ulm university universität ulm

Ulm University | 89069 Ulm | Germany

Faculty of Engineering and Computer Science
Institute of Databases and Information Systems

Technical Conception and Implementation of a Configurator Environment for Process-aware Questionnaires Based on the Eclipse Rich Client Platform

Master thesis at Ulm University

Submitted by:
Juri Schulte
juri.schulte@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert
Dr. Vera Künzle

Supervisor:
Johannes Schobel

2014

Version April 23, 2014

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

Questionnaires are one of the fastest and easiest methods for inquiring information about a required topic. Especially the more and more advancing online connectivity and mobile accessibility offer additional possibilities, like working collaboratively from different places or store results centrally, to make it an even faster and more comfortable tool for data collection. Several existing software approaches to create questionnaires – called questionnaire configurators – are available and expensively tailor functionality to the needs of the target group. In this thesis an approach is presented, which outsources tasks to functionality provided by a process-aware information system (PAIS). To offer extensibility for upcoming needs, a generic questionnaire model is the basis for an integration of a PAIS into a questionnaire configurator environment. The result is called a process-aware questionnaire configurator and is discussed regarding its architecture and implementation. With an implemented prototype of a process-aware questionnaire configurator an insight is granted into a concrete implementation based on the Eclipse Rich Client Platform.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Summary of Contribution 2

2 Fundamentals **3**

2.1 Questionnaires 3

2.1.1 Definitions 4

2.1.2 Measuring Scales 5

2.1.3 Elements 7

2.1.4 Construction of a Questionnaire 8

2.2 Process-aware Workflow 9

2.2.1 Activities 10

2.2.2 Data Elements 11

2.2.3 Branches 12

2.2.4 Loops 13

2.3 Eclipse Rich Client Platform 13

2.3.1 RCP Components 14

2.3.2 RCP Workbench 16

2.3.3 RCP Application Model 18

3 Related Work **21**

3.1 Web-based Applications 21

3.2 Desktop Applications 24

3.3 Mobile Applications 26

3.4 Summary 27

4 Process-aware Questionnaires **29**

4.1 Requirements on the PAIS environment 29

4.2 A Technical View on Generic Questionnaires 30
4.3 Integration of Questionnaires in Process-aware Information Systems 33
4.3.1 Constraints within Questionnaires 36

5 Architecture and Implementation **39**
5.1 Core Components 39
5.1.1 User Interface Components 40
5.1.2 Background Components 43
5.2 Persistence 46
5.2.1 Direct Database Connection 47
5.2.2 Web Services 49
5.2.3 Summary 49
5.3 Design Patterns 50
5.4 Usage of the Rich Client Platform 53
5.5 Summary 56

6 Conclusion **57**

7 Future Work **59**

A The Process-aware Questionnaire Configurator Prototype **61**

Bibliography **67**

List of Figures

2.1 Filter including the questions from example 1 - 3. 8
2.2 ADEPT2 example for an ordering process. 10
2.3 Overview of important RCP components. 14
2.4 An RCP workbench example including a window with a menu, tool bar, part stack and three parts. 17
2.5 The internal steps for building a workbench. 18

3.1 The QuestionSys questionnaire configurator (see [28]). 22
3.2 The LimeSurvey questionnaire configurator. 23
3.3 The SurveyGold questionnaire configurator. 25
3.4 The StatPac questionnaire configurator. 25

4.1 A schematic representation of a questionnaire as a container for question pages (questionnaire layer). 31
4.2 A schematic representation of a questionnaire page as a container for question elements (page layer). 31
4.3 Example for a process-aware questionnaire at element level. 34
4.4 Example for a process-aware questionnaire at page level. 35

5.1 Core components of the architecture of a process-aware questionnaire configurator. 40
5.2 Excerpt of the element builder environment of a process-aware questionnaire configurator prototype. 41
5.3 Excerpt of the modelling environment of a process-aware questionnaire configurator prototype. 42
5.4 Status bar of a process-aware questionnaire configurator prototype. 44
5.5 Architecture of the persistence components of a process-aware questionnaire configurator prototype. 47

5.6 Excerpt of the questionnaire management of a process-aware questionnaire configurator prototype. 51
5.7 The hierarchical agent structure of the PAC pattern for the questionnaire managment in figure 5.6. 52
5.8 The hierarchical agent structure of the PAC pattern for the questionnaire managment using RCP. 54

A.1 Welcome perspective of a process-aware questionnaire configurator prototype. 61
A.2 Workspace perspective of a process-aware questionnaire configurator prototype. 62
A.3 Questionnaire perspective of a process-aware questionnaire configurator prototype. 62
A.4 Questionnaire configuration perspective of a process-aware questionnaire configurator prototype. 63
A.5 Simple element builder of a process-aware questionnaire configurator prototype. 63
A.6 Complex element builder of a process-aware questionnaire configurator prototype. 64
A.7 Adding a sensor for a complex element in a process-aware questionnaire configurator prototype. 64
A.8 Multimedia elements in a process-aware questionnaire configurator prototype. 65
A.9 Questionnaire builder of a process-aware questionnaire configurator prototype. 65

1 Introduction

Questionnaires are an inexpensive method to collect large amounts of data in rather short time. Nowadays, the creation of such questionnaires is supported by software applications, called questionnaire configurators, by providing an (more or less) extensive set of tools and mechanisms to manage a questionnaire's life cycle. If this life cycle is transferred to a process-aware information system (PAIS) [25] common tasks, like the questionnaire execution, could be automated and even extended [9]. Although questionnaires have been extensively studied in the last decades, the focus lays primarily on structural [2, 14], representational [21] and content-related [2, 27] aspects, disregarding the technical view. Available configurators are mostly proprietary software applications tailored to the specific needs of a target group and unavailable for research purposes. This thesis therefore develops an approach, which integrates PAIS functionality in a questionnaire configurator, and validates the resulting benefits by implementing a process-aware questionnaire configurator for the first time.

Therefore the sociological fundamentals of questionnaires and technical aspects of a PAIS are outlined in chapter 2. This includes an overview of the Rich Client Platform (RCP) as well, which is used for the implementation of a prototype of a process-aware questionnaire configurator. In chapter 3 existing questionnaire configurators are analysed to create a generic questionnaire model in chapter 4. The latter defines the basic construction pattern of every questionnaire and catches deviating characteristics through extension points. Afterwards this generic model is transferred to the ADEPT2 PAIS theoretically and then in chapter 5 practically. For the practical realisation an architecture is provided, which discusses important components, aligns them with well-known design patterns and enriches them by RCP functionality.

1.1 Motivation

Questionnaires are used in a wide area of applications, including marketing research, employee feedback, recording medical backgrounds, etcetera. Data can be collected anonymously and through different channels (e.g., pen-and-paper based or internet based) without the mandatory presence of an interviewer. Furthermore, questionnaire configurators – and therefore software applications – offer supplementary tools to integrate multimedia content, share questionnaires online, store results centrally, analyse results and many more. In addition, computer supported questionnaires can be beneficial, if the survey topic is computer-related [29] in the first place. As a result, feature-rich configurators are of great concern for any kind of data collection and therefore of high commercial interest. With the access to PAIS functionality a configurator is extended by several strong features, like automatic questionnaire execution or access rights through organisational models, which are already available and do not require an implementation from scratch.

1.2 Summary of Contribution

The main contribution of this thesis is an approach for an extensible and flexible process-aware questionnaire configurator. This includes initially an analysis of consistent questionnaire parts and extensions. A generic model, which describes the structure of every possible questionnaire, is extracted and discussed regarding its similarities to a process model. On this basis an approach is developed to represent this generic model as a special kind of process and therefore use PAIS functionality to support a questionnaire's life cycle. With an architecture including components, design patterns and a development platform, a recommendation for an implementation is provided and validated by an exemplary prototype for a process-aware questionnaire configurator.

This paves the way for thin questionnaire configurators, which outsource functionality to existing applications. From a general point of view, a new challenge is transferred to an existing problem for which a solution is already available.

2 Fundamentals

A configuration system for process-aware questionnaires requires the merge of research areas from two diverse domains: On the one hand psychology knowledge regarding the structure and construction of questionnaires, on the other hand a profound technical understanding of process-aware information systems. In the following sections we cover the requirements for a fundamental understanding of both branches. Special focus lays on the technical aspects.

In section 2.1 questionnaires are analysed regarding their construction and relevant parts. Afterwards in section 2.2 the fundamentals of process-aware workflows are covered. The content of section 2.1 and section 2.2 is required for the approach of a process-aware questionnaire in chapter 4. Section 2.3 gives an insight to the Eclipse Rich Client Platform which is used in chapter 5 for the software architecture of a process-aware questionnaire configurator.

2.1 Questionnaires

Questionnaires belong to the empiric research in social sciences. In the empiric research there are two different approaches of gaining information about a predefined topic. Firstly, *qualitative research methods* which define a thematic guideline, but avoid standardised proceedings. These methods are flexible and exploratory approaches with the aim of building a hypothesis. Take for an example a group discussion of the topic *'Should smoking be prohibited'*. The topic is predefined and a moderator lets the participants offer their opinions. The opinions are written down and analysed later on.

Secondly, *quantitative research methods* which focus on building a universal model with a representable number of samples. This model usually proves or refutes a predefined hypothesis. To gain comparable information, the same requirements have to be provided for

each respondent. The quantitative methods are therefore usually fully standardised. *Questionnaires* in written form belong to this quantitative research method. [31]

Definition 1 (*Questionnaire*)

> *"A questionnaire is defined as a document containing questions and other types of items designed to solicit information appropriate to analysis." [3]*

The great number of questionnaire types is categorised by its process. There are personal, telephone, written and online questionnaires. As described in definition 1 questionnaires provide analysable measurable data. Depending on the question this data can be interpreted more or less precisely (see section 2.1.2). To fulfil this goal a set of elements (see section 2.1.3) is necessary. Besides questions, text and media fragments are important parts of a questionnaire. These elements are ordered to determine a sequence through the questionnaire. A modification of the sequence is possible through filters (see section 2.1.4). First of all section 2.1.1 covers the basic vocabulary to work with questionnaires.

2.1.1 Definitions

Questionnaires gather data and interpret this data to obtain information. Processable and therefore interpretable and meaningful data has four properties. It is *valid*, *reliable*, *representative* and *unbiased*. [19, 23] These properties are discussed using the following definitions.

Definition 2 (*Validity*)

> *"A study has a high validity if the results allow only a unique interpretation." [23]*

Valid data has a high content-related expressiveness.

Definition 3 (*Reliablity*)

> *"The repeatability of an instrument." [27]*

Reliability describes the fact that an extracted result is constant through several repeated experiments. Reliable data requires always a *representative* number of samples.

Definition 4 (*Representative*)

> *"Random samples are representative if their characteristic reflect the characteristic of the entire population." [23]*

If the number of random samples is too small or does not cover a heterogeneous group the data is not representable. Reliable data requires always a representative number of samples.

Definition 5 (*Unbiased*)

> *"The stability of an instrument independent from environment variables and the person who uses it." [27]*

For the avoidance of disturbances and errors an unbiased environment has to be created. For example setting a default answer like 'Yes' or 'No' for a Yes-or-No question will affect the choice of the respondents.

If a questionnaire produces valid, reliable, representative and unbiased data we call it a *standardised instrument*. For a standardised instrument the validity of its elements is as important as the valid order of the elements.

2.1.2 Measuring Scales

Questions provide the data needed for analysis of a questionnaire. To analyse empiric magnitudes, answers are mapped to measurable numbers. Therefore a scale is defined for every question. Every scale has a different set of comparative operators M. Comparative operators define in which ways gathered data can be transformed without any information loss. The larger the set of operators, the higher is the expressiveness of the scale. Four different scales with a static set of comparative operators are distinguished:

The *Nominal Scale* ($M = \{=, \neq\}$) contains the smallest set M of comparative operators and is therefore the most inaccurate scale. It only checks the data for equality and thus for inequality either. As shown in example 1 the answers only indicate whether position `1 = Yes` or `2 = No` was chosen. There is no information about the hierachic relation between the two options included.

Example 1 (*Nominal Scale*)

Do you smoke?
1. Yes
2. No

Example 2 (*Ordinal Scale*)

How many cigarettes do you smoke a day?
1. 0 - 5 cigarettes
2. 5 - 15 cigarettes
3. > 15 cigarettes

This is provided by the *Ordinal Scale* ($M = \{=, \neq, <, >\}$). In addition to the check for equality the answer options can be compared to each other (less or greater). In example 2 option 2 includes a greater amount of cigarettes than option 1, i.e., respondents who choose option 2 smoke more cigarettes than respondents who choose option 1.

The *Interval Scale* ($M = \{=, \neq, <, >, +, -\}$) allows through the comparative operators '+' and '-' a more precise comparison of the replies. Next to the rough information about the relation (less or greater) the precise difference can be calculated. Applied to example 3 the time difference between two respondents started smoking allows very clear statements: If respondent A started smoking in the year 1960 and respondent B started in the year 1980, respondent A smokes 20 years longer than respondent B. Although it is obvious for a human that respondent B has been smoking twice as long as respondent A, if the survey had taken place in the year 2000, this is not directly included in the data. Therefore the proportion of the numbers does not correlate with the manifestation of the characteristic.

Example 3 (*Interval Scale*)

In which year did you start smoking?
In year ____ .

Example 4 (*Ratio Scale*)

How many years have you been smoking?
____ years.

This fact is covered by the *Ratio Scale* ($M = \{=, \neq, <, >, +, -, \times, /\}$) which requires an absolute zero. If the question from example 3 is rewritten as shown in example 4 the value directly reflects the years the respondent has been smoking. The value zero defines that a respondent has not been smoking a year yet. As a result, statements about the ratio can be made, like 'Respondent C smokes for 20 years and respondent D smokes for 40 years, therefore respondent D smokes twice as long as respondent C'.

In practice the distinction between the Interval Scale and the Ratio Scale is mostly irrelevant. With the operators from the Interval Scale essential statistic methods can already be performed. [2, 23]

2.1.3 Elements

Questionnaires host a wide variety of different element types. *Questions*, *text* and *media content* are the most fundamental element groups.

The most important and complex element type is the *question* (also called *item*). Questions generate pure data and properly gathered data lead to usable information – the aim of every survey. A variety of question types exist and therefore many possible categorisations. A well-documented way to categorise questions is to discriminate between *open*, *closed* and *hybrid* questions. Open questions allow respondents to express the answer in their own words. No restrictions – besides the wording of the question itself – are imposed. Afterwards researchers interpret and categorise these answers. Closed questions define a static set of replies. Each respondent has to pick one (or more) replies which match the respondents opinion best. The static sets of replies are a well-known source of errors because every reply has to be exclusive and the entire set has to cover every possible answer. In contrast, closed questions can be interpreted fully automatically and therefore much faster than open questions [14]. A hybrid question is a closed question where one or more replies are constructed like an open question. These questions are used if the set of replies does not cover the respondent's answer. [21]

Another categorisation of questions is based on the tasks of the respondent. Next to the widespread *single choice* and *multiple choice* questions, *ranking* and *distribution* questions exist. Whereas single choice and multiple choice questions simply prompt the respondent to pick from a set of replies, ranking and distribution questions demand more interaction. In ranking questions a set of replies has to be ordered (e.g., by the importance or acceptance for the respondent). Distribution questions instead define a pool of points which can be distributed to the available replies. The number of points on each reply represents the importance of the reply to the respondent.

Besides questions, a questionnaire contains a lot of standalone *text* parts. Text paragraphs are used to inform (e.g., of the purpose of the survey), to prepare (e.g., explain the next steps) or to help the respondent (e.g., if the wording of a question is not understood). A typical questionnaire contains the following text fragments:

- Introduction text (at the beginning of a questionnaire stating the involved organisations)
- Headlines (introduce new questionnaire parts)

- Information text (can bridge to new topics or provide information for the recent topic)
- Help text (provides the respondent with additional information if a question or text is not fully understood)
- Instructions for the interviewer (regarding the behaviour of the interviewer)

Furthermore, *media content* and *logo*s are parts of a questionnaire. Logos identify the involved organisations and media content provide additional information for the respondent. Media content includes videos, audio files and images. For example a respondent can be confronted with pictures of lung cancer patients in connection with a question about the attitude towards smoking. [3, 27]

2.1.4 Construction of a Questionnaire

A questionnaire is a container including elements in a strict defined order. Some of the elements are visible to the respondent, others are not. Consider example 1 to 4 from section 2.1.2. If the question from example 1 is answered with 'No' the questions from example 2 to 4 are not relevant for the further course of the interview. In fact, in this case they are contra-productive since they bore respondents and result in a loss of concentration. [14, 27]

To assure only the relevant elements for a specific group of respondents are shown *filter* are an important tool. Filters define paths through a questionnaire by creating dependencies between a reply and the set of following elements (i.e., shown in figure 2.1). Depending on the construction of a filtering question several new paths can be created with one filter question.

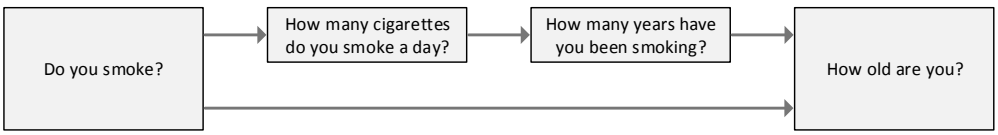

Figure 2.1: Filter including the questions from example 1 - 3.

For questionnaires in paper form a text has to be added to define which element will be the next one. Software applications instead are able to hide unused elements and show the

next element directly. Because of dependencies between questions, filter can aggregate several filter questions to determine the following path.

The typical setup of a questionnaire starts with an introduction. The responsible organisations and authors are introduced and the topic of the survey is stated. Afterwards, simple questions, like demographic information, should accustom the respondent to the survey. At the end simple questions should be used as well to react to the loss of concentration during long surveys. The complex and more relevant and sensible questions have to be placed between the start and the end elements. To structure the questionnaire the use of thematic paragraphs and text passages is recommended, as seen in section 2.1.3. [27]

2.2 Process-aware Workflow

A *process* describes a sequence of single activities to accomplish a predefined assignment. The structure of a process features a clear definition of its activities and their dependencies. Complex dependencies between activities require special constructs, like branches or loops, which manipulate the sequence of activities during runtime. Because a process is designed once and executed several times, the term *process instance* is used for a single run. All parts of a process which can be performed computerised are called a *workflow* and can be supported by *process-aware information systems* (PAIS). The latter offer controls for the management of process instances (e.g., the assignment of an authorised user to outstanding activities can be managed fully automatically). Sections 2.2.1 - 2.2.4 provide an overview of the fundamental parts of a PAIS focusing on the example ADEPT2 [25]. The ADEPT2 PAIS covers all of the elementary functionality of a PAIS and enables additional operations for dynamic process evolution and ad-hoc deviations. The approach is based on a *relaxed block-structure*. With this structure every branch and every loop has exactly one entry and one exit node. An entry node represents the start of a new block and an exit node the corresponding end. Per convention a process has always a static start and exit node defining its start and termination point.

Though two blocks are allowed to be nested, no two blocks may overlap. Because this fact limits the expressiveness of ADEPT2 supplementary operations have been added in the form of *synchronisation edge*s. A synchronisation edge expresses a constraint whereby the start of an activity depends on the termination of a different activity.

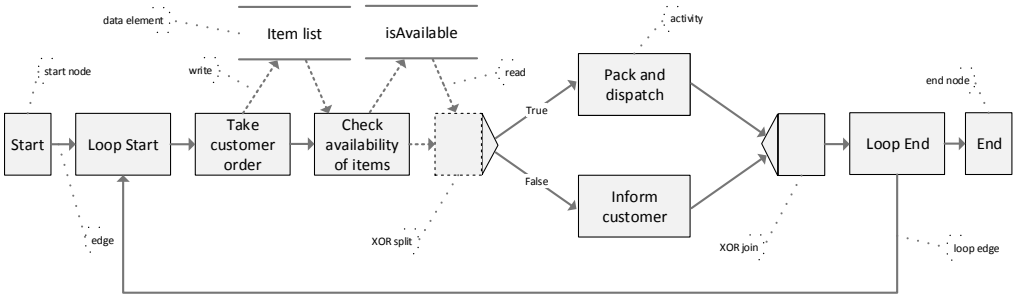

Figure 2.2: ADEPT2 example for an ordering process.

The further sections will cover the basic workflow constructs of the ADEPT2 PAIS. Figure 2.2 shows a simple ordering process using the ADEPT2 notation. This example process takes orders from a customer, checks the availability of the requested items and dispatches them if possible. The example consists of four activities (*Take customer order*, *Check availability of items*, *Pack and dispatch* and *Inform customer*). Every activity defines a task or an entire nested process (called a subprocess) and has different states defining its behaviour (see section 2.2.1). Solid edges represent the control flow and dashed edges the data flow. Through data edges an activity can read or write data elements like the *Item list* in figure 2.2. The functionality of data elements in ADEPT2 will be discussed in more detail in section 2.2.2. Branches operate with the boolean logic 'AND' or 'XOR'. They decide which activity or activity sequence is executed next (see section 2.2.3). Finally, loops are introduced in section 2.2.4. They define parts of the process, which should be executed more than once during one process instance, depending on data processed within the instance.

With AristaFlow a software environment for a process-aware information system is available which is based on the ADEPT workflow technology.

2.2.1 Activities

An activity represents a simple *task* or an entire *subprocess*. A simple task can involve sending or receiving an email, opening or closing a form, reading or writing data, etcetera. Figure 2.2 describes a process for a mail order company by the use of the four activities *Take customer order*, *Check availability of items*, *Pack and dispatch* and *Inform customer*. After the arrival of a customer order (*Take customer order*) the availability of the items is

checked (*Check availability of items*). Therefore the item list is extracted from the order. If all items are available, they will be packed and dispatched to the customer (*Pack and dispatch)*, otherwise the customer is informed that the order can not be handled (*Inform customer*). Depending on the companies size, in this example the activity *Pack and dispatch* could be an entire subprocess for the logistics department. For the simplicity it is handled as a simple task in this case.

Each activity has a list of preconditions. Only if all preconditions are fulfilled the activity is ready for execution. Since there may be more than one activity ready for execution at a time and because of the need to provide an overview of the process progress, each activity has a state. There are the four distinguishable super states *waiting*, *running*, *terminated* and *skipped*, which define the general purpose of a set of sub states. The super state waiting holds the sub states of an activity which waits to be executed. If all preconditions are met, an activity changes its state from the sub state *not activated* to the sub state *activated*. *Not activated* represents the initial state of all activities. An authorised user can select one of the activated activities which changes thus from activated to *selected*. After being selected an activity is automatically assigned to a sub state of the super state running. A running activity performs its task by setting it to the sub state *started*. If there are any interruptions during the task the state changes to *suspended*. If the task of an activity is finished or aborted the sub states of the super state terminated are used for the activity. For finishing a task successfully the sub state *completed* is assigned, for aborting *failed*. The last super state contains a special case for conditional branches. If a path is not selected, the attached activities are unable to run. This is marked by the state skipped.

Dependencies between two activities are defined by the *control flow*. The control flow is presented as unilateral control flow edges which link activities, branches and loops. A link between such two constructs defines a precedence relation. In figure 2.2 the activity *Take customer order* has to be terminated before the activity *Check availability of items* is activated.

2.2.2 Data Elements

Data elements are global process variables, which enable the data exchange between activities. The only two data elements in the ordering process example of figure 2.2 are *Item list* and *isAvailable*. Data elements are accessed via *data flow* edges – in contrast to

the control flow edges presented as dashed links. If an activity writes a data element, a unilateral data flow edge leads from the activity to the data element describing that the data element is now connected to the activity. Alternatively, reading a data element is defined by a unilateral data flow edge leading from the data element to the activity. For an activity each connection to a data element is internal modelled as exactly one input or output parameter. For example the activity *Check availability of items* holds one output parameter for the data element *isAvailable* and one input parameter for the data element *Item list*. Besides primitive data elements, like string, integer, double and boolean, complex data elements (e.g., arrays) are supported and even user-defined constructs (e.g., SQL ResultSets) are possible. Regarding the ordering process example (see figure 2.2), the list of items could be presented as a string (comma-separated item identifiers), as array or as an entirely new construct. Whereas *isAvailable* presents a simple boolean.

Constraints ensure the correctness of the data flow by for example checking if a data element is written before it is read – except for a data element which is marked as *optional*. Furthermore, versions of the data elements are managed by ADEPT2. Every write access adds a new version to the data element. This can be necessary in many cases, especially when parallel branches work on the same data elements requiring the same data start values.

2.2.3 Branches

Branches represent decisions during the runtime of a process instance. Following the boolean logic, 'XOR'- and 'AND'-splits cover different use cases for branches. 'XOR'-splits decide which of the following paths should be the only one to be executed. The activities of the non-selected paths are transferred to the status skipped (as described in section 2.2.1). For such a decision a data element (see section 2.2.2) is read and based on the dimension, value ranges are mapped to the following paths. In the ordering process example the simple boolean value from *isAvailable* is read to determine which path for the further execution is chosen. If all items are available the value `true` is written to the data element *isAvailable*. `True` is mapped to the upper path of the 'XOR'-branch and activates the activity *Pack and dispatch*, whereas `false` activates the activity *Inform customer*.

'AND'-splits divide the path into multiple paths. Activities of all paths are executed independently. If the 'XOR'-branch from the ordering process example would be replaced by an 'AND'-branch, the activity *Pack and dispatch* as well as *Inform customer* are executed.

2.2.4 Loops

Although processes can be repeated several times, loops are of great concern. In many cases only parts of a process have to be repeated and the number of iterations is determined during runtime. As already described in section 2.2, a loop is represented by exactly one start and one end node. If the end node is activated, the condition for the next loop iteration is evaluated. Regarding the ordering process example a loop is useful if a customer has several orders. Each order will be processed successively. Therefore the fragment of the process, which is nested in the loop construct, has to be passed for each order. For a more precise example consider a customer with three orders. The first two orders contain only available items and are therefore packed and dispatched. The third one misses an item and hence the customer is informed that the order could not be dispatched probably. Although the same fragment is run over again, the execution order can be different.

2.3 Eclipse Rich Client Platform

The *Eclipse Rich Client Platform* (RCP) is a Java[1]-based development tool for standalone desktop applications. It provides a framework, which can be freely enriched by functionality to build a client-side software system. Because of the great basic functionality the term *Rich Client Platform* was established.

The Eclipse RCP is based on an *OSGi* runtime environment to guarantee modularity and expandability [24]. Every software module is encapsulated in a *bundle* or in RCP-terms a *plugin*. An OSGi runtime manages these plugins through their entire life cycle including resolving dependencies between plugins. To add supplementary functionality, new plugins are developed and linked to the OSGi runtime. The Eclipse IDE[2] for example is a fully plugin-based application with a great number of plugins. RCP was originally extracted from

[1] Java is an object-oriented programming language developed by Sun Microsystems. [34]
[2] The Eclipse integrated development environment is a collection of tools for computer programmers to develop software.

the Eclipse IDE and is still strongly connected. With every major Eclipse IDE release comes a RCP release [18]. The RCP content covered in this paper is related to the RCP 4 version.

In the following sections an overview of important basic RCP components is provided (see section 2.3.1). Based on the plugin structure a set of basic functionality is available reaching from user interface components over structural models to back-end functionality. In section 2.3.2 the workbench component is described in more detail. With the workbench and its elements the user interface for the front-end user communication is rendered. The logical model, the latter is based on, is called the application model and is outlined in section 2.3.3. The application model is a framework to generate a workbench dynamically and allow changes to the workbench model even at runtime.

2.3.1 RCP Components

Components bundle software functionality in frameworks. In RCP every component, except the OSGi runtime, is build on plugins. Most of the components consist of a very large number of plugins.

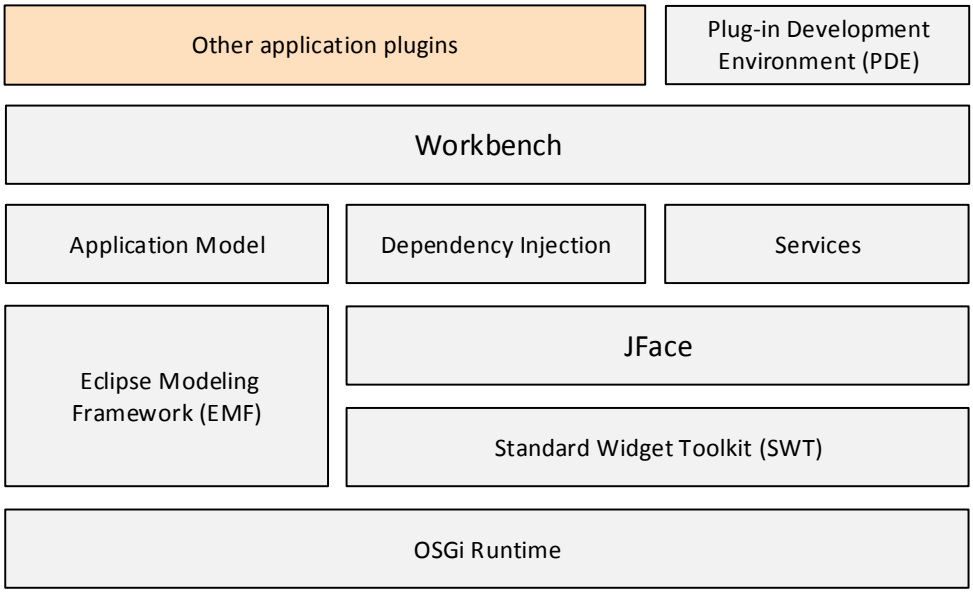

Figure 2.3: Overview of important RCP components.

To separate these amounts of plugins, *features* allow a logical separation by simply bundling a list of plugins and other features. The OSGi runtime breaks these features down into plugins again and starts the *life cycle* management for each plugin. The life cycle installs the required plugin and resolves its dependencies. There are two types of dependencies: The import dependencies define which other plugins are necessary for a plugin to run correctly. Export dependencies in contrast describe the functionality a plugin offers for other plugins – in RCP called *extension point*. After its dependencies are resolved, the plugin is started and according to its activation policy deferred or directly activated. During its life cycle the plugin can be stopped and started arbitrary. After the application is terminated it is uninstalled. [33, 35]

In figure 2.3 the most important RCP components are shown. The access point for user developed plugins is highlighted in orange. For the development of such plugins the *plug-in development environment* (PDE) offer support. These tools are organised in three parts: The *PDE UI* part leads the user through several development and deploying steps with the help of wizards, launchers and editors. With the *PDE API Tools* the documentation and maintenance of plugins are supported (especially regarding the JavaDoc). At last the *PDE Build* eases the build process of an RCP application by fetching the required files and properties and combine them in a finished and runnable product. [8]

The *Standard Widget Toolkit* (SWT) and the *JFace* toolkit create a uniform user interface throughout different operating systems by using native user interface widgets. SWT represents a low-level abstraction layer for the user-interface facilities of the operation system. It offers the essential functionality to display user interface elements, like buttons, text fields, canvas, etcetera. JFace appends supplementary registries for images and fonts and provides frameworks for dialogues, preferences and wizards. Furthermore, the handling with SWT is simplified with JFace. Concepts as *viewer*s let user interface widgets be supplied by data from model objects. These model objects are called *adapter*s and offer especially advantages for more complex data structures like lists, tables or trees. [12]

The *Dependency Injection* belongs to the programming model of RCP 4. If an object references another object via dependency injection, the reference is not resolved until runtime. Therefore the referenced object is stored at a central place (in RCP the *IEclipseContext* class) and injected at the time the referencing object is instantiated. If the referenced object is not available (e.g., because it was not created or stored yet) an error occurs and the

instantiation of the object fails. In RCP the injected objects are tracked and if they change they can be automatically re-injected. [35]

Another important component of RCP are *services*. Services offer controls over several RCP functionality. Via the *IThemeManager* service for example the style of the RCP application can be changed at runtime. The service is accessed by an IThemeManager object injected by dependency injection. Further important services are:

- EModelService (programmatic access to the application model)
- ESelectionService (programmatic access to the active selections)
- EPartService (programmatic access to parts)
- ECommandService (programmatic access to commands)
- EHandlerService (programmatic access to handler)

Each of these services refer to the application model and allow different elements of it to be accessed and manipulated [8]. The elements of the application model and their construction in a workbench are described in the sections 2.3.2 and 2.3.3 in more detail.

2.3.2 RCP Workbench

The *RCP workbench* itself only represents an empty shell of the user front-end. To create a fully equipped graphical user interface this empty shell is enriched by several RCP elements. The main control is the workbench *window*, which represents the visual area for the containing user interface elements of an RCP application. Windows have several properties defining their operations (minimising, maximising and closing), settings (visible, initial size, etcetera) and behaviour (modal, resizeable, etcetera). Windows contain *menu*s, *tool bar*s or *tool control*s and *part*s as shown in figure 2.4.

A menu hosts a set of submenus and simple entries called *item*s. An item holds functionality, which is executed whenever it is selected. This item functionality is declared directly in a class or through *command*s. With commands a declarative description of the functionality is created independently from its implementation. For the concrete implementation a *handler*, which holds the uniform resource identifier (URI) of the executable class, is linked to the command. [24] The benefit of commands and handler lays in a flexible access to the implementation of functionality. Several commands can point to the same handler and

several handler can be linked to the same class. Commands and handler are accessed programmatically at runtime through the services *ECommandService* and *EHandlerService* as mentioned in section 2.3.1.

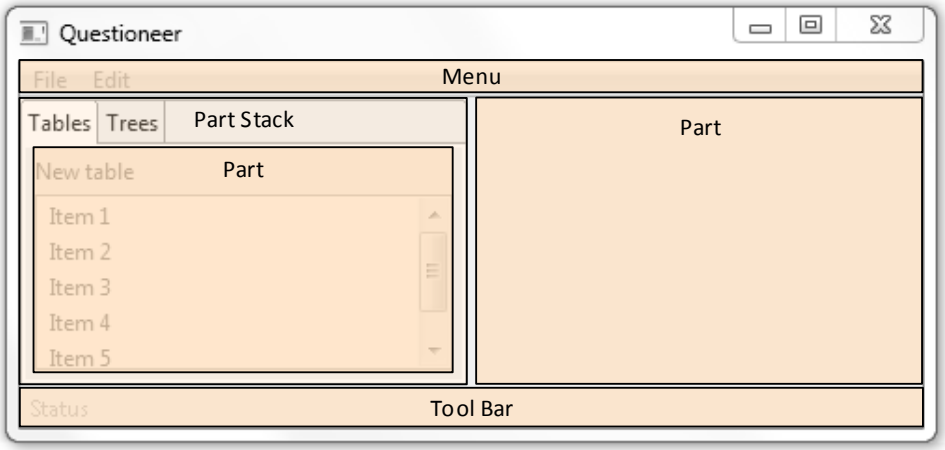

Figure 2.4: An RCP workbench example including a window with a menu, tool bar, part stack and three parts.

Tool bars are similar to menus, but avoid nested items. A tool bar displays items as a blank spot, an image, a text, or an image with text, depending on the information available. To avoid this predefined behaviour, a tool bar can be replaced or extended by a tool control. With a tool control a simple canvas is provided, which is filled arbitrary (to display a status bar for example).

The most important controls are *part*s. They constitute a plain area, which is filled by user-created forms and elements. In figure 2.4 the part on the right has no content and part *Tables* holds a table with five items. The content of part *Trees* is unknown. Parts can be arranged in different ways. A *part stack* holds several parts as tabs and always displays only the selected one. The selection is tracked and accessed through the *ESelectionService* service. With a *PartSashContainer* several parts and part stacks are placed next to each other in a restricted area. Parts are accessible through the *EPartService* at runtime. The entire set of parts and part containers displayed in a window is called a *perspective*. They therefore define views and are not visible for the user. Because the number of perspectives is not limited, *perspective stack*s keep track of all available perspectives.

All these workbench elements have a predefined declarative structure and are placed automatically in the workbench. To add, delete or change a workbench element, RCP allows, via a graphical editor, access to the workbench model holding the workbench elements. It is called the application model and is described in the next section.

2.3.3 RCP Application Model

The application model is based on the *Eclipse Modeling Framework* (EMF). EMF is a framework to build applications grounded on a structured data model. This data model is called the meta model and is specified in the XML[3] Metadata Interchange format (XMI). Based on this specification, source code is generated and displayed using JFace [8]. In other words the abstract model is transformed into a concrete implementation.

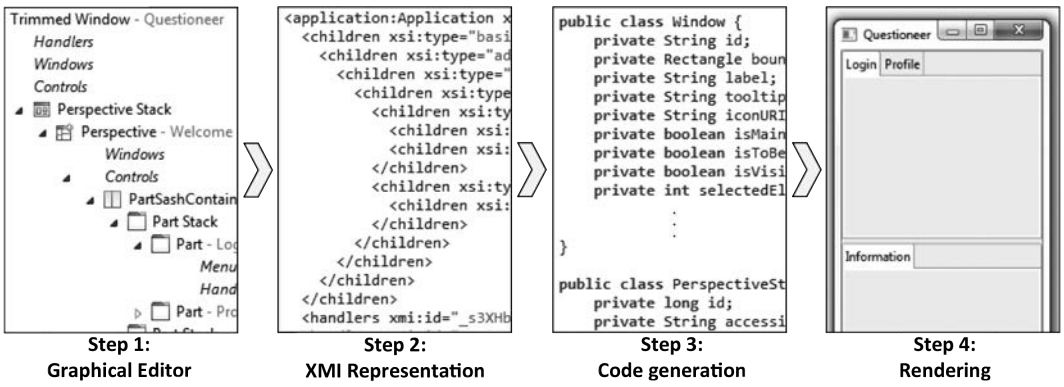

Figure 2.5: The internal steps for building a workbench.

An example is the RCP *application model*. The application model allows a user to generate a fully-equipped workbench with only the help of a declarative graphical editor. With the graphical editor an abstraction layer for the EMF XMI notation is available. Based on this user generated model the workbench is rendered. As figure 2.5 shows, the graphical editor is only the first step, but the only one the user has to handle. Within a tree-structure windows and related perspectives are defined, parts are created and arranged, and properties are set. Every time the user makes such a change the entire presentation is automatically transformed into the internal EMF XMI format in the background (see step two in figure 2.5). At startup the meta model is extracted from the XMI file and classes representing the

[3]The Extensible Markup Language is a declarative language to express hierarchical structured data.

workbench elements are generated by the RCP runtime environment (shown in step three). In step four the generated code is executed and the workbench is displayed as defined in step one. If extensive changes are made to the application model during runtime, steps two to four are repeated. With the help of the service *EModelService*, workbench elements can be accessed and therefore for example added, deleted and replaced. [8]

Not only the workbench user interface elements itself, but also commands and handler, are created in the application model. The definition of dependencies between commands, handler and classes follows a uniform resource identifier pattern, which adds a supplementary separation between classes and all other resources. To refer to a class the prefix `bundleclass://` is used, for resources, like images or fonts the prefix `platform://`. Furthermore, *binding*s and *addon*s are declared in the application model. Bindings define which key combinations have a special function in a defined context. Addons are global objects which are managed by the dependency injection framework. With addons additions and changes to the application are possible in a very early stage. Because addons are called before the workbench is rendered changes to the user interface are feasible. [35]

3 Related Work

Questionnaires are a rather inexpensive method to gather a large amount of data in short time. The analysis of this data is mostly done automatically by statistical software, like R [22], SAS [26] and SPSS [11]. Even for paper-based questionnaires these tools can be used by transforming the written answers into a digital representation. Regarding the used measuring scales (see section 2.1.2), the statistical software is able to provide more or less complex methods to automatically extract information from the answers given.

Besides the automation of the data analysis, the procedure of data collection itself can be automated as well. Based on a set of required elements (as described in section 2.1.3) a questionnaire can be constructed (see section 2.1.4). Software applications, that are able to create such a questionnaire, are called *questionnaire configurator*s. In the sections 3.1 to 3.3 an excerpt from the large pool of questionnaire configurators is described ordered by its application type. The chosen excerpt focuses on providing an overview of as heterogeneous as possible approaches. In section 3.1 the web-based applications *Generic Questionnaire System* and *LimeSurvey* are depicted, in section 3.2 the desktop applications *SurveyGold* and *StatPac* and in section 3.3 the mobile application *Pollcode*. The summary in section 3.4 aggregates the information from section 3.1 to 3.3 in a brief overview. For descriptions of further configurator environments refer to [13] and [32].

3.1 Web-based Applications

Web-based questionnaire configurators constitute the largest number of configurator applications. With a thin client application (e.g., a browser) remote server functionality is accessed and executed. Because this functionality is based on one or more central servers, thin clients require minimum hardware resources. Moreover, the scalability depends on the number of accessible servers.

The *Generic Questionnaire System* (GQS) is one of these web-based questionnaire applications. It is the predecessor of the configurator prototype originated by this thesis. GQS is a system consisting of the three components *configurator*, *middleware* and *client* to cover the entire questionnaire life cycle. With the configurator component questionnaires are constructed, edited and deleted [28]. To store the created questionnaires centrally, the middleware component offers interfaces, which can be accessed via the Simple Object Access Protocol (SOAP) [13, 36]. A stored questionnaire can be downloaded from the middleware and executed via the client component, which is available as web-based and mobile applications [17].

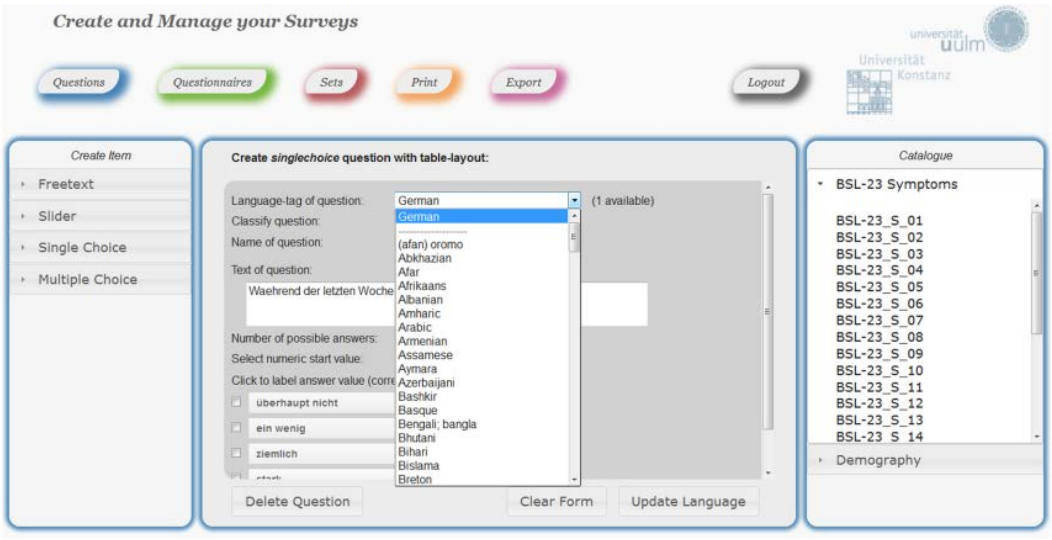

Figure 3.1: The QuestionSys questionnaire configurator (see [28]).

The configurator of GQS has a control concept focusing on Drag and Drop events. The center of the application has a large drop area for several tasks (as shown in figure 3.1), e.g., the content of a questionnaire is displayed by dragging it into the drop area. To create a questionnaire, questions have to be defined and assigned to a group. With *Freetext*, *Slider*, *Single Choice* and *Multiple Choice* four question types are provided. Single and multiple choice questions can be displayed vertical or horizontal, restricting the answer items to 15 or six respectively. At the creation of a new question an intern template function automatically preallocates redundant parts and properties by using the content of the previous question. The question groups can be complemented by text elements and page breaks. With text elements intermediate text fragments are created, whereas page breaks define

the point where the recent page ends and a new one is started. If several questionnaires should be executed successively, they can be bundled in sets. The configurator provides multi-language support and an export method for statistical tools.

LimeSurvey [6] is a non-profit questionnaire managing application. Based on an open source approach an initial release was published in 2003. Similar to the GQS configurator LimeSurvey defines a static set of question types, which can be adjusted individually by the user. Besides single and multiple choice, free text and slider questions, LimeSurvey offers a variety of additional types including e.g., question arrays, ranking questions and file uploads. Questions are also bundled in groups, which are ordered via Drag and Drop to design the survey. Different is the handling of text elements. LimeSurvey reduces text elements to a description text for each question group and three global text fragments defining the description, the welcome and end text for a survey. Page breaks can not be defined explicitly, but are automatically inserted after every question group.

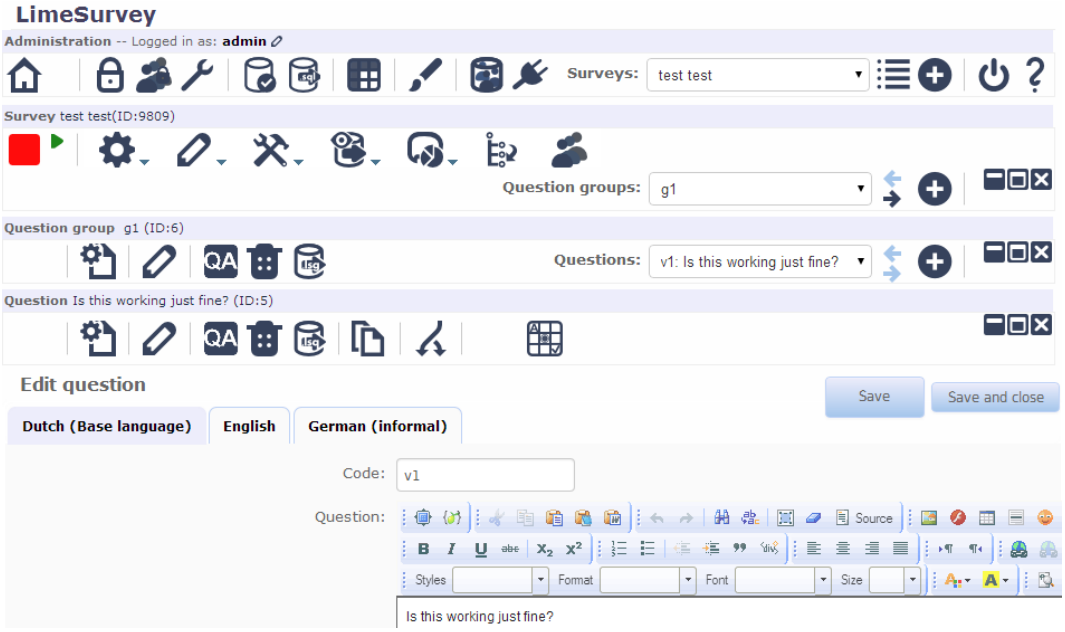

Figure 3.2: The LimeSurvey questionnaire configurator.

The concept to control the application focuses on a hierarchical tool bar structure. As shown in figure 3.2, the topmost tool bar holds the administration tools (e.g., user management, database export and backup, plugin framework, style editor) where a survey can be created or chosen from a list. If a survey is selected, a survey tool bar is displayed under the

administration tool bar offering functionality for the global survey settings (e.g., ordering the question groups to a questionnaire or executing a test run). Based on the same principle the question group and question tool bars are accessed, displaying four tool bars with over 50 tools. In contrast to the GQS configurator, filtering questions, integrated statistical tools for data analysis, survey access functionality (e.g., access token generator or sending invitation emails) and automated error checks are available among other supplementary functionality.

Although both web-based configurators offer a large number of questionnaire elements, several restrictions reduce their integration options in a questionnaire. The GQS configurator offers text elements, but only between question groups, whereas LimeSurvey does not offer free text elements at all. Filtering questions are not provided by GQS, while LimeSurvey lacks explicit page breaks. Furthermore, both configurators require a permanent connection to the server. Every change is stored by manipulating the content of a remote database. As a result, working offline is not possible.

3.2 Desktop Applications

In contrast to web-based applications, desktop applications are designed for *rich clients*. The latter provide enough hardware resources to calculate and display the entire application functionality. Therefore a connection to a remote server is possible, but not required.

With *SurveyGold* [10] an offline configurator environment is available, which offers additional online features for publishing or sending surveys and receiving survey results for subsequent analysis. The application is based on a tree structure, which represents the hierarchical order of every questionnaire (as shown in figure 3.3). Each leaf defines a section, which bundles a set of questions. SurveyGold distinguishes between single and multiple choice question types and offers filter functionality for the former. With filter, results are mapped to sections (which will be executed immediately after the filter question is answered) or directly to the end of the questionnaire. The specific order of questions, besides an introduction text and properties regarding the graphical representation, is defined in *section*s. A set of sections is assigned to exactly one survey. A survey belongs to a user-defined folder and determines the list of respondents (e.g., for an email survey) and which personal user data should be collected. With an additional property page breaks

are automatically inserted after every section. Through export functions a survey can be transformed into a plain text-, a Microsoft Word- or a web-format.

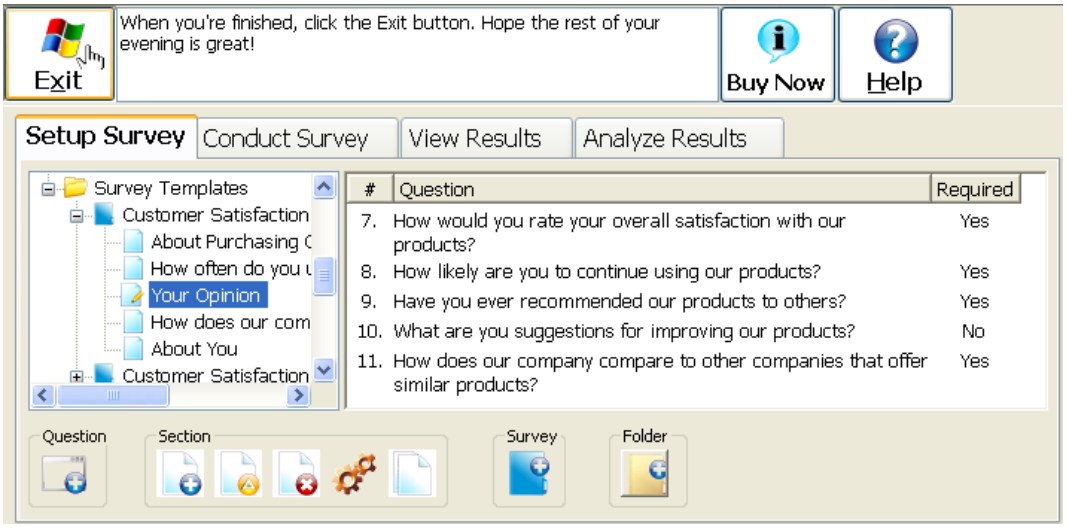

Figure 3.3: The SurveyGold questionnaire configurator.

SurveyGold is a rather simple and limited configurator and stays therefore in direct contrast to *StatPac* [30]. With this application the style of a survey is strictly separated from its content.

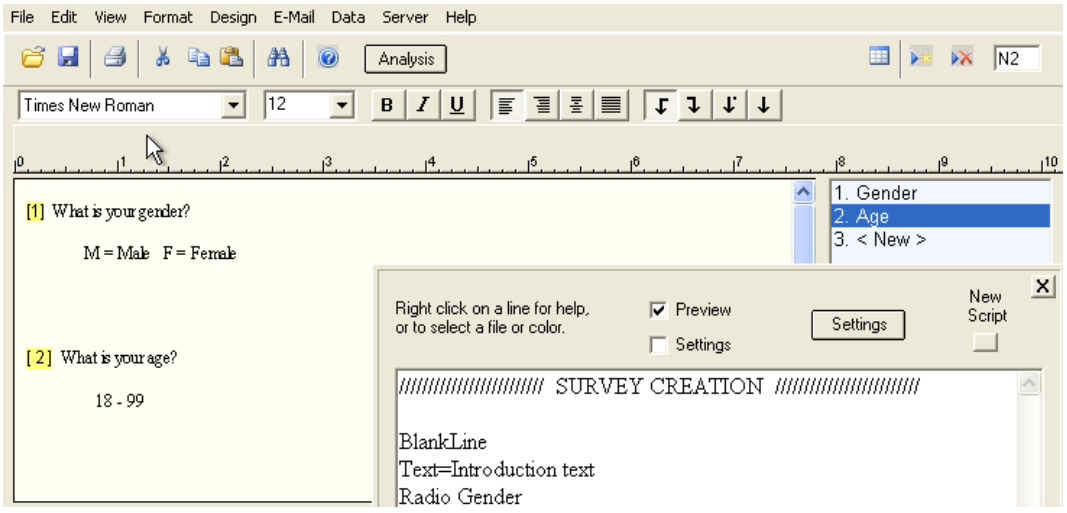

Figure 3.4: The StatPac questionnaire configurator.

A so called *codebook* holds all questions with their answer options (in figure 3.4 a codebook overview is shown on the right side with the questions *Gender* and *Age*). The codebook can be created by manually adding the questions, by using an existing file or the internal StatPac editor (represented as a text editor on the left side in figure 3.4). Consider for example a Microsoft Word document, which contains a fully styled questionnaire. If this document is saved as rich text file [20], it can be viewed in the graphical StatPac editor. By highlighting a text fragment in the editor, it can be linked to an empty question in the codebook. Based on a given codebook StatPac can generate a questionnaire for print media or web. An internet questionnaire requires a user script, defining its structure and content. To add the question *Gender* for example to an internet survey, the command *'Radio Gender'* is defined in the script, whereby the fragment *'Radio'* indicates that radio buttons are used to provide the answers available. An intermediate text fragment is added by the command '*Text=*' following the required text.

StatPac, as well as SurveyGold, uses several file types to save and load the user generated content and therefore provides independent configurator environments. Additional online features are accessible and mostly correlate with internet surveys.

3.3 Mobile Applications

Mobile applications are designed primarily for devices with small screens and limited hardware resources, like smartphones or tablet pcs. The control concept for these devices is mostly based on touch gestures including the input of text using an on-screen keyboard.

Although these restrictions lay in contrast to the rich functionality, which should be provided by a questionnaire configurator, several approaches for creating a questionnaire with mobile devices are available. With *Pollcode* [4] an approach besides the previous presented configurators is available. Pollcode does not provide an environment for the construction of a survey, it just focuses on the generation of a single question instead. A single or multiple choice question with up to 30 answer options is predefined and configured regarding its style. The result is the source code of an HTML [38] form, which can be embedded in a website. Initially in this generated form the answers are forwarded to a pollcode server, which stores and displays the results. This behaviour can be freely adjusted in the generated source code.

The idea to simplify the required functionality for a specific use case of a configurator as far as possible, matches the limitations for a mobile application. In a wider sense, the reduction of functionality to a necessary minimum is beneficial for low hardware resources and small displays. Consider for example a mobile application which is implemented to handle only a single static questionnaire. Only occurring elements are handled, style changes are limited and available space is hard-wired to the view of every single element. Even device-specific optimisations are possible. For example in the medical environment such applications exist [1].

Although specialised configurators fulfil their goal, extensibility and flexibility are not ensured. Moreover, the approach to create an application for every single questionnaire is a very time-consuming process, which comes at great cost.

3.4 Summary

In this chapter, five different configurator approaches (GQS, LimeSurvey, SurveyGold, StatPac and Pollcode) are described varying from a minimal question generator to entire questionnaire life cycle environments. Every approach offers different functionality regarding the construction, the persistence and the element variety of a survey. In nearly every approach several elements – reaching from simple text fragments to complex filtering questions – are available, but limited to specific use cases or conditions. These restrictions are in many cases related to a strict logical questionnaire model, which does not support any deviations. Consider for example the GQS configurator in section 3.1, which allows text fragments only between question groups. If every question requires a description and introduction text, one group has to be created for every single question – causing an overhead of several unnecessary question groups.

Furthermore, none of these approaches considered a questionnaire as a sequence of activities and therefore as a process. If a questionnaire is executed, the respondent is asked a number of questions (enriched with text and other elements) in a specific order. If this order can be mapped to a PAIS (as described in section 2.2), several tasks can be performed automatically and additional functionality can enrich the questionnaire's creation and execution. In chapter 4 such a technical mapping of a questionnaire to a PAIS is discussed.

4 Process-aware Questionnaires

In section 2.1 the questionnaire is introduced as a social research method for data collection. With most research focusing on the content of a questionnaire or the order of its elements, like the ideal wording or the optimal placement of complex and sensitive questions, the realisation of a flexible and generic questionnaire configurator is not immediately possible. An approach is required, which maps every possible questionnaire to a universal model – or in other words, a generic questionnaire model. If this generic model is mapped to a PAIS, the execution of a questionnaire could be automated and therefore simplified (based on the idea of [9] and the design approach of [32]).

The goal of this chapter is to define such a generic questionnaire model to map it to a PAIS. This approach focuses on a maximisation of PAIS support functionality for questionnaires on the one hand, and on the minimisation of restrictions regarding the structure and content of questionnaires on the other hand. In section 4.2 a generic questionnaire model is developed and presented. This model is based on a three-layer architecture defining a layer for the questionnaire, its questionnaire pages and questionnaire elements. In section 4.3 the generic model is mapped to a PAIS by linking activities to different layers of the model. The result is defined as a process-aware questionnaire and depending on the requirements on the PAIS environment, as described in the next section 4.1.

4.1 Requirements on the PAIS environment

A PAIS is an environment for the entire life cycle of processes. Questionnaires can be handled – as described in more detail in section 4.3 – as a special type of process and therefore benefit from an expressive PAIS. As a result, the focus of this chapter lays on the integration of an questionnaire configurator in a PAIS, rather than in the development of a new process life cycle environment, which is particularly designed to fulfil the needs of questionnaires. There are two fundamental approaches to combine a PAIS with questionnaire functionality.

Firstly, an integrated approach, which extends an existing PAIS application by adding supplementary graphical user interfaces and background functionality through a manipulation of the source code of a PAIS application. Therefore an implementation of the PAIS has to be available and accessible. This binds the questionnaire configurator directly to the further development of the PAIS and limits the technical realisation of a configurator. In contrast, the full functionality is accessible and avoids redundant source code fragments. Consider for example a PAIS application, which is based on RCP (as described in section 2.3) and displays the configurator as a single RCP part of the user interface. Although the part itself can be filled arbitrary (regarding the underlying runtime environment), all comprehensive functionality is predefined by the PAIS application through RCP. Any changes to these functionality would interfere with existing components and thereby endanger the stability of the entire system. The flexibility is therefore decreased, but the available functionality increased. Furthermore, the complexity of the PAIS application would be raised and resulting from that discourage the target group. Because the target group covers mostly professions without an affinity to PCs (e.g., sociologists, psychologists, doctors, etcetera), no knowledge concerning processes may be assumed.

The second approach is based on a standalone questionnaire configurator, which accesses PAIS functionality through predefined interfaces. Through this programmatic interface access the complexity of the PAIS can be hidden or reduced to a minimum. A new developed graphical user interface for example could filter PAIS functionality, which is not required or present it in a way which is more suitable for the target group. Resulting from that, no PAIS functionality – besides from those provided by interfaces – is reusable and therefore has to be explicitly created.

Both approaches require a PAIS, which offers activities, data elements and XOR-branches (with their related control and data flow), to map the technical questionnaire model, which is described in section 4.2.

4.2 A Technical View on Generic Questionnaires

Questionnaires consist of a variety of elements reaching from simple information texts to complex question constructs (as described in section 2.1.3). As seen in the examples from chapter 3, this set of elements is under steady development and may be extended by new upcoming needs of the target groups. The GQS configurator for example has four question

types, whereas LimeSurvey defines over 30. With logical elements, like page breaks or filter, the structure of elements and their dependencies are controlled. This logical elements offer comprehensive functionality for the questionnaire elements. Page breaks define which elements are displayed at once, whereas filter determine the sequence of upcoming pages. Finally, specific global settings influence the behaviour and functionality of a questionnaire e.g., by defining its interview mode (performed self-contained or by an interviewer), required languages, authorised users, etcetera.

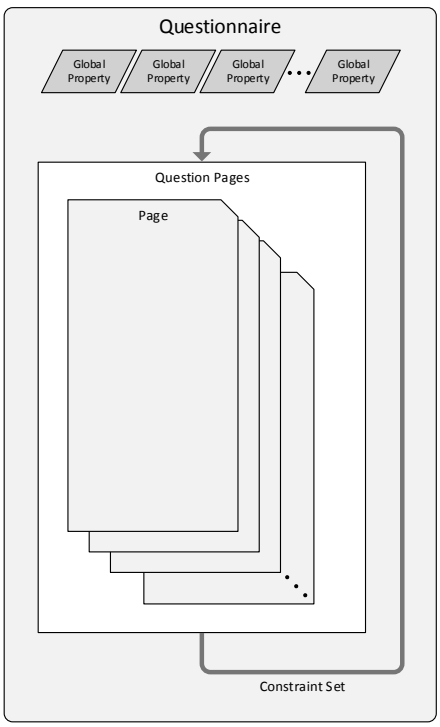

Figure 4.1: A schematic representation of a questionnaire as a container for question pages (questionnaire layer).

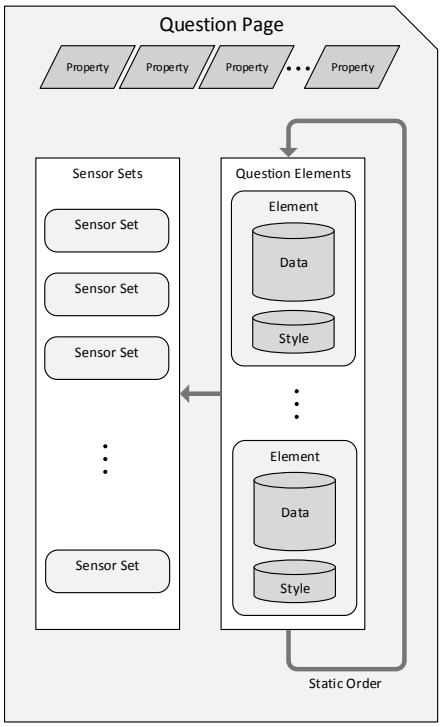

Figure 4.2: A schematic representation of a questionnaire page as a container for question elements (page layer).

To transform these requirements into a generic model, a three-layer architecture is defined, which consists of a questionnaire layer, a page layer and an element layer. The *questionnaire layer* considers the questionnaire as a container. As shown in figure 4.1, every questionnaire hosts a set of question pages and global properties. The logical page break construct is expressed as a question page in form of a container for elements, instead of an

exclusive questionnaire element – equal to a question or a text element. With this abstraction layer supplementary interfaces for the control flow (represented as an arrow in figure 4.1) of pages are provided. Every control flow edge is enriched by an optional constraint set, which influences – depending on its evaluation – the flow of upcoming pages. For example a questionnaire page about '*Smoking*' will be skipped, if the information that the respondent is a non-smoker is provided through a question on a previous page. To avoid a limitation of the filter functionality, the constraint set is not restricted in its number of constraints. Nevertheless, the occurrence of these constraints is important for the constraint set. To guarantee an always executable page sequence, constraints have to be evaluable as soon as the related page is displayed and therefore only allow constraints which are defined in previous, inevitably executed pages.

Beside the constraint set, global properties (defined in a questionnaire) represent settings regarding every page of a questionnaire. As an example consider the possibility to enable or disable the functionality of jumping one page back in a questionnaire. If it is globally enabled, each page is enriched by an additional button to return to the previous page. Other examples are the earlier in this section mentioned interview mode, the available language set and the list of authorised users.

The *page layer* defines every page as a container similar to the questionnaire container it is nested in (as shown in figure 4.2). Several properties describe comprehensive settings or behaviour regarding every element of the page. For example a property could define if the page is tagged as optional and therefore can be skipped by a respondent during the survey. As presented in figure 4.2, each element has a data and style content, which can be freely defined during the process of the questionnaire creation. Consider a text element, which holds an introduction text in different languages. These languages are defined by the global property of the questionnaire. A single text item could be saved as a text string and the entire element content as a list of text strings where each item represents the introduction text for a specific language. For the representation of the content additional style information are provided to define the layout, font, font size, color, etcetera. These style settings could be declared individually or follow the syntax of a standardised language for style sheets, like the cascading style sheets (CSS [37]). Because a page in a digital survey is not restricted in its size (as in contrast to pen-and-paper questionnaires with mostly the DIN A4 format), the page always fits the overall size of its content (or in other word its nested elements). The order of the elements is strictly determined during the

construction of the page resulting in an explicit control flow (shown as arrow in figure 4.2). Every element on a page has therefore an assigned index.

The collection of information is not only provided by the given answers of a respondent and therefore restricted to questions. The body signals from an respondent confronted with a particular situation (or question), like gestures, facial expressions and vital signs (e.g., pulse), deliver important supplementary information about the attitude to the related topic. A high blood pressure for example, may indicate tension and taken photos reveal emotions, which are hard to capture with questions. With this in mind, a sensor framework is inserted in the technical questionnaire model (see figure 4.2). Each sensor belongs to exactly one of two sensor types: The *discrete sensor*s collect only data at one point of time (like a photo camera or a thermometer device), whereas *continuous sensor*s measure data through a time interval (e.g., pulse measuring devices and video cameras). Every sensor is linked to the index of an element in a page. Via this index link the same element is able to refer to different sensors even if it is used many times in the same page. For all sensors linked to one index (i.e., element of a page), the term *sensor set* is used (in figure 4.2 shown on the left side next to the elements). To ensure data privacy, every sensor has an attribute defining the starting point of a sensor. The explicit starting point requires the respondent to allow the specific sensor to collect the required data, whereas an implicit starting point begins with the collection automatically. In the latter the respondent should be informed about the used sensors in advance of the survey.

4.3 Integration of Questionnaires in Process-aware Information Systems

With the questionnaire model from section 4.2, every questionnaire consists of three hierarchical abstraction layers defining different access points to the model. With the questionnaire layer the questionnaire in its entirely is described with its access to global properties and included question pages. In addition, the page layer determines the structure of pages with their properties and elements, whereas the details of the element construction are defined in the element layer. Because a PAIS offers performable tasks in form of activities, only one access point can be mapped to an activity natively at once. With three abstraction layers, three different approaches to map a questionnaire to a process are possible:

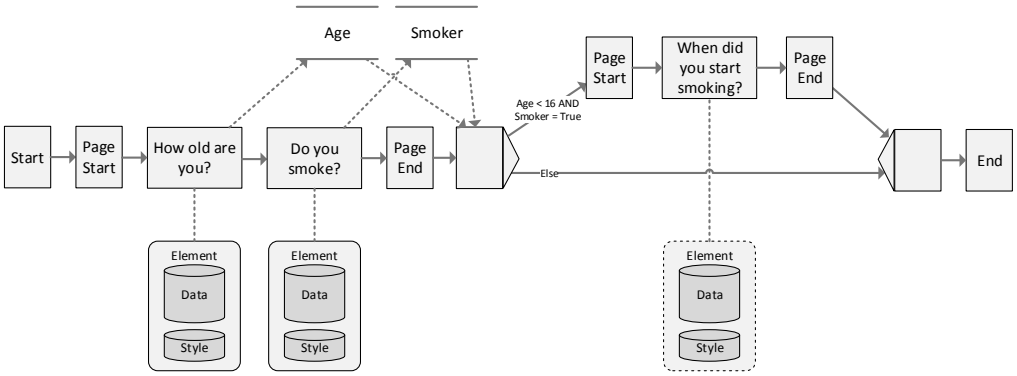

Figure 4.3: Example for a process-aware questionnaire at element level.

The mapping of the element layer to a PAIS is shown in figure 4.3. All questionnaire elements are directly represented as an activity. If an activity is executed, the involved element is displayed and able to store user input in data elements. The activity '*How old are you?*' in figure 4.3 holds a question element, which inquires the respondents age and saves the answer in the data element *Age*. Each data element can be used as a constraint for branches and therefore define which path is chosen. With the constraint set of the questionnaire model an analogy is available. This constraint set is therefore representable through a group of data elements restricted to a specific value range. In figure 4.3 the data elements *Age* and *Smoker* are combined to the constraint set `Age < 16 AND Smoker = TRUE` containing the two constraints `Age < 16` and `Smoker = TRUE`. Only if both constraints are fulfilled (expressed by the `AND` operator) the related path (in this case the upper path) is chosen and within the upcoming activities and their included elements. Because every element belongs to a question page, the page construct has to be mapped to the PAIS as well. One possibility to present pages lays in subprocesses. For every page a subprocess is defined, which holds a sequence of activities equal to the sequence of elements. Another possibility is provided by special activities determining the start and the end of a page (as shown in figure 4.3). Both approaches are possible, but contradict the meaning of a questionnaire page. A page holds a set of elements, which is displayed as a whole and allows the respondent to decide which elements are considered and manipulated first. This free element handling is not natively mappable to a PAIS. Further functionality would have to be added, like bidirectional control flow edges or iterative transactional sequences.

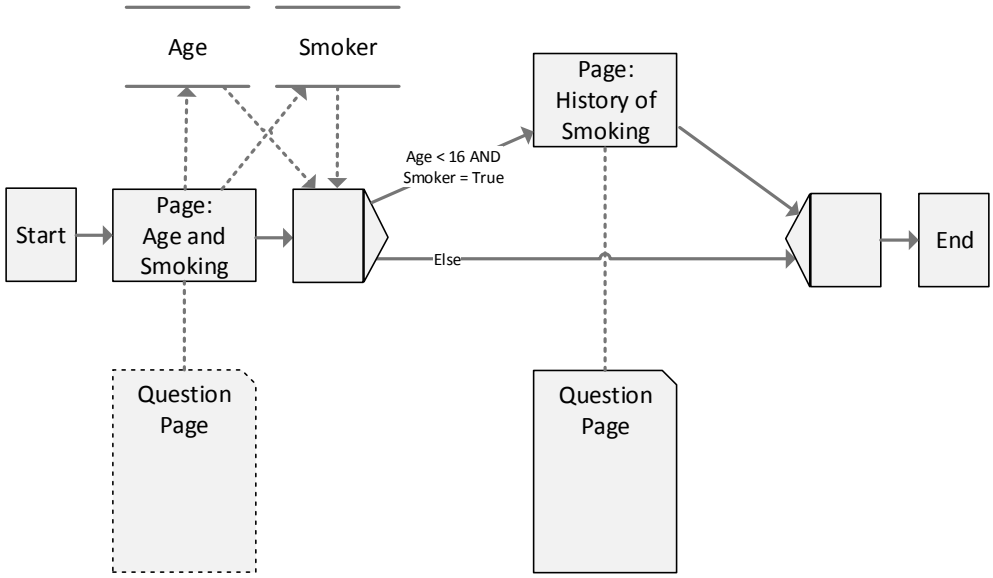

Figure 4.4: Example for a process-aware questionnaire at page level.

A more appropriate approach is based on the mapping of the page layer to a PAIS. As shown in figure 4.4 every activity represents a question page and therefore an entire set of elements in a specific order. Each element can write data elements by connecting the activity, which includes the superior page, with data elements. If the same element occurs several times in the same page, special naming conventions, like using a prefix, which contains the related index of a page, avoid confusion when executing the questionnaire. Furthermore, the activities exactly present the behaviour of an entire page. In a questionnaire always a page in its entirely is displayed and therefore manipulable. Only if the next (or previous) page is called the recent page is closed. Transferred to the activities, the manipulation of a page is a running task which is terminated by the respondent or in PAIS terms the authorised user. The sequence of pages is defined by branches and their constraint sets (identical to the element layer approach).

The third approach captures all questionnaire data in a single construct and links it to an activity. Therefore the entire process only consists of this single activity. Branches, data elements and further activities are expendable, because the page sequence is defined in the questionnaire construct and not with the use of the PAIS. As a result, the use of a PAIS is expendable in this approach.

4.3.1 Constraints within Questionnaires

All three approaches to map the generic questionnaire model to a PAIS require constraint sets to determine the sequence of displayed pages. The first two approaches transfer the constraint sets to the data flow of a PAIS, the third one defines it internally. To avoid the complexity of the explicit data flow, an automation of the data element creation, with its corresponding data edges, is necessary. Constraints are defined and internally transformed into a hidden PAIS data flow. Because only elements, which collect user input – in a questionnaire only elements of the type question – can influence the sequence of upcoming pages and therefore define analysable constraints, the four question categories – single choice, multiple choice, ranking and distribution (as seen in section 2.1.3) – must be created by fulfilling the following regular expression:

$$constraint \ := \ < answer \ items > \ (< answer \ data \ type >) \ (< result \ data \ type >)*$$

The construct $< answer \ items >$ selects the answer items (at least one) for which the same comparative operators are applied. The latter are defined via the expression $(< answer \ data \ type >)$, which is required for free user inputs and distinguishes between text strings and numbers. With numbers the comparative operators $<$, $\leq$, $=$, $\neq$, $\geq$, $>$ are available, whereby text strings focuses on search operators, like $CONTAINS$. The data type should be defined within the question's construction and therefore automatically resolved into the corresponding operators by the configurator. With the last fragment $(< result \ data \ type >)*$ additional manipulations of the answer items are determined. For example the question type 'ranking question' allows the user to arrange the single answer items to a preferred order. An user-defined order is an user input where every answer item is attached to a new index and therefore to a number. As an example consider a ranking question where a respondent should order the four choices *Teacher*, *Policeman*, *Politician* and *Reporter* by appreciation. A fifth choice is a free text field called *Other* allowing the respondent to enter and order a fifth profession. Consider an constraint to filter answers, which had chosen the fifth free text answer item. Furthermore, the entered profession should contain the word *Fireman* and take the first position in the order chosen by the respondent. The corresponding constraint is described by $\{\{5. \ Other\}\} \quad CONTAINS \ 'Fireman' \quad = 1$. For a single choice question where a respondent can only choose between static text strings only the $< answer \ items >$ fragment is required to form constraints. Furthermore, the $< answer \ items >$ expression only contains a single answer item – the item selected by the respondent.

All in all a constraint is reducible to a selection of answer items for which a set of operators define a value or a value range. The data types can therefore be created (inclusive reading and writing edges) fully automatically.

5 Architecture and Implementation

With a generic questionnaire model and an approach to integrate PAIS functionality into such questionnaires (as described in chapter 4), the basis for a standalone process-aware questionnaire configurator is created. Such a configurator requires additional features to handle the complexity of the supplementary PAIS support in a user-friendly way. Furthermore, the generic questionnaire model (as shown in section 4.2) defines several extension points for a flexible expandability of configurators. In this chapter an architecture is presented, which lists and structures required components including their dependencies, to pave the way for a concrete implementation of a process-aware questionnaire configurator. To validate this architecture an implemented prototype is developed as part of this thesis and grants an insight into a concrete implementation based on RCP.

The first step of implementing a configurator is the listing of requirements and their packaging into different software components. Section 5.1 provides an overview of necessary core components reaching from front-end to persistence modules. A closer look at the persistence components is given in section 5.2. To avoid restrictions by only providing either online or offline functionality, this persistence components describe a hybrid approach, which offers both functionalities. These components are structured in section 5.3 by using well-known design patterns, like the Presentation-Abstraction-Control (PAC) pattern, which divides the graphical representation from the data model on several hierarchical layers. In section 5.4 the Rich Client Platform (see section 2.3) is integrated into the architecture of the process-aware configurator to benefit from available functionality for common programming tasks. The results of this chapter are provided as an overview in section 5.5.

5.1 Core Components

The components of a process-aware questionnaire configurator constitute three categories: The *user interface component*s contain the visual representation and therefore focus on

the user communication with the application. In contrast, *background component*s determine the data processing independently from any user input. A special kind of background components are the *persistence component*s, which constitute the third category and are described in section 5.2. The latter aim at extending the storage of processed input data over the runtime of the application. In figure 5.1 an overview of all components is provided.

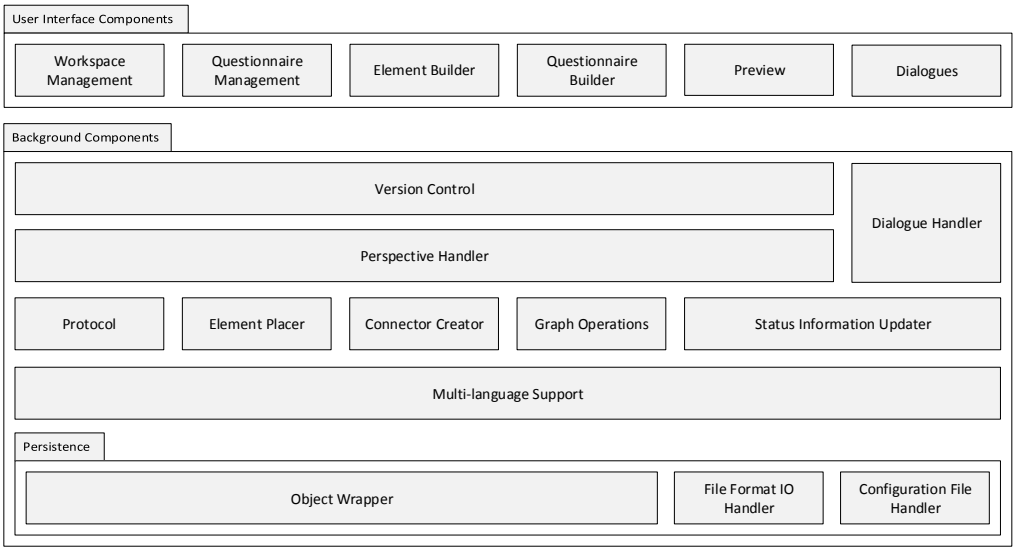

Figure 5.1: Core components of the architecture of a process-aware questionnaire configurator.

5.1.1 User Interface Components

As shown in figure 5.1 there are six user interface components: Workspace Management, Questionnaire Management, Element Builder, Questionnaire Builder, Preview and Dialogues. The *Workspace Management* component provides a graphical management layer for a set of questionnaires. This management layer is expressed via *workspace*s, which are logical containers for the categorisation and ordering of questionnaires. Each workspace has a title and a description offering the possibility to provide information about the included questionnaires, their purposes and aims. Furthermore, functionality to share a workspace or forward it to other users is provided by an explicit management of rights. Every user man-

ages for all self-created workspaces a set of authorised users and their regarding rights. Authorised users obtain the right to read or modify the content of a workspace, whereas the owner has the supplementary right to delete it and manage the set of authorised users.

The questionnaires within a workspace are handled by the *Questionnaire Management* component. Besides the finalised and executable questionnaires, corresponding questionnaire pages and elements, configuration settings, comprehensive properties and the supported languages are displayed. The latter determine the languages in which the finished questionnaire can be executed. Because the languages affect all corresponding questionnaire elements with a multi-language part, the use of a specific *input mask* is beneficial. Instead of manipulating every related element with a multi-language part, only the input mask changes when the supported languages of a questionnaire are varied. If a multi-language element is modified, a form, which includes the input mask, is generated, displayed and manipulatable through a user interface. With the input mask for languages navigation instruments are defined as well. Labels of buttons to navigate to the next (or previous) question page for example are required in the chosen language. This configuration of navigation instruments is defined in the configuration settings. At last the properties represent the global properties of the generic questionnaire model as seen in section 4.2.

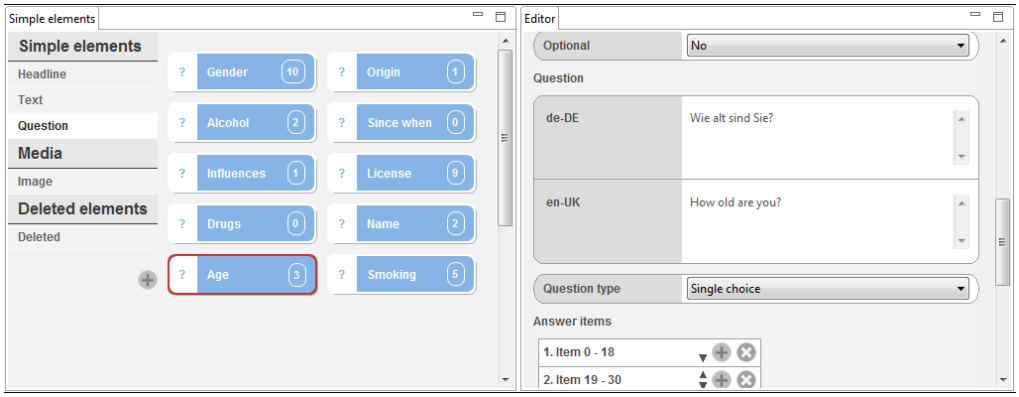

Figure 5.2: Excerpt of the element builder environment of a process-aware questionnaire configurator prototype.

With the *Element Builder* the life cycle of questionnaire pages and elements is handled. Figure 5.2 shows an excerpt of the Element Builder implemented in the configurator prototype. New pages and elements (in figure 5.2 represented with the blue elements on the left)

can be created, whereas existing ones can be modified or deleted (in figure 5.2 possible through the editor view on the right). Because pages represent a container for elements (in reference to the generic model in section 4.2) on the one hand, but share much functionality with elements on the other hand, the term *complex elements* is used. Furthermore, the abstraction of the page as a container offers new complex elements to be derived from a page. For an example consider a standardised instrument (described in section 2.1.1). Because the occurrence and order of specific elements is predefined in an instrument, it represents a page, which is restricted in its allowed modification functionality. Regular questionnaire elements in contrast are called *simple elements*. Simple elements contain text elements (e.g., headlines), questions (e.g., single choice questions) and multimedia content (e.g., images). With a template for simple as well as complex elements extension points are available for upcoming self-created elements.

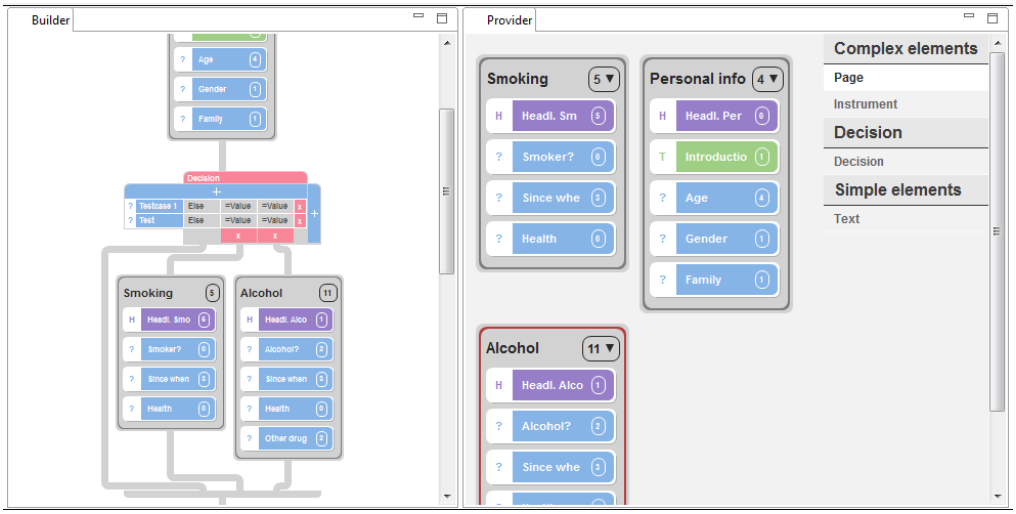

Figure 5.3: Excerpt of the modelling environment of a process-aware questionnaire configurator prototype.

The *Questionnaire Builder* component (as shown in figure 5.3) is the part of the configurator which uses PAIS functionality to create a process-aware questionnaire. A modelling approach (following the design approach of [32]) provides an environment where pages (and its derivations) are combined to an executable questionnaire. Constraint sets are definable via modelling constructs called *filter*s. In figure 5.3 the available pages for creating a questionnaire are shown on the right, whereas the modelling area for the questionnaire

is displayed on the left side. The resulting graph reflects the generic questionnaire model (from section 4.2) and is automatically transformed into the required PAIS format by using interfaces provided by the PAIS (as described in section 4.3).

With the *Preview component* an overview of a selected simple or complex element is provided. Especially questions with a large number of answer items or a text in several languages benefit from a fast preview (during the creation of the element) to check, if the created element is complete. Moreover, decisions regarding the style, like the font or font size, require an immediate feedback for the user.

Feedback of the system is provided by dialogues as well. The *Dialogue* component therefore offers a collection of necessary dialogues and dialogue templates. Furthermore, status and tool bars provide relevant information throughout the entire application.

All in all the user interface components are designed to guide the user in four steps through the creation of a questionnaire. First of all a workspace, which defines the general topic, is set up with the Workspace Management component. Afterwards a new questionnaire is created in this workspace with the use of the Questionnaire Management component. Required languages are adjusted, navigation instruments are labelled and general information are entered. Then, elements and pages are created, which should occur in the new questionnaire by using the Element Builder. Finally, this pages and elements are assembled to an executable questionnaire with the functionality provided by the Questionnaire Builder. Each of these components require background components to process and store the data of the user input, which are therefore described in section 5.1.2.

5.1.2 Background Components

The Dialogue Handler, Version Control, Perspective Handler, Protocol, Element Placer, Connector Creator, Graph Operations, Status Information Updater, Multi-language Support component and the persistence components belong to the background components (as shown in figure 5.1).

The *Dialogue Handler* component is responsible for the management of dialogues (see the Dialogue component in section 5.1.1). Because of the wide variety of dialogues, the Dialogue Handler provides comprehensive functionality regarding the creation of necessary objects (e.g., another window), for calculations for the required dialogue position (e.g.,

window-centred or predefined) and for displaying several dialogues at once. Even functionality affecting the visual appearance, like the use of transparency, or behaviour of common dialogue parts, like the initial focus, are adjustable through such a dialogue handler. Because the Dialogue Handler provides only support functionality for user interface components and is not part of any user interactions, it belongs to the category of background components.

With a *Version Control* component different versions of an object are tracked and saved. A simple questionnaire element for example can be modified several times and included into several finalised questionnaires. If a questionnaire has reached the state *finalised*, it is put into practice and therefore going to be executed by respondents. In this state the questionnaire is immutable to avoid divergent and not comparable answers. Nevertheless, parts of the questionnaires are required in other questionnaires as well and may demand adjustments. To modify an element, which is nested into a finalised questionnaire, versions are needed. With versions different editing states of an object are available at the same time. Versions are immutable to provide data integrity, but offer the functionality to load the versions content, manipulate it and save it as a new version. Therefore, not the entire object with all its versions is referenced, but a single version of the object. Besides the simple elements, complex elements, questionnaires and workspaces require a version control as well and resulting from that, every user interface component except Dialogues.

The *Perspective Handler* component provides functionality for user interface components as well. Every window consists of a variety of view areas with distinct forms and input possibilities for user interaction. Because the overview clarity suffers under a growing number of view areas, sets of view areas, which are shown at once, are bundled in perspectives. The latter require additional functionality to administrate the nested view areas. For example methods to initialise and dispose view areas are linked to a call to change the recent perspective. With a Perspective Handler comprehensive perspective mechanisms are therefore available.

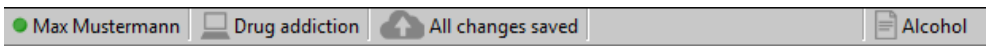

Figure 5.4: Status bar of a process-aware questionnaire configurator prototype.

To help the user to keep track of relevant information (e.g., entered data through different perspective changes) a status bar is beneficial (as shown in figure 5.4). The functionality to feed this data from everywhere within the application is provided by the *Status Information Updater* component. An object registers itself to one of the available data sources and is informed every time the state of this data source changes – such an object is therefore called a *receiver*. In contrast, objects are able to take over the functionality to trigger changes by registering themselves as a *sender*. With the sender functionality tool bars are feasible in addition to status bars.

Another component to ease usability is the *Protocol*. A protocol saves actions from a user and automatically creates reverse actions to undo the origin ones. Therefore, an intern called method is mapped to a set of methods required to reset the functionality of the initial method call. With every undo call the set of undo methods for an action is executed successively. As a result, the user is able to undo or redo taken actions, like inserting a question page into a questionnaire.

The *Element Placer*, *Connector Creator* and *Graph Operations* components are especially relevant for the graphical modelling environment of the Questionnaire Builder component. Figure 5.3 shows an excerpt from the modelling environment of a process-aware questionnaire prototype (based on the design approach of [32]). On the right of figure 5.3 the available (and categorised) elements are located and placed – depending on the available space of the right view area – with the Element Placer component. The latter places a set of elements as space-saving as possible on a defined area. On the left side of figure 5.3 the modelling of a questionnaire is shown. Elements are inserted into the questionnaire by simply dragging them on an available edge. With every insertion (and deletion), edges are adjusted (and created) automatically via the Connector Creator component. After the Element Placer component positions the questionnaire elements on the modelling view area the Connector Creator calculates paths between each linked element pair. Those paths are not allowed to overlap to avoid ambivalent interpretations. At last the Graph Operations component offers direct access to the modelling graph and a set of predefined operations, like setting a comprehensive zoom level or searching for a specific element.

With the *Multi-language support* component a template is defined, which offers common functionality to handle a multi-language part. Methods to integrate a language mask are covered by this component as well.

Finally, the persistence components Object Wrapper, File Format IO Handler and Configuration File Handler provide functionality to save the processed data on disk and optional on a remote server. Section 5.2 gives a more detailed view on the structure and mechanisms of the persistence components.

5.2 Persistence

The persistence layer constitutes an Object Wrapper, File Format IO Handler and Configuration File Handler component. These components provide functionality to persist processed data provided by user input through user interface components. This application data can be stored as local files and on a remote server. Because both approaches have different advantages, a hybrid approach is beneficial and described in this section. An offline storage saves data in specific file formats in local directories. As a result, the local files are accessible and manageable by the user and can be used for backups or to share them with other users (if not restricted by the file format or an encrypted file content). Furthermore, the required data is directly available and does not depend on the bandwidth, workload and availability of a remote server. On the other side, online storage eases the management of files by triggering automatic backups or holding the logic of the data structure, and provide supplementary functionality to work on the same data set from different places and devices.

Figure 5.5 shows the architecture of a hybrid approach based on the persistence components, which store files locally as well as remotely. The *Application Logic* is the basis by processing input data and displaying output data. This in- and output data is available in form of objects and allows logical sets of data to be encapsulated into a single construct. For example all relevant information of a specific workspace, like a title, description, or a set of authorised user, are bundled in a single workspace object and are manipulated through the user interface component Workspace Management. Because these objects are structured based on a specific programming language, they have to be adjusted to match a different structure on a remote server. The latter can be a direct database for example, communicating via a database connector in form of SQL queries, (see section 5.2.1) or a web service (as described in section 5.2.2), which uses SOAP or HTTP messages with an XML or JSON content for the communication (e.g., REST [5]). To provide application logic independently from the data structure of a remote server, a wrapper layer is provided by

the *Object Wrapper* component (as shown in figure 5.5). The functionality to store the data offline is provided by the *File Format IO Handler* component, which uses the already existing wrapper to store the objects locally in the same way as the remote server (or additional wrappers for a different data structure). Finally, the *Configuration File Handler* component manages (independently structured) application files to change variables of the application runtime environment or to manage user preferences.

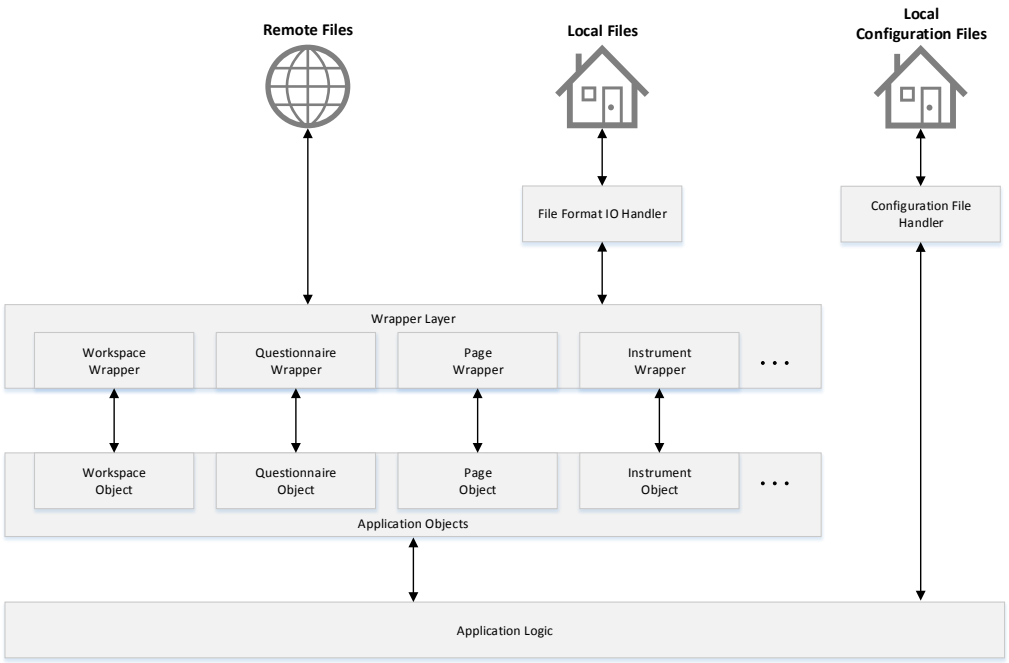

Figure 5.5: Architecture of the persistence components of a process-aware questionnaire configurator prototype.

5.2.1 Direct Database Connection

To connect a questionnaire configurator to a database, connectivity drivers are required, providing methods for the direct access to stored data. These methods are called *queries* and exist in different forms throughout relational, hierarchical, network databases and many more. A query reads or writes data parts from an underlying database. These data parts are sets of single variables, which reflect a data type respectively, like an integer or a string.

A wrapper therefore has to assemble an object (e.g., a workspace object) by firstly executing queries until all required variables are received and finally assigning these variables to the right object variables. On the other hand, a given object, which needs to be stored in a remote database, has to be disassembled into single queries. Because the integrity of the transmission is depending on the successful executing of every single query for an object, a transactional content has to be available either by the server or by the application. Furthermore, the database structure may be different, but nevertheless has to cover the offered functionality of the application. If for example a workspace offers the manipulation of a description text, an related string variable for the description has to be available in the database.

Because not every manipulated object requires to be stored immediately on the remote database, the wrapper layer (as seen in figure 5.5) has to provide a *queue*. A queue holds track of every manipulated object and transfers finalised objects to the database. This transmission can depend on the state change of an object or is triggered by a timer. To keep track of the states, all objects in the queue are tagged with a state, like `LOCAL_ONLY` (for an object which is temporarily stored locally), `READY_FOR_TRANSMISSION` (for an object ready to be stored in the database), `IN_WORK` (for an object which is in use and therefore is not stored yet), etcetera. With the termination of the application, this states and their related objects have to be stored. The Configuration File Handler component therefore saves this mapping in an extra file, which is read at every application start.

The offline storage of the objects uses the same data structure as on the server to avoid redundant source code. The definition which content is stored in a single file is arbitrary, but to ensure a clear file system it is beneficial to nest the content of every workspace into an own file. This file has to match the structure on the remote server, for example a *Structured Query Language Data* file for a SQL database.

To ensure data integrity between the offline and online data set, a priority of significance needs to be defined. At the start of the application all local data is read. The online data requires a login and through a possible loss of connection it does not act as a continuously data source. Because a large number of users can work on a validated online data set, this data has the highest significance and therefore is always considered as correct if it conflicts with a local data set. For example the manipulated offline data by a user can therefore not replace valid data stored on the remote server. The local data is always adjusted to the

online data set. To avoid the loss of offline work, the Version Control component offers a merge functionality to create new versions based upon old data sets.

5.2.2 Web Services

Web services provide interfaces described in a machine-processable format. Every interface is accessible by an URI via messages of a specific protocol (e.g., HTTP). The content of these messages is mostly defined in XML and therefore allows more complex requests than a query (as described in section 5.2.1). With XML the structure and content of an entire object can be transferred at once. The interfaces of a web service are therefore able to offer entire objects for transmission. Resulting from that, every wrapper of the Object Wrapper component only requires the mapping from a single XML object to a single application object. Because XML is a widespread and well documented descriptive language, several software libraries are available which cover the functionality to transform an object into an XML representation and vice versa. Besides XML further formats, like the Java Script Object Notation (JSON) are available. Moreover, such formats offer comprehensive functionality to read and write local representations, which reduces the functionality the File Format IO Handler component needs to provide.

5.2.3 Summary

With the management of online data on the one hand and local data on the other hand, the complexity to guarantee data integrity increases. A hybrid approach to take advantage by combining the online and offline functionality requires a strict defined interaction pattern to coordinate the data access. Furthermore, the different technologies to store data sets remotely demand the responsible components to allow a flexible adaption. With a wrapper layer this adaption is provided and offers supplementary functionality, like queueing and state assignments. To ensure a validated data set, the online data is weighted higher than the local files and therefore avoid manually data manipulation. Nevertheless, strong tools and libraries for web services are available to provide support functionality for this hybrid approach.

5.3 Design Patterns

> ”Design patterns are basically design tools to improve existing code. [...] They standardize common programming tasks into recognizable forms, giving your projects better cohesiveness.“ (Lasater, 2010) [7]

With design patterns components and component parts are structured (as mentioned in the quote) in a standardised way. This standardisation validates the structure and reduces the introductory period. In this section the well-known Presentation Abstraction Control (PAC), Observer and Factory Pattern are discussed regarding their advantages for a process-aware questionnaire configurator.

The *PAC pattern* separates the functional core of an application from the graphical representation. With a set of hierarchic organised *agent*s a multi-layer architecture is created. Every agent represents a part of the application logic and defines a presentation, abstraction and control component. The *presentation* component displays data in a graphical user interface and therefore manages the user in- and output. Displayed data is located in the *abstraction* component in form of a structured data model. Via the *control* component the presentation is connected with the abstraction and supplementary functionality for the communication with other agents is provided. There are three types of agents: The *top-level* agent constitutes the core functionality and therefore mostly manages no user interface parts. Only one single top-level agent is allowed, which defines the root layer of the architecture. *Intermediate* agents constitute the underlying layers (at least one) by defining dependencies between agents of any type. The bottom layer is defined by *bottom* agents, which provide encapsulated concepts for user interaction.

The result is an architecture where application parts are continuously subdivided into smaller parts with every hierarchical layer. Especially complex applications with a lot of user interaction benefit from the possibility to encapsulate semantic parts independent from comprehensive functionality. Consider for example the excerpt of the questionnaire management in figure 5.6. A questionnaire hosts several editable information and settings (as described in section 5.1), which are encapsulated into the three different semantic parts Description Dialogue, Dependency Dialogue and Operations Dialogue. A *Description Dialogue* holds all the descriptive information about a questionnaire, whereas the *Dependency Dialogue* determines the dependencies to other questionnaires (and an additional setting to lock the questionnaire). With the *Operations Dialogue* comprehensive functionality for the manage-

ment of a single questionnaire is provided. Each of these parts concentrate on different aspects of a questionnaire and therefore invoke different functionality. Because all parts belong to the same superior construct – a questionnaire – they are displayed and handled as a unit in form of a *Questionnaire Dialogue*. If for example the height of the Description Dialogue exceeds the overall height of the Questionnaire Dialogue (e.g., by adding new supported languages), the size of the superior Questionnaire Dialogue is extended. Because the overview of questionnaires contains mostly more than one questionnaire, the change of size affects following questionnaires. As a result, the locations of the latter has to be recalculated.

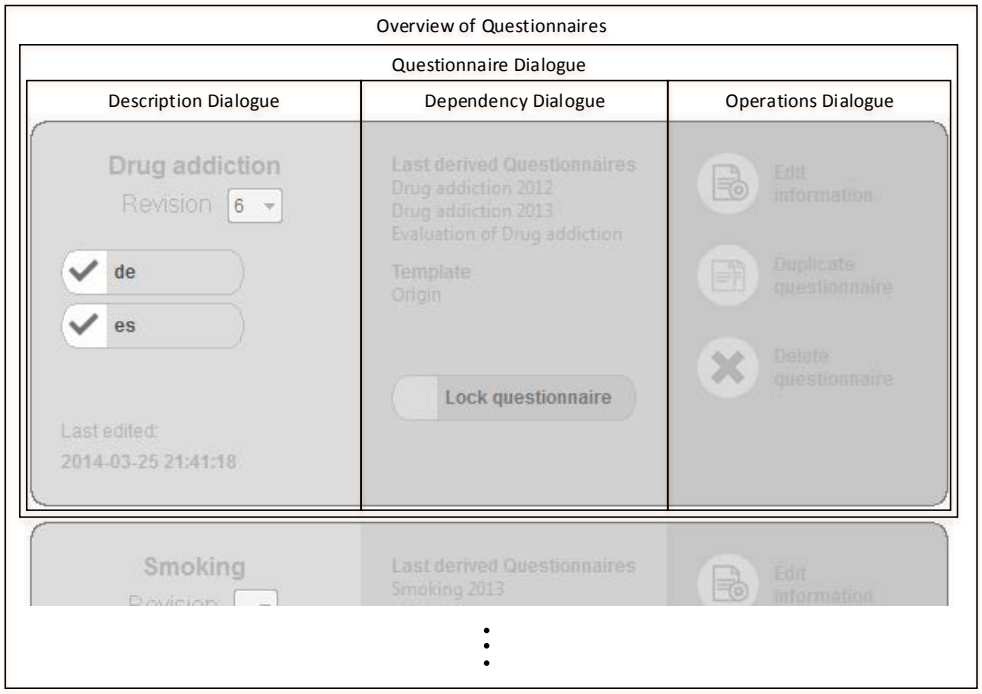

Figure 5.6: Excerpt of the questionnaire management of a process-aware questionnaire configurator prototype.

Figure 5.7 shows the transformation of the questionnaire management from figure 5.6 to the multi-agent structure defined by the PAC pattern. The top-level agent is the container of questionnaires which offers no user interfaces, but comprehensive functionality to position the questionnaire dialogues in a specific way (e.g., with a predefined margin to avoid

overlappings). A questionnaire dialogue is an intermediate-level agent, which contains a description, dependency and operations dialogue in form of a dependency to a bottom-layer agent respectively. The intermediate agent provides functionality to combine the bottom-layer agents in a uniform construct and offers additional user interface aspects (e.g., a border which separates between active and non-active questionnaires). With three bottom-layer agents different user in- and outputs are provided by independent user interfaces. The communication to inferior layers is expressed as function call, whereas events define the communication to superior layers (as shown in figure 5.7). The architecture defined by the PAC pattern is expandable to all user interface components mentioned in section 5.1.

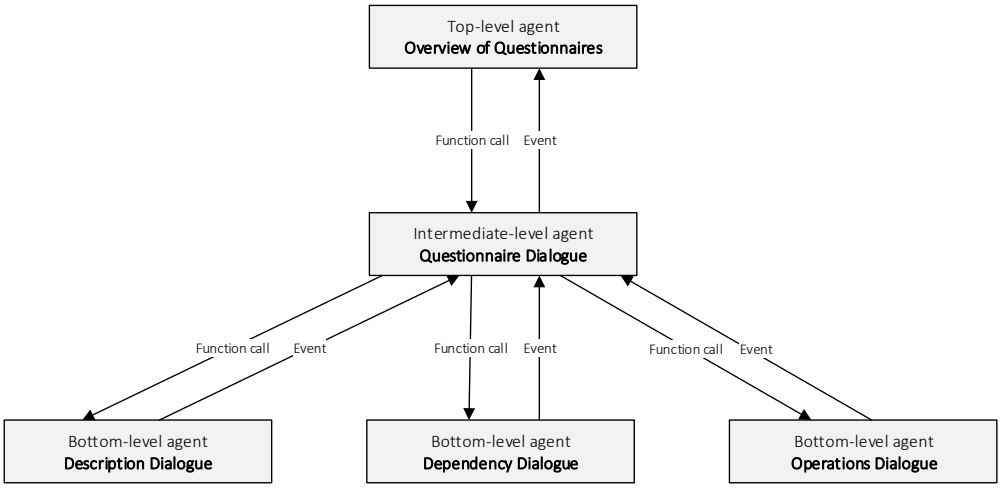

Figure 5.7: The hierarchical agent structure of the PAC pattern for the questionnaire managment in figure 5.6.

The *Observer* pattern is a behavioural pattern, which focuses on unilateral dependencies between components. One component is called the *subject* and states the object on which the *observer* components depend on. Every time the subject changes, all registered observers are notified. This behaviour is especially relevant if data needs to be displayed or updated in real time. In contrast to automatic polling, the observers are independent from timers, which pushes the next subject call and therefore require less hardware resources. A common example is a status bar (see figure 5.4), which displays information, which are spread over different components. Every time a component changes its information (e.g., through a user input), the status bar requires the new state as fast as possible to keep the

user up-to-date. This is exactly the task of the Status Information Updater component (described in section 5.1.2) – but independently from a user interface – which therefore benefit from the subject-observer structure.

With a *Factory pattern* the creation of application constructs, like questionnaire elements and pages, is managed by *factories*. The latter encapsulate the creating logic of an construct – also called *product* – in a central location. Resulting from that, initialisation procedures are not spread across the application and can therefore be adjusted faster and less error-prone. Moreover, supplementary management functionality is integrable. A factory can operate as a repository by saving the references to every created product and with that perform comprehensive tasks. Consider for example a factory which encapsulates the creating logic of every questionnaire element. Every time an element is created a reference is saved in this element factory. If the user requires an overview of the created elements since application start (e.g., to validate that every required element has been created successfully), the element factory can provide a list of created elements by analysing the saved references and extracting the necessary data. In the developed configurator prototype product repositories are available in the persistence components. Every created element needs to be stored and therefore referenced by the persistence module.

5.4 Usage of the Rich Client Platform

The Rich Client Platform (as described in section 2.3) supports JAVA application developers by offering functionality for common tasks. With the RCP workbench and the RCP application model (see section 2.3.2 and 2.3.3) a window management tool is provided for graphical software applications. The application is divided into windows defining their nested view areas. Several constructs to manage these view areas, especially RCP perspectives, RCP part stacks and RCP parts (see section 2.3.3), can be declared and accessed through specific services at runtime.

In section 5.3 the PAC pattern is described to present a structure for interactive software systems. Figure 5.8 shows an approach to integrate the RCP application model to the PAC structure of a process-aware questionnaire configurator (as shown in figure 5.7). Basically every user interface component (in figure 5.8 the Questionnaires Management component is used exemplary) is packed into an RCP part. Each of these RCP parts is then linked to an RCP perspective, which is anon linked to an RCP window. With this structure the

arrangement of all view areas provided by the user interface components, is defined in a central and independent place.

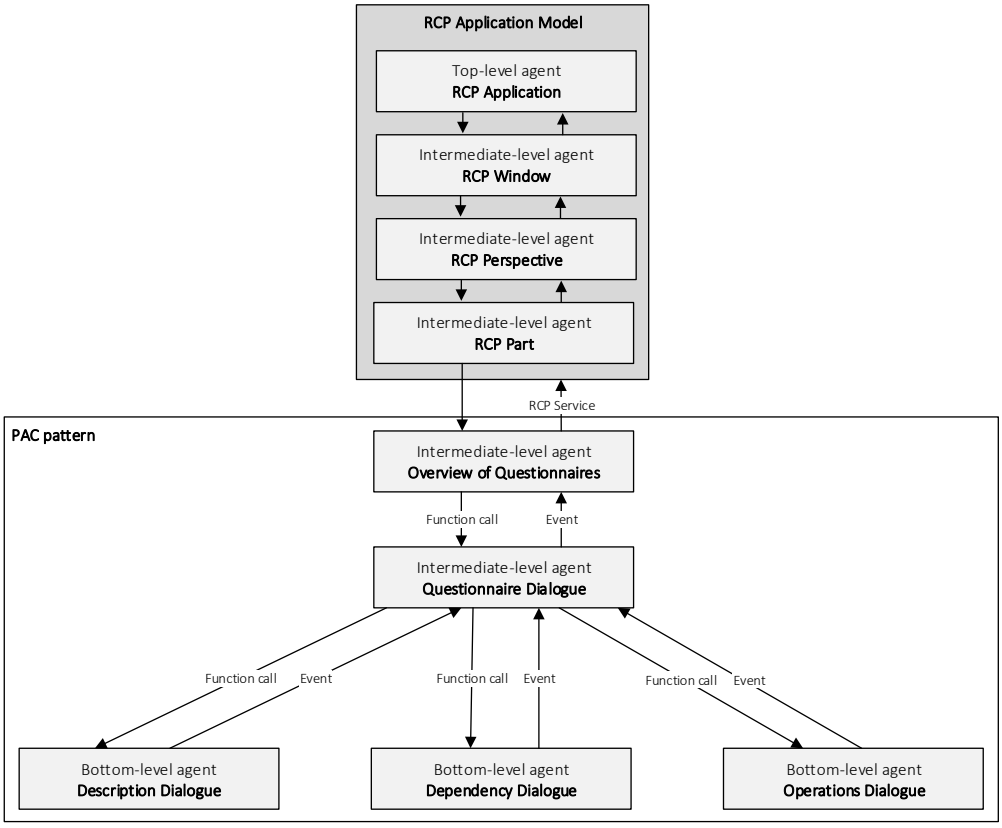

Figure 5.8: The hierarchical agent structure of the PAC pattern for the questionnaire managment using RCP.

Figure 5.8 shows an example of an RCP application, which displays the user interface component Questionnaire Management (as seen in figure 5.6). Firstly, an RCP window has to be defined including a related RCP perspective with one RCP part. Secondly, the size is defined by declaring an RCP window size and an RCP part size (which is a relative value to the size of the RCP window). The arrangement of view areas is dispensable with only one RCP part. Finally, the RCP part is mapped to the Questionnaire Management component by providing a view area (in RCP through an object called *Composite*), which can be filled arbitrary. Because RCP parts provide comprehensive functionality for their content, they interact – transferred to the PAC pattern – like a layer of intermediate agents. Nested

components are able to access the superior RCP part with the available RCP services – in this case the EPartService (as described in section 2.3.1 and 2.3.3). Every RCP part is nested directly or through other containers (e.g., an RCP PartStack or RCP PartSashContainer) into an RCP perspective. The latter offers additional comprehensive functionality for the arrangement of RCP parts and therefore function as an independent intermediate-layer of agents as well. This repeats until the root of the RCP application model (in form of a single RCP application construct, which holds references of all RCP windows) is reached. In figure 5.8 the corresponding PAC pattern is shown. If a view area for another user interface component should be displayed, a new RCP part is created, which provides then again a view area for this component. Afterwards the RCP part is connected to the RCP perspective, which is adjusted to arrange both RCP parts.

Because most components depend on the functionality of other components a central mechanism to manage dependencies is required. For this purpose, RCP offers dependency injection and provides a central storage place for registered components with the IEclipseContext (see section 2.3.1). Because the order in which the IEclipseContext is accessed is important (dependencies can only be resolved if the required components are registered first), it is beneficial to fill the IEclipseContext at the start of the application with all required components. Especially the availability of the background components for the user interface components is of great concern. Because the order of access to user interface components can lay in the hands of the user, an ad-hoc component registration can increase the complexity of the application and cause redundant registration calls.

A core requirement of most RCP applications is to offer extensibility. With the flexible and on several layers extensible generic questionnaire model (as described in section 4.2) this is of particular concern (e.g., a new questionnaire element has to be included). Because the entire RCP is based on an OSGi environment, extensibility is offered through plugins. Every plugin bundles a set of functionality and makes it available through extension points. Therefore the configurator can be extended by simply including and accessing required plugins. Moreover, functionality can be developed and used independently from a specific application. To access plugins programmatically, RCP provides an add-on mechanism, which allows the manipulation of plugins and their dependencies before the application starts. Therefore event handler are connected to different states of the application's user interface life cycle (e.g., APP_STARTUP_COMPLETE). Consider a configurator which holds all relevant text labels of the application in a single plugin for each language. Resulting from that, new languages can be dynamically added by developing new language plugins. All

these language plugins are read before the user interface is generated and therefore are available at application start.

Another important feature of the RCP is portability. Every RCP functionality supports the operating systems Linux, Apple MacOSX, Oracle Solaris and Microsoft Windows natively. For special configuration settings each supported operation system can be configured individually by defining different launcher arguments and plugin start levels for example. Furthermore, tools to create an executable application based on the source code are provided. Application icons, a runnable executable and a launcher configuration file are generated automatically. No installation is required, the application is instantly executable and even brings a chosen runtime environment to decrease the complexity for the user and ease access to a configurator environment.

5.5 Summary

In this chapter an architecture is presented, which covers all requirements for a process-aware questionnaire configurator. First of all, fundamental requirements are collected and bundled logically into components (see section 5.1). A set of these components is responsible for the user in- and output and define a user guidance through the questionnaire creation process. Questionnaires are bundled into supplementary workspaces to provide logical collections and nest simple and complex elements, which are put together to a finalised questionnaire. To get a better impression of the components, excerpts from an implemented configurator are additionally provided. Section 5.2 analyses the storage of data in more detail. To restrict the user as little as possible, the advantages of a remote server and local files to save the data are combined in a hybrid approach. Furthermore, design patterns (like the Presentation Abstraction Control pattern), which provide a better cohesiveness and code clarity, are discussed in section 5.3. These are complemented by the available functionality provided by RCP for common programming tasks, in particular considering a configurator environment (e.g., portability or extensibility). This chapter therefore paves the way for a concrete and complete configurator implementation.

6 Conclusion

Questionnaires are a fast method to collect data in empiric research. Much study in the recent years focused on the versatile aspects of questionnaires reaching from wording [27] over graphical representations [21] to element structures [2, 14]. With advancing online connectivity and mobile accessibility, technical aspects of questionnaires are of growing interest. Additional features to ease the process of managing a questionnaire, like distributing questionnaires via email, making them accessible through the internet, or working on them collaboratively from different places, are provided by software applications. A large number of these questionnaire configurator applications are available (six of them are described in chapter 3) with a different scope of functionality. None of these configurators consider a questionnaire as a process and therefore uses the rich functionality of process-aware information systems (PAIS).

With this thesis such an approach for process-aware questionnaire configurators is developed and implemented for the first time. Initially, all characteristics of a questionnaire are extracted, collected and transferred into a generic questionnaire model, which describes every questionnaire by providing a clear basis structure with extension points on several layers. This model is integrated into the process-structure of a PAIS by mapping questionnaire pages to activities and filter to branches. To validate this approach, a process-aware questionnaire configurator is implemented and in excerpts used to define an architecture for further implementations. This should pave the way for approaches, which integrate a PAIS into a questionnaire configurator, but aim to use different PAIS functionality or PAIS functionality in a different way. With the automatic execution of process-aware questionnaires only an excerpt from the capabilities of a PAIS is used. A PAIS is still a research topic, but with the integration in a configurator environment, every upcoming feature may offer another beneficial functionality for questionnaires. A few features, whose integration into questionnaires is not considered yet, are presented and described in more detail in chapter 7.

7 Future Work

In this thesis the execution of a created questionnaire is provided entirely by the PAIS environment with a developed process-aware questionnaire configurator. Resulting from that, a standardised access to questionnaires on the available clients is defined, which reduces the client-side complexity.

But a PAIS offers additional functionality – besides an execution environment – which is yet unexplored in a configurator environment. With an organisational model, agent structures are definable by attaching specific abilities, roles or positions to a set of agents. Positions are connectible to projects or units, whereas the latter is nestable into groups. Based on this network, agents can be assigned and therefore authorised to handle activities or entire processes. Consider for example a team of researchers, which plans a survey on the health risks of smoking. One part of the researchers are doctors, the other psychiatrists. The interview is divided into two phases: The first one contains questions covering the respondent's background related to smoking, the second questions about the respondent's health. To ensure that only a specialist – a psychiatrist for the first phase and a doctor for the second phase – is performing the interview, every researcher is assigned to a role in the organisational model – either to *Psychiatrist* or to *Doctor*. The role *Psychiatrist* is then authorised for the activities holding the questions for phase one, whereas the role *Doctor* is authorised for the activities related to phase two. As a result, phase one is always performed by a psychiatrist and phase two by a doctor.

Furthermore, the process schema evolution [15] is an promising PAIS functionality for a configurator. Because surveys mostly require a development in several iterations (including iterations where early versions of questionnaires are put into practice), the continuously adjustment of processes is possible through different process schemes. With every adjustment a new process scheme is created, which allows instances of earlier process variants to be finished without interruption. This mechanism could replace or extend the required version control functionality. With the PAIS instance migration even ad-hoc schema changes [15] for running instances are realisable allowing a questionnaire in execution to

be adjusted to the new structure. Because constraint sets define question page dependencies, deadlocks are possible. As a result strong checks are necessary for the use of the instance migration tool and only questionnaires for test purposes should be affected (to ensure the validity of the survey).

Another interesting aspect of a PAIS is time [16]. Although time dependency is a topic still in development, a wide range of applications is conceivable. Consider the example above, where a survey is divided into two phases. To ensure that the respondent is not influenced by the first interview, the second interview has to be performed at least a month later. If this time dependency could be expressed via a PAIS, the configurator is able to create questionnaires with additional constraint types.

All in all the combination of a questionnaire configurator with a PAIS is in the beginning stage. In this thesis the useful and supporting functionality of a PAIS for a configurator is analysed in its basics, but through several milestones, the utilisation and integration is eased for further research.

A The Process-aware Questionnaire Configurator Prototype

Figure A.1: Welcome perspective of a process-aware questionnaire configurator prototype.

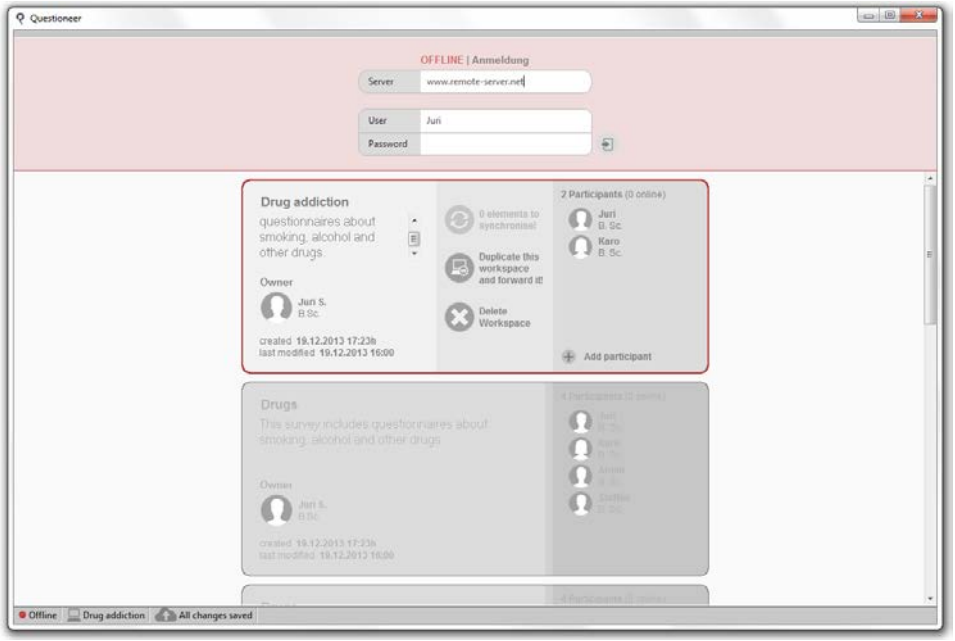

Figure A.2: Workspace perspective of a process-aware questionnaire configurator prototype.

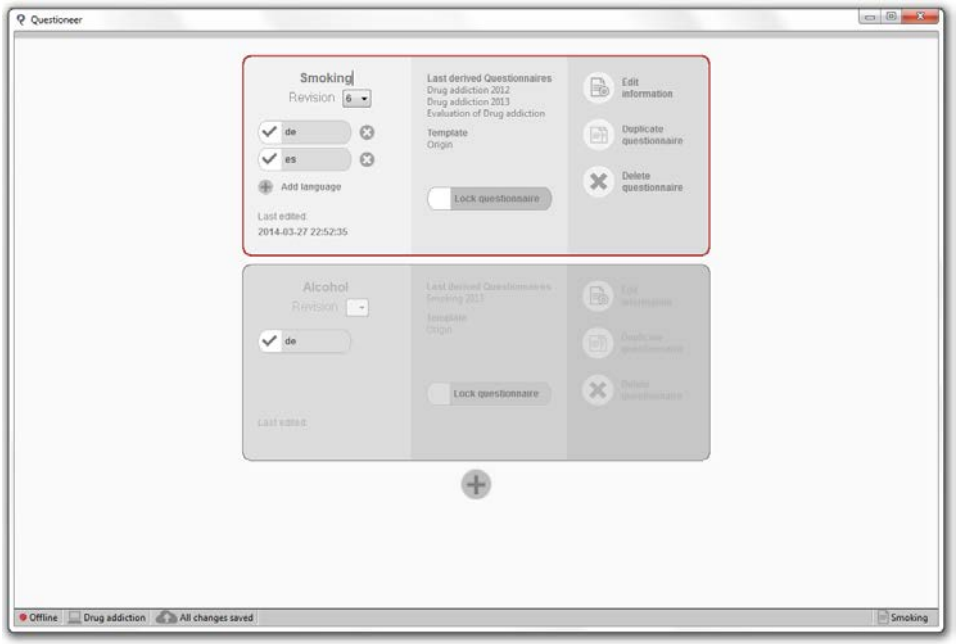

Figure A.3: Questionnaire perspective of a process-aware questionnaire configurator prototype.

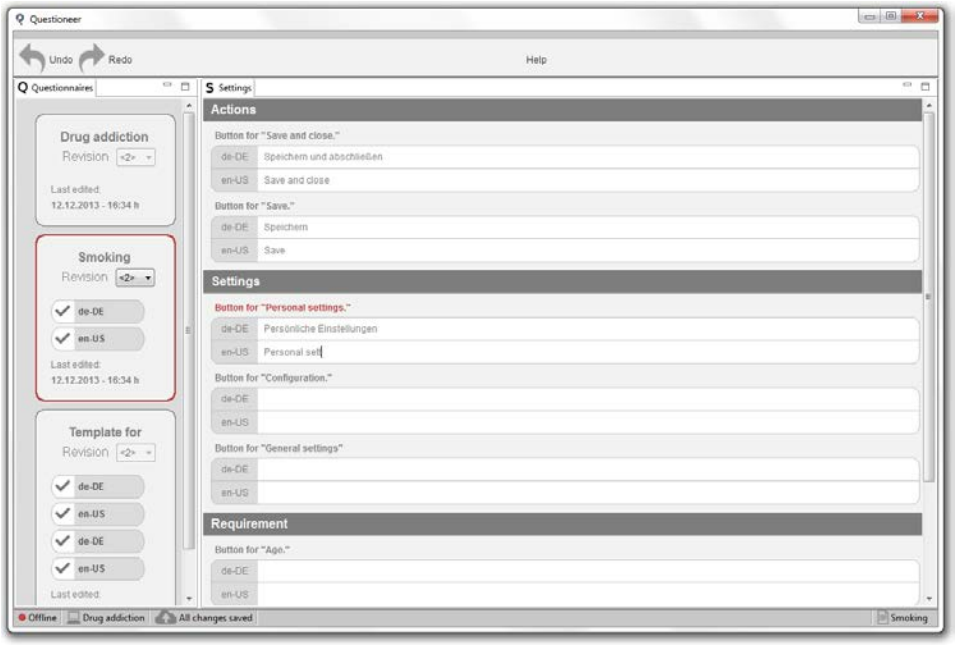

Figure A.4: Questionnaire configuration perspective of a process-aware questionnaire configurator prototype.

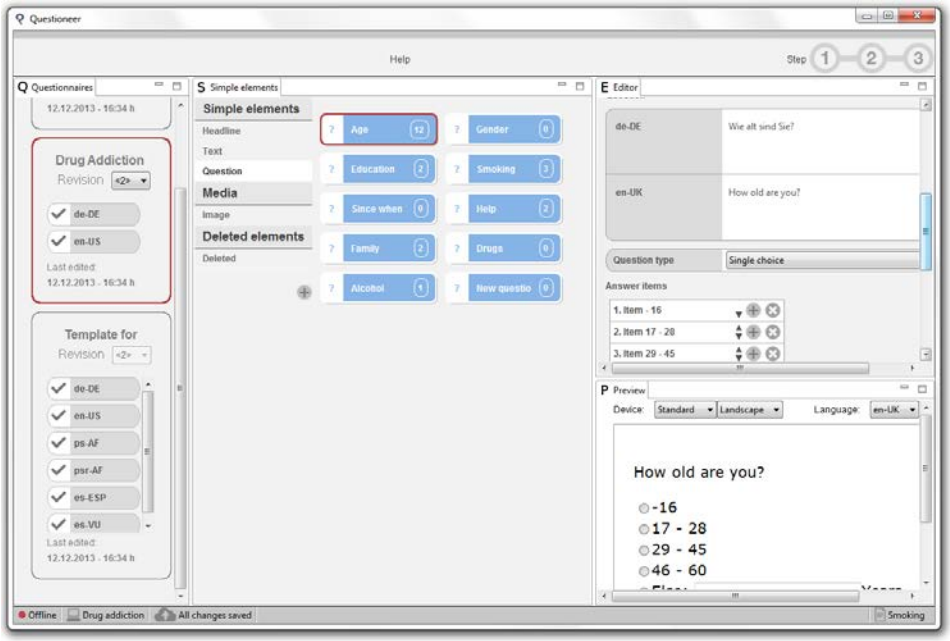

Figure A.5: Simple element builder of a process-aware questionnaire configurator prototype.

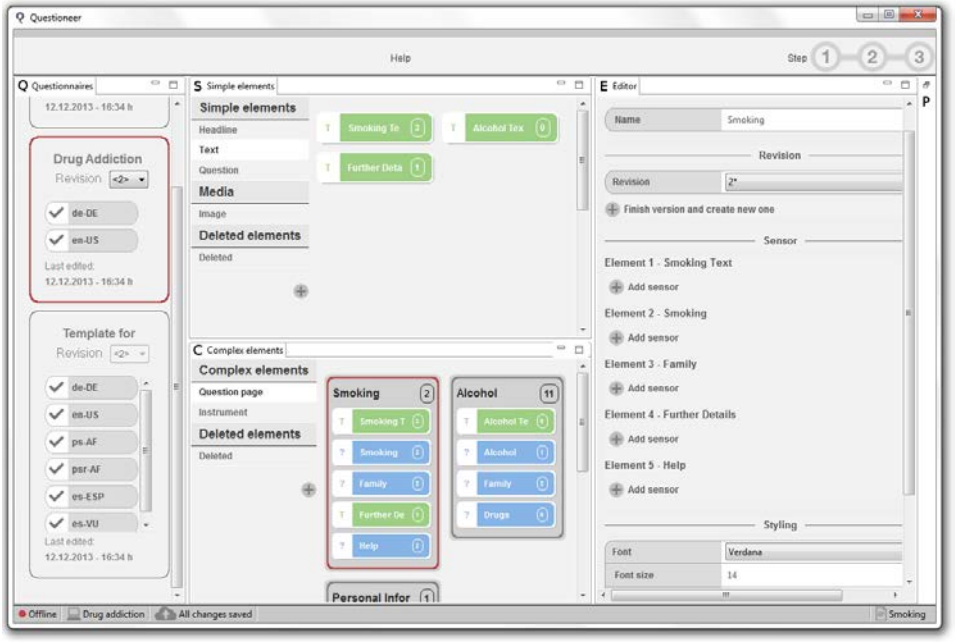

Figure A.6: Complex element builder of a process-aware questionnaire configurator prototype.

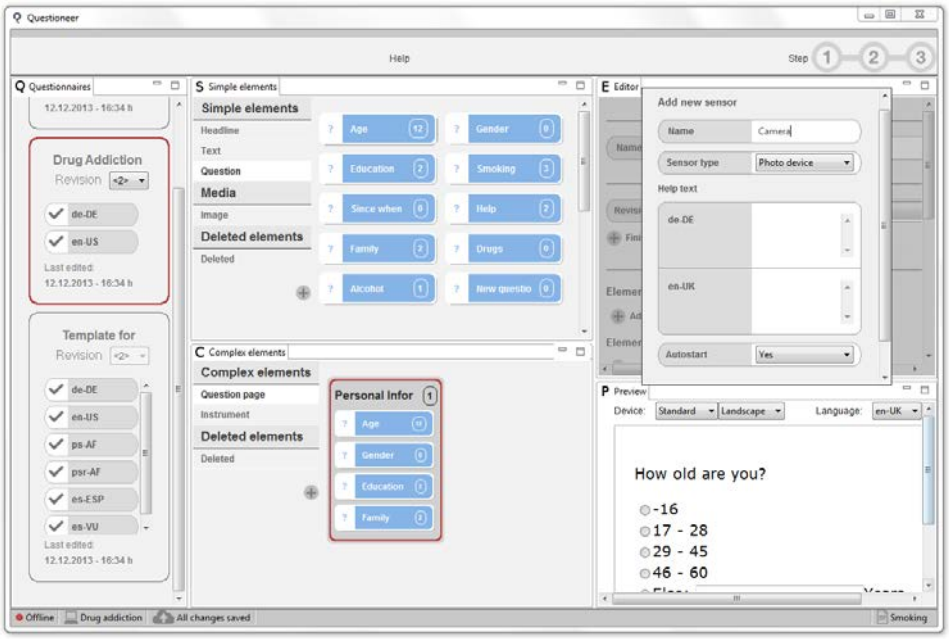

Figure A.7: Adding a sensor for a complex element in a process-aware questionnaire configurator prototype.

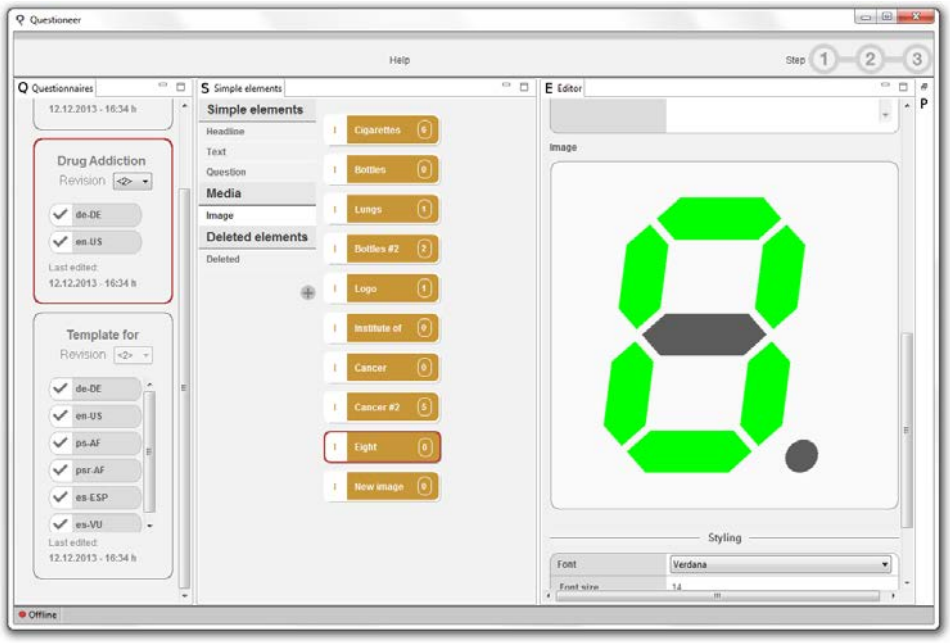

Figure A.8: Multimedia elements in a process-aware questionnaire configurator prototype.

Figure A.9: Questionnaire builder of a process-aware questionnaire configurator prototype.

Bibliography

[1] ARNIM SCHINDLER: *Technische Konzeption und Realisierung des MACE-Tests mittels mobiler Technologie*. University Ulm, 2013

[2] ATTESLANDER, Peter: *Methoden der empirischen Sozialforschung*. 10., neubearb. und erw. Aufl. Berlin : Walter de Gruyter, 2003 (De Gruyter Studienbuch). – ISBN 9783503093212

[3] BABBIE, Earl R.: *Survey research methods*. 2nd ed. Belmont and Calif : Wadsworth Pub. Co., 1990. – ISBN 9780534981969

[4] BOARDHOST.COM INC.: *Pollcode*. – `http://pollcode.com/`, visited 2014-03-28

[5] BURKE, Bill: *RESTful Java with JAX-RS*. Sebastopol and CA : O'Reilly, 2010. – ISBN 0596158041

[6] CARSTEN SCHMITZ: *LimeSurvey Manual*. 2014. – `http://manual.limesurvey.org/index.php?title=LimeSurvey_Manual&oldid=59930`, visited 2014-03-28

[7] CHRISTOPHER G. LASATER: *Design Patterns*. Jones & Bartlett Learning, 2010

[8] ECLIPSE FOUNDATION: *Eclipse documentation - Current Release: Eclipse Kepler*. – `http://help.eclipse.org/kepler/index.jsp`, visited 2014-03-28

[9] FABIAN MAIER: *Entwicklung eines mobilen und Service getriebenen Workflow-Clients zur Unterstützung von evaluierten Studien der klinischen Psychologie am Beispiel der AristaFlow BPM Suite und Android: Bachelorarbeit an der Universität Ulm*. University Ulm, 2012

[10] GOLDEN HILLS SOFTWARE INC.: *SurveyGold User Guide*. – `http://www.surveygoldsolutions.com/user-guide.html`, visited 2014-03-28

[11] IBM: *IBM SPSS Statistics 22 Core System User's Guide*. – `ftp://public.dhe.ibm.com/software/analytics/spss/documentation/`

statistics/22.0/en/client/Manuals/IBM_SPSS_Statistics_Core_System_User_Guide.pdf, visited 2014-03-28

[12] IBM: *Welcome to Eclipse*. 2005

[13] JÖRG MICHAEL GRÜNING: *Technische Konzeption und Realisierung der Laufzeitumgebung für ein generisches Fragebogensystem zur IT-gestützten Durchführung von evaluierten Studien der Klinischen Psychologie*. University Ulm, 2012

[14] KITCHENHAM, Barbara A. ; PFLEEGER, Shari L.: Principles of survey research. In: *ACM SIGSOFT Software Engineering Notes* 27 (2002), Nr. 2, pp. 20. http://dx.doi.org/10.1145/511152.511155. – DOI 10.1145/511152.511155. – ISSN 01635948

[15] KROGSTIE, John ; OPDAHL, Andreas ; SINDRE, Guttorm: *Lecture Notes in Computer Science*. Bd. 4495: *Advanced Information Systems Engineering: 19th International Conference, CAiSE 2007, Trondheim, Norway, June 11-15, 2007. Proceedings*. Berlin and Heidelberg : Springer-Verlag Berlin Heidelberg, 2007. – ISBN 3540729887

[16] LANZ, Andreas ; WEBER, Barbara ; REICHERT, Manfred: Time patterns for process-aware information systems. In: *Requirements Engineering* (2012). http://dx.doi.org/10.1007/s00766-012-0162-3. – DOI 10.1007/s00766–012–0162–3. – ISSN 0947–3602

[17] MAXIMILIAN XAVER SCHMID: *Technische Konzeption und Realisierung der Anwenderumgebung für ein generisches Fragebogensystem zur IT-gestützten Durchführung von evaluierten Studien der Klinischen Psychologie*. University Ulm, 2012

[18] MCAFFER, Jeff ; LEMIEUX, Jean-Michel ; ANISZCZYK, Chris: *Eclipse Rich Client Platform*. 2nd ed. Upper Saddle River and NJ : Addison-Wesley, 2010 (The eclipse series). – ISBN 9780321603784

[19] MCCOLL, E. ; JACOBY, A. ; THOMAS, L.: *Health technology assessment*. Bd. vol 5 no. 31: *Design and use of questionnaires: A review of best practice applicable to surveys of health service staff and patients*. National Co-ordinating Centre for HTA.Great Britain, 2001

[20] MICROSOFT: *Rich Text Format (RTF) Specification 1.7*. 2003

[21] PORST, Rolf: *Fragebogen: Ein Arbeitsbuch*. 1. Aufl. Wiesbaden : VS, Verl. für Sozialwiss, 2008 (Lehrbuch). – ISBN 3531908979

[22] R FOUNDATION: *An Introduction to R*. – http://www.r-project.org/, visited 2014-03-28

[23] RAAB-STEINER, Elisabeth ; BENESCH, Michael: *UTB Schlüsselkompetenzen*. Bd. 8406: *Der Fragebogen: Von der Forschungsidee zur SPSS-Auswertung*. 3., aktualisierte und überarb. Aufl. Wien : Facultas-Verl, 2012. – ISBN 978–3–8252–8496–1

[24] RALF EBERT: *Eclipse RCP: Entwicklung von Desktop-Anwendungen mit der Eclipse Rich Client Platform 3.7*. 2011

[25] REICHERT, Manfred ; DADAM, Peter: Enabling adaptive process-aware information systems with ADEPT2. In: *Handbook of research on business process modeling, pp. 173–203* (2009)

[26] SAS: *SAS/STAT 13.1 User's Guide*. – http://support.sas.com/documentation/onlinedoc/stat/131/statug.pdf, visited 2014-03-28

[27] SCHOLL, Armin: *UTB*. Bd. 2413: *Die Befragung*. 2., überarb. Aufl. Konstanz : UVK-Verl.-Ges, 2009. – ISBN 3825224139

[28] SEBASTIAN JEHLE: *Technische Konzeption und Realisierung der Konfiguratorumgebung für ein generisches Fragebogensystem zur IT-gestützten Durchführung von evaluierten Studien der Klinischen Psychologie*. University Ulm, 2012

[29] SENKBEIL, Martin ; IHME, Jan M.: Wie valide sind Papier-und-Bleistift-Tests zur Erfassung computerbezogener Kompetenzen? In: *Diagnostica* 60 (2014), Nr. 1, pp. 22–34. http://dx.doi.org/10.1026/0012-1924/a000114. – DOI 10.1026/0012–1924/a000114. – ISSN 0012–1924

[30] STATPAC INC.: *StatPac For Windows User's Manual*. – http://www.statpac.com/download/Manual.pdf, visited 2014-03-28

[31] STEFANIE WINTER: *Quantitative vs. Qualitative Methoden*. Karlsruhe, 2000. – http://imihome.imi.uni-karlsruhe.de/nquantitative_vs_qualitative_methoden_b.html, visited 2014-03-28

[32] STEFFEN SCHERLE: *Konzeption und Evaluierung einer domänenspezifischen Modellierungsumgebung für prozessorientierte Fragebögen*. Ulm, 2014

[33] THE OSGI ALLIANCE: *OSGi Core Release 5*. 2012. – http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf, visited 2014-03-28

[34] ULLENBOOM, Christian: *Java ist auch eine Insel: Das umfassende Handbuch ; [aktuell zu Java 7 ; Programmieren mit der Java Platform, Standard Edition 7 ; Java von A bis Z: Einführung, Praxis, Referenz ; von Klassen und Objekten zu Datenstrukturen und Algorithmen]*. 10., aktualisierte und überarb. Aufl. Bonn : Galileo Press, 2012 (Galileo computing). – ISBN 9783836218023

[35] VOGEL, Lars: *Eclipse 4 RCP: The complete guide to Eclipse application development*. Second edition. 2013 (Vogella series). – ISBN 3943747077

[36] W3C: *SOAP Version 1.2*. – `http://www.w3.org/TR/2007/REC-soap12-part0-20070427/`, visited 2014-03-28

[37] W3C: *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. 07.07.2011

[38] W3C: *HTML5*. 2014. – `http://www.w3.org/TR/html5/`, visited 2014-03-28

Name: Juri Schulte Matrikelnummer: 677063

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Juri Schulte