Ulm University | 89069 Ulm | Germany

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

Concept and Implementation of a Rule Component Enabling Automatic Analysis of Process-Aware Questionnaires

Master Thesis at Ulm University

Submitted by:
Bernd Mertesz
bernd.mertesz@uni-ulm.de

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

Supervisor:
Johannes Schobel

2014

Version October 22, 2014

Satz: PDF-LaTeX 2_ε

Abstract

Psychological studies are usually done using paper-based questionnaires, which are a common and inexpensive way to collect data. However, this leads to problems, especially with very big studies. Usually, the evaluation of thousands of completed questionnaires needs the help of a computer application. Therefore, the answers of the subjects need to be transfered manually to electronic worksheets (e.g., Microsoft Excel spreadsheets). The manual transfer opens the possibility for errors when transcribing handwritten text and causes a lot of work.
One possible solution for this problem could be, to support the complete lifecycle of a questionnaire digitally. *QuestionSys* is one system aiming to provide a full digital support for domain experts and is developed at the University of Ulm.
This thesis presents the theoretical concept and the development of the rule editor *Questionrule*. This component enables domain experts to create and manage rules, which are then used to evaluate a completed questionnaire. This has to be achieved in an abstract and intuitive way, as domain experts usually have little or no experience in boolean algebra.
The different concepts and technologies that are used for the development of *Questionrule*, are presented in this thesis. In addition, an outline of other rule editors as well as a comparison is provided.

Acknowledgment

First of all, i would like to thank Johannes Schobel for his support and important guidance through all of this thesis.
My deepest appreciation goes to Michael, Raphael, Steini, Thinh and Wolfgang, my fellow students, for their moral support and advice. A special thanks goes to Markus Brenner, who recommended ANTLR and was always there to answer questions regarding ANTLR.
Last but not least, my thanks goes to my family who believed in me the whole time, not only while writing this thesis, but all of my life. Without you, this thesis would be not possible.

Contents

1 Introduction

Nowadays, most studies are built on paper questionnaires. This means, a domain expert creates a questionnaire within a word processing application and prints it on paper. Later, the subject fills in this paper-based questionnaire. This approach is also used in case of a very big study with thousands of subjects, which results in a big amount of completed questionnaires that need to be evaluated at a later point in time.
This is, however, the main issue with this approach. For the evaluation of these completed questionnaires and to draw conclusions from the given answers, the help of a computer application is needed. However, to do so, the answers of the subjects need to be transfered manually to electronic worksheets (e.g., Microsoft Excel spreadsheets). This is, on the one hand, a lot of work and on the other hand opens the possibility for errors while transcribing handwritten text.
One possible solution for this problem could be, to support the complete lifecycle of a questionnaire digitally. That means, that the creation of a questionnaire takes place

with a computer application and that the subjects fill in the questionnaire on a computer. Thereby, it is no longer necessary to print the questionnaires and to transcribe the data collected. In addition, the analysis of a single questionnaire can occur directly after completion. One system aiming to provide a full digital support for domain experts is called *QuestionSys* and is being developed at the University of Ulm [Ulm].

When designing the system, an adequate representation for electronic questionnaires has to be developed. In other research papers [SSP+14] mapping for questionnaires to business processes has already been proposed. That means, single pages of a questionnaire are *activities* and the order of them are defined through the structure and control flow between *activities*. Additionally, questions can be displayed depending on already given answers using specific gateways. Figure 1.1 shows an example questionnaire modeled as a business process. This business process is modeled in BPMN 2.0 [OMG11]. Note, that specific elements used in this context are annotated.

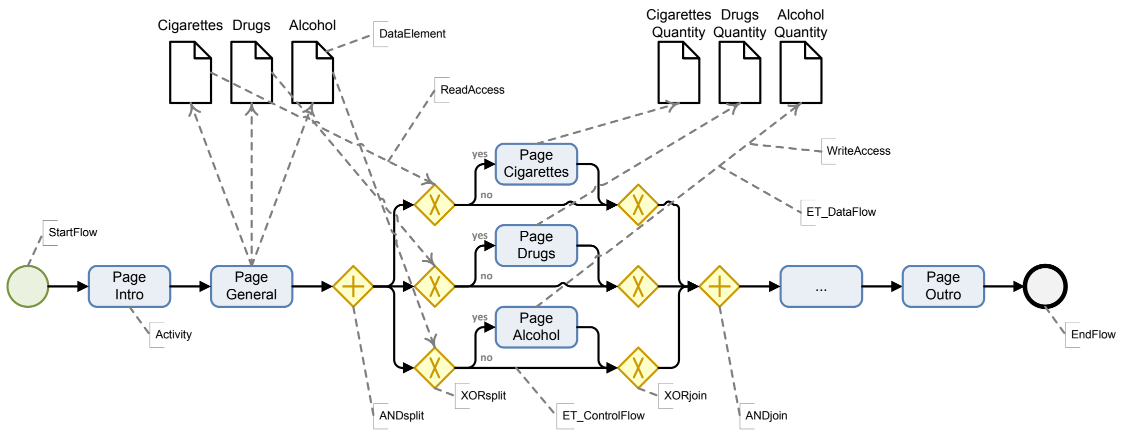


Figure 1.1: Example Business Process

At first an introduction page is shown. Then the subject answers the questions if he smokes cigarettes, takes drugs or drinks alcohol. His answers are stored in the *data elements Cigarettes*, *Drugs* and *Alcohol*. These data elements are then used to decide, whether he must answer questions regarding his daily consumption of the corresponding substance. In the end an conclusion page is shown.

In addition, *QuestionSys* contains a light-weight *process engine* to enact questionnaire models (i.e., business processes) on a smart mobile device.

1.1 Purpose of the Thesis

The next step in the lifecycle of such an electronic questionnaire is the analysis of the data stored in the data elements of the respective business process. Therefore, an application is needed, which enables domain experts to define rules, which can be used to evaluate a completed questionnaire.
This thesis covers the creation of rules to analyze the data collected using these electronic questionnaires. These rules should be created by domain experts, although they have little to no experience in boolean algebra.
A rule thereby is a boolean expression which can be evaluated by the application. In addition to this rule, texts in different languages can be defined, for when the rule is either fulfilled or is not (e.g., not all requirements for this rule are met). Depending on the language of the questionnaire chosen by the user, the text in the appropriate language is presented when evaluating the data collected. This thesis also looks for powerful concepts to extend this approach and evaluates these concepts. Rules should be built using a defined structure and the definition of such rules should be easy for an inexperienced user. Additionally, the domain experts should have the possibility to add new behavior in terms of the subsequent evaluation to rules. This is achieved by *user defined functions*, which are introduced in section 4.1.1 and presented in detail in section 5.2. Thereby, *Questionrule* can adapt to new circumstances. Two possible use cases for evaluating the data collected are presented in chapter 7.1.

1.2 Structure of the Thesis

The thesis is structured as follows: Section 2 covers necessary fundamentals, which are needed for the further course of this thesis. Subsequently, section 3 presents the functional and nonfunctional requirements for the *Questionrule* application. Section 4 discusses the concepts and the architecture used to implement *Questionrule*. Rules are defined, the use of grammars is explained and an overview of the architecture of *Questionrule* is given. Section 5 presents various implementation aspects of *Questionrule*. For example, ANTLR, which is a parser generator or the creation and integration of user-defined functions and the graphical rule editor. Section 6 looks at other rule editors and provides related work. Finally, section 7 discusses the features of *Questionrule*, concludes this thesis and provides an outlook on how to further extend the *QuestionSys* project.

2 Fundamentals

This section covers fundamental knowledge that serve as a basis for the further course of this thesis. Section 2.1 thereby introduces the *QuestionSys* system. Section 2.2 presents the Eclipse RCP Framework, which is used to realize the *QuestionRule* component developed in this thesis. Finally, section 2.3 covers formal grammars, which are used for rule validation.

2.1 QuestionSys

The *QuestionSys* system [Ulm] aims at covering the whole lifecycle of an electric questionnaire. This involves creating, distributing, enacting electronic questionnaires on smart mobile devices and even evaluating them.

The system itself is based on a process-driven approach [SSP+14]. This means a questionnaire is realized as a business process.

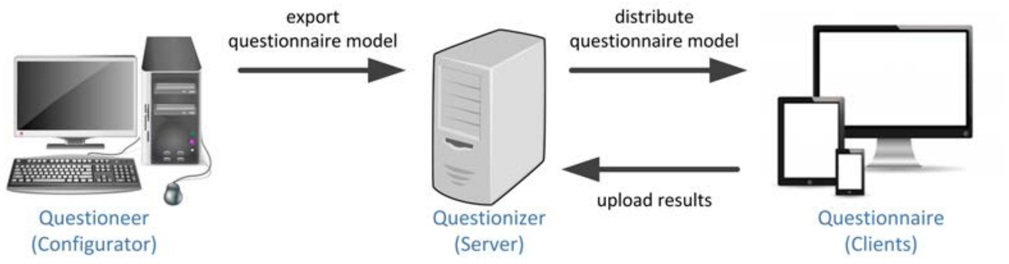


Figure 2.1: Architecture of QuestionSys

As of now *QuestionSys* consists of three main components (see figure 2.1): the configuration program *Questioneer*, the server-component *Questionizer* and the client application *Questionnaire*. The following sections present the components of *QuestionSys* in detail:

2.1.1 Questioneer (Configurator)

With the configurator *Questionneer* domain experts are able to create and manage questionnaires in multiple languages. Therefore they can create the different parts of a questionnaire (i.e., pages, questions, etc.) and define their order and dependencies. The questionnaire model is then stored on to the server-component *Questionizer* for further processing.

2.1.2 Questionizer (Server-Component)

The central server-component *Questionizer* stores the questionnaires created by domain experts and offers the possibility to distribute them to different clients for enactment. Furthermore, the server archives the result of the completed questionnaires. These results can then be evaluated on the server using rules defined by domain experts. The definition of these rules will be part of this thesis.

2.1.3 Questionnaire (Client Application)

Questionnaire is a client application running on smart mobile devices or web browsers, which is used to fill in the questionnaires. Since questionnaires are mapped to business processes, the client application contains a process engine [SSP$^+$14] to enact the process. In case the questionnaire is enacted using a normal web browser, a remote process engine is used. If a questionnaire is completed a client can evaluate the result or send the filled in questionnaire to the server for later evaluation.
The rule component *Questionrule*, which is presented in this thesis, will be added to the architecture of *QuestionSys* (see section 7.1 for more details).

2.2 Eclipse RCP Framework

Eclipse RCP is a framework for developing Rich Client Applications with Java [Vog13]. A Rich Client (also called fat, heavy or thick client) is a computer client, that provides its functionality independent of a central server. It often works with local data and contains business logic.
Historically, Eclipse RCP emerged from the Eclipse IDE [Ecl]. Many aspects and components of Eclipse IDE are general in nature and may be reused in other applications like, for example, the Workbench-Design of the user-interface or the extensible Plug-In system. All general parts of Eclipse IDE were extracted in 2004 and are released since Eclipse 3.0 as *Eclipse RCP*, allowing developers to use the benefits of Eclipse IDE when developing Rich Client Applications with Java.
In Section 2.2.1 the basic architecture of an application using the Eclipse RCP framework is presented. In Section 2.2.2 a brief overview of the Eclipse RCP features is given.

2.2.1 Basic Architecture of Eclipse RCP

The architecture of Eclipse RCP is composed of multiple layers (see Figure 2.2). The lowest layer consists of *OSGi*, *Equinox* and *EMF*. *OSGi* [OSG] is a specification which describes a modular approach for Java applications. The programming model of OSGi

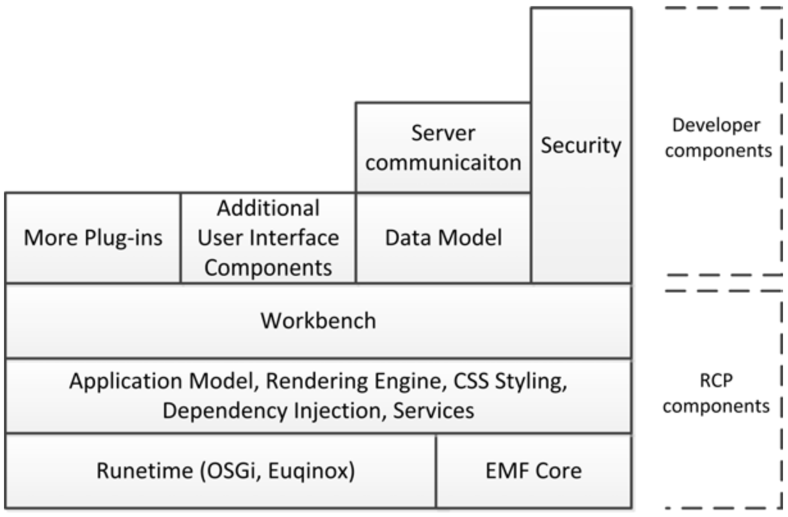


Figure 2.2: Eclipse RCP components

allows one to define dynamic software components. These components (also known as *bundles* in OSGi) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot of the application. *Equinox* is one implementation of the OSGi specification and is used by the Eclipse platform. The Equinox runtime provides the necessary framework to run a modular Eclipse application. *EMF* (Eclipse Modeling Framework) is a modeling framework code generation facility for building applications based on a structured data model.
The layer above consists of *Application Model*, *Rendering Engine*, *CSS Styling*, *Dependency Injection* and various *Services*, which can be used when programming an application. The Application Model is a logical model, which describes the structure of an application. It contains the visual elements (e.g., windows) as well as some non-visual elements (e.g., handlers) of the Eclipse RCP application. The Rendering Engine is responsible for generating the user-interface. CSS Styling enables Eclipse widgets to be configured via external (CSS like) files. With Dependency Injection the Eclipse RCP is able to implicitly create objects. This means, a developer can let the framework handle the object creation and doesn't need to create them himself. The next layer is the so called *Workbench*. It is an empty graphical application, which supports the basic concepts and interaction patterns of Eclipse RCP, like perspectives or menus. Eclipse RCP applications can expand this workbench to fit their specific needs. On top of these

components a developer can add his specific components, which are important for his Eclipse RCP application.

2.2.2 Features of the Eclipse RCP

Eclipse RCP has a lot of advantages when developing a desktop-application. It offers fully developed basic components for graphical applications, that have proven themselves in many use cases. The workbench of Eclipse RCP is a graphical user-interface and provides a consistent, sophisticated concept for operating with the user-interface and can be used by all Eclipse RCP applications. Therefore, end-users only have to familiarize the basic control of a Eclipse RCP applications one time. The basic structure for the design of the graphical user-interface is prespecified by Eclipse RCP and doesn't need to be developed first. Eclipse RCP is consistently designed for modularization and extensibility. This is an advantage especially for bigger applications, as they must be split up in small modules to remain manageable. Extensions (e.g., Plug-Ins) on the basis of Eclipse RCP can harmonize without knowing each other. Moreover, a big application can be broken down in several Plug-Ins, which are developed separately. Later the Plug-Ins can be coupled to create the big application. Nowadays, there are a lot of providers for extensions, tools, support and training for Eclipse RCP.
Of course, the Eclipse RCP has some drawbacks. A developer is forced into a straitjacket, because the basic structure of an Eclipse RCP application is prespecified. The basic structure is dictated by the application model. For example, the application model only allows nesting of certain user-interface elements. Another point is, that the developer needs to learn the ropes at first.
Eclipse RCP was selected for the development of *Questionrule*, because the advantages outweigh the drawbacks. On the one hand the adaptation to the structure of an Eclipse RCP application was necessary, but on the other hand made Eclipse RCP the development a lot easier and saved a lot of time during work. In addition, Eclipse RCP allows for a great amount of extensibility in the future of *Questionrule*.

2.3 Formal Grammars

Formal grammars are mathematical models, which define and describe *formal languages*. A formal grammar consists of a set of rules for rewriting strings and a *start symbol* from which rewriting starts. Therefore, a grammar is usually regarded as a language generator, but it can also be used as a language *recognizer*. This is a function that determines whether a given string belongs to the language (i.e., the string is grammatically correct) or not.
To create a new string with a formal grammar one starts with the *start symbol* S and continues to apply *production rules* of a specific rule set P until strings only contain *terminal symbols*. A production rule is applied by replacing one occurrence of production rule's left-hand side in the string by that production rule's right-hand side. This process is called *derivation*.
The vocabulary of a formal grammar consists of terminal symbols Σ and nonterminal symbols V and specifies which symbols can be used for derivation. The set of terminal symbols defines which characters of a word can not be derivated. Words, which only consist of terminal symbols, define the language, which is described by the formal grammar. A production rule is a tuple (α, β), which can also be written as $\alpha \to \beta$. The production rule is applied to a word $\omega \in (V \cup \Sigma)^*$ by simply replacing every occurrence of α with β. There can be multiple production rules for the same α with $\alpha \to \gamma$ and $\gamma \neq \beta$.
In the following, a definition of a formal grammar is given [Sch08]:

A formal grammar is a tuple $G = (V, \Sigma, P, S)$ whereas:

- V, is the finite set of nonterminal symbols
- Σ, is called alphabet and the elements are called terminal symbols
- $P \subset ((V \cup \Sigma)^+ \times (V \cup \Sigma)^*)$ is a finite set of production rules
- $S \in V$ start symbol

An example grammar is shown in Table 2.1. This grammar consists of four productions rules, whereas one contains the start symbol **S**. **A**, **B S** are so called nonterminal symbols, whereas c and d are terminal symbols. An example word of this formal language would be $ccddcc$.

S $\rightarrow$ **B**
A $\rightarrow c$
B $\rightarrow$ **ASA**
B $\rightarrow dd$

Table 2.1: Example Grammar

To derivate $ccddcc$, one starts with the production rule which contains start symbol **S** and obtains **B**. Subsequently, one applies production rule **B** $\rightarrow$ **ASA** to **B** and receives **ASA**. Now one applies production rule **S** $\rightarrow$ **B** and obtains **ABA**. After applying production rule **B** $\rightarrow$ **ASA** one receives **AASAA**. The next step is to apply production rule **S** $\rightarrow$ **B** to receive **AABAA**. After applying **B** $\rightarrow dd$ and **A** $\rightarrow c$ one receives $ccddcc$. Note, that all occurrences of **A** and **B** has to be replaced at once!
As mentioned before, a formal grammar can also be used to determine if a given word belongs to a language, which is described by a formal grammar. The word $ccddc$ is not part of the language described by the grammar in table 2.1. The reason for this is that the production rule **B** $\rightarrow$ **ASA** adds the same amount of **A** with each appliance of the production rule. Therefore the number of **A**s on the left hand side has to be the same as the number of **A**s on the right hand side.
Formal grammars can be assigned to classes, which are defined by similarities. The best known classification is the *Chomsky hierarchy* [Cho56]. The *Chomsky hierarchy* groups formal grammars depending on the kind of productions rules in classes Type-0 to Type-3. The following list shows the different requirements for the classes:

- Type-0: no restrictions
- Type-1 (context-sensitive grammars): $\forall\ (\omega_1 \rightarrow \omega_2) \in P\ :\ \mid \omega_1 \mid \leq \mid \omega_2 \mid$
- Type-2 (context-free grammars): $\forall\ (\omega_1 \rightarrow \omega_2) \in P\ :\ \omega_1 \in V$
- Type-3 (regular grammars): $\forall\ (\omega_1 \rightarrow \omega_2) \in P\ :\ \omega_2 \in \Sigma\ \cup\ \Sigma V$

Type-1 means, that the number of symbols on the right hand side of a production rule has to be at least as big as the number of symbols on the left hand side. In addition to the restriction of *Type-1*, consists the left hand side of a production rule in *Type-2* only of one nonterminal symbol. In addition to the restriction of *Type-2*, consists the right hand side of a production rule in *Type-3* only of either a single terminal symbol or a terminal symbol followed by a nonterminal symbol.
The explanation of the types hints the relation between the different types. The Chomsky hierarchy is a containment hierarchy as can be seen in figure 2.3. That means each more restricted type is a proper subset of the less restricted type.

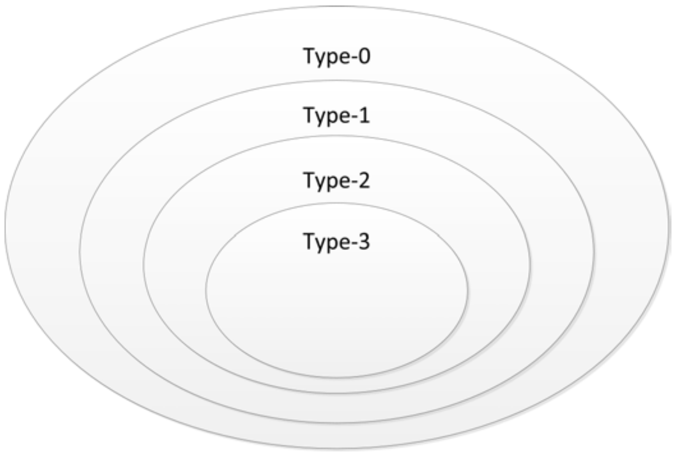


Figure 2.3: Chomsky Hierarchy

A subset of the languages described by context-free grammars (Type-2) is the theoretical basis for the syntax of most programming languages. Regular grammars describe so called *regular languages*, which are often used for search patterns and for describing the lexical structure of programming languages.
Formal grammars are an important part of *Questionrule*. They enabled an easy rule validation and are the theoretical basis for the graphical rule editor, which is presented in section 5.3 of this thesis.

3 Requirements

This section presents the requirements for *Questionrule*. Section 3.1 introduces the functional requirements and section 3.2 discusses nonfunctional requirements.

3.1 Functional Requirements

Functional requirements define what a system is supposed to accomplish. The functional requirements of *Questionrule* are presented in the following:

FR1 Create Projects It should be possible to create different projects within the application. It consists of rules and variables from the questionnaire model.

FR2 Load Questionnaire Model When creating a rule, it should be possible to choose the questionnaire template and load it in the project.

FR3 Delete Project It should be able to delete a project. If a project is deleted all corresponding variables and rules should be deleted as well.

FR4 Replace Questionnaire Model It should be possible to change the questionnaire model of a project, after the project has been created.

FR5 Create Rule Domain experts should be able to define new rules. The definition should be on the one hand easy and intuitive and on the other hand shouldn't complicate the creation of more complex rules. Using the rule editor should result in a correct rule (i.e., following a defined structure and grammar)

FR6 Edit Rule Rules should be editable.

FR7 Copy and Paste Rule A rule should be copyable to another project.

FR8 Delete Rule A rule should be deletable.

FR9 Export Rule The domain expert should be able to transfer rules to the server *Questionizer*.

FR10 Exchange Rule Domain experts should be able to exchange created rules.

FR11 Validation of Rules Rules should be checked for their validity (e.g., follow the defined grammar, use known variables).

FR12 Edit Variable Loaded variable should be editable and their "Meta" informations should be editable.

FR13 Display of Errors and Warnings If something unexpected happens, *Questionrule* should display warnings and errors in a proper way.

The rule creation (**FR5**) is very important, because domain experts usually have little or no experience in boolean algebra. Therefore a graphical rule editor should be created, to help the domain experts to get started. One possible solution for this issue is presented in section 5.3.

3.2 Nonfunctional Requirements

Nonfunctional requirements defines the characteristics of a system. The nonfunctional requirements of *Questionrule* are as follows:

NR1 **Programmatic Extensibility** It should be easy for a software developer to add new functionality to *Questionrule*.

NR2 **Rules Extensibility** A domain export should be able to add more functionality to the rules and the evaluation of rules.

NR3 **Reliability** The system functions in *Questionrule* should be mature and *Questionrule* should have a high tolerance for errors.

NR4 **Usability** *Questionrule* should be comprehensible and easy to learn. The operation should be intuitive and clear. Wizards should guide the domain experts in using the functions of *Questionrule*.

NR5 **Portability** The installation process of *Questionrule* should be clear. The port to another system should be no problem.

NR6 **Provide Detailed Errors and Warnings** If an error occurs (i.e., not connected to the server) the user should get informed by displaying meaningful error- or warning-messages.

NR7 **Platform Independent** *Questionrule* should provide its functionality independent of the platform used on.

Especially, nonfunctional requirement **NR2** is important, as domain experts must add functionality to the rule evaluation depending on their respective use cases. Therefore it is crucial to provide the possibility to easily create new functions. To cope with this challenge a solution should be designed to add new functions in an easy way and use them within *Questionrule*. One solution to achieve this goal is presented in section 5.2.

4 Concept and Architecture

This chapter presents the concepts as well as the architecture, which were used for implementing *Questionrule*. The concepts introduced in section 4.1 are more general and serve as a theoretical foundation for *Questionrule*. Section 4.2 presents the general structure and the individual components of the implementation in more detail.

4.1 Concept

This section presents the different concepts, which where used for the design of the rule component. In section 4.1.1 *rules* are defined and *functions* are introduced. Subsequently, section 4.1.2 presents the workspace and projects. Furthermore, section 4.1.3 explains the use of formal grammars in *Questionrule* for validity checks of rules.

4.1.1 Rules

A rule consists of a number of comparisons, which are connected using boolean operators *AND* (&&) and *OR* (||). Comparisons consists of two operands and a single operator in between. An operand thereby either is a constant or a variable. Constants are simple data types like strings, floats, integers or booleans. The actual value for a variable is gathered when enacting a questionnaire (i.e., they are data elements within the business process). If the questionnaire is mapped to a business process the data elements of this business process are used to store the given answers of the user. That means a rule can trigger on a certain combination of answers. Figure 4.1 shows an example of a rule with further annotations.

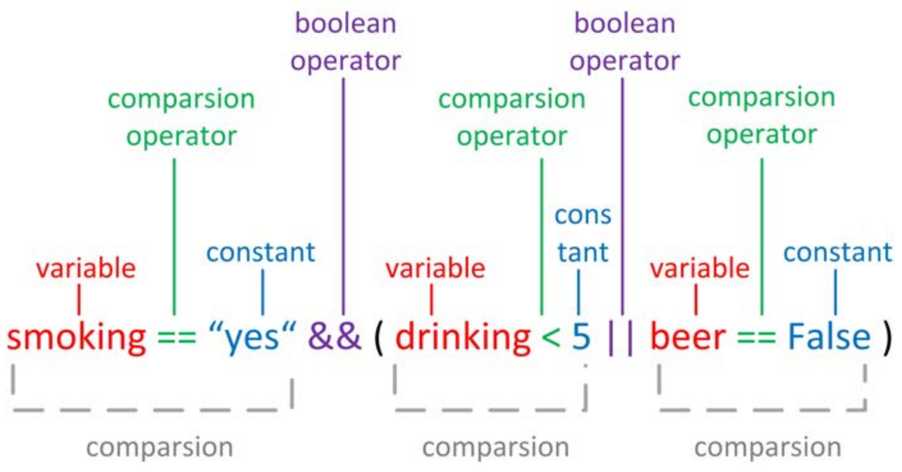


Figure 4.1: Example for a Rule

The given example consists of 3 comparisons which are connected with two boolean operators. `Smoking`, `drinking` and `beer` are variables and `"yes"`, `5` and `False` are constants.
To allow domain experts to create more complex rules, an operand can also be a *function*. A function is created by the domain expert and adds a specific behavior in terms of the subsequent evaluation to rules. Figure 4.2 shows an example for a rule using a user-defined function.

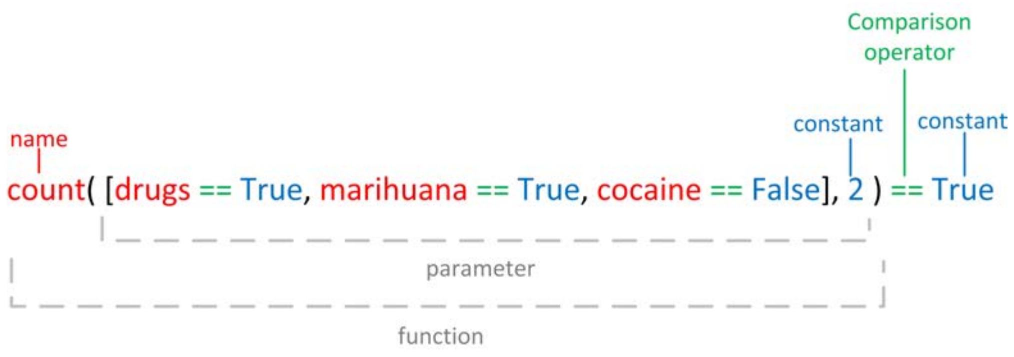

Comparison operator
constant
constant
name
count([drugs == True, marihuana == True, cocaine == False], 2) == True
parameter
function

Figure 4.2: Example of a Rule containing a Function

The function is called `count` and the input parameters are an array of comparisons and an integer. The function, for example, checks if a certain number (in this example, `2`) of comparisons are fulfilled, it doesn't matter which. If so, the function will return `true`. If the specified number of comparisons is not fulfilled, the function will return `false`.
To create a new user-defined *function* a lightweight framework is provided, which allows the domain expert to implement his own logic. This framework enables *Questionrule* to load all the user-created *functions* in a consistent way. During the evaluation of a rule, the user-created functions are ran and then replaced with their return value. By doing so, a comparisons of two constants is created, which can be easily evaluated as described before.
The concept of these user-defined *functions* adds another level of extensibility to *Questionrule*. This was already described as nonfunctional requirement **NR2** in section 3.2.
The data model of a rule is defined in a XSD-file and the rules are stored in XML-files following this schema. By using XML to store the rules, the output is human-readable. Furthermore, the XSD-file is used to automatically created the needed classes for the implementation. More about XSD and XML can be found in [W3Cb] and respectively in [W3Ca]. Additionally, *DAO*s (Data Access Object) are used, which encapsulate a data source (here XML-file) and the available methods for the data. The data is only accessed through the DAO itself and not manipulated directly. This approach ensures the changeability of the application, as only the XSD needs to be adapted to the new requirements without changing the underlying implementation.

4.1.2 Workspace of *Questionrule*

The projects which a domain expert creates are stored in the workspace of *Questionrule*. Upon creation, each project is assigned a unique name and a questionnaire model. A project consists of the created rules and the data elements of a questionnaire. The latter are extracted from the given model and added to the variable store. After the project is successfully created, the domain experts may create rules to allow for an electronic evaluation. Rules can be *published* and *exported*. *Publishing* rules means, to export all selected rules into a single file and send it to the server using Web Services. *Exporting* rules, however, means, to export each selected rule into a different file for an easy exchange of rules between users. The *Publishing Rules* functionality covers functional requirement **FR9**, while the *Export Rules* functionality covers function requirement **FR10**.

4.1.3 Formal Grammar

We defined a formal grammar, which describes the structure of the rules in section 4.1.1. This formal grammar provides an easy way to check the rules for validity in many different ways. Aside from general syntax errors (e.g., a comparison consists of only one operand) it is checked if a variable used in a rule is also defined in the questionnaire model for this project. This means in other words, there exists a data element within the business process with the respective variable name. Furthermore, if the variable exists, it is checked if the data type of the other operand in the comparison suits the type of the variable. This check can be done easily, as the data types are defined in the formal grammar. If a *function* is used within the rule it is checked if the respective *function* is successfully loaded in *Questionrule*. Therefore, the application checks, whether there exists a *function* with the name defined. As with the variable, it is checked if the data type of the other operand suits the function's return type.
The next step is to adopt the presented concepts in the architecture of the rule component. Therefore, the architecture of *Questionrule* is presented at first and the concrete implementation of these concepts will be shown in a later section.

4.2 Architecture

This section introduces the overall architecture of *Questionrule*. Section 4.2.1 covers the *Model-View-Presenter* software design pattern, which is mainly set through the use of the Eclipse RCP framework used for the development. Section 4.2.2 shows an overview of the architecture of *Questionrule*. The individual components of the structure of *Questionrule* are presented in sections 4.2.3 to 4.2.8.

4.2.1 Model-View-Presenter

Model-View-Presenter (MVP) [Pot96] is a software design pattern, which emerged from the widely used *Model-View-Controller (MVC)* [KP$^+$88] pattern. It describes a new approach to completely isolate the model and the view and connect them with a presenter. The advantage over MVC is a stricter separation of the individual parts resulting in a better testability of the application. Figure 4.3 shows the structure and dependencies of MVP.

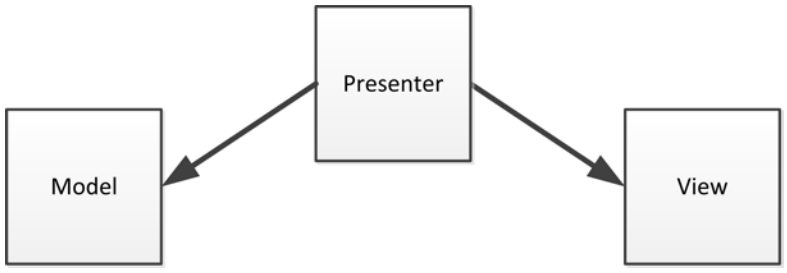


Figure 4.3: MVP Structure

MVP consists of 3 parts: *model*, *view* and *presenter*. They are discussed in the following:

Model The *model* represents the business logic of the application. That means it provides all functionality to run the *view*. The *presenter* alone, however, controls the *model*. The *model* is completely isolated and doesn't know either *view* or *presenter*.

View The *view* doesn't contain any controlling logic and is only responsible for displaying the data and providing data input and possibilities for manipulating the data. Neither does it access the functionality of the *presenter* nor the *model* itself. The *view* is controlled only by the *presenter*.

Presenter The *presenter* is the connector between *model* and *view*. It controls the logical activities between both other parts and ensures that the *view* can display the correct data.

To fully benefit from the advantages from MVP over MVC, *interfaces* for both *view* and *model* are used. The interfaces define the access and methods to both parts, while the *presenter* connects to these interfaces. This ensures a complete replaceability and reusability of both *model* and *view*. The *view* or the *model* can be replaced without the need to change the other two parts. The new part only needs to implement the specified interface.
Another concept, often used in conjunction with MVP, is the *event bus*. The event bus solves the problem, how presenters interact among themselves. *Presenters* can fire *events* onto the event bus and register on the event bus, to get informed if a certain event occurs. That means a *presenter* can interact with other *presenter* by publishing events to the event bus. The latter notifies *presenters*, which have subscribed for this event. This approach allows a very flexible interaction between *presenters* and the replaceability of single presenters. A new *presenter* needs to listen for specific events it is interested in and implement the interaction between the corresponding *view* and *model*.

4.2.2 Architecture of *Questionrule*

Questionrule uses the *Eclipse RCP framework* as solid foundation and is composed of different managers. The managers are structured in layers and build upon each other as shown in figure 4.4. The layer, which consists of the *Project Manager* and the *Questionnaire Model Manager* takes care of the import and export of the questionnaires and rules. They pass the data to the *Validation Manager*, *Rule Manager* and *Variable Manager*. In the next layer the data is further processed and may be edited by the domain expert working with the application. The *User Interface* and the *Web Services*, which are used to communicate with the domain expert and the server component *Questionizer*, are build on top of these managers.

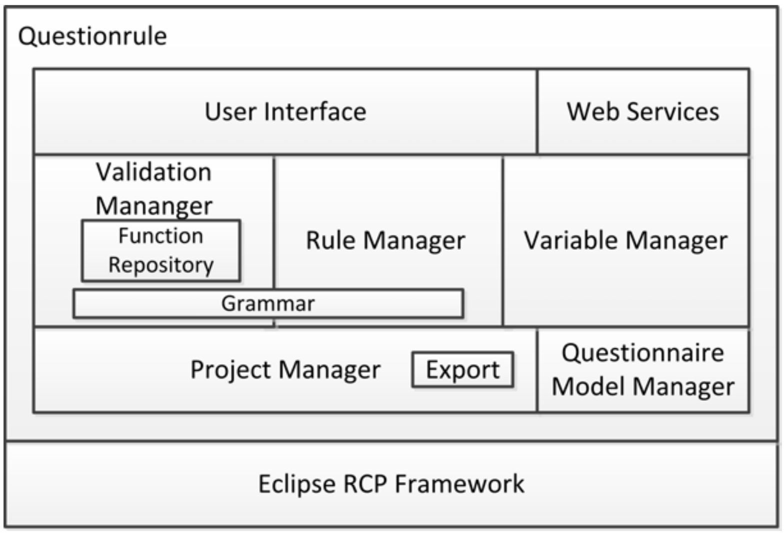


Figure 4.4: Architecture of *Questionrule*

The different managers will be presented in detail in the following sections. Thereby, they will be presented in a bottom to top order. The different task of the managers will be explained and the connection to other managers will be pointed out.

4.2.3 Project Manager

The *Project Manager* handles the workspace of *Questionrule* and everything related to the projects within the workspace. This includes loading existing projects, creating new ones or deleting old projects. Furthermore, it allows for changing the questionnaire model of an existing project, defining new rules and exporting rules.
When starting *Questionrule* the *Project Manager* loads all existing projects which are stored in the workspace of *Questionrule* and presents them to the domain expert. When creating a new project, a domain expert must choose a name for the project and select a questionnaire model using a wizard. Thereupon, the *Project Manager* creates all relevant folders, forwards the questionnaire to the *Questionnaire Model Manager* and inserts the received variables into the created project. Upon deleting a project, the *Project Manager* simply removes the project folder with all content. Flexibility is an important aspect for business processes. Reichert, Dadam and Weber have shown in [RD09, RW12], that business processes must be adaptable to changing requirements.

Therefore a questionnaire model in a project may be replaced. When changing the questionnaire model (e.g., because a new version is available), all variables must be reloaded. Of course, existing rules must be reevaluated, as variables could be deleted. This would result in invalid rules (i.e., the rule is not satisfiable any more). When creating a new rule, the *Project Manager* creates a new empty rule (XML-file) in the rule folder of the project. To copy or paste a rule, the *Project Manager* reads in the rule and uses the export to create the rule in another project.
The export component within the *Project Manager* is divided into two functionalities: *exporting* and *publishing*. For the export of rules a domain expert selects the desired rules in a wizard and sets a location folder to save the exported rules. The *Project Manager* then creates for every rules a new file containing the rule. When publishing the rules all selected rules are merged into one single file, which can then be used to send it to the server component *Questionizer*.
The *Project Manager* covers function requirements **FR1** - **FR5**, **FR7**, **FR9** and **FR10** as described in section 3.1.

4.2.4 Questionnaire Model Manager

The *Questionnaire Model Manager* provides the functionality to read questionnaire models. This is required for the *Project Manager*, *Variable Manager* and *Validation Manager*. When a new project is created, the *Project Manager* sends the questionnaire model to the respective manager. The *Questionnaire Model Manager* extracts the required information (e.g., data elements used in the model) from the underlying business process, which represents answers for questions. This business process is modeled in Aristaflow, which is an implementation of the ADEPT2 concept [DR09]. These data elements are then converted to an internal format and are now called *variables*. Variables consists of the properties of the data elements derived from the questionnaire model and additional user-defined properties (e.g., a custom description providing additional information for the domain expert). Internally, *Questionrule* works with these variables. The conversion into this internal format is important, because a later change of the questionnaire model format shouldn't result in massive changes of *Questionrule*. The variables, which are extracted from the questionnaire model are sent back to the *Project*

Manager. The *Project Manager* stores them the current projects folder.
When checking for validity of a rule, the *Validation Manager* asks the *Project Manager* for the variables within the rule. Then the *Validation Manager* performs validation checks for these variables. The *Variable Manager* receives the variables from the *Project Manager* to present them to the domain expert and let the domain expert edit the variables.
The *Questionnaire Model Manager* covers the functional requirement **FR1**, **FR2** and **FR4**.

4.2.5 Variable Manager

The *Variable Manager* enables the domain expert to edit variables and provide additional information. For this purpose, the *Variable Manager* receives the variables from the *Questionnaire Model Manager*. The domain expert can only add new properties to variables but can not change the original properties extracted from the questionnaire model (e.g., the data type of the variable). The reason for this is that, the domain expert could create inconsistencies otherwise (e.g., change name of a variable to a non existing data element).
The *Variable Manager* covers functional requirement **FR12**.

4.2.6 Rule Manager

The *Rule Manager* receives rules from the *Project Manager* to display them to the domain expert. In addition, it also enables the domain expert to edit and delete a specific rule. In case the domain expert deletes a rule, the XML-file is removed from the project. When saving changes after editing, the *Rule Manager* overwrites the existing rule in the rule folder of the project with the new one. The *Rule Manager* also sends the rule to the *Validation Manager*, where it is checked for validity. To do so, a grammar is used for a graphical representation of a rule, which can be defined using a rule editor. This grammar also defines the structure of the rules created with the graphical rule editor and implements the *correctness by construction* principle. This concept means, that a domain expert is not able to create syntactic wrong rules. Domain experts need to be supported and should only be allowed to use certain functions. Thereby, domain

experts can make less mistakes. The *Rule Manager* also receives the name of all functions available from the *Validation Manager* as well as the name of all variables from the *Project Manager* for the graphical rule editor. Thereby, only existing variables and functions can be used when defining a new rule.
The *Rule Manager* covers functional requirement **FR6** and **FR8**.

4.2.7 Validation Manager

The *Validation Manager* checks if a given rule is valid. Therefore it receives the rule from the *Rule Manager* and the variables of the project from the respective manager. The *Validation Manager* checks for the syntax of the rules, which is defined in the grammar. In addition it checks if a variable used in the rule is existent in the project and if the data type of the variable suits respective the comparisons. Furthermore, it is also checked if the used functions exist. All known functions are stored in the *Function Repository* of the application. When starting *Questionrule*, the *Validation Manager* reads all functions from the Function Repository. To check the existence of a function, the *Validation Manager* uses the imported functions. It is also checked, if the data type of the function's result suits the comparison.
The *Validation Manager* covers functional requirement **FR11** and **FR13**.

4.2.8 Web Services

The *Web Services* are responsible for the communication with the external server component *Questionizer*. These Web Services cover functional requirement **FR9**. An introduction to Web Services can be found in [CDK+02].

4.3 Conclusion

These concepts form a basis for the functionality of *Questionrule*. The architecture enables programmatic extension and splits *Quesitonrule* in multiple components. Each component has its own distinct task and communicates with other components.

5 Implementation Aspects

This section presents selected highlights of the *Questionrule* application. Section 5.1 introduces ANTLR, which is a parser generator. It is used for defining a grammar, which describes the structure of rules. From this grammar, ANTLR creates a parser. Section 5.2 present the approach for dynamically loading user-defined functions to extend the overall functionality of the rules. The graphical rule editor is shown in section 5.3.

5.1 ANTLR

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files [Par13]. It's widely used to build languages, tools and frameworks. From a so called grammar, ANTLR generates two classes: lexer and parser. The lexer runs first and splits the

input into different pieces, the so called tokens. Each token represents a piece of input (e.g., a variable or a constant). The stream of tokens is passed to the parser, which builds and walks parse trees, interprets the code or translates it into some other form. A grammar file contains all required information ANTLR needs to generate the lexer and corresponding parser. Most importantly, this grammar file describes how to split the input into the different tokens and how to build the tree from the derived tokens. In other words, the grammar file contains lexer rules and parser rules. The defined grammars are context-free (see section 2.3).
In this thesis ANTLR was used to define a grammar which describes the structure of rules which can be generated using *Questionrule*. This grammar is shown in listing 5.1.

Listing 5.1: Grammar to define the Structure of Rules

```
grammar BooleanRules;

//Lexer rules
//Operators
OPERATOR: '+' | '-' | '*' | '/' | '%';
EQOP:  '==' | '!=' | '<' | '>' | '>=' | '<=';
BOOLOP: '&&' | '||';

//Data Types
BOOL: 'True' | 'False';
IDENTIFIER: LETTER (DIGIT | LETTER)*;
fragment DIGIT: [0-9];
fragment LETTER: [a-zA-Z];
INT: DIGIT+;
FLOAT: DIGIT+'.'DIGIT+('d'|'D');
STRING: '"'IDENTIFIER'"';

//Braces
BRACKOPEN: '('; BRACKCLOSE: ')';
ARROPEN: '['; ARRCLOSE: ']';
```

```
SEPARATOR: ',';

WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+ -> skip ;

//Parser Rules
//entry-point
start: boolexpression;

//rule serves as a chaining or nesting of comparison
boolexpression: BOOL | BRACKOPEN boolexpression BRACKCLOSE |
boolexpression BOOLOP boolexpression |
BRACKOPEN test BRACKCLOSE | test;

//a comparsion consists of relational operator and a expression
test: functioncall | expression EQOP expression ;

expression: IDENTIFIER | functioncall | INT | BOOL |FLOAT |
STRING | BRACKOPEN expression BRACKCLOSE |
expression OPERATOR expression;

functioncall : IDENTIFIER BRACKOPEN paramlist BRACKCLOSE;

paramlist: param | param SEPARATOR paramlist;

param: array | test | expression;
//Arrays
array: ARROPEN arrlist ARRCLOSE;

arrlist: arrelement | arrelement SEPARATOR arrlist;

arrelement: test | expression;
```

The grammar starts with the lexer rules. Each lexer rule (e.g., `INT: DIGIT+;`) describes one token. Lines 5-7 defines the operators for comparisons. Next, the data types (e.g., `BOOL: 'True' | 'False';`) are defined in lines 10-16. The `IDENTIFIER` in line 11 acts as name for a variable or function, whereas `STRING` is an actual constant, which may be used for comparison. Digits and letters (line 12 and 13) are defined as a `fragment`. Thereby they can be used in lexer rules, which simplifies the grammar and makes it more readable. The lines 19-21 define the braces and separators. Line 23 tells the lexer to skip whitespaces and line breaks.

Next, the parser rules are listed, which describe the structure of the rules for the evaluation of data collected. The appliance of these parser rules is the same as presented in section 2.3. The `start` rule is the entry point for the parser. A `boolexpression` either consists of a boolean constant (`BOOL`), an embraced boolexpression (`BRACKOPEN boolexpression BRACKCLOSE`), a comparison of two boolexpressions (`boolexpression BOOLOP boolexpression`), an embraced `test` (`BRACKOPEN test BRACKCLOSE`) or a normal `test`. A `test` thereby is a simple comparison, which consists of either a function (`functioncall`) or a comparison of two expressions (`expression EQOP expression`). An `expression` is a variable name (`IDENTIFIER`), a function (`functioncall`), a simple data type (`INT | BOOL | FLOAT | STRING`), an embraced expression (`BRACKOPEN expression BRACKCLOSE`) or a comparison of two expressions (`expression OPERATOR expression`). A `functioncall` consists of the name of the function (`IDENTIFIER`) and the list of parameters (`paramlist`). The `paramlist` is composed of one or multiple parameters (`param`). A `param` may be an `array`, a `test` or an `expression`. The `array` has a list of arrays (`arrlist`), which consists of at least one element (`arrelement`). An array element is either a `test` or an `expression`.

Figure 5.1 shows the example rules of section 4.1.1 annotated with various parts of the grammar.

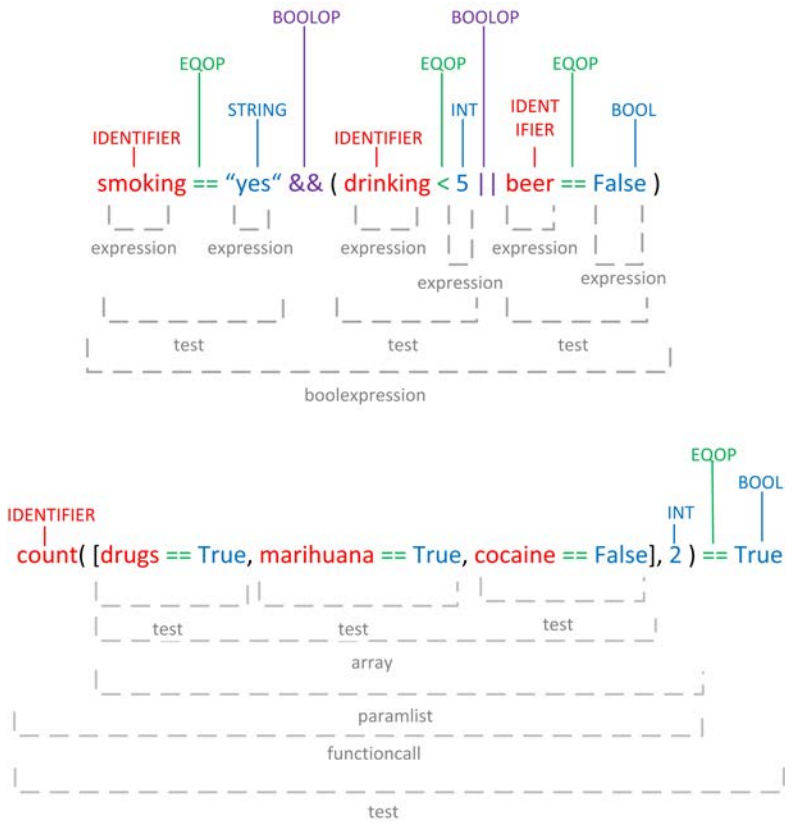


Figure 5.1: Grammar Examples

With this grammar, ANTLR creates the following classes, which are described in the further course of this chapter.

- BooleanRulesBaseListener.java
- BooleanRulesLexer.java
- BooleanRulesListener.java
- BooleanRulesParser.java
- BooleanRules.tokens
- BooleanRulesLexer.tokens

The most important class is `BooleanRulesBaseListener`. It implements the interface `BooleanRulesListener`, which consists of methods presented in listing 5.2. The methods are called when the parse tree of a rule is traversed and a specific element of the grammar is found.

Listing 5.2: Excerpt of BooleanRulesListener

```
/**
* Enter a parse tree produced
* by {@link BooleanRulesParser#expression}.
* @param ctx the parse tree
*/
void enterExpression(@NotNull BooleanRulesParser.
   ExpressionContext ctx);
/**
* Exit a parse tree produced
* by {@link BooleanRulesParser#expression}.
* @param ctx the parse tree
*/
void exitExpression(@NotNull BooleanRulesParser.
   ExpressionContext ctx);
/**
* Enter a parse tree produced
* by {@link BooleanRulesParser#test}.
* @param ctx the parse tree
*/
void enterTest(@NotNull BooleanRulesParser.TestContext ctx);
/**
* Exit a parse tree produced
* by {@link BooleanRulesParser#test}.
* @param ctx the parse tree
*/
void exitTest(@NotNull BooleanRulesParser.TestContext ctx);
```

Furthermore, a class `BooleanRulesListenerImpl` was automatically created, which extends `BooleanRulesBaseListener` and implements selected methods like `enterExpression(@NotNull BooleanRulesParser.ExpressionContext ctx)`. These methods are used to check the validity of a rule. Listing 5.3 shows an excerpt of `BooleanRulesListenerImpl`, which uses the `enterExpression` method to check if a variable used within the given rule exists in the project (i.e., exist in the corresponding questionnaire model).

Listing 5.3: Excerpt of BooleanRulesListenerImpl

```
1  public void enterExpression(@NotNull BooleanRulesParser.ExpressionContext ctx){
2          boolean matchingVariableFound = false;
3          if(ctx.IDENTIFIER() != null){
4          //checks if the variable is part
5          //of the list of all variables
6          for(int i=0;i<listOfInternalDataElements.size();i++){
7                  if(ctx.IDENTIFIER().toString().equals(listOfInternalDataElements.get(i).getName()))
8                          matchingVariableFound = true;
9          }
10         if(!matchingVariableFound)
11                 warningList.add(new RuleWarning("The variable named" + ctx.IDENTIFIER() + " doesn’t exist!", ErrorCodes.QuestionRule_Variable_DoesntExit));
12         }
13 }
```

Therefore, `BooleanRulesListenerImpl` receives all variables in a list `listOfInternalDataElements` and checks if the name of variable used in the rule exists in the list of all variables (line 6-9). If so, it creates a warning which can be

displayed within the user-interface of the application (line 10-12).
The implementation of the validation of a rule works as follows: The `BooleanRulesLexer` imports the rule and creates the tokens. Then the `BooleanRulesParser` creates a parse tree for the rule with the help of the tokens. This parse tree is given to a `ParseTreeWalker`, which traverses the tree and runs methods implemented in `BooleanRulesListenerImpl`. After the `ParseTreeWalker` is finished, the list of occurred errors can be received and may be displayed in the user-interface, to present them to the user.
An important fact of our grammar is, that common data types are defined. For example, when checking the data type of a variable for a comparison, no Java mechanism are needed, as only the data type of the variable has to be compared to the one of the constant.

5.2 Adding User-Defined Functions to Rules

Adding user-defined functions to rules offer domain experts the possibility of evaluating these rules in a new way. To do so, a framework, which allows to create their own function is provided to domain experts. In practice, this framework is a *JAR-File*, which must be added to a new Java-Project. This JAR-File contains mostly *Interfaces*, which must be implemented by a software developer. Figure 5.2 shows the content of the *Function Template JAR-File*.

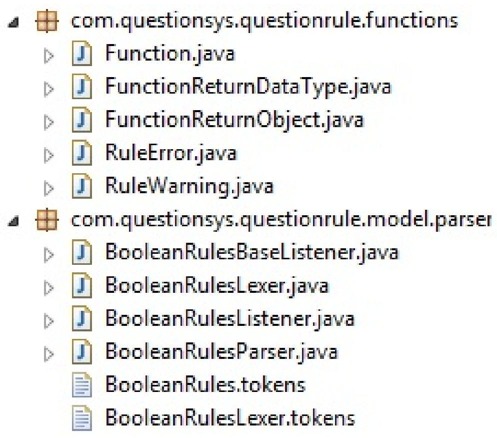

```
com.questionsys.questionrule.functions
    Function.java
    FunctionReturnDataType.java
    FunctionReturnObject.java
    RuleError.java
    RuleWarning.java
com.questionsys.questionrule.model.parser
    BooleanRulesBaseListener.java
    BooleanRulesLexer.java
    BooleanRulesListener.java
    BooleanRulesParser.java
    BooleanRules.tokens
    BooleanRulesLexer.tokens
```

Figure 5.2: JAR-File (Framework) for creating User-Defined Functions

This JAR-File is divided in 2 packages: The `functions` package contains everything which is directly related to the function, whereas the `parser` contains all classes generated by ANTLR (see section 5.1).
The interface `Function` is the most important part of the `functions` package as it contains all methods, which must be implemented later. The interface is shown in Listing 5.4.

Listing 5.4: Interface Function

```
package com.questionsys.questionrule.functions;
/**
* defines all methods for user-defined functions
*/
public interface Function {
/**
* Returns the name of the function
* @return name of the function
*/
public String getIdentifier();

/**
* evaluates the function
* @param ctx function to evaluate
* @return String name of function
*/
public FunctionReturnObject eval(FunctioncallContext ctx);

/**
* checks input function for errors. If function isn't valid
* the method returns a list of errors. If it is valid
* the list is empty
* @param ctx
*/
public ArrayList<RuleError> checkForErrors(BooleanRulesParser.
    FunctioncallContext ctx);

/**
* checks input function for warnings. If function isn't valid
* the method returns a list of warnings. If it is valid
* the list is empty
* @param ctx
*/
public ArrayList<RuleWarning> checkForWarnings(
    BooleanRulesParser.FunctioncallContext ctx);

/**
* returns the FunctionReturnObject, where the value
* and type is stored
* @return
*/
public FunctionReturnObject getFunctionReturnObject();
}
```

In case, the domain experts wants to create a new function, this interface has to be implemented. The method `getIdentifier()` returns the name of the function. This method is called, when the *Validation Manager* checks if a function used within a rule exists in the function repository. The method `eval(FunctioncallContext ctx)` evaluates the function. The function is passed as a data type of ANTLR (`FunctioncallContext ctx`) to the method and not as a simple String. Thereby the structure of the grammar can be used (e.g., iterate array elements of an array) for evaluation and no complex string manipulation operation are needed. The domain expert needs to engage in ANTLR, but the actually evaluation is a lot easier with the ANTLR data type in contrast to a simple String. The methods `checkForErrors(BooleanRulesParser.FunctioncallContext ctx)` and `checkForWarnings(BooleanRulesParser.FunctioncallContext`

`ctx)` return errors or warnings, which happened during the check for validity of the rule. Note, that a domain expert can define own errors and warnings and check for them. The last method `getFunctionReturnObject()` returns the function's ReturnObject. This method is called, when the *Validation Manager* checks if the data type which is used in a comparison with a function, suits the functions return type.
The interface `FunctionReturnObject` in Listing 5.5 has to be implemented by the ReturnObject of the custom function.

Listing 5.5: Interface FunctionReturnObject

```java
package com.questionsys.questionrule.functions;

public interface FunctionReturnObject {
        public Object getValue();
        public FunctionReturnDataType getType();
        public void setValue(Object value);
        public void setType(FunctionReturnDataType  type);
}
```

These methods are only getters and setters for the actual value as well as the data type of the value. The available types are defined in the enum `FunctionReturnDataType`. Next, the approach of implementing a custom function is presented. As an example, the `count` function (see Section 4.1.1) is discussed, which is invoked with 2 arguments: an array of comparisons and an integer. The latter checks if a certain number of comparisons is fulfilled, however, it doesn't matter which. If so, the function will return true. If the specified number of comparisons is not fulfilled, the function will return false. Listing 5.6 shows the class of the function `count`.

Listing 5.6: User-Defined Function count. Implemented using the provided Framework

```java
package com.questionsys.questionrule.functions;

public class CountingFunction implements Function {
        String identifier = "count";
        int paramCount = 2;
```

```
	private FunctionReturnObject
	countingFunctionReturnObject;

	public CountingFunction(){
		countingFunctionReturnObject = new
		   CountingFunctionReturnObject();
		countingFunctionReturnObject.setType(
		   FunctionReturnDataType.Boolean);
	}

	public String getIdentifier() {
		return identifier;
	}

	public ArrayList<RuleError> checkForErrors(
	   FunctioncallContext ctx) {
		ArrayList<RuleError> listOfFunctionErrors = new
		    ArrayList<RuleError>();
		return listOfFunctionErrors;
	}

	public ArrayList<RuleWarning> checkForWarnings(
	   FunctioncallContext ctx) {
		ArrayList<RuleWarning> listOfFunctionWarnings =
		    new ArrayList<RuleWarning>();
		//iterate parameter list
		ParamlistContext paramlistContext = ctx.
		   paramlist();
		int count = 0;
		while(paramlistContext != null){
			count++;
```

```
30                 paramlistContext = paramlistContext.paramlist();
31             }
32             //compare parameter count
33             if(count !=  paramCount){
34                 listOfFunctionWarnings.add(new RuleWarning("Anzahl der Parameter bei der Funktion mit dem Namen: "+ ctx.IDENTIFIER()+ " stimmt nicht!", 2301));
35             }
36             return listOfFunctionWarnings;
37         }
38 
39         public FunctionReturnObject eval(FunctioncallContext ctx) {
40             //TODO implement evaluation
41             countingFunctionReturnObject.setValue(true);
42             return countingFunctionReturnObject;
43         }
44 
45         public FunctionReturnObject getFunctionReturnObject() {
46             return countingFunctionReturnObject;
47         }
48 }
```

Note, that the user-defined *CountingFunction* implements the interface `Function` in line 3, which was discussed already. The Identifier (line 4) defines the name of the function. Then the amount of parameters is defined, which is used in a function specific check for warnings. The constructor defines the return data type. The method `checkForErrors (FunctioncallContext ctx)` doesn't do anything, as the implementation for the *count* function doesn't define any errors. However, the method `checkForWarnings(`

`FunctioncallContext ctx)` verifies, if the correct amount of parameters are provided. If not, a warning is added to the warning list. The `eval` method needs to be implemented with actual business logic.
When starting the application, *Questionrule* loads all user-defined functions from the Function Repository using *Reflection* [Orc]. This can be done, because all functions have to implement the `Function` interface, which was described earlier, enabling *Questionrule* to work consistently with these user-defined functions. When *Questionrule* detects a function within a rule, it verifies if this function has been loaded already. If so, it calls the function's `checkForWarnings()` and `checkForErrors()` methods for the custom validation checks defined by the domain expert.

5.3 Graphical Rule Editor

The rules in *Questionrule* are text-based, which is not very user friendly especially for user new to the application. To allow for an easier work, a *Graphical Rule Editor* was integrated in *Questionrule*. It enables the user to create such rules using a tree-based drag & drop approach. It uses the grammar to determine the structure of created rules. Figure 5.3 shows an empty editor, when creating a new rule.

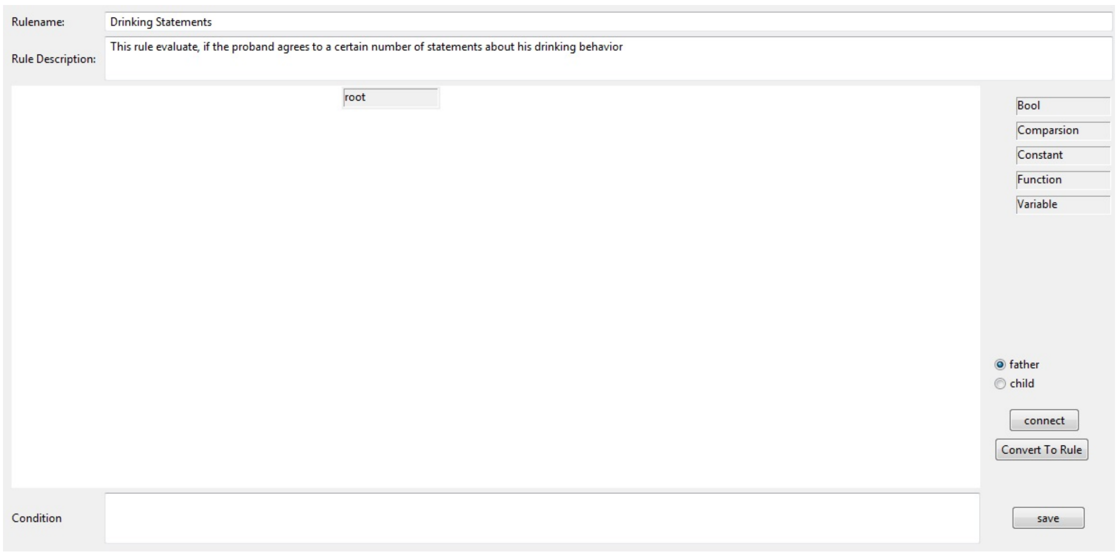

Rulename: Drinking Statements
Rule Description: This rule evaluate, if the proband agrees to a certain number of statements about his drinking behavior
Condition:

Figure 5.3: Overview of the Graphical Rule Editor when Creating a new Rule

On top of the view are several fields for the name of the rule as well as a description. Below these two fields is the graphical rule editor pane and on the bottom is the description field for the actual rule. The editor pane consists of a tool area on the right hand side and a drawing area on the left hand side.
The rule `(True == count(INSERT YOUR DATA HERE)) && (Age <= 18)` evaluates if the subject agrees to a certain number of statements about his drinking behavior. To create this rule, the domain expert simply drags various parts of the rule (`Bool`, `Comparison`, `Constant`, `Function` and `Variable`) and drops them on the editor pane (figure 5.4).

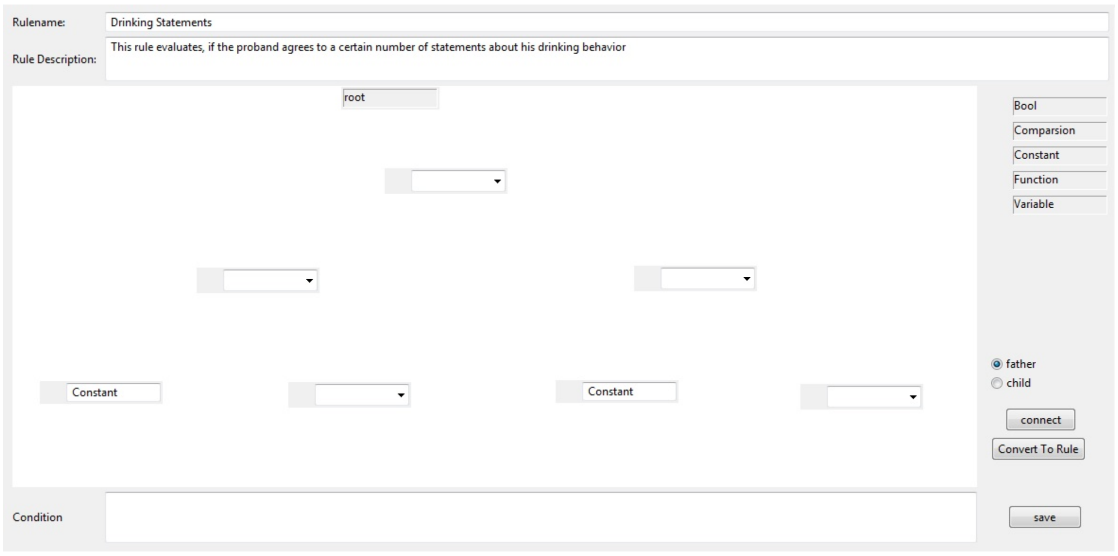

Rulename: Drinking Statements
Rule Description: This rule evaluates, if the proband agrees to a certain number of statements about his drinking behavior
Condition

Figure 5.4: Adding Various Parts to the Rule

In the next step, the domain expert can specify the different parts of the rule. Therefore he can use the dropdown fields for `Bool` (`||` or `&&`), `Comparison` (`==`, `!=`, `<=`, `<`, `>=` or `>`), `Function` and `Variable`. The dropdown fields for `Function` and `Variable` are generated dynamically. To fill the `Function` and `Variable` dropdown fields, the *Rule Manager* receives the functions names from the *Validation Manager* and the variable names from the *Project Manager* (figure 5.5).

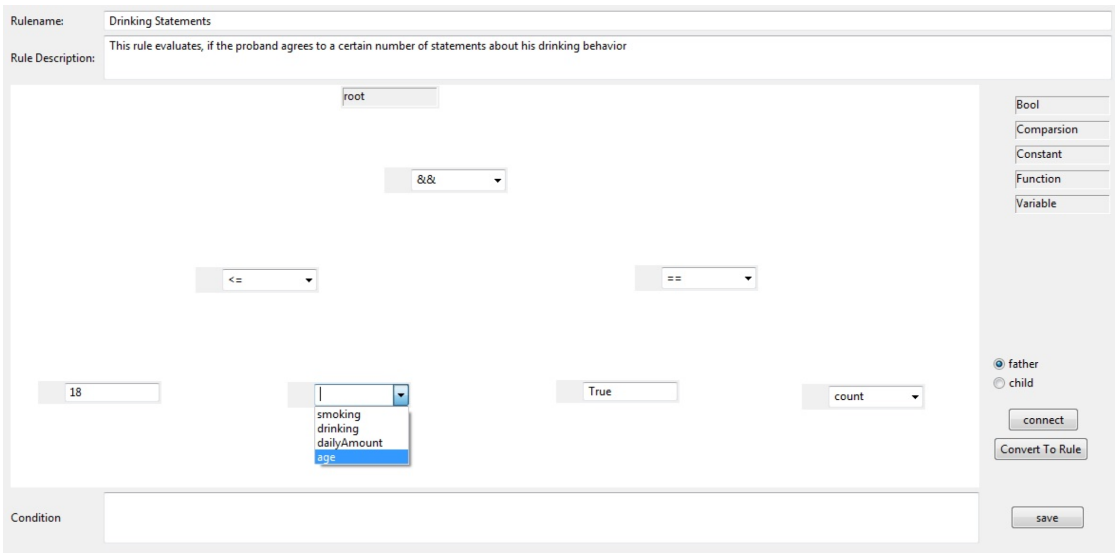


Figure 5.5: Dynamically Selecting Parts of the Rule

Next, the domain expert selects two parts of the rule, which should be linked and uses the *connect* button to do so. To be more precisely, he needs to select which part of the rule is the `father` (red) and which part is the `child` (green) in order to set them in a hierarchical structure shown in figure 5.6.

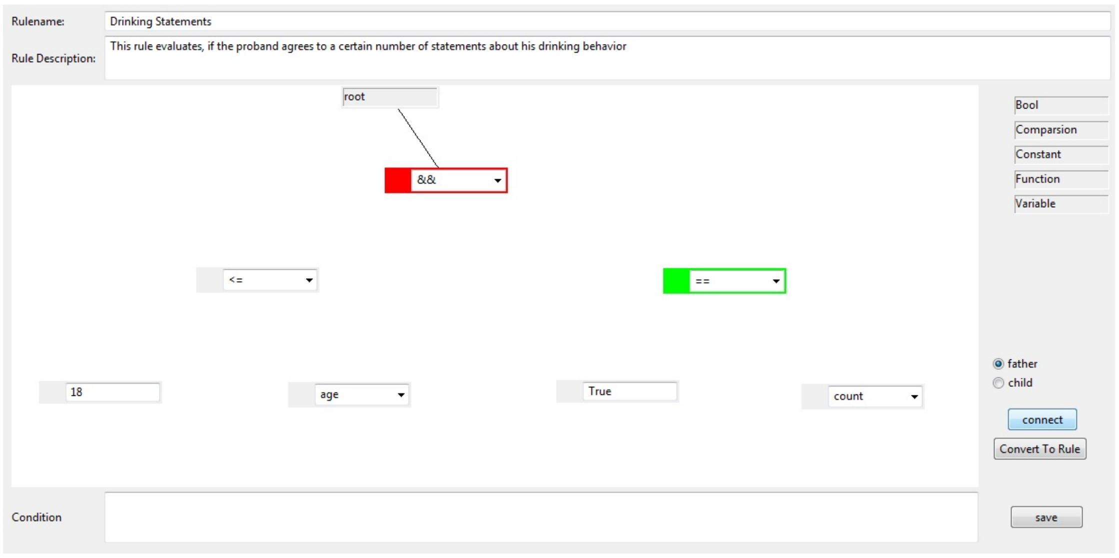
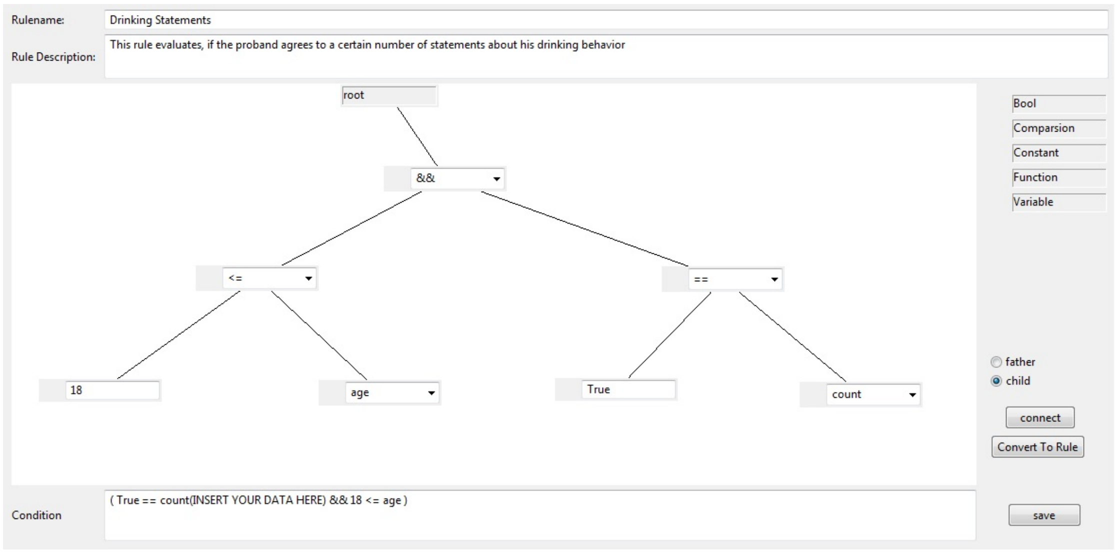


Figure 5.6: Hierarchically Connecting Parts of the Rule

The connection mechanism enforces a correct rule and implements the correctness by construction principle. For example, it is not possible to connect two `Constants` or more than two parts for a `Comparison`.
When the expert has finished building the rule, he uses the *Convert to Rule* button and the rule is displayed in the *condition* description field as seen in figure 5.7. Note, that the parameters for a function still has to be assigned after the rule has been converted. This has to happen manually, because *Questionrule* doesn't know the desired input of the functions.


Rulename: Drinking Statements

Rule Description: This rule evaluates, if the proband agrees to a certain number of statements about his drinking behavior

Condition: (True == count(INSERT YOUR DATA HERE) && 18 <= age)

Figure 5.7: Complete Rule Graph with Condition

Of course the rule editor provides additional functionalities like deleting and repositioning different parts of the rule. When a rule part is deleted all lines connected to it are deleted as well.

5.4 Conclusion

ANTLR offers the possibility to easily validate rules and thereby helps to enforce a valid rule. User-defined functions enables domain experts to add new ways to evaluate the data collected and therefore increase the flexibility and expressiveness of *Questionrule*. The Graphical Rule Editor allows for an easy introduction to *Questionrule*.

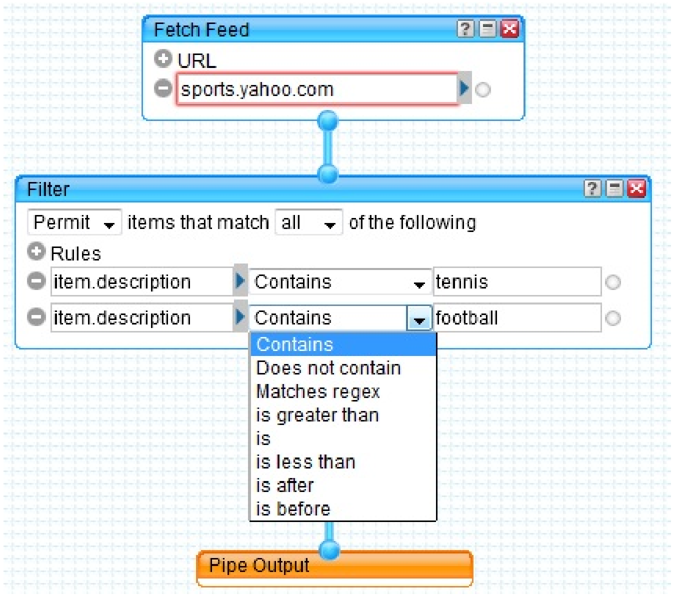
6
Related Work

There exists a variety of rule editors available on the market. Three examples for applications using editors to compose boolean logic are: *Yahoo! Pipes*, *Axure RP Pro* and *Apple Itunes*. These applications are discussed in the following sections 6.1 to 6.3. Finally, section 6.4 compares these applications against the concept of *Questionrule*, which was developed in this thesis.

6.1 Yahoo! Pipes

Yahoo! Pipes (YP) [Yah] is a web application from Yahoo! that provides a graphical user-interface to build data mashups. These data mashups may aggregate web feeds, web pages, and other services, to create Web-based apps from various sources, and to publish those apps. Users are able to `pipe` information from different sources (e.g.,

Flickr, RSS) and can then create rules for how that content should be modified (e.g., filtering, merging). An example is New York Times through Flickr, thereby a pipe takes the New York Times RSS feed and adds an appropriate photo from Flickr based on the keywords of each item.
Yahoo! Pipes provides many predefined modules which can be used either to grab data from sources or to edit and manipulate the data grabbed. To create a new pipe, the user drags the modules onto a working pane and connects them afterwards. These modules are grouped into categories. These categories are for example *sources*, *user inputs* and *operators*. In the *sources* category are modules, which grab data from one or multiple sources on the internet. The modules of the *user input* category enable the user to add an input in the pipe. The modules in the *operator* category are used to filter or transform the data. Figure 6.1 shows the *Filter* module in a pipe, which is part of the *operator* category.


Fetch Feed
URL: sports.yahoo.com

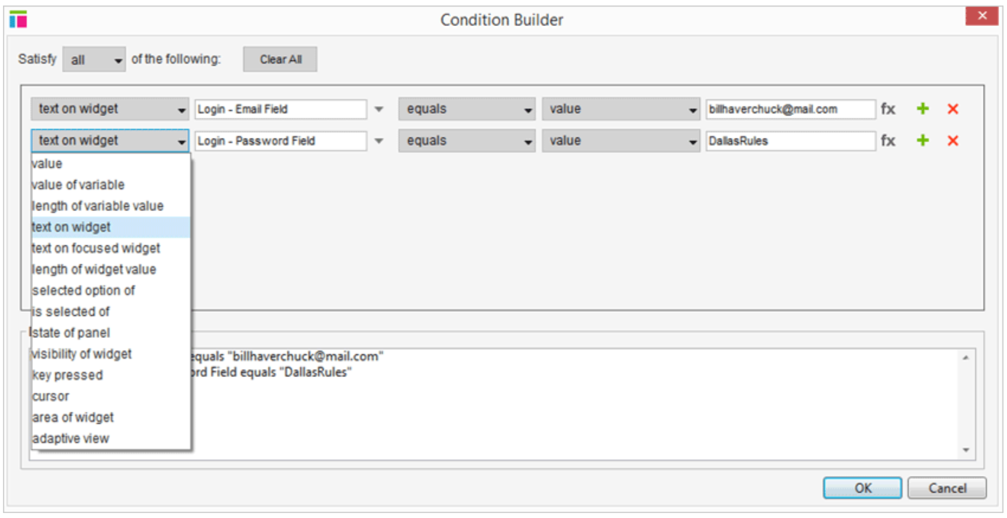
Filter
Permit items that match all of the following
Rules
item.description Contains tennis
item.description Contains football

Figure 6.1: Yahoo! Pipes Filter

This *filter* module can receives input from a module of the *sources* category (in this example, *Fetch Feed*), filters the content depending on user-created rules and forwards the filtered input to another module. The user working with YP can create boolean rules with these filters by simply using the dropdown choices and the textfields. These rules either consist of comparisons connected with boolean `AND` or with boolean `OR`. It is not possible to nest `AND` and `OR`.

6.2 Axure RP Pro

Axure RP Pro [Axu] is a wireframing, rapid prototyping, and specification software tool aiming at web and desktop applications. It offers possibilities like drag and drop placement, resizing, and formatting of user-interface widgets. Additionally, it enable the developer to annotate widgets and define interactions such as linking, conditional linking, simulating tab controls and show or hide elements.
The Condition Builder of Axure RP Pro allows to add functionality to the prototype, which can help the user when testing the application. The example in figure 6.2 creates a login functionality.


Condition Builder

Satisfy: all of the following

text on widget	Login - Email Field	equals	value	billhaverchuck@mail.com
text on widget	Login - Password Field	equals	value	DallasRules

equals "billhaverchuck@mail.com"
rd Field equals "DallasRules"

Figure 6.2: Axure Condition Builder

The first and second field in each row are the specific widget and the type of value which are the first operand. Next the type of comparison follows. The last two fields are the type of value and the specific value of the other operand. Like the *filter* module within *Yahoo! Pipes*, it is not possible to nest boolean rules.

6.3 Itunes

Apple Itunes is a media player, media library, and mobile device management application developed by Apple Inc. [App]. It is used to play, download, and organize digital content like audio and video on personal computers running on OS X and Microsoft Windows operating systems.
In addition, it offers the possibility to create *smart playlists* using boolean logic (figure 6.3).

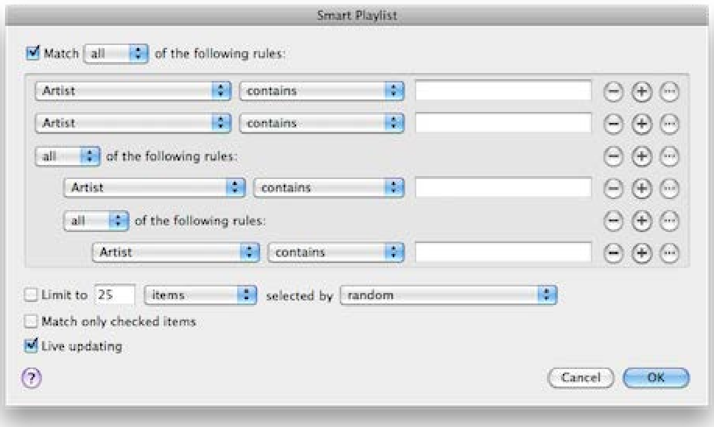

Smart Playlist
Match all of the following rules:
Artist contains
Artist contains
all of the following rules:
Artist contains
all of the following rules:
Artist contains
Limit to 25 items selected by random
Match only checked items
Live updating
Cancel
OK

Figure 6.3: Itunes

To create a new *smart playlist*, users use the dropdown choices (e.g., Artist, Album) and the textfields. The comparisons are connected with boolean `AND` (all) or with boolean `OR` (any). In contrast to Yahoo! Pipes and Axure the user can create nested boolean rules here very easy. The brace is presented through the indenting of the corresponding comparisons.

6.4 Comparison

In comparison to the presented rule editors, *Questionrule* offers both an easy and intuitive way to create boolean rules with the graphical rule editor. In addition, *Questionrule* doesn't complicate the creation of more complex rules, because complex rules can be created using the text input. In contrast to all presented rule editors, which use forms to create and represent the rule, *Questionrule* uses a tree-based approach. This is a more intuitive approach, because when creating a new rule, the parenthesis doesn't have to be put explicitly by the user, but results implicit from the structure of the tree. Additionally, the use of a formal grammar in *Questionrule* helps to define the structure of the tree and allows for a validation of the rules. None of the presented rule editors offers a comparable validation of rules as discussed in this thesis. Moreover, these rule editors have a fixed set of operators, which can't be extended through the user. *Quesitionrule*'s concept of user-defined *functions* enables the later addition of operators. Thereby, *Questionrule* isn't restricted to a certain use case like for example Itunes managing a music collection, but can adapt to the current application scenario. Supplementary, the *functions* of *Questionrule* enable the domain expert to define errors and warnings for the function himself. Thereby, a *function* can be checked for its specific errors and warnings. *Questionrule* validates rules for these errors and warnings.

7 Conclusion

Section 7.1 provides a critical discussion of the features of *Questionrule*. Section 7.2 summarizes the result and lessons learned form thesis, whereas section 7.3 provides an outlook for further improvements and extensions.

7.1 Discussion

Questionrule adds additional functionality to the *QuestionSys* project. The presented application allows to create rules which are used to evaluate the data collected using electronic questionnaires. Figure 7.1 shows, how the developed concept for *Questionrule* can be integrated into *QuestionSys*.

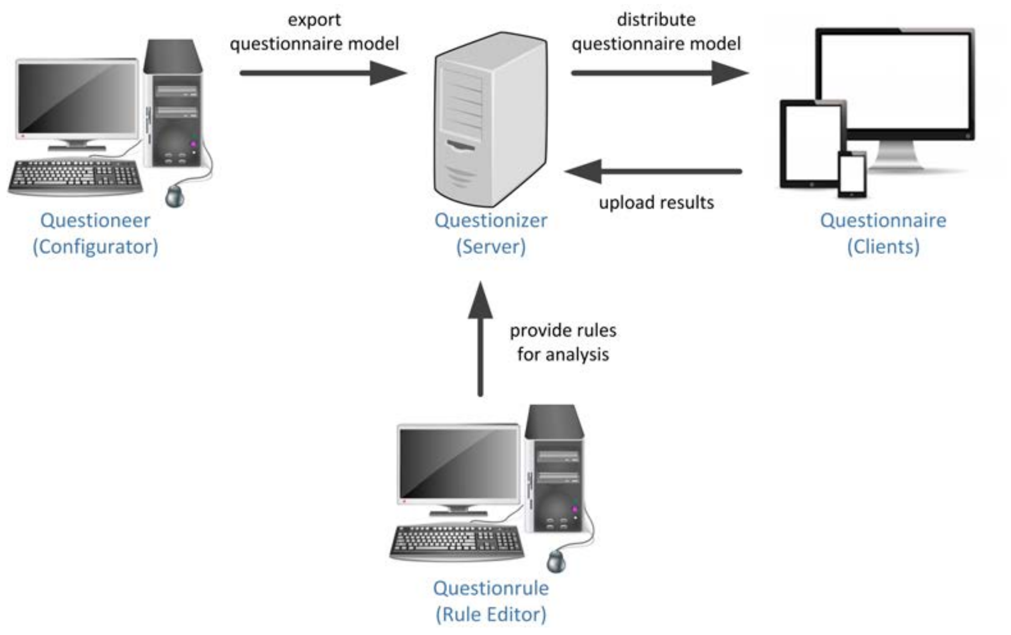


Figure 7.1: QuestionSys Architecture enriched with the Questionrule Component

The workflow to enable the analysis of the data collected using these rules could be as follows: At first the questionnaire is modeled [Sch14a, Sch14b]. Afterwards the questionnaire model is exported to the server component *Questionizer* and *Questionrule* can import this questionnaire model using Web Services. Now the domain expert can create rules and define corresponding texts. These texts can be defined in multiple languages and are shown if a rule applies. *Questionrule* transfers the finished rules to the server, where they are stored along with the corresponding questionnaire model. The client *Questionnaire*, which runs on a smart mobile device, can download questionnaires with their corresponding rules. This model is then enacted using a lightweight process engine. All data collected is stored directly on the device. The completed questionnaire can then be evaluated on the smart mobile device using the rules. Furthermore, results can be uploaded to the server and the server evaluates the results. This approach allows to evaluate the rules both on the server and the client, as the rules are stored on both components. This leads to two use cases for the evaluation of the data collected.
The first use case is the evaluation on the client. In some cases it is important to have the result immediately after completing a questionnaire (e.g., german TÜV inspection for

a car or a medical questionnaire about previous injuries). The presented approach will offer the possibility to evaluate the completed questionnaire directly on the smart mobile device and present the result right away. Therefore, the smart mobile device must enact the business process with a process engine.
The second use case is the evaluation on the server. In a large case study a big amount of data maybe gathered, which needs a powerful application to evaluate the data collected. As the clients export all data collected to the server component, this server is able to evaluate numerous questionnaire with their corresponding rules. The data stored on the server may be used for further evaluation in terms of Business Intelligence [AAS03] and process mining [ARW$^+$07].
To summaries in short, *QuestionSys* allows both for a later evaluation of big amounts of data on the server and an immediate evaluation on the client.

7.2 Conclusion

Questionrule enables domain experts to create rules, which are used for evaluating electric questionnaires. The graphical editor allows for an intuitive and fast creation of such rules. By using the Eclipse RCP framework for the implementation, *Questionrule* is easy to extent and add further features. Moreover, the concept of user-defined functions was introduced. These functions enable domain experts to enhance their rules with own functionality to cover additional requirements in respect to the evaluation of the data collected. In addition, a grammar for rules was defined allowing for a syntax for the creation of rules. The grammar provides the basis for the graphical rule editor and the *Validation Manager*, which checks the rules for errors.

7.3 Outlook

The next step for *Questionrule* is the upload of completed rules to the server *Questionizer* using a REST interface. For this purpose, a REST client needs to be implemented in *Questionrule*. REST was introduced by Roy Fielding in his doctoral dissertation [FT02].

In addition often used, standard user-defined function should be implemented, so the domain experts can use them. One example is the evaluation of the `count` function, which must be implemented. A new function is the `sum` function, which adds passed integers and returns the sum. This function may be used when the subjects must answers questions with for example *never*, *somestimes*, *often* or *always*. These possible answers are assigned to integers and then are added with the help of the sum function. Depending on the sum, the subject can receives different advices.
Furthermore *Questionrule* needs to be tested in practice by domain expert, so the experience of real world tests can be incorporated in *Questionrule*. Especially a study on the usability of the graphical rule editor is important to determine how comfortable the editor actual is. Another step is the translation of *Questionrule* into multiple languages.

List of Figures

Listings

Bibliography

[AAS03] ANANDARAJAN, Murugan ; ANANDARAJAN, Asokan ; SRINIVASAN, Cadambi A.: *Business intelligence techniques: a perspective from accounting and finance*. Springer, 2003

[App] APPLE INC.: *Apple Itunes*. `https://www.apple.com/de/itunes/`. – last visited: October 21., 2014

[ARW$^+$07] AALST, W. M. P. d. ; REIJERS, H. A. ; WEIJTERS, A. J. M. M. ; DONGEN, B. F. ; MEDEIROS, A. K. d. ; SONG, M. ; VERBEEK, H. M. W.: Business Process Mining: An Industrial Application. In: *Inf. Syst.* 32 (2007), Juli, Nr. 5, S. 713–732. – ISSN 0306–4379

[Axu] AXURE SOFTWARE SOLUTIONS: *Axure RP Pro*. `http://www.axure.com/`. – last visited: October 21., 2014

[CDK$^+$02] CURBERA, Francisco ; DUFTLER, Matthew ; KHALAF, Rania ; NAGY, William ; MUKHI, Nirmal ; WEERAWARANA, Sanjiva: Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. In: *IEEE Internet Computing* 6 (2002), März, Nr. 2, S. 86–93. – ISSN 1089–7801

[Cho56] CHOMSKY, Noam: Three models for the description of language. In: *IRE Transactions on Information Theory* 2 (1956), S. 113–124. – `http://www.chomsky.info/articles/195609--.pdf` – last visited: October 21., 2014

[DR09] DADAM, Peter ; REICHERT, Manfred: The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support - Challenges and Achievements. In: *Computer Science - Research and Development* 23 (2009), Nr. 2, S. 81–97

[Ecl] ECLIPSE FOUNDATION: *Eclipse IDE*. `https://www.eclipse.org/downloads/`. – last visited: October 21., 2014

[FT02] FIELDING, Roy T. ; TAYLOR, Richard N.: Principled Design of the Modern Web Architecture. In: *ACM Trans. Internet Technol.* 2 (2002), Mai, Nr. 2, S. 115–150. – ISSN 1533–5399

[KP^{+}88] KRASNER, Glenn E. ; POPE, Stephen T. u. a.: A description of the model-view-controller user interface paradigm in the smalltalk-80 system. In: *Journal of object oriented programming* 1 (1988), Nr. 3, S. 26–49

[OMG11] OMG SPECIFICATION, OBJECT MANAGEMENT GROUP: *Business Process Model and Notation (BPMN) Version 2.0*. 2011

[Orc] ORCALE CORPORATION: *Trail: The Reflection API*. `http://docs.oracle.com/javase/tutorial/reflect/`. – last visited: October 21., 2014

[OSG] OSGI ALLIANCE: *OSGi Core Release 5*. `http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf`. – last visited: October 21., 2014

[Par13] PARR, Terence: *The Definitive ANTLR 4 Reference*. O'Reilly, 2013. – ISBN 1934356999

[Pot96] POTEL, Mike: MVP: Model-view-presenter the taligent programming model for c++ and java. In: *Taligent Inc* (1996)

[RD09] REICHERT, Manfred ; DADAM, Peter: Enabling Adaptive Process-aware Information Systems with ADEPT2. In: CARDOSO, Jorge (Hrsg.) ; AALST, Wil van d. (Hrsg.): *Handbook of Research on Business Process Modeling*. Hershey, New York : Information Science Reference, March 2009, S. 173–203

[RW12] REICHERT, Manfred ; WEBER, Barbara: *Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies*. Berlin-Heidelberg : Springer, 2012

[Sch08] SCHÖNING, Uwe: *Theoretische Informatik- kurz gefasst*. Spektrum Akademischer Verlag, Heidelberg, 2008. – ISBN 3827418240

[Sch14a] SCHERLE, Steffen: *Konzeption und Evaluierung einer domänenspezifischen Modellierungsumgebung für prozessorientierte Fragebögen*. January 2014. – Diploma thesis, University Ulm

[Sch14b] SCHULTE, Juri: *Technical Conception and Implementation of a Configurator Environment for Process-aware Questionnaires Based on the Eclipse Rich Client Platform*. March 2014. – Master thesis, University Ulm

[SSP$^+$14] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; MAIER, Fabian ; REICHERT, Manfred: Towards Process-Driven Mobile Data Collection Applications: Requirements, Challenges, Lessons Learned. In: *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, 2014, S. 371–382

[Ulm] ULM UNIVERSITY: *QuestionSys*. `http://www.uni-ulm.de/in/iui-dbis/forschung/projekte/questionsys.html`. – last visited: October 21., 2014

[Vog13] VOGEL, Lars: *Eclipse 4 RCP: The complete guide to Eclipse application development*. (Vogella series), 2013. – ISBN 3943747077

[W3Ca] W3C: *Extensible Markup Language (XML) 1.1 (Second Edition)*. `http://www.w3.org/TR/2006/REC-xml11-20060816/`. – last visited: October 21., 2014

[W3Cb] W3C: *XML Schema Definition Language (XSD) 1.1*. `http://www.w3.org/TR/xmlschema11-1/`. – last visited: October 21., 2014

[Yah] YAHOO! INC.: *Yahoo! Pipes*. `https://pipes.yahoo.com/`. – last visited: October 21., 2014

Name: Bernd Mertesz Matrikelnummer: 699008

Erklärung

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

Ulm, den ....

Bernd Mertesz