common data structures that are used generally. Most of them are derived from the XForms specification and extended with kxforms specific attributes or child elements.

Each control is related to an XML node in the underlying data XML document, which holds the reference data, that is edited with the rendering application. This relationship is stated in the `ref` attribute, which is a XPath expression to the according node. Every control may also have a set of properties, which will override the global default properties. The properties are listed in 6.1.3.

kxforms defines the following GUI elements:

- **The `xf:input` Element**

  This element is derived from XForms. It is used as an input field which enables data entry. Nothing is said about the type of the data, however, which is done with properties ().

- **The `xf:textarea` Element**

  This element is also derived from XForms. A textarea is used for free-form text data and can contain arbitrary text. In contrast to the `xf:input` element it can also control sequences, e.g. linebreaks.

- **The `list` Element**

  This element enables handling of lists of elements. The XForms list element can only be used for a list with only one type of elements in it. For kxforms however, it was important to also allow different elements in the same list, because this is crucial to represent some data structures. An example is an item catalogue which is separated into categories. Each category can then have a list of items which belong to it, but it might also have subcategories.

  Thus, a kxforms list can contain one or more types of elements which are specified by the `itemClass` child element. One such element has to exist for each type of element that can appear in the list. Besides the XPath expression to the XML node in the data

document, it also contains the definition of what data of the element should be shown in the list.

Therefor, one `itemlabel` element per column is added as a child of the `itemClass` element. The label of the items can be either some static text, some data from inside the element or a combination of both. This allows to polish the presentation of the data in order to increase usability and readability. It is even possible to combine several fields of the item in one column. The inclusion of item data is done with the `itemLabelArg` element. It just references a node or attribute inside the element and it will be replaced with the actual data in the list. This only applies to lists of ComplexType elements, as with SimpleType elements, there is only one data field that can be shown in the list.

The headers of the columns can be defined using the `headers` child element. It just contains a list of `header` elements giving the text for each of the columns that are used within the `itemlabel` elements.

The following example shows the usage of the `itemclass` and `headers` elements. The combination of static text and item data can be seen with the "version" column, which prefixes the actual data with a "V" string.

```
1  <list id="list_productcontext" showHeader="true">
2   <xf:label>Productcontexts</xf:label>
3   <itemclass ref="/productcontext[1]">
4    <itemlabel><itemLabelArg ref="/product[1]/name[1]"/>
5         </itemlabel>
6    <itemlabel>V<itemLabelArg ref="/product[1]/version[1]"
7         truncate="40"/></itemlabel>
8    <itemlabel><itemLabelArg ref="/status[1]"/></itemlabel>
9    <itemlabel><itemLabelArg ref="/architecture[1]"/>
10        </itemlabel>
11   <itemlabel><itemLabelArg ref="/@legacyinfo[1]"
12        truncate="40"/></itemlabel>
13  </itemclass>
14  <headers>
15   <header>Product</header>
16   <header>Version</header>
17   <header>Status</header>
18   <header>Architecture</header>
19   <header>Legacy</header>
20  </headers>
```

```
21 </list>
```

Listing 6.2: Example list definition

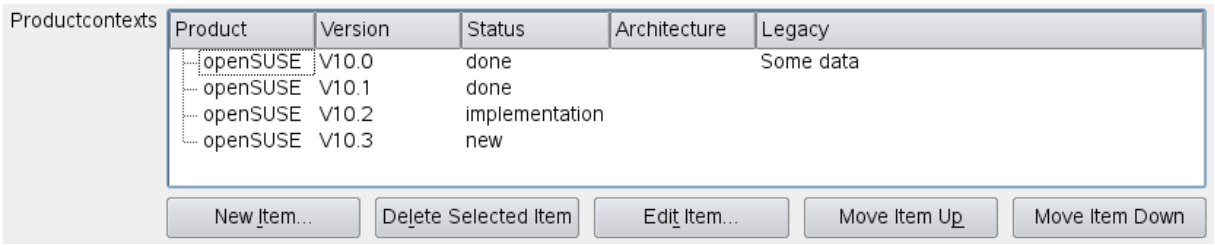

Figure 6.1: Example list

Lists can be customized to show or hide the headers and a filter bar using the `showHeaders` and the `showSearch` attributes of the `list` element. This can be useful for long lists. It only shows the items of the list, which contain the given search string.

Figure 6.2 shows a list control which has both the headers and the filter bar activated.

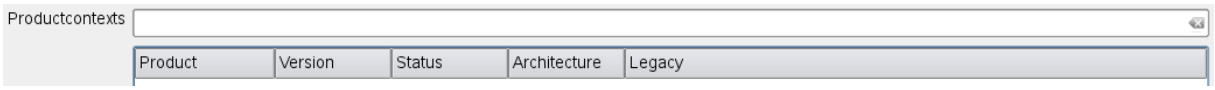

Figure 6.2: List control with headers and filter bar

- **The `section` Element**

  The section element is used to visually encapsulate a set of elements. That is appropriate when a complexType element with several child elements is shown, for example. This will result in a less cluttered GUI.

- **The `xf:select` Element**

  This element presents a set of options to the user and lets him select one or more of them. The items are defined with `xf:item` child elements. There can be an arbitrary number of items. They will be shown in the same order as they are listed in the XML document. Each of the items has two child items: One defining the label of the item, which is used in the GUI and an value, which is the data that will be used in the XML document if this item is selected.

- **The `xf:select1` Element**

  This element is basically the same as the `select` element. The only difference is, that it limits the number of chosen options to only one.

6.1.3 Control Properties

Controls can be customized in their appearance and behaviour. This can be done by adding a `properties` child element to the control, which contains the property elements.

kxforms defines three different types of properties, which are explained in the list below:

- **Common Properties**

  These properties are applicable to all controls and are behaviour properties.

  - **The `readonly` Property**

    This property determines, whether an element can be edited by the user or not. If it is false, the control behaves just as expected. If it is set to "true" however, the control still displays its data, but it can not be changed. This state should also be visually indicated by the application.

  - **The `relevant` Property**

    Sometimes, parts of an XML document are only interesting if some condition is fulfilled. If there is a document describing an order by a customer, for example, the fields for the creditcard number and the expiration date of the card is only relevant, if the payment method is set to "creditcard". They have no effect if the payment is done via bank transfer or account.

    This can betransferredd to the according GUI element with the `relevant` property. It contains a reference attribute, which points to the other element, the content of which is to be watched. The condition is then given with a regular expression. If the expression matches, the elements state is set to be writeable, if it does not, it is set to read only.

    The following example shows a solution to the scenario described above.

```
1 <relevant ref="/paymentmethod">creditcard</relevant>
```

Listing 6.3: `relevant` property example

- **Input Properties**

  Input properties can be applied to input controls, being `xf:input` and `xf:textarea`.

– **The `type` Property**

This property defines the data type of a control. It defaults to “xs:string”, but all data types that are defined in the XML Schema specification are possible. This allows the application to verify that the data of the control is a valid entry for the specified data type, e.g. that an “xs:integer” consists of digits only and so on.

Another advantage is, that the application can render specific controls for some types. For an input control that expects “xs:integer”, it might show a SpinBox, while for “xs:string” it would show a LineEdit.

– **The `constraint` Property**

The `constraint` property allows to define a regular expression that has to match the content. If it does not match, the contents state would be invalid, thus preventing the form being saved.

The table below shows some use cases for this property and the according regular expression.

Constraint	**Regular Expression**
A number	$\backslash d*$
A number bigger than 2	$\backslash d*[3-9] \mid \backslash d*[1-9]\backslash d+$
”true“	`true`
A XML tag	$<[/]?\backslash w+>$

- **Layout Properties**

Finally, the layout properties can be used to alter the appearance of controls on the screen.

– **The `groupRef` Property**

This property defines to which group the control belongs. See 6.2 for more information.

– **The `position` Property**

With this property it is possible to specify the position of an element inside of the group.

– **The `appearance` Property**

This property tells the application how to render the control it is applied to. ”full“ indicates that all options should always be visible. ”compact“ means that an adequate number of options should be shown at once while ”minimal“ should result

in a presentation that always shows just one option at once and thus takes up the minimum space.

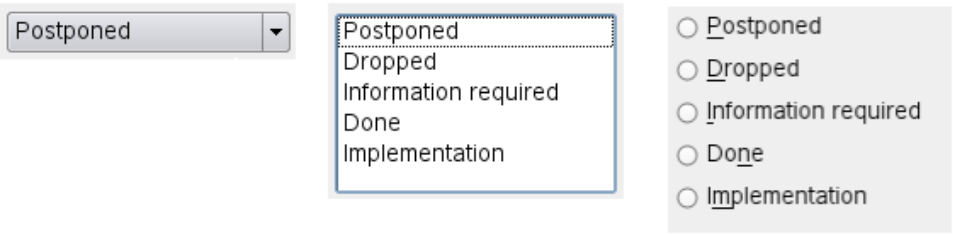

Figure 6.3: Select1 control with minimal, compact and full appearance

– **The `layoutstyle` Property**

Elements usually exist of two parts: The widget containing the actual data and a label widget, showing a descriptivetitlel for the element. This element can be used to change the way, the application should arrange these two widgets. There are two options available:

* `horizontal`
  If this option is chosen, the application should place the label widget left of the widget.
* `vertical`
  If this option is chosen, the application should place the label above the widget.

6.2 Element Positioning

The format has to provide capabilities for positioning the GUI elements on the forms. The structure in an XML-Schema file is completely arbitrary and no assumptions regarding a suitable representation in the GUI can be made out of it. It is sometimes necessary to change the order or place some items next to others.

In this thesis, two approaches were evaluated. After the first one turned out to not suite the requirements well, the second one was developed, addressing the issues. The following sections describe the two approaches in detail.

6.2.1 Static Approach

In a first approach an explicit position description for each element was used. That means, that each element had a `page`-element defining the page the element has to be placed in case of a multi-page form. Additionally the position could be specified by the `position`-element.

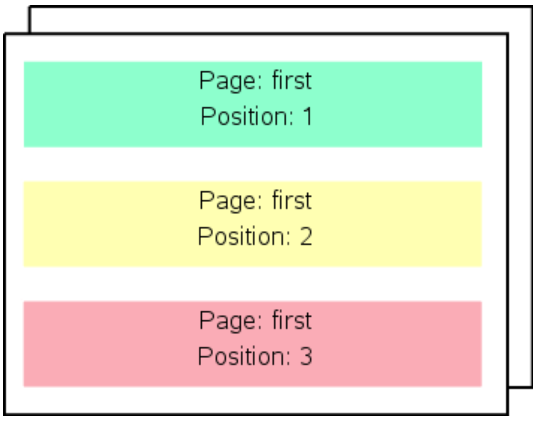

Figure 6.4: Original Form - Static Approach

This approach however turned out to be too static for some use cases. If for example the schema is extended by an element which should be placed in the middle of the form, many other `position` elements would have to be updated, too.

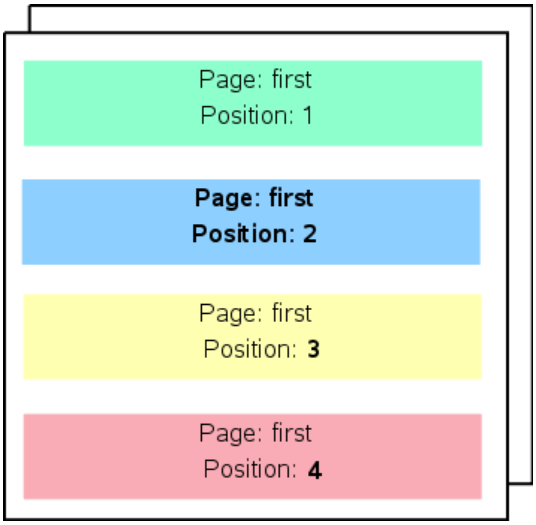

Figure 6.5: Form After Extension - Static Approach

Another disadvantage was the fact that this way, elements could only be positioned in 1 dimension, which made it hard to create appealing GUIs sometimes. Those problems led to the new, relational approach.

6.2.2 Relational Approach

In the relational approach relations are used to describe the positioning between elements.

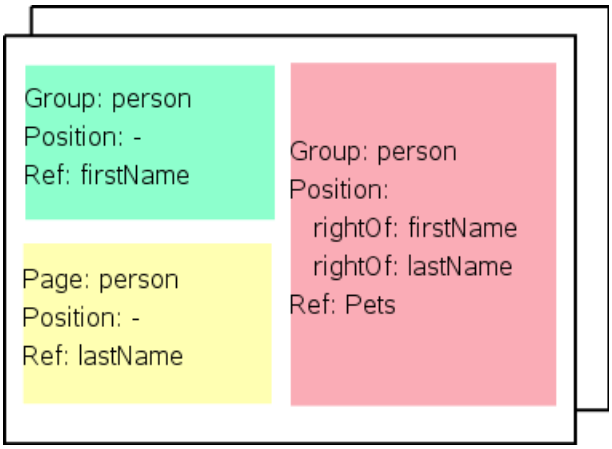

Figure 6.6: Original Form - Relational Approach

As you can see in the example, the elements are no longer positioned on pages but they are grouped together into the logic group “person”. The application can then decide if it is necessary to place a group on another page or if there can be more than one group on it. This depends on the number of elements in the group and their positions.

The same applies to the position. It is no longer defined as the absolute position of the element on the page but the relations between elements in the group are described. In the example above, the “Pets”-element is said to be placed right of the “firstName” element.

The elements have an “expanding“ property which defines whether the elements fill available space or not. This property is set to `true` by default, thus the ”Pets“-element spans 2 rows. If this behavior is not intended, the ”expanding“ property has to be set to `false`. Then the span can be defined by also specifying the lowest neighbour with a relation.

If the schema is now extended there is only one change to apply:

The following relations are supported:

- **rightOf**

  Places the element right from the other element.

- **leftOf**

  Places the element left from the other element.

- **below**

  Places the element below the other element.

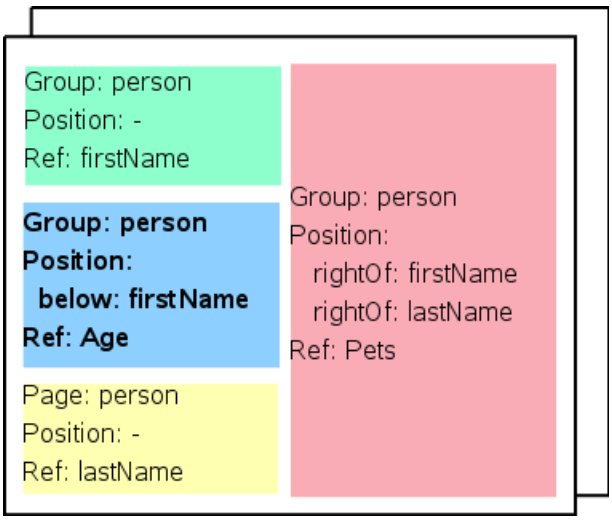

Figure 6.7: Form After Extension - Relational Approach

- **above**

  Places the element above the other element.

If an element does not have any relation it is simply placed at the end of the form in the first column. Thus, if there are no relations at all, all elements will be put on top of each other in the same order as they are defined in the XML schema. This relation approach would allow impossible descriptions, such as ”A rightOf B“ and ”B rightOf A“. The application that is used to create or modify this should not allow such a state. If however, the description was created manually for example and it contains an impossible relation, the instantiating application should discard those relations and continue with the next relation.

6.3 Group Positioning

The Relational Positioning Approach, which was chosen over the Static one groups elements in groups instead of absolute page positions. It is then left to the rendering application to place each group on a own page or to place two are more onto the same page. 9.3

This behaviour can be influenced by providing a `sizeThreshold` property in the according `form` element. The size of each group is calculated with some metric and compared to that threshold. If the threshold is exceeded, the group should be placed on the next page.

The size of a group depends on the toolkit used, the appearance styles of the controls and their positions. The size of the display also varies between different systems. So the calcula-

tion has to be done by the application and be adapted to the environment. This means, that the size value has to be normed somehow.

In kxforms, the value of the `sizeThreshold` property is 100 by default. That means that a GUI of one or more groups, that has a combined size of 100 should use the space perfectly, e.g. fit on the screen but not leave much empty space.

6.4 Hints

Usually, a GUI that is generated only from the schema describing the data will not fulfill all requirements on a modern user interface. Therefore, hints can be created by the developer or usability expert and merged into the kxforms document in order to improve the presentation of the GUI.

The hints can either be delivered as an extra file along with the schema description of the data or it can be embedded into the schema document itself. While the combined document eases the shipment of a GUI, the standalone hints document allows the dynamic modification of an already shipped GUI. Consider an application that fetches a hints file from a server atstart upp, for example.

6.4.1 Technique

A hint is an XML fragment and states a directive to the application, to change a property of an element. Thus, the hint has to describe which element of the kxforms description it addresses. This is done using a XPath expression (see 2.4).

The hint can now specify a property of the element and the value it should be overridden with. There are two different types of hints available:

- **Key-Value hints**

  This kind of hints is used to specify the value of one property. This is applicable to simple type properties such as a label.

```
1 <hint ref="/title[1]">
2  <label>Title of the Feature</label>
3 </hint>
```

Listing 6.4: Example of a Key-Value hint

- **Valuelist hints**

  There are however properties that can not be described with one value. Therefore, this type of hint can be used. The content of the hint is again an XML fragment. The parent XML element describes the property that is changed. This element can then have a list of values as child elements.

```
1 <hint ref="/title[1]">
2  <position>
3   <rightOf>/description[1]</rightOf>
4   <above>/list_actors[1]</above>
5  </position>
6 </hint>
```

Listing 6.5: Example of a valuelist hint

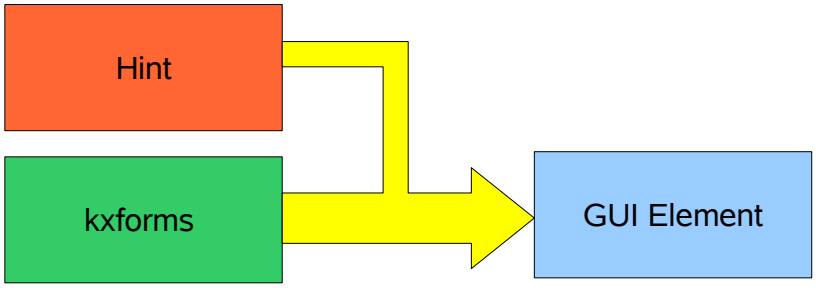

Figure 6.8: Hint architecture

6.4.2 Implemented Hints

Hints Modifying The Displayed Data

The plainest and probably most common modification, a developer or GUI-designer wants to do is changing the data that is shown. The following list will describe the hints that make such changes possible.

- **label**

  Usually an element is labeled with its “name” attribute. In some cases these attributes might be abbreviations, non-readable ids or simply not understandable enough so that

it is advisable to display a different label in the GUI. That can be achieved using the label-hint.

This hint simply overrides the automatic label generation mechanism and defines the label of the specified element.

```
1 <hint ref="needinfo">
2  <label>Information required </label>
3 </hint>
```

Listing 6.6: label hint

ref defines the element of the XML-Schema this hint should be applied to

label specifies the label that should be applied

This example would change the displayed label of the “needinfo” element from “needinfo” - or the automatically generated “Needinfo” if the rendering application supports humanization of strings - to “Information required”.

- **appearance**

  This hint modifies the “appearance” property of an element.

```
1 <hint ref="/status[1]">
2  <appearance>full</appearance>
3 </hint>
```

Listing 6.7: appearance hint

- **layoutStyle**

  This hint modifies the “layoutStyle” property of an element.

```
1 <hint ref="/status[1]">
2  <layoutStyle>vertical</layoutStyle>
3 </hint>
```

Listing 6.8: layoutStyle hint

Behavioural Hints

- **readOnly**

  This hint modifies the “readOnly” property of an element.

```
1 <hint ref="/status[1]">
2  <readOnly>false</readOnly>
3 </hint>
```

Listing 6.9: readOnly hint

- **inputType**

  With the ”inputType“ hint, it is possible to override the data type of an Input element.

```
1 <hint ref="/duration[1]">
2  <inputType>xs:integer</inputType>
3 </hint>
```

Listing 6.10: inputType hint

Positioning hints

- **groups**

  The groups hint can be used to define a set of element groups where the elements can be put into.

```
1 <hint ref="/feature">
2  <groups>
3   <group id="feature">Feature</group>
4   <group id="documentation">Documentationstatus</group>
5  </groups>
6 </hint>
```

Listing 6.11: groups hint

- **groupRef**

  This hint puts an element into a specific group

```
1 <hint ref="/duration[1]">
2  <groupRef>feature</groupRef>
3 </hint>
```

Listing 6.12: groupRef hint

- **position**

  With this hint a relational position description of an element can be created.

```
1 <hint ref="/title[1]">
2  <position>
3   <rightOf>/description[1]</rightOf>
4   <above>/list_actors[1]</above>
5  </position>
6 </hint>
```

Listing 6.13: position hint

- **formSizeThreshold**

  This hint sets the size threshold of a form, which is used to determine the placement of the groups.

```
1 <hint ref="/feature">
2  <formSizeThreshold>120</formSizeThreshold>
3 </hint>
```

Listing 6.14: formSizeThreshold hint

List hints

There are two different types of lists, lists of simple elements and lists of composited elements. Simple type lists are straightforward and show the content of the elements. In the case of complex elements however, there are many possibilities to modify the appearance of the lists. The following list describes those possibilities and the corresponding hints.

- **listItemLabel**

  KXForms is capable of presenting lists of elements. Usually, the elements are simple elements which can be shown in the list directly. If the elements are complex elements, it is important to show a subelement in the list which describes or identifies the element best.

  If no further information is given, KXForms chooses the first simple element. This might not always be the best choice, so it can be overridden using the `listItemLabel` hint.

```
<hint ref="/product[1]">
 <listItemLabel>Version <itemLabelArg ref="/@version[1]"
   truncate="40"/></listItemLabel>
</hint>
```

Listing 6.15: ListItemLabel hint

- **listShowSearch**

  This hint sets the filter bar to visible or hidden. Showing the filter bar is advisable for lists which contain many entries, so that filtering for a string might help finding a specific item.

```
<hint ref="list_productcontext">
 <listShowSearch>true</listShowSearch>
</hint>
```

Listing 6.16: listShowSearch hint

- **listShowHeader**

  By default, the headers of lists are hidden. In most cases it is not necessary to describe the data that is shown because it is either the content of the elements itself or in cases complexTypes a descriptive subelement, eventually chosen with a `listItemLabel` hint.

  Sometimes it still might be necessary to show the header, for example if the list has more than one column. That can be achieved using the `listShowHeader` hint as demonstrated below.

```
<hint ref="list_productcontext">
 <listShowHeader>true</listShowHeader>
</hint>
```

Listing 6.17: listShowHeader hint

- **listHeader**

  The `listHeader` hint lets one override the label of a column in a list, e.g. if the automatically generated one is not understandable.

```
<hint ref="/status[1]">
 <listHeader>Feature status</listHeader>
```

```
3  </hint>
```

Listing 6.18: listHeader hint

- **listItemList**

  The `listItemList` hint can be used to mark a type of list item as a list item. That means, that it contains other elements, which can be shown by expanding this item.

```
1  <hint ref="/category[1]">
2   <listItemList>true</listItemList>
3  </hint>
```

Listing 6.19: listItemList hint

Chapter 7

Generation Of The kxforms Document

Generating a kxforms document only from a schema, which describes the underlying data, is a hard task. The application has to provide a lot of intelligence, in order to create a GUI description which picks the right amount of detail of the data that will be shown and also shows it appealingly.

This chapter describes the different cases it has to consider and how they can be handled.

7.1 Forms

Before the application can translate the single elements, it first has to detect how many and which forms it has to provide. It therefor resolves the XML schema by following the references in it. Elements, which are part of another element, but can not be entirely shown on its form are appended to the root as a new element.

This results in a tree which contains the elements according to their relations. The nodes connected to the root are the elements which need an separate form.

7.2 SimpleType Elements

The easiest type of data are SimpleType elements. They can simply be translated into the corresponding kxforms elements according to the `type` attribute.

```
1 <xs:element name="email" type="xs:string">
```

for example would be translated to

```
1 <xf:input ref="/email[1]">
2  <xf:label>Email</xf:label>
3 </xf:input>
```

with the `ref` attribute pointing to the “email” element in the XML document accordingly.

The label is generated automatically by inspecting the `name` attribute of the element. Usually, the first character is simply made uppercase. In cases, where the element denotes a plural, the application can try to convert it to plural form. If the label ends on “y” for example, it is turned into “ies”.

7.3 ComplexType Elements

With ComplexType elements, the application inspects the elements that are contained. If the element contains only one child, the according element is created. If there are more than one, a `section` element is created at first, encapsulating all sub-elements. All child elements are then put into this section.

7.4 Lists

Defining a list from the XML Schema only can be a hard task. If the datatype of the list is a SimpleType, the list consists of just one column and the data that is shown in the list, is just the data of the XML elements itself.

If the data is a ComplexType however, there are several decisions that need to be made:

1. **Which and how many columns to show**

   Lists can have either one column, which makes it a plain list, or more than one, resulting in a table. On the one hand, it is desirable to present as much information in the list as required in order to identify the list items at first glance. On the other hand the GUI gets cluttered and overcrowded if there are too many columns. If there are several candidates available that could be shown as columns in the list, the application has to pick the more relevant ones. Mostly, the attributes are less interesting to show than other elements. But if the element is named ”name“ for example it for sure is important and should be shown in any case.

   Therefore, the application scans the data structure of the list item. At first, it only considers elements on the first level and does thereby not include attributes. Note that only SimpleType elements or ComplexType elements with a SimpleContent can be used.

   If this scan fails, that means no element could be found, another scan is started on the first level, but this time includes the attributes, too. If this scan also finds no elements/attributes, or there were only attributes, but none of them was a “name” attribute,

the procedure is started again but is extended to the second level. This means that if the list item contains a ComplexType, it is scanned for applicable elements as well. This time, it requires 3 elements to be found overall. If there were still not enough elements, the third level is added, too. If it still is not able to find elements, the application stops.

To recapitulate the application will show only all elements on the first level, or the "name" attribute if there were no elements. If there is also no "name" attribute, it collects 3 elements from the second level or also adds the elements of the third level if there less than 3 elements on the second level.

2. **Text of the columns**

   The text that is shown in a column, usually is just the content of the corresponding element. In case of elements which are likely to have a long content, the "truncate" attribute is set to 40 chars, e.g. with `textarea` elements.

   If the list contains more than just one type of items, the name of the according element is prepended. That makes the items more easy to recognise.

3. **Header of the columns**

   Additionally, after the columns have been chosen, the headers of the columns have to be defined. This is done the same way as with the label of an element using the `name` attribute.

Chapter 8

The KXForms Application

The KXForms application can be used to both generate a kxforms document from a schema and to create a GUI from such a description and thus covers the whole workflow from the schema to the GUI. This chapter presents the prototype application in detail.

8.1 Use Cases

The KXForms applications can be used to

- **Create a GUI for editing XML data**

  It is capable of generating a GUI for a given XML schema and load some data for further processing. It is also possible to additionally load hints, which polish the appearance of the GUI.

- **Create a kxforms document from a schema**

  It follows the directive that are described in 7. The kxforms document can be either exported as a file for later usage or can be stored only in memory and be used to generate a GUI from it directly.

- **Generate a GUI from a kxforms document**

  KXForms can use a kxforms description and render it into a usable GUI, which can then be used to load, edit an save the corresponding data.

- **Modify a GUI**

  Finally, the prototype application can be used to modify a rendered GUI and generate the corresponding hints. This mechanism is presented in more detail in 9.

It is also possible to create a library from this application, which could be embedded into other applications in order to create parts of the GUI dynamically. This was not part of this work and thus is not yet implemented.

8.2 Architecture

Figure 8.1 on page 67 shows a class diagram of the classes involved. For parsing of the XML schema, KXForms uses kxml_compiler (5.2.2). It is a wrapper around another schema parser (5.2.1) and transforms an XML schema into a data structure.

This data is further processed in a `FormCreator` instance. This class takes such a parsed XML data structure as input and translates it into a kxforms document. See 7 for more information about the challenges it faces.

Finally, this kxforms document is loaded into the `Manager`. There is one instance of this class, which functions as the central coordinator of the whole application workflow. The `Manager` then parses the document and stores the individual forms in `Form` objects (`parseForms()`). They hold the description of the form and some information about the relation to the underlying data.

The actual GUIs are instantiated on demand, that means only if they are to be shown on the screen. In order to generate the GUI of a form, the `Manager` asks a `GuiHandler` to create a `FormGui` object (`createGui()`). It is possible to create different `GuiHandlers`, which will generate different GUIs. That is possible, because the `FormGUI` uses the `GuiHandler` to create the layout of the GUI. It for example calls a method of the `GuiHandler`, that places a widget onto the current layout (`addWidget()`). Thus, it is possible to modify the general style of an application by using the desired `GuiHandler`.

The `FormGUI` retrieves the kxforms snippet from the corresponding `Form` object and creates and places the elements. The abstract base class of these elements is the `GuiElement` class. This class is derived from the widget base class of the Qt framework, `QWidget`, and thus is the actual representation of that element in the GUI. These objects additionally have information about the element, such as its properties, the applicable editor actions (see 9) or the tooltip and so on. There is one derived class of `GuiElement` for each type of control, that kxforms offers.

Finally, the `GuiHandler` registers the created `FormGui` with the `Manager` (`registerGui()`), so that it is aware of all forms. That is needed for managing the data flow inside the application, showing the currently needed form and for the edit mode, which is described in 9.

8.3 GUI Generation Details

When the generation of the kxforms document succeeded or the KXForms application is given an already existing document, it has to create a GUI for this description. It traverses the XML tree and creates a GUI element for each kxforms element, taking care of the positioning of the elements.

That means, that it can not put the elements into a layout right after they were created. Instead, a list of the elements is created. When an element is to be created, which has a relative position to an already existing element, it is not put at the end of the list but at the desired position according to the provided positioning information. If all elements are handled and correctly sorted, they can be finally put into a GUI layout.

The application also has to respect the sizeThreshold which is set for the current form. That means it has to calculate the space of each group and if the threshold is exceeded, it has to create another tab and put the group there instead of on the current tab.

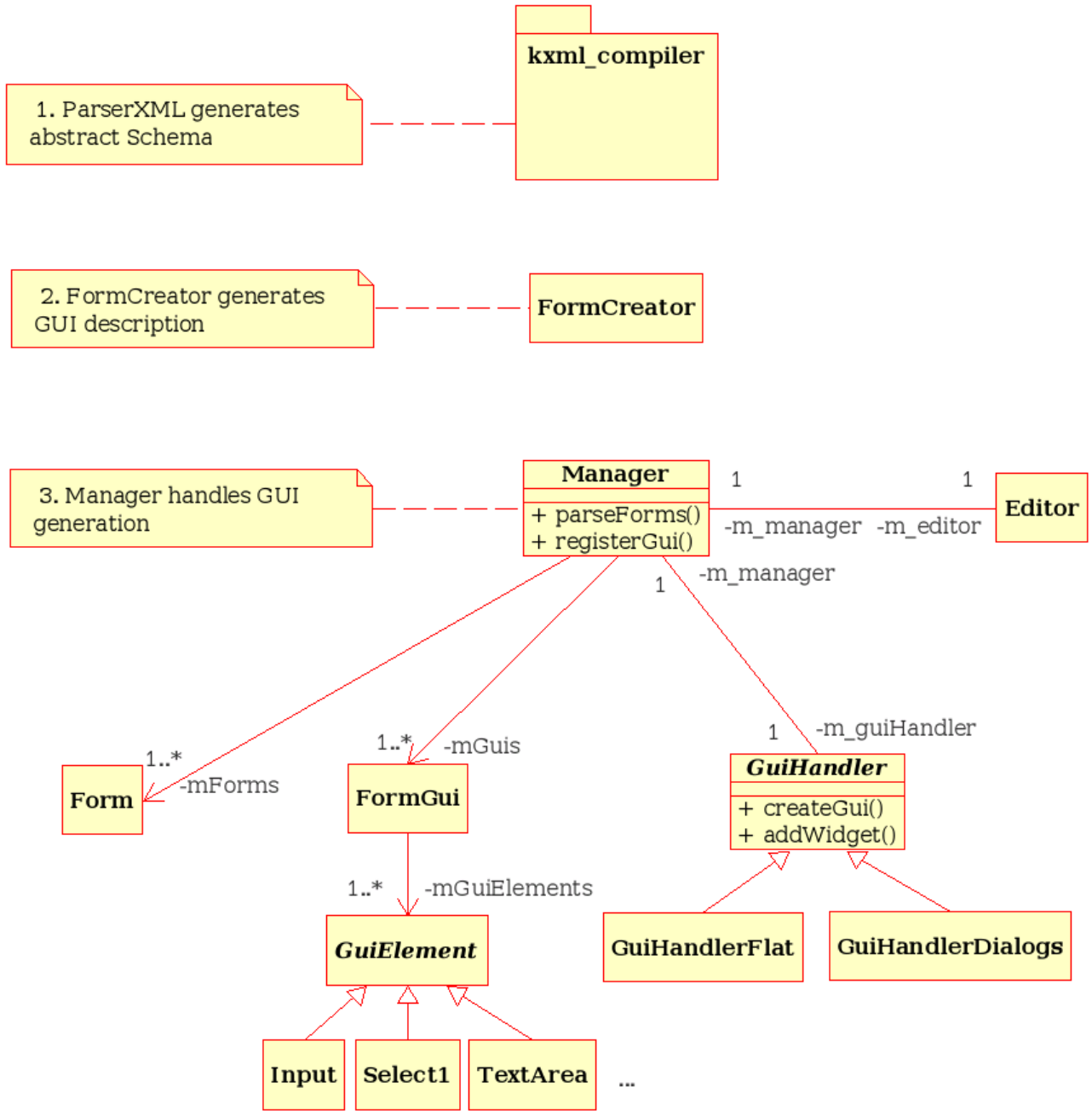

Figure 8.1: Class diagram of KXForms

Chapter 9

Live Editing Of GUIs

As already stated in previous chapters, the generated GUIs do not always match all requirements in terms of usability and layout. When the kxforms description was generated and a GUI instantiated from it, the developer or usability expert might notice things that should better be changed. This can be done with hints (see 6.4 for more information).

This chapter presents an implementation that allows the convenient generation of those hints.

9.1 Required Functionality

KXforms should provide an integrated editor mode that can be enabled at runtime and which should be able to edit the GUI in its current state. This editor mode should provide facilities to edit the interface in WYSIWYG style, in order to enable non-technical usability experts or first-time users to achieve appealing results. The result of such an editing process should be a file containing the hints that are sufficient to transform the automatically generated GUI into the modified state by the editor.

The editing mode should support the generation and modification of all hints that are described in the KXForms specification [5].

9.2 Architecture

The editor of KXForms lives in its own subdirectory `editor/`. It can be divided into three major parts:

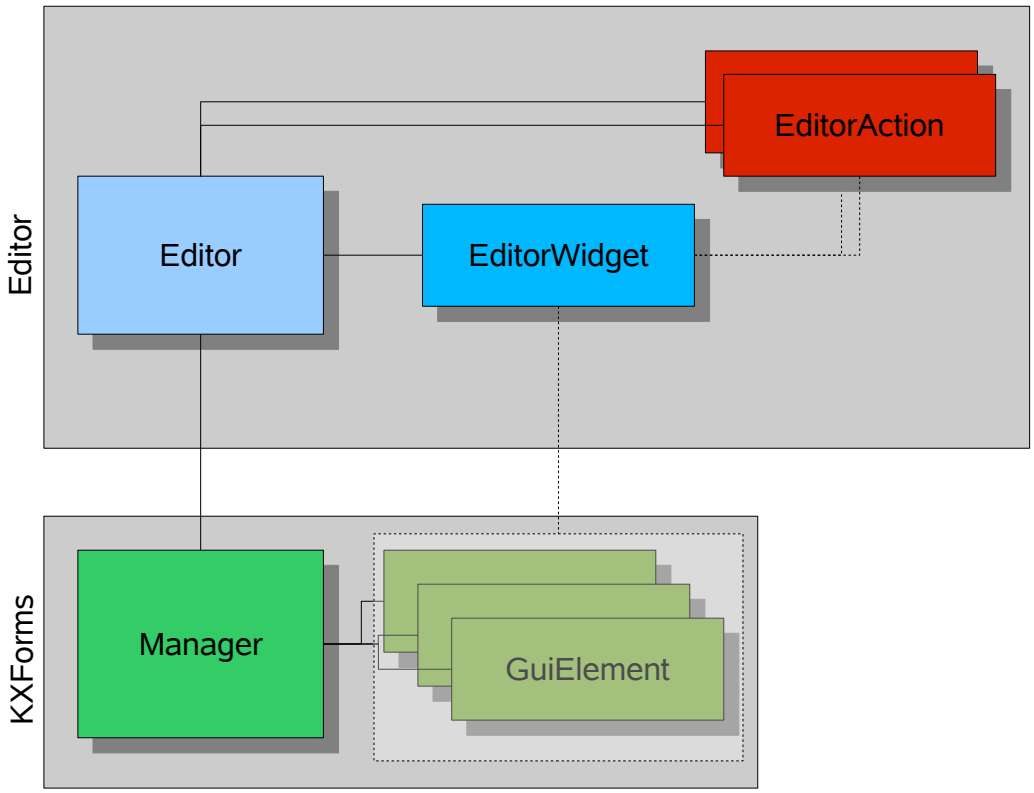

Figure 9.1: Editor Architecture

- **The Editor**

  The Editor itself is the central instance of the editing mechanism. It coordinates the editing process and forwards the generated hints to the application, which will use them to rebuild the GUI. It also knows all GUI elements so that it can switch each of them into editing mode when activated.

  The Editor resides between the main application interface - the Manager - and the editor's internals, specifically the Editor Widget which is described below (see Figure 9.1). The Editor object is created when the editor mode is activated. It then retrieves the currently shown GUI part from the manager and propagates the elements of this GUI to the editor widget, which uses this information to draw an interface for the user.

  The Editor has a function `actionMenu( GuiElement *e )` which can be used to generate a menu for a specific element of a GUI. The Editor examines the capabilities of the GUI element and adds the according actions to the menu. This allows to have different actions for different types of elements. A list for example has an action that allows to

hide the headers while an element that allows the user to choose one option out of several possibilities has an action that can be used to set the appearance to use a checkbox, a list or radio buttons. A list of implemented actions is given in 9.4.

Figure 9.2 shows two different generated action menus. At the top of the menu there are the generic actions that are applicable to all GUI elements. If appropriate, these actions are followed by a separator and the specific actions. In the left example there is an action that lets the user modify the properties of a list element while on the left the appearance of a select element can be modified.

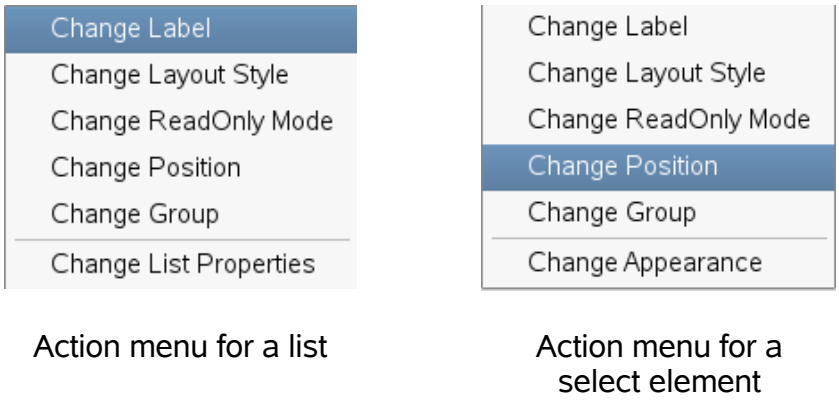

Figure 9.2: Generated Action Menus

The Editor has a slot `applyHint( const Hint &h )` to which the Editor Actions are connected. Whenever a hint is propagated through this slot, the Editor merges the hints with the already existing list of hints and triggers a rebuild of the GUI with the latest changes.

- **The Editor Widget**

  The Editor Widget is the visible part to the user of the editor mode. It is a widget which provides access to the features that the Editor provides. The widget is placed on top of the actual GUI that it represents when the edit mode is active. It only exists as long as the editor mode is active. In order to increase usability, the Editor Widget is semi-transparent, thus both exposing the underlying original GUI and indicating that the the edit mode is active. It then offers the user functionality by drawing additional information on top of it (see Figure 9.3).

  There are two different types of interfaces:

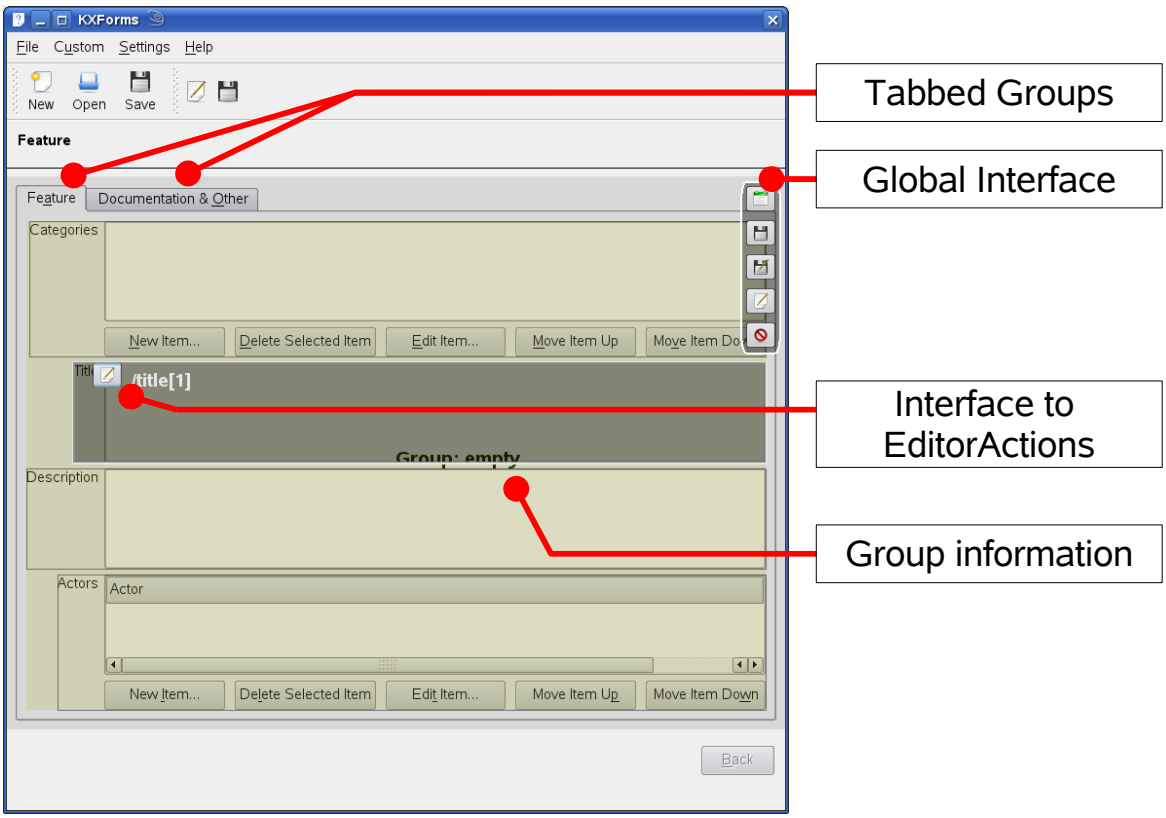

Figure 9.3: Screenshot of Editor

– **Global Interface**

  The kxform specification defines some global settings, which can be changed with this interface. This includes the default values for the appearance, layout style and read-only property for example. The groups that are available on a form can also be specified here.

  Another value that can be set is the size threshold for the form. This threshold is used to determine how many groups can be placed on one form.

  Other options are available for saving and exporting the generated hints and for displaying the current list of hints.

– **Interface for a specific element**

  There is also one interface per element. It currently only shows the available actions for the according element, but one might also think of a list, showing the hints for the element etc.

The Editor Widget also draws semi-transparent overlays enclosing the elements which belong to the same groups. This makes it easier to understand the relations between the elements and eases the logical separation of elements into groups.

- **The Editor Actions**

  An Editor Action applies a change to a specific GUI element and generates the corresponding hint for that change. It might ask for additional information from the user. The Editor Actions are offered to the user by the Editor Widget and when all required information is gathered, it propagates the generated hint to the Editor which will trigger a rebuild of the GUI.

9.3 The Editing Process

When the edit mode is activated, the Editor communicates with the Forms Manager in order to retrieve the current FormGui. A FormGui is the object in the system's memory, representing the GUI that is currently shown in the application. This object is aware of all information about the GUI and the underlying XML document, e.g. the XPath expression leading to the relevant XML fragment and the GUI elements that exist in this GUI.

The Editor now creates an EditorWidget, with a pointer to the FormGui. Thus, the EditorWidget can retrieve the list of GUI elements that are relevant for the editing process.

Each of these GUI elements is derived from QWidget, the base widget class of the Qt Toolkit. Thus, the Editorwidget is able to calculate the exact position of the elements on the screen.

As stated above, the EditorWidget forms a transparent layer above the actual GUI. With the geometry of the elements, it can simply draw the editing interface on the widgets. The interface is not static, however.

In the idle state, the application waits for the user to choose the element which should be edited. Thus, it draws the references of the elements, so that they can be easily identified. Additionally, the existing groups are visually highlighted. This is important at a later point in the process, which will be explained later.

If an element is hovered, a button with an edit symbol appears. If the button is pressed, the EditorWidget asks the Editor for an action menu which offers all available EditorActions for the corresponding element. Therefor, the GUI element has flags for the relevant action groups. When the user made the choice and selected one of the actions, the EditorWidget propagates the choice to the Editor.

The Editor in return invokes the desired EditorAction with the reference of the chosen element. The EditorAction might perform additional tasks in order to generate the hint for the desired change. When the hint is created, it is propagated to the Editor again.

The new hint is then merged with the already existing hints and the Manager is triggered to rebuild the GUI with the new set of hints, resulting in the GUI with the new change applied.

The user can export the currently underlying kxforms document or the set of hints from within the edit mode, too. These files can then be used later to recreate the GUI with the hints or to simply use the kxforms document directly instead of creating it from the schema and the hints again.

Both types of exports have advantages and disadvantages which are shown in the table below:

kxforms Document	**Set of Hints**
+ standalone usage	+ can adopt changes of the schema
+ fast application start up	- longer start up time
- inflexible with regard to schema changes	- schema required at runtime

9.4 List Of Implemented Editor Actions

- **ChangeLabelAction**

  This action is used to change the label of an element. The label is the text that is shown next to the element and which should describe its content. This is often required since the quality of the automatically generated label from the schema is often not good enough.

- **PositionAction**

  The PositionAction can be used to position an element on the GUI. The layout is described with relations between the elements, so this action asks for the neighbour and the type of relation, being one of ”Above”, ”RightOf”, ”Below” or ”LeftOf”.

- **LayoutStyleAction**

  With the LayoutStyleAction the editor can choose the layout style between ”vertical” and ”horizontal”, placing the label and the widget of a GUI element either above or next to each other.

- **GroupAction**

  This action allows to define the group to which the element should belong. The groups have to be defined in the global interface first, before they can be applied to an element.

- **ReadOnlyAction**

  The ReadOnlyAction lets one set the read-only property of an element. The content of an element can only be changed by the user if the read-only property is set to false.

- **InputTypeAction**

  The InputTypeAction is specific to input elements. With this action, the type of an input element can be defined. The type describes which kind of data the user can provide using this element. If the type is ”xs:string” for example, the application would show a lineedit, while if it is ”xs:integer” a spinbox would be appropriate.

- **AppearanceAction**

  The AppearanceAction is specific to select and select-one elements. It defines how much space such an element should take up, e.g. if all options are shown at once or only a subset or even only one at a time.

- **ListAction**

  This action is specific to lists and lets the editor set options such as if the headers or a filter bar should be shown.

9.5 Working With The Editor

This section describes the typical steps that need to be taken in order to apply a change to a GUI:

1. **Navigate to the piece of GUI you want to edit**

   After the application has been started, the editor mode can be switched on/off via the settings menu or a button in the editor toolbar. Whenever the edit mode is active however, it can no longer be navigated through the GUI, so the edit mode has to be activated at the place where the changes are to be applied.

2. **Select an action for the desired element**

In the next step, the according element is chosen and the menu requested by clicking on the edit button that is shown on top of the element. This menu will contain all applicable actions for this element.

3. **Provide required information**

   In the final step, the user has to provide the required information for the action. That information differs between the different actions and ranges from providing a simple label for an element to filling a complete dialog of information.

4. **Repeat or export hints or kxforms document**

   The user can then repeat the process from step 1 or export the generated GUI. The export can happen either in form of a kxforms document or as a set of hints.

Chapter 10

Case Study: The KDE Feature Plan

The KDE Feature Plan [2] holds all features that are planned to be implemented in future versions of KDE. Each feature has a summary, describing the feature, a target KDE version and a status, telling the process of this feature. Additionally, a list of responsible persons can be added in order to tell who is responsible for the implementation.

This data is represented as XML, so that it can be validated against an XML-Schema and be transformed into a HTML page using XSLT for example. This also qualifies it to be used with Kode. This chapter describes the steps taken to create an appealing GUI for the Feature Plan.

10.1 Generating A kxforms document

The XML Schema for the Feature Plan looks as follows:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3   elementFormDefault="qualified"
4   xmlns:ugh="http://www.kde.org/standards/ugh/0.1" >
5
6   <xs:element name="features">
7    <xs:complexType>
8     <xs:sequence>
9      <xs:element maxOccurs="unbounded" ref="category"/>
10    </xs:sequence>
11   </xs:complexType>
12  </xs:element>
13  <xs:element name="category">
14   <xs:complexType>
15    <xs:choice minOccurs="0" maxOccurs="unbounded">
```

```
16      <xs:element ref="feature"/>
17      <xs:element ref="category"/>
18     </xs:choice>
19     <xs:attributeGroup ref="attlist.category"/>
20    </xs:complexType>
21   </xs:element>
22   <xs:attributeGroup name="attlist.category">
23    <xs:attribute name="name" use="required"/>
24   </xs:attributeGroup>
25   <xs:element name="feature">
26    <xs:complexType>
27     <xs:sequence>
28      <xs:element minOccurs="0" ref="summary"/>
29      <xs:element minOccurs="0" maxOccurs="unbounded" ref="responsible"/>
30     </xs:sequence>
31     <xs:attributeGroup ref="attlist.feature"/>
32    </xs:complexType>
33   </xs:element>
34   <xs:attributeGroup name="attlist.feature">
35    <xs:attribute name="status" default="todo">
36     <xs:simpleType>
37      <xs:restriction base="xs:token">
38       <xs:enumeration value="todo"/>
39       <xs:enumeration value="inprogress"/>
40       <xs:enumeration value="done"/>
41      </xs:restriction>
42     </xs:simpleType>
43    </xs:attribute>
44    <xs:attribute name="target" use="required">
45     <xs:simpleType>
46      <xs:restriction base="xs:token">
47       <xs:enumeration value="3.4"/>
48       <xs:enumeration value="3.5"/>
49       <xs:enumeration value="4.0"/>
50      </xs:restriction>
51     </xs:simpleType>
52    </xs:attribute>
53   </xs:attributeGroup>
54   <xs:element name="responsible">
55    <xs:complexType>
56     <xs:attributeGroup ref="attlist.responsible"/>
57    </xs:complexType>
```

```
58  </xs:element>
59  <xs:attributeGroup name="attlist.responsible">
60   <xs:attribute name="name"/>
61   <xs:attribute name="email"/>
62  </xs:attributeGroup>
63  <xs:element name="summary">
64   <xs:complexType mixed="true">
65    <xs:choice minOccurs="0" maxOccurs="unbounded">
66     <xs:element ref="i"/>
67     <xs:element ref="a"/>
68     <xs:element ref="b"/>
69     <xs:element ref="em"/>
70     <xs:element ref="strong"/>
71     <xs:element ref="br"/>
72    </xs:choice>
73   </xs:complexType>
74  </xs:element>
75  <xs:element name="i" type="xs:string"/>
76  <xs:element name="b" type="xs:string"/>
77  <xs:element name="em" type="xs:string"/>
78  <xs:element name="strong" type="xs:string"/>
79  <xs:element name="br">
80   <xs:complexType/>
81  </xs:element>
82  <xs:element name="a">
83   <xs:complexType mixed="true">
84    <xs:attributeGroup ref="attlist.a"/>
85   </xs:complexType>
86  </xs:element>
87  <xs:attributeGroup name="attlist.a">
88   <xs:attribute name="href"/>
89   <xs:attribute name="title"/>
90  </xs:attributeGroup>
91 </xs:schema>
```

Listing 10.1: XML Schema of the Feature Plan

The root element of the data structure thus is the `features` element. It consists of a list of categories. Each category can in return contain features and/or categories. A feature consists of a summary and a list of responsibles. Both categories, features and responsibles also have some attributes.

Besides the basic XML Schema features, this example schema also contains restrictions, for example in the `status` attribute of the `feature` element.

Some example data is shown below:

```
<features>
 <category name="KDE PIM (Personal Information Management)" >
  <category name="Kontact" >
   <feature status="done" target="4.0" >
    <summary>Merged configuration view.</summary>
    <responsible email="Kretz@kde.org" name="Matthias Kretz" />
   </feature>
   <feature status="todo" target="3.5" >
    <summary>Add alternative tab-based viewmode.</summary>
    <responsible email="molkentin@kde.org" name="Daniel Molkentin" />
   </feature>
   <feature status="inprogress" target="3.5" >
    <summary>Fix sidebar to use other icon sizes</summary>
    <responsible email="molkentin@kde.org" name="Daniel Molkentin" />
   </feature>
[...]
```

Listing 10.2: Excerpt from a Feature Plan

The automatically generated kxforms document from the KXForms application is given below:

```
<kxforms>
 <form ref="features">
  <xf:label>Features</xf:label>
  <list id="list_category">
   <xf:label>Categories</xf:label>
   <itemclass ref="category[1]">
    <itemlabel><itemLabelArg ref="@name[1]" truncate="40"/></itemlabel>
   </itemclass>
   <headers>
    <header>Category</header>
   </headers>
  </list>
 </form>
 <form ref="responsible">
  <xf:label>Responsible</xf:label>
  <attributes>
   <xf:input ref="@name[1]">
    <xf:label>Name</xf:label>
```

```
19      </xf:input>
20      <xf:input ref="@email[1]">
21       <xf:label>Email</xf:label>
22      </xf:input>
23     </attributes>
24    </form>
25    <form ref="feature">
26     <xf:label>Feature</xf:label>
27     <attributes>
28      <xf:select1 ref="@status[1]">
29       <xf:label>Status</xf:label>
30       <xf:item>
31        <xf:label>Todo</xf:label>
32        <xf:value>todo</xf:value>
33       </xf:item>
34       <xf:item>
35        <xf:label>Inprogress</xf:label>
36        <xf:value>inprogress</xf:value>
37       </xf:item>
38       <xf:item>
39        <xf:label>Done</xf:label>
40        <xf:value>done</xf:value>
41       </xf:item>
42      </xf:select1>
43      <xf:select1 ref="@target[1]">
44       <xf:label>Target</xf:label>
45       <xf:item>
46        <xf:label>3.4</xf:label>
47        <xf:value>3.4</xf:value>
48       </xf:item>
49       <xf:item>
50        <xf:label>3.5</xf:label>
51        <xf:value>3.5</xf:value>
52       </xf:item>
53       <xf:item>
54        <xf:label>4.0</xf:label>
55        <xf:value>4.0</xf:value>
56       </xf:item>
57      </xf:select1>
58     </attributes>
59     <xf:textarea ref="/summary[1]">
60      <xf:label>Summary</xf:label>
```

```
61    </xf:textarea>
62    <list id="list_responsible" showHeader="true">
63     <xf:label>Responsibles</xf:label>
64     <itemclass ref="/responsible[1]">
65      <itemlabel><itemLabelArg ref="/@name[1]" truncate="40"/></itemlabel>
66      <itemlabel><itemLabelArg ref="/@email[1]" truncate="40"/></itemlabel>
67     </itemclass>
68     <headers>
69      <header>Name</header>
70      <header>Email</header>
71     </headers>
72    </list>
73   </form>
74   <form ref="category">
75    <xf:label>Category</xf:label>
76    <attributes>
77     <xf:input ref="@name[1]">
78      <xf:label>Name</xf:label>
79     </xf:input>
80    </attributes>
81    <list id="list_feature+category">
82     <xf:label>Item</xf:label>
83     <itemclass ref="/feature[1]">
84      <itemlabel>Feature: <itemLabelArg ref="/summary[1]" truncate="40"/>
85      </itemlabel>
86     </itemclass>
87     <itemclass ref="/category[1]">
88      <itemlabel>Category: <itemLabelArg ref="/@name[1]" truncate="40"/>
89      </itemlabel>
90     </itemclass>
91     <headers>
92      <header>Feature</header>
93     </headers>
94    </list>
95   </form>
96  </kxforms>
```

Listing 10.3: Generated kxforms Document

KXForms generated a kxforms document which contains 4 forms: `features`, `category`, `feature` and `responsible`. This is sufficient to display all information described by the schema.

The relations between the kxforms description and the XML Schema are clearly visible. The list of categories in the `features` element in the schema was mapped to a `list` element in the kxforms document. The list control contains one `itemclass` element which references the category. With the header element, the mapping of this element is complete.

The same correlation can be found at the other elements, too. Thus, the generated document is just another description of the given XML Schema.

10.2 Generating A GUI

The GUI that is shown after running KXForms with the generated document can be seen in Figure 10.1.

There is one piece of GUI created for each of the forms. There are no tabs, as the different elements are all nested and thus can not be displayed at the same time.

While this GUI is quite good and appealing, there are still some improvements that could make the GUI better. These are described in the next section.

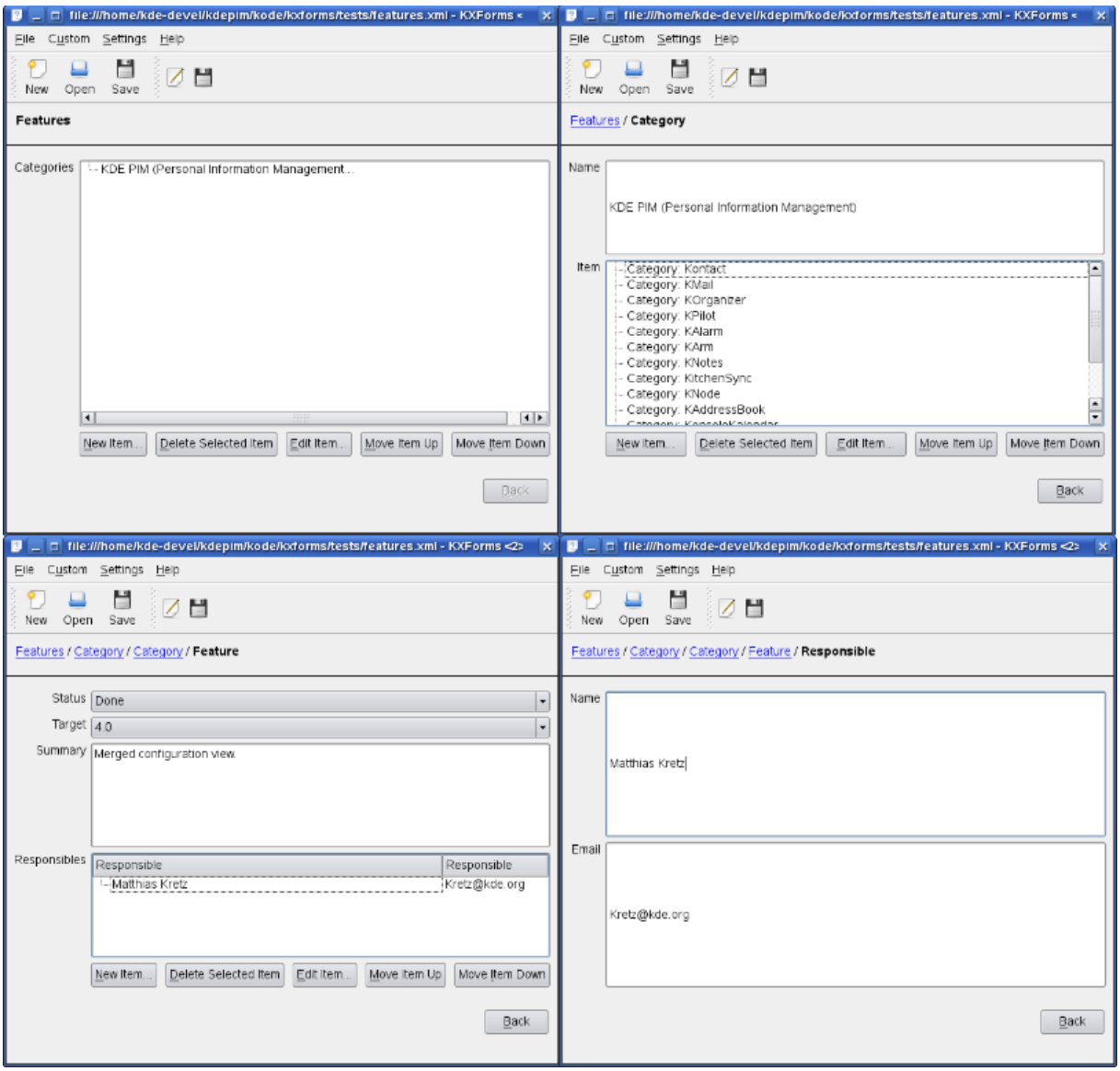

Figure 10.1: KDE Feature Plan GUI

10.3 Tweaking The GUI

In the “Feature” GUI, there are two ComboBoxes, one for the status and one for the target of the feature. KXForms has put these controls on top of each other. The ComboBoxes do not

look good, however, if they span the whole width of the form. Thus, it would be better to place they next to each other.

Therefore, the GUI is navigated to the “Feature” form and the edit mode is activated (see Figure 10.2).

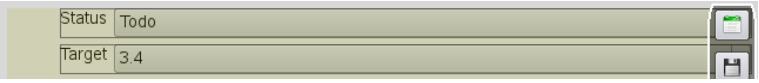

Figure 10.2: Editing the Feature Plan 1

Then, the “target” ComboBox is dragged and dropped onto the “status” ComboBox, initiating a PositionAction editor action. The editor then asks for the new relative position of the control, which is answered with “rightOf”. The result can be seen in Figure 10.3.

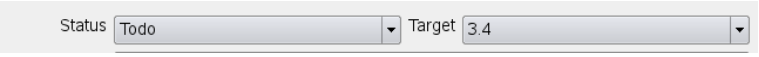

Figure 10.3: Editing the Feature Plan 2

Another change that was made to the GUI is the change of the label of the name attribute of the category element. The generated one was “Name”, the new one “Title” (Figure 10.4).

Figure 10.4: Editing the Feature Plan 3

These changes were then saved as a hints document for later usage, which is shown below:

```
1 <ugh>
2   <hint ref="@name[1]">
3     <label>Title</label>
4   </hint>
5   <hint ref="@status[1]">
```

```
6       <position>
7         <leftOf>@target[1]</leftOf>
8       </position>
9     </hint>
10  <ugh>
```

Listing 10.4: Hints for the Feature Plan

Chapter 11

Conclusion And Outlook

This thesis has presented the development of a framework for automatic GUI generation with the kxforms GUI description language and the application KXForms. It can handle those cases, where a description of the underlying data is available. Much information can be extracted from the description, allowing for a basic but working GUI. Furthermore, a flexible approach for modifying those GUI was developed, called hints. Therewith, a generated GUI can be modified and adapted, e.g. to match some guidelines.

The prototype application handles all tasks of the workflow, namely parsing the data description, generating a GUI description and finally instantiating a GUI from it. It also has an edit mode built in, which allows to edit a GUI on the fly in WYSIWYG style and export the required hints for the changes that were made.

For the application to be a generic solution for automatic GUI generation, some work has yet to be done though. The XML Schema parser does not support all features of XML Schema (derivation for example). Additionally, the edit mode does not support all hints of the kxforms specification and some other functions are not working properly yet.

But these issues can be sorted out. If more parsers for other data formats are created and the functionality is offered in form of a library, many applications could benefit from this solution.

As a next intermediate step, another format which is not as complex as XML Schema could be supported, for example. One option is KConfig XT, which is a configuration framework for KDE. It also has a description of the available options which could be automatically turned into a GUI. Supporting KConfig XT, KXForms could be used in every KDE application to automatically generate the configure dialogs.

Appendix A: Bibliography

[1] Christophe Coenraets. *An overview of MXML: The Flex markup language*. 2003. `http://www.adobe.com/devnet/flex/articles/paradigm.html`.

[2] KDE developers. *KDE 4.0 Feature Plan*. 2007. `http://developer.kde.org/development-versions/kde-4.0-features.html`.

[3] KDE developers. *KDE Human Interface Guidelines*. 2007. `http://usability.kde.org/hig/`.

[4] KDE developers. *Kode*. 2007. `http://rechner.lst.de/~cs/kode/`.

[5] Andre Duffeck. *KXForms Specification*. Suse Linux Products GmbH, 2007. `http://websvn.kde.org/trunk/KDE/kdepim/kode/kxforms/doc/`.

[6] JSusan L. Fowler. *GUI Design Handbook*. 1997.

[7] Ian Griffiths. *Inside XAML*. 2004. `http://www.ondotnet.com/pub/a/dotnet/2004/01/19/longhorn.html`.

[8] IBM. *Design basics*. 2007. `http://www-03.ibm.com/easy/page/6`.

[9] Jeff Johnson. *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. 2000.

[10] Soffa M.L. Memon, A.M. and M.E. Pollack. *Coverage Criteria for GUI Testing*. Vienna University of Technology, Austria, 2001.

[11] Inc. OASIS Open. *User Interface Markup Language (UIML) Specification*. 2004. `http://www.uiml.org/specs/uiml3/DraftSpec.htm`.

[12] The Mozilla Project. *Firefox Add-ons*. 2007. `https://addons.mozilla.org/en-US/firefox/`.

[13] The Mozilla Project. *XUL*. 2007.
http://developer.mozilla.org/en/docs/XUL.

[14] Brent Rector. *Introducing Longhorn for Developers, Chapter 3 Inside XAML*. 2003.
http://msdn2.microsoft.com/en-us/library/aa186116.aspx.

[15] W3C. *XForms - The Next Generation of Web Forms*. 2002.
http://www.w3.org/MarkUp/Forms/.

[16] W3C. *XML Schema Part 0: Primer Second Edition*. 2004.
http://www.w3.org/TR/xmlschema-0/.

[17] W3C. *XML Schema Part 1: Structures Second Edition*. 2004.
http://www.w3.org/TR/xmlschema-1/.

[18] W3C. *XML Schema Part 2: Datatypes Second Edition*. 2004.
http://www.w3.org/TR/xmlschema-2/.

[19] W3C. *XForms 1.0 (Second Edition)*. 2006.
http://www.w3.org/TR/xforms/.

[20] Ltd. Wireless Application Protocol Forum. *Wireless Markup Language*. 2001.
http://www.openmobilealliance.org/tech/DTD/index.htm.

[21] Larry E. Wood. *User Interface Design: Bridging the Gap from User Requirements to Design*. 1998.

[22] XULPlanet. *XULPlanet.com*. 2006.
http://www.xulplanet.com.

Appendix B: Glossary

API	Application Programming Interface. A source code interface that allows to use services of libraries or applications.
CSS	Cascading Style Sheets. A stylesheet language, that can be used to define the presentation of a underlying document.
Form	A form is a part of a graphical user interface. It encapsulates several widgets and only one form can be shown at once
GUI	Graphical User Interface
HIG	Human Interface Guideline
HTML	Hypertext Markup Language. A markup language for the creation of web pages.
OSS	Open Source Software
RelaxNG	RelaxNG is an XML schema language such as XML Schema. In contrast to other schema languages, it is known to be very simply and elegant.
UI	User Interface
UIML	User Interface Markup Language
widget	Single graphical element of a GUI, such as a button, an icon or a progressbar
WYSIWYG	What-You-See-Is-What-You-Get describes a mode of editing, where the final result is immediately visible while editing
XML	Extensible Markup Language
XUL	XML User Interface Language. A user interface markup language developed by the Mozilla project.

Appendix C: The kxforms Specification

Chapter 1

Introduction

KXForms is a description language which provides facilities for describing user interfaces. The language, which itself is presented in XML 1.0, is based on parts of the W3C XForms.

Chapter 2

KXForms

2.1 KXForms Common

2.1.1 Common Attributes

ref

The ref attribute describes a reference to the XML element which is represented through the element. This attribute is mandatory for all GUI elements.

Consider the following xml document:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3      xmlns:k="http://inttools.suse.de/sxkeeper/schema/keeper"
4      elementFormDefault="qualified">
5  <xs:element name="feature">
6   <xs:complexType>
7    <xs:element ref="title"/>
8    <xs:element ref="partnercontext"/>
9   </xs:complexType>
10 </xs:element>
11
12 <xs:element name="title" type="xs:string"/>
13
14 <xs:element name="partnercontext">
15  <xs:complexType>
16   <xs:sequence>
17    <xs:element ref="organization"/>
18    <xs:element ref="externalid"/>
```

```
19   </xs:sequence>
20  </xs:complexType>
21 </xs:element>
22
23 <xs:element name="organization" type="xs:string"/>
24
25 <xs:element name="externalid" type="xs:string"/>
26 </xs:schema>
```

Listing 2.1: Example Schema

The main form would in this case relate to the “feature” element, thus the ref argument would be set to “/feature”. The references are heritable, which means that the child elements of this form automatically refer to the feature element. Thus, the reference to the “title” element would be “/title[1]” instead of “/feature/title[1]”.

The number at the end of a reference refers to the occurrence of this element, so [1] addresses the first “title” element, [2] the second one and so on.

A possible kxform document for the example schema could look like this:

```
1 <form ref="/feature">
2  <xf:label>Feature</xf:label>
3  <xf:textarea ref="/title[1]">
4   <xf:label>Title</xf:label>
5  </xf:textarea>
6  <section ref="/">
7   <xf:label>Partnercontext</xf:label>
8   <xf:textarea ref="/partnercontext[1]/organization[1]">
9    <xf:label>Organization</xf:label>
10  </xf:textarea>
11  <xf:textarea ref="/partnercontext[1]/externalid[1]">
12   <xf:label>Externalid</xf:label>
13  </xf:textarea>
14 </section>
15 </form>
```

Listing 2.2: Example KXForms Document

2.1.2 Common Child Elements

`xf:label`

The `xf:label` child element is derived from XForms. It is used to define the label that is shown for the corresponding GUI element. If this element is omitted, the application should try to generate an appealing label from the available information (the element name in most cases).

`tip`

The `tip` child element allows to provide a tooltip for controls. The presentation of the tip is left to the application but usually it will just be displayed after the mouse rested a bit over the control.

```
1 <xf:input ref="/password">
2  <xf:label>Password:</xf:label>
3  <tip>Please enter your private password.</tip>
4 </xf:input>
```

Listing 2.3: `xf:label` and `tip` element

`attributes`

The `attributes` child element encapsulates GUI Elements, that map to attributes of the parent element in the XML Schema. These elements might be visually separated from the other normal elements in order to make clear they are attributes.

- Valid Child Elements: `xf:input`, `xf:textarea` (2.3)

```
1  <xf:textarea ref="/productid[1]">
2   <xf:label>Productid</xf:label>
3   <attributes>
4    <xf:input ref="@legacypriotype">
5     <xf:label>Legacypriotype</xf:label>
6    </xf:input>
7    <xf:input ref="@legacytype">
8     <xf:label>Legacytype</xf:label>
9    </xf:input>
10   <xf:input ref="@legacyinfo">
11    <xf:label>Legacyinfo</xf:label>
12   </xf:input>
```

```
13   </attributes>
14 </xf:textarea>
```

Listing 2.4: `attributes` element

2.2 KXForms Core

2.2.1 The kxforms Element

Description: This element is the root element of a kxforms document. It contains all forms that are available in the kxforms document as its child elements. In a valid kxforms document the forms have to be sufficient to display and edit all elements of a XML instance of the corresponding XML Schema.

Common Attributes: none

Special Attributes: none

Common Child Elements: none

Special Child Elements:

- `form` (2.2.2)
- `defaults` (2.4.1)

Example:

```
1  <kxforms>
2   <defaults>
3    <properties>
4     <appearance>Full</appearance>
5     <layoutstyle>vertical</layoutstyle>
6    </properties>
7   <defaults>
8   <form ref="category">...</form>
9   <form ref="productcontext">...</form>
10 </kxforms>
```

Listing 2.5: kxforms Element

2.2.2 The form Element

Description: This element encapsulates one form. A form describes a GUI for one part of the XML Schema. One form has to be available for the toplevel element and one for the elements of each list.

Common Attributes: Common (2.1.1)

Special Attributes: none

Common Child Elements: Common (2.1.2)

Special Child Elements: `groups` (2.4.2), GUI Elements (2.3)

Example:

```
1 <form ref="category">
2  <xf:label>Category:</xf:label>
3  ...
4 </form>
```

Listing 2.6: form Element

2.3 KXForms GUI Elements Descriptions

2.3.1 The `xf:input` Element

Description: This element is derived from XForms. An input field enables free-form data entry. In contrast to the `xf:textarea` Element that is described below, it only allows single-line input.

Common Attributes: Common (2.1.1)

Special Attributes: none

Common Child Elements: Common (2.1.2)

Special Child Elements: `inputproperties` (2.4.11), `properties` (2.4.12)

Example:

```
1 <xf:input ref="/externalid">
2  <xf:label>Name:</xf:label>
3 </xf:input>
```

Listing 2.7: `xf:input` Element

2.3.2 The `xf:textarea` Element

Description: This element is derived from XForms. A textarea is used for free-form data and can contain arbitrary text and control sequences, e.g. linebreaks.

Common Attributes: Common (2.1.1)

Special Attributes: none

Common Child Elements: Common (2.1.2)

Special Child Elements: `inputproperties` (2.4.11), `properties` (2.4.12)

Example:

```
1 <xf:textarea ref="/partnercontext[1]/externalid[1]">
2  <xf:label>Externalid</xf:label>
3 </xf:textarea>
```

Listing 2.8: `xf:textarea` Element

2.3.3 The `list` Element

Description: This element enables handling of lists of elements. A list can contain one or more types of elements which are specified by the `itemClass` child element. The number of columns and their content are also defined there per type of element. The headers can be defined using the `headers` child element. The `list` element is capable of both showing plain lists and tree structures.

Common Attributes: Common (2.1.1)

Special Attributes:

- `showHeader` [optional] [default=false]

  An optional attribute that defines whether the list should display a header describing the columns or not. It can be either "`true`" or "`false`". If it is omitted, no header should be shown.

- `showSearch` [optional] [default=false]

  An optional attribute that defines whether the list should present a facility to search items in the list. This is very useful for editing lists with a large number of items.

- `minOccurs` [optional] [default=0]

  An optional attribute specifying the minimum number of elements the list has to contain in order to build a valid xml document. The default minimum number is 0.

- `maxOccurs` [optional] [default=0]

  An optional attribute specifying the maximum number of elements the list may contain in order to build a valid xml document. The default maximum number is 0. A maximum of 0 means that the occurrence is unbounded.

Common Child Elements: Common (2.1.2)

Special Child Elements:

- `headers` (2.4.4)

  Defines the headers of the columns.

- `itemClass` (2.4.6)

  Defines the element that the list corresponds to when the list operates in single-column mode.

Example:

```
<list showHeader="true">
 <xf:label>Item</xf:label>
  <itemclass ref="/feature">
    <itemlabel><itemLabelArg ref="/summary[1]"
        truncate="40"/></itemlabel>
    <itemlabel><itemLabelArg ref="/@status[1]"
        truncate="40"/></itemlabel>
    <itemlabel><itemLabelArg ref="/@target[1]"
        truncate="40"/></itemlabel>
  </itemclass>
 <itemclass ref="/category">
  <itemlabel>Category: <itemLabelArg ref="@name"/></itemlabel>
 </itemclass>
 <headers>
  <header>Summary</header>
  <header>Status</header>
  <header>Target</header>
 </headers>
</list>
```

Listing 2.9: `list` Element

2.3.4 The `section` Element

Description: The `section` element is used to visually encapsulate a set of elements. That is appropriate when a complexType element with several child elements is shown, for example. As a result the GUI should be less cluttered.

Common Attributes: Common (2.1.1)

Special Attributes:

- `externalLabel` [optional] [default=false]

  Most widgets that are used to group elements provide a possibility to attach a title or label. In most cases this will look more polished and less cluttered than an external label as it is used with the other GUI elements. If desired it can still be used by setting the `externalLabel` attribute to true.

Common Child Elements: Common (2.1.2), GUI Elements (2.3)

Special Child Elements: none

Example:

```
<section ref="/">
 <xf:input ref="/organisation"/>
 <xf:input ref="/externalid"/>
</section>
```

Listing 2.10: `section` Element

2.3.5 The `xf:select1` Element

Description: This element is derived from XForms. It allows the user to make a single selection from multiple choices.

Common Attributes: Common (2.1.1)

Special Attributes: none

Common Child Elements: Common (2.1.2)

Special Child Elements: `xf:item` (2.4.9)

Example:

```
<xf:select1 ref="/documentationstatus[1]">
 <xf:label>Status of the documentation: </xf:label>
 <xf:item>
  <xf:label>Postponed</xf:label>
  <xf:value>postponed</xf:value>
 </xf:item>
 <xf:item>
  <xf:label>Information required </xf:label>
  <xf:value>needinfo</xf:value>
 <xf:item>
</xf:select1>
```

Listing 2.11: `xf:select1` Element

2.3.6 The `xf:select` Element

Description: This element is derived from XForms. It allows the user to make one ore more selections from multiple choices.

Common Attributes: Common (2.1.1)

Special Attributes: appearance (2.7.3)

Common Child Elements: Common (2.1.2)

Special Child Elements: `xf:item` (2.4.9)

Example:

```
1  <xf:select ref="/attendees">
2   <xf:label>Attendees of the meeting: </xf:label>
3   <xf:item>
4    <xf:label>Developer</xf:label>
5    <xf:value>developer</xf:value>
6   </xf:item>
7   <xf:item>
8    <xf:label>Manager</xf:label>
9    <xf:value>manager</xf:value>
10  <xf:item>
11 </xf:select>
```

Listing 2.12: `section` Element

2.4 KXForms Special Elements Descriptions

2.4.1 The `defaults` Element

Description: This element lets you set defaults that will be used for the whole document. If this element does not exist, the kxforms defaults are applied, however it might be advisable to override these defaults in some cases.

Common Attributes: none

Special Attributes: none

Common Child Elements: none

Special Child Elements: `inputproperties` (2.4.11), `properties` (2.4.12)

Example:

```
1 <defaults>
2  <inputproperties>
3   <type>xs:integer</type>
4  </inputproperties>
5  <properties>
6   <appearance>Full</appearance>
7   <layoutstyle>vertical</layoutstyle>
8  </properties>
9 </defaults>
```

Listing 2.13: `defaults` Element

2.4.2 The `groups` Element

Description: This element defines the groups that are available on the corresponding form. By default, all controls are placed at the same page. When there are too many controls or there is a strong logical separation between them it might be appropriate to put them into groups. These groups can be declared using the `groups` element.

The placement and presentation of the groups is left to the application. In most cases, a tab widget might the most appropriate kind of presentation.

Common Attributes: none

Special Attributes: none

Common Child Elements: none

Special Child Elements: `group` (2.4.3)

Example:

```
<groups>
 <group id="feature">Feature Data</group>
 <group id="product">Product Data</group>
 <group id="doc">Documentation</group>
</groups>
```

Listing 2.14: `groups` Element

2.4.3 The `group` Element

Description: This element specifies the titles of one group on the form.

Common Attributes: none

Special Attributes:

- `id`

  This attribute specifies the title of the a group. This group can then be referenced from GUI Elements by its id in order to be put in this group.

Common Child Elements: none

Special Child Elements: none

Example:

```
<group id="feature">Feature Data</group>
```

Listing 2.15: `group` Element

2.4.4 The `headers` Element

Description: This element is used to specify the columns of a list.

Common Attributes: none

Special Attributes: none

Common Child Elements: none

Special Child Elements: `header` (2.4.5)

Example:

```
1 <headers>
2  <header>Summary</header>
3  <header>Status</header>
4  <header>Target</header>
5 </headers>
```

Listing 2.16: `headers` Element

2.4.5 The `header` Element

Description: This element defines the header of a column in a list element.

Common Attributes: none

Special Attributes: none

Common Child Elements: none

Special Child Elements: none

Example:

```
1 <header>Summary</header>
```

Listing 2.17: `header` Element

2.4.6 The `itemClass` Element

Description: This element describes the element a list corresponds to.

Common Attributes: Common (2.1.1)

Special Attributes:

- `list` [optional] [default=false]

  This attribute specifies if items of this class should be displayed recursively, building a tree. That sometimes gives a better overview over the data and makes navigation easier.

Common Child Elements: none

Special Child Elements: `itemLabel` (2.4.7)

Example:

```
1 <itemclass ref="/productcontext">
2  <itemlabel>
3   <itemLabelArg ref="/product/productid" truncate="20"/>
4  </itemlabel>
5 </itemclass>
```

Listing 2.18: `itemClass` Element

2.4.7 The `itemLabel` Element

Description: This element describes which element provides the label for a single-column list.

Common Attributes: none

Special Attributes: none

Common Child Elements: none

Special Child Elements: `itemLabelArg` (2.4.8)

Example:

```
1 <itemlabel>Product-Id: <itemLabelArg ref="/product/productid"
2     truncate="20"/></itemlabel>
```

Listing 2.19: `itemLabel` Element

2.4.8 The `itemLabelArg` Element

Description: This element defines a reference to an xml entity which will be substituted into the label for the list.

Common Attributes: Common (2.1.1)

Special Attributes:

- `truncate` [optional]

  Optional attribute that defines after how many chars the label should be truncated. If the label succeeds the defined length, it will be truncated and "..." should be appended to indicate the truncation.

Common Child Elements: none

Special Child Elements: none

Example:

```
1 <itemLabelArg ref="/product/productid" truncate="20"/>
```

Listing 2.20: `itemLabelArg` Element

2.4.9 The `xf:item` Element

Description: This element is derived from XForms. It defines one choice in controls that allow the user to choose between several items.

Common Attributes: none

Special Attributes: none

Common Child Elements: Common (2.1.2)

Special Child Elements: `xf:value` (2.4.10)

Example:

```
1 <xf:item>
2  <xf:label>Implementation</xf:label>
3  <xf:value>implementation</xf:value>
4 </xf:item>
```

Listing 2.21: `xf:item` Element

2.4.10 The `xf:value` Element

Description: This element is derived from XForms. It defines a value that is returned for a selection, for example in a select1 control (2.3.5).

Common Attributes: none

Special Attributes: none

Common Child Elements: none

Special Child Elements: none

Example:

```
1 <xf:value>implementation</xf:value>
```

Listing 2.22: `xf:value` Element

2.4.11 The `inputproperties` Element

Description: The `inputproperties` element encapsulates further specifications of the content of input controls. This can be the type or regular expressions as constraints for example.

Common Attributes: none

Special Attributes: none

Common Child Elements: none

Special Child Elements: Input Property Elements (2.5)

Example:

```
<inputproperties>
 <type>xs:integer</type>
 <constraint>\d\w*</constraint>
</inputproperties>
```

Listing 2.23: `properties` Element

2.4.12 The `properties` Element

Description: The `properties` element encapsulates further specifications of the parent control, such as relevance or layout information.

Common Attributes: none

Special Attributes: none

Common Child Elements: none

Special Child Elements: Property Elements (2.6)

Example:

```
<properties>
 <layout>
  <halign>right</halign>
 </layout>
</properties>
```

Listing 2.24: `properties` Element

2.5 KXForms Input Property Elements Descriptions

2.5.1 The `type` Element

Description: The `type` element defines the type of the data that is hold in the corresponding control. All XML Schema types should be supported. Depending on the type, the application can display other widgets for the control, e.g. a lineedit for `xs:string`, a spinbox for `xs:integer` and so on.

Additionally, the `type` element should be used to validate the content of the control. A `xs:integer` control containing chars should be qualified as invalid, for example.

Possible Content: One of the datatypes defined in the XML Schema specification.

Default: `xs:string`

Example:

```
1 <type>xs:string</type>
```

Listing 2.25: `type` Element

2.5.2 The `constraint` Element

Description: This element holds a regular expression that the content of the control has to match in order to be qualified as valid. If invalid the control should visually indicate the invalid state and circumvent the data to be stored as XML.

Possible Content: A regular expression.

Default: none

Example:

```
1 <constraint>\w{3}\d</constraint>
```

Listing 2.26: `contraint` Element

2.6 KXForms Property Elements Descriptions

2.6.1 The `readonly` Element

Description: This element is used to mark controls as read-only, thus preventing the user to edit the content.

Possible Content: `true` — `false`

Default: false

Example:

```
1 <readonly>true</readonly>
```

Listing 2.27: `readonly` Element

2.6.2 The `relevant` Element

Description: With the `relevant` element it is possible to activate controls depending on the state of another control. Therefor the value of the element specified by the `ref` attribute is evaluated and compared to the value of the `relevant` element. If they match the control is set to read-and-write state, if they don't it is set to read-only state.

Possible Content: A regular expression.

Default: none

Example:

```
1 <relevant ref="id/isExternal">true</>
```

Listing 2.28: `relevant` Element

2.6.3 The `layout` Element

Description: This element is used to define the layout of the GUI more fine-grained.

Possible Content: Layout elements (2.7)

Default: none

Example:

```
<layout>
 <page>1</page>
 <position>-1</position>
</layout>
```

Listing 2.29: `layout` Element

2.7 KXForms Layout Elements Descriptions

2.7.1 The `groupRef` Element

Description: Per default, all controls of a form are placed on one form in the order they appear in the kxforms document. If there are too many of them the resulting GUI might be cluttered or might not fit on the screen. A possible solution is to encapsulate the controls in groups. Elements which belong to the same group will then be placed next to each other and if the groups occupy too much space, they are put into tabs.

If the groupRef property of an element is empty, it is put into an abstract group after all defined groups.

Possible Content: String

Default: empty

Example:

```
1 <groupRef>other</other>
```

Listing 2.30: `groupRef` Element

2.7.2 The `position` Element

Description: This element defines the position of the control. The positions are given in a relational approach, that means that the position is defined in relation to another element. Both elements have to be in the same group, otherwise the directive will have no effect.

The relations can be given with four different tags:

- leftOf
- rightOf
- below
- above

Possible Content: Arbitrary combination of `leftOf`, `rightOf`, `below` and `above` elements

Default: empty

Example:

```
<position>
 <rightOf>/title</rightOf>
</position>
```

Listing 2.31: `position` Element

2.7.3 The `appearance` Element

Description: This element tells the application how to render the control it is applied to. "`full`" indicates that all options should always be visible. "`compact`" means that an adequate number of options should be shown at once while "`minimal`" should result in a presentation that always shows just one option at once and thus takes up the minimum space.

Possible Content: `full` — `compact` — `minimal`

Default: `minimal`

Example:

```
<appearance>Full</appearance>
```

Listing 2.32: `appearance` Element

2.7.4 The `layoutstyle` Element

Description: Elements usually exist of two parts: The widget containing the actual data and a label widget, showing a descriptive title for the element. This element can be used to change the way, the application should arrange these two widgets. There are two options available:

- `horizontal`

  This option is chosen, the application should place the label widget left of the widget, showing the data.

- `vertical`

  This option is chosen, the application should place the label above the widget, showing the data.

Possible Content: `horizontal` — `vertical`

Default: `horizontal`

Example:

```
1 <layoutstyle>vertical</layoutstyle>
```

Listing 2.33: `layoutstyle` Element

Chapter 3

Hints

Usually, a GUI that is generated only from the schema describing the data will not fulfill all requirements on a modern user interface. Therefor, hints can be merged into the kxforms document.

3.1 Technique

A hint is an XML fragment and states a directive to the application, to change a property of an element. Thus, the hint has to describe which element it addresses. This is done using a XPath expression (see 2.4).

The hint can now specify a property of the element and the value it should be overridden with. Thereby there are two different types of hints available:

- **Key-Value hints**

  This kind of hints is used to specify the value of one property. This is applicable to simple type properties such as a label.

```
1 <hint ref="/title[1]">
2  <label>Title of the Feature</label>
3 </hint>
```

Listing 3.1: Example of a Key-Value hint

- **Valuelist hints**

  There are however properties that can not be described with one value. Therefor, this type of hint can be used. The content of the hint is again an XML fragment. The parent XML element describes the property that is changed. This element can then have a list of values as child elements.

```
<hint ref="/title[1]">
 <position>
  <rightOf>/description[1]</rightOf>
  <above>/list_actors[1]</above>
 </position>
</hint>
```

Listing 3.2: Example of a valuelist hint

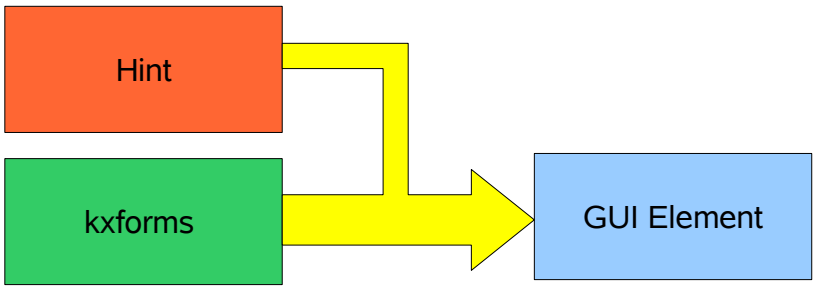

Figure 3.1: Hint architecture

3.2 Implemented Hints

Hints Modifying The Displayed Data

The plainest and probably most common modification, a developer or GUI-designer wants to do is changing the data that is shown. The following list will describe the hints that make such changes possible.

- **label**

  Usually an element is labeled with its "name" attribute. In some cases these attributes might be abbreviations, non-readable ids or simply not understandable enough so that it is advisable to display a different label in the GUI. That can be achieved using the label-hint.

  This hint simply overrides the automatic label generation mechanism and defines the label of the specified element.

```
1 <hint ref="needinfo">
2  <label>Information required </label>
3 </hint>
```

Listing 3.3: label hint

ref defines the element of the XML-Schema this hint should be applied to

label specifies the label that should be applied

This example would change the displayed label of the “needinfo” element from “Need-Info” to “Information required”.

- **appearance**

  This hint modifies the “appearance” property of an element.

```
1 <hint ref="/status[1]">
2  <appearance>full</appearance>
3 </hint>
```

Listing 3.4: appearance hint

- **layoutStyle**

  This hint modifies the “layoutStyle” property of an element.

```
1 <hint ref="/status[1]">
2  <layoutStyle>vertical</layoutStyle>
3 </hint>
```

Listing 3.5: layoutStyle hint

Behavioural Hints

- **readOnly**

  This hint modifies the “readOnly” property of an element.

```
1 <hint ref="/status[1]">
2  <readOnly>false</readOnly>
3 </hint>
```

Listing 3.6: readOnly hint

- **inputType**

  With the ”inputType“ hint, it is possible to override the type of an Input element.

```
1 <hint ref="/duration[1]">
2  <inputType>xs:integer</inputType>
3 </hint>
```

Listing 3.7: inputType hint

Positioning hints

- **groups**

  The groups hint can be used to define a set of element groups, where the elements can be put into.

```
1 <hint ref="/feature">
2  <groups>
3   <group id="feature">Feature</group>
4   <group id="documentation">Documentationstatus</group>
5  </groups>
6 </hint>
```

Listing 3.8: groups hint

- **groupRef**

  This hint puts an element into a specific group

```
1 <hint ref="/duration[1]">
2  <groupRef>feature</groupRef>
3 </hint>
```

Listing 3.9: groupRef hint

- **position**

  With this hint a relational position description of an element can be created.

```
1 <hint ref="/title[1]">
2  <position>
3   <rightOf>/description[1]</rightOf>
4   <above>/list_actors[1]</above>
```

```
5   </position>
6  </hint>
```

Listing 3.10: position hint

- **formSizeThreshold**

  This hint sets the size threshold of a form, which is used to determine the placement of the groups.

```
1  <hint ref="/feature">
2   <formSizeThreshold>120</formSizeThreshold>
3  </hint>
```

Listing 3.11: formSizeThreshold hint

List hints

There are two different types of lists, lists of SimpleType elements and lists of ComplexType elements. SimpleType lists are straightforward, they just show the content of the elements. In the case of ComplexType elements however, there are many possibilities to modify the appearance of the lists. The following list describes those possibilities and the corresponding hints.

- **listItemLabel**

  KXForms is capable of presenting lists of elements. Usually, the elements are simple elements which can be shown in the list directly. If the elements are complex elements, it is important to show a subelement in the list which describes or identifies the element best.

  If no further information is given, KXForms chooses the first simple element. This might not always be the best choice, so it can be overridden using the `listItemLabel` hint.

```
1  <hint ref="/product[1]">
2   <listItemLabel>Version <itemLabelArg ref="/@version[1]"
3     truncate="40"/></listItemLabel>
4  </hint>
```

Listing 3.12: ListItemLabel hint

- **listShowSearch**

  This hint sets the filter bar to visible or hidden. Showing the filter bar is advisable for lists which contain many entries, so that filtering for a string might help finding a specific item.

```
1 <hint ref="list_productcontext">
2  <listShowSearch>true</listShowSearch>
3 </hint>
```

Listing 3.13: listShowSearch hint

- **listShowHeader**

  By default, the headers of lists are hidden. In most cases it is not necessary to describe the data that is shown because it is either the content of the elements itself or in cases complexTypes a descriptive subelement, eventually chosen with a `listItemLabel` hint.

  Sometimes it still might be necessary to show the header, for example if the list has more than one column. That can be achieved using the `listShowHeader` hint as demonstrated below.

```
1 <hint ref="list_productcontext">
2  <listShowHeader>true</listShowHeader>
3 </hint>
```

Listing 3.14: listShowHeader hint

- **listHeader**

  The `listHeader` hint lets one override the label of a column in a list, e.g. if the automatically generated one is not understandable.

```
1 <hint ref="/status[1]">
2  <listHeader>Feature status</listHeader>
3 </hint>
```

Listing 3.15: listHeader hint

- **listItemList**

  The `listItemList` hint can be used to mark a type of list item as a list item. That means, that it contains other elements, which can be shown by expanding this item.

```
1 <hint ref="/category[1]">
2  <listItemList>true</listItemList>
3 </hint>
```

Listing 3.16: listItemList hint