university of groningen

faculty of mathematics and natural sciences

Quby

A domain-specific language for non-programmers

Master's thesis computer science

August 18, 2012

Student: M. Veldthuis

Primary supervisor: Prof. M. Aiello

Secondary supervisor: Prof. N. Petkov

Secondary supervisor: A.C. Emerencia

Abstract

Periodically taken questionnaires are used by mental health-care institutions to monitor the well-being and satisfaction of patients. It is preferable to let patients fill out these questionnaires digitally from home. Therefore, web-based applications that display these questionnaires and record the answers are used in practice.

Several web applications that deal with this already exist. In order to get such an application to display a questionnaire, it first needs to be defined using its editing interface, which is done by non-programmers. Questionnaires often require a large set of features like input validation and non-linearity due to dependencies between possible answers and further questions. Because of this, the graphical user interfaces for defining questionnaires in the existing tools are complex.

An alternative to defining questionnaires using a graphical user interface would be to use a text-based interface, using a domain-specific language (DSL). No research has been found that surveyed the use of domain-specific languages for constructing questionnaires by non-programmers. In fact it has been stated that further research into the more general case of DSL usage by non-programmers is needed.

This thesis examines the use of DSLs for the purpose of defining questionnaires. We designed, implemented and tested Quby: a web-based application written in Ruby that reads in questionnaires defined in a custom DSL and presents these to psychiatric patients. These questionnaire definitions can be built by psychiatric research staff.

We show a method of working with DSLs that cleanly seperates application logic from the code that supports the DSL by using expression builders, a method which has not been in wide use in the Ruby programming community.

We found that non-programmer domain experts were not only capable of using the DSL we designed, but they preferred it over traditional GUI-based tools. Additionally, experts who work day-to-day with GlobalPark were able to create new questionnaires as quickly using our DSL as they were using GlobalPark.

Based on our findings, we conclude that it is possible to design domain-specific languages based on the Ruby language, and have non-programmers work effectively with them with minimal support from on-site developers.

Keywords: domain-specific Languages, Language Design, Ruby, Rails, Questionnaires, Psychiatrics, Web-Applications

Contents

Chapter 1

Introduction

Domain-specific languages (DSLs) have existed for a very long time. Ancient Unix utilities like `sed`, which dates back to 1973, offer mini-languages that serve to aid users to accomplish their goals within a small, well-defined problem domain.

In recent years, domain-specific languages have seen a large uptake in usage. In his book on DSLs [9], Fowler gives his thoughts on why this uptake is happening now. He outlines how around the year 2000, the software development community was largely standardizing towards C-like languages, specifically Java and C#. However, not everything felt right being expressed in these languages, which led to XML taking a leading position in being the place to store all sorts of declarative information, most importantly configuration data.

Of course, it did not take long for developers to start feeling overwhelmed by the amount of syntactical overhead that XML has. Around this time, in 2004 the web application framework Ruby on Rails was released. Its approach to configuration was not to use XML, but instead have lots of small domain-specific languages for everything ranging from logging levels to describing relations between models in its object-relational mapper. While the Ruby programming community has long had a love for making interacting with libraries as fluent as possible, Rails was probably the most prominent project that caused this style to become popular in the wider programming community.

A domain-specific language can be a great way of making a piece of code easier to understand, sometimes even clear enough for domain experts to be able to read it, providing a great communication channel that developers and experts can use to talk about how their domain is codified in the software.

Taking this idea a step further is the notion of making DSLs so understandable by domain experts that they are able to write programs in them by themselves. However, it is far from clear what the elements of success are for a given DSL to actually be usable by domain experts. Very little research has actually focused on this problem [7] even though software projects greatly benefit from better understanding between the development team and the business experts [2, 8].

In this thesis, we will give an account of a DSL that we implemented for a company developing web-based psychiatric software.

In the psychiatric health-care industry, it is valuable to periodically assess the well-being of patients. In order to do so in an objective fashion, standardized questionnaires are used, a practice called routine outcome assessment [5, 17]. In the last few years, Dutch health insurance companies have started requiring that mental health-care institutions perform these periodic surveys, which has rapidly expanded the market for software applications that can help lift the administrative overhead in performing these surveys.

One such software application is called RoQua, which is a web application that integrates with electronic patient dossier (EPD) software to provide these routine outcome assessments in an automated manner. In order to let patients and psychiatric professionals fill out questionnaires, the questionnaires themselves have to be defined in a subsystem called a questionnaire engine.

Commercial web services that implement questionnaire engines exist, and RoQua was using one of them called GlobalPark. However, in the pages that follow it will be argued that these services have certain undesirable properties when used on a large scale, such as is done within RoQua, where a database of nearly 200 psychiatric questionnaires is built up.

1.1 Non-programmer DSLs

Domain-specific languages can be helpful in the communication between the customer and developers. They can even help offload work from the developers on to domain experts.

The objective of this thesis is to determine whether it is possible to build an internal domain-specific language that non-programmer domain experts are able to use to write declarative programs, based on the Ruby programming language.

To determine this, we will show the domain-specific language created Quby, which is a web-based questionnaire engine we built. We will explain the syntax of the DSL, the implementation of the DSL parser, and evaluate how well-suited the DSL and the web-based editor we built to support it are for usage by domain experts by comparing it against state-of-the-art professional solutions.

1.2 Thesis organisation

The next chapter will describe the history of domain-specific languages, and explain a few classifications and key concepts. We will also discuss work related to the use of DSLs by nonprogrammers, and discuss the state of web-based questionnaire engines. In chapter 3 we will discuss the overall design of the Quby application. We will briefly outline the different parts of the questionnaire engine, and go into a more detailed overview of the architecture relating to the DSL. Chapter 4 describes the various elements found in psychiatric questionnaires, and discusses how the DSL is set up to capture these elements in a manner that is close to how domain experts reason about these elements. In chapter 5 we discuss how we implemented the domain-specific language using expression

builders and nested closures. In chapter 6 we go into our findings with regard to building a DSL, and building a DSL specifically for non-programmer domain experts. Finally, chapter 7 discusses the results we found.

Chapter 2

Related Work

We will be looking at domain-specific languages: what kinds of classifications exist, what are the advantages and disadvantages. We will also look at what kind of research there is with regard to usage of domain-specific languages by nonprogrammers. Finally, we will elaborate on the state of the art with regards to web-based questionnaire builders.

2.1 Domain-Specific Languages

A domain-specific language is a type of programming language which is distinct from general purpose languages such as C or Java. General purpose languages are designed to be able to handle pretty much any kind of programming task. Domain-specific languages on the other hand are designed with a specific problem domain in mind, and are set up to be able to solve a specific problem as easily, quickly and/or elegantly as possible.

Domain-specific languages have existed for a long time. Unix utilities have used them to make tasks easier to accomplish, and the use of DSLs has been considered idiomatic in dynamic languages like Lisp for a long time. They are commonly divided into external and internal DSLs, as described by Fowler [10]:

> External DSLs are written in a different language than the main (host) language of the application and are transformed into it using some form of compiler or interpreter. The Unix little languages, active data models, and XML configuration files all fall into this category. Internal DSLs morph the host language into a DSL itself - the Lisp tradition is the best example of this.

External DSLs are built by designing a grammar and implementing a parser for the language. The upside is that you have a lot of control over the intricacies of the design of your language, but the obvious downside is that for more complicated languages it can be a lot of work to implement. Because the language is external, you do not have access to any of the functions already

built in to the host language from it. This means that if, for example, you want to support basic integer math in your DSL, you will have to build that.

On the other hand, internal DSLs take advantage of the host language, using its parser and data structures. For internal DSLs, you mold the host language to look like a DSL. The degree to which easily usable internal DSLs can be constructed varies between programming languages. The Ruby language is one of the more flexible languages when it comes to internal DSLs, and this has caused its developer community to embrace the usage of DSLs as a design pattern. Ruby does not require much syntax, which allows you to design elegant languages. Libraries like RSpec (for unit testing [16]), and Cucumber (for system testing [4]) are great examples of this.

In recent years the style of programming where you develop and make use of many small DSLs has been called language oriented programming, a term first coined by Ward [20]. In this style of programming, you solve problems by creating languages in which it is easy to abstract them.

However, working with DSLs takes time and effort of its own. Especially editing programs written in a DSL is something that can be harder to do than working in a general purpose language, because programming environments do not offer the same level of editing features for custom DSLs as they do for well known programming languages. This point was reiterated in 2008 by a panel at COOPSLA [11]. The overall trend observed in the positions of the panelists is that while DSLs offer a number of benefits, tool support is behind compared to the state of the art for general purpose languages.

Language workbenches are one possible solution to this tooling problem [10, 12]. In essence, language workbenches are supposed to be tools that make not only the development of DSLs easier, but also make it easy to create editing environments to support programming in a DSL. Since we developed both a DSL and a web-based editor to support it, this could have been very useful to us. However, these tools are still in the early stages and considered far from mature [9]. So while there is great potential here, it is still too early to tell how well language workbenches will work in practice. In addition, none of the language workbenches currently available support the Ruby programming language, which unfortunately made them unsuitable for our purposes.

Interestingly close to the topic of this thesis, in the 1990's a group of software engineers at Cap Volmac worked on a DSL for questionnaires called RISQUEST [3], which translated DSL text into Tcl/Tk programs which could then be used by financial engineers to build up financial products in yet another DSL called RISLA. RISLA was a DSL that compiled into COBOL code. However, the DSL programs written in RISQUEST were still written by software developers.

2.2 Nonprogrammer domain experts using DSLs

A debatable topic is the viability of non-programmer domain experts (people who are not programmers, but who do have knowledge of the domain) reading and/or writing DSL programs.

Certainly, if important areas of software projects where business logic is encapsulated can be written in a DSL that is clear enough for domain experts to be able to read and understand, it can be much easier for development teams to communicate with their customers, and vice versa. Given that better communication is one of the main points of current software development methodologies [1, 8], the viability of getting domain experts to understand parts of the software code by writing it in a customized programming language is an important research topic.

Donohue [7] found evidence that non-programmers can successfully read, validate, write and maintain DSL programs. However, he found that success was far from uniform, and there were also accounts where DSLs specifically designed for non-programmers had only limited success.

It would be interesting to see an analysis of what types of DSLs worked, and what types of DSLs didn't work. For instance, it might be possible that purely declarative DSLs, such as the one described in this thesis, are more often successful than DSLs that support some form of imperative programming (that contain a "state" beyond the lexical scope of the current line of DSL text). Unfortunately, to the best of our knowledge no such analysis has been performed. As Deursen, Klint, and Visser [6] noted, literature around DSLs is fragmented, and a lot of DSLs are not described in software engineering literature, which might be a possible reason why no survey of DSL styles has been performed to date.

2.3 Web-based survey editors

Most web-based questionnaire editors currently employ a set of forms to let the user define questionnaires. We will look at two big players in our space of questionnaire editors: GlobalPark and NetQ. Both of these systems present a GUI editor that provides users a series of web forms to build questionnaires.

The areas of these systems that we will look at are text formatting, data validations and relations between questions.

Text Formatting

It is often needed to add some simple formatting like **bold** or *italics* to the texts within a questionnaire. Although occurring less frequently, bulleted and numbered lists are also needed.

For displayed texts, GlobalPark supports writing HTML into the text boxes (Figure 2.1) in order to add formatting. NetQ has a WYSIWYG editor for descriptions (Figure 2.2) but not for any other places where text is entered.

Data Validation

Before saving data, it is of course important to ensure that the entered data is in a valid format. The simplest validation is probably the one that ensures that

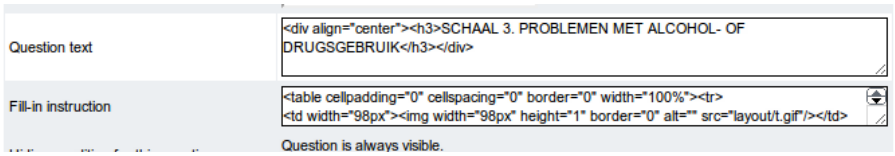


Figure 2.1: GlobalPark lets users write HTML to add formatting

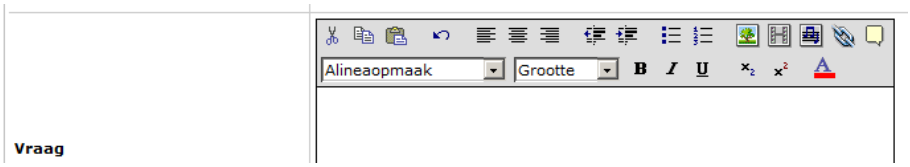


Figure 2.2: NetQ's WYSIWYG editor

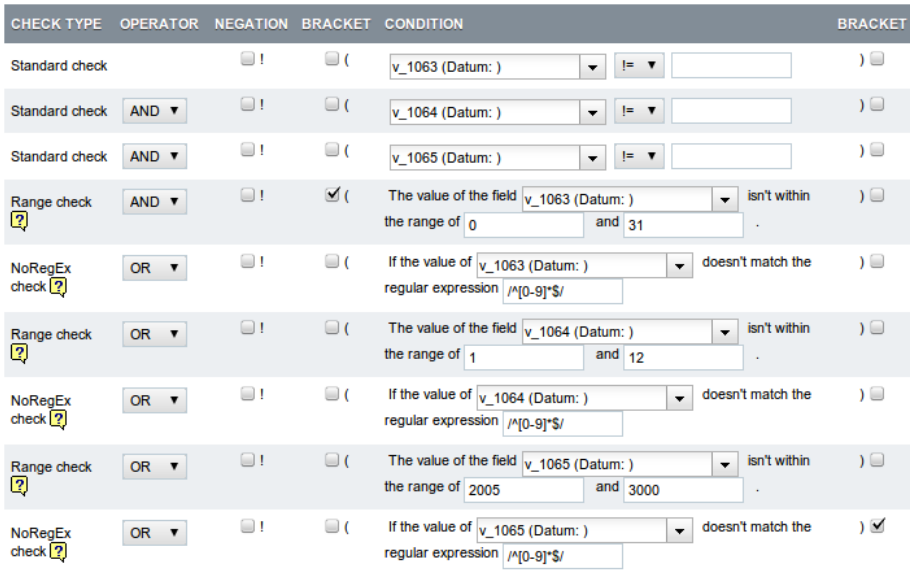


Figure 2.3: Checking if a date is correctly entered in GlobalPark

a question is actually answered. Validations that are a bit more complex are those that validate the format of some entered string, or validate that an entered integer, float or date lies within some predefined limits. Some validations can also span multiple questions.

In terms of validation, both GlobalPark and NetQ supports making questions required by simply checking a check box.

NetQ has no support for conditions beyond marking questions as required. GlobalPark also supports more complicated validations (e.g. requiring that at least one question is filled in within a group of questions, or ensuring that text entered in a text field matches a certain format) as can be seen in Figure 2.3. In that figure, the validations for a date field (which is seen as three fields in

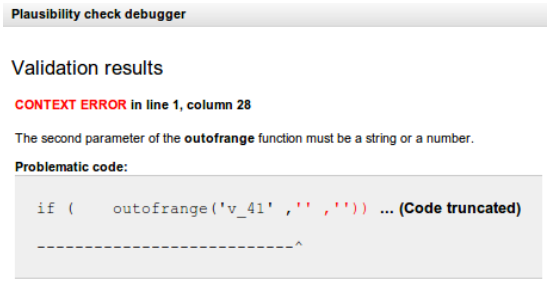


Figure 2.4: The error presented to the user when leaving both values for a range check empty

GlobalPark) are shown.

It seems that based on these graphical controls, behind the scenes some form of script gets automatically generated, because leaving some fields in for instance a range check empty, results in the error that can be seen in Figure 2.4.

Relations between questions

In psychiatric questionnaires, most questions are not dependent on each other. However, some questions can have dependencies between them. One of the things a survey editor must do is be able to work with these dependencies in a simple and straight-forward manner.

One type of relation that happens often is that some questions are only applicable when a preceding question is answered in a certain way. For instance, one question might ask "Do you drink alcohol?". If the patient then answers "Sometimes" or "Often", follow-up questions about drinking habits need to be filled out. But if the patient answers "Never", these are not applicable and should be hidden or at least disabled.

A somewhat special case of this type of relation is when an option of a multiple-choice question asks for more details in a text field. GlobalPark supports subquestions under multiple choice questions, but only one subquestion can be added to each option. This is designated by writing **\%s** somewhere into the text for that option, and upon saving an extra item will appear which represents a new question. However, this question does not have the same configurability as a regular (top-level) question, but instead shows far fewer configurable options.

In chapters 4 and 5 we show how Quby offers greatly simplified constructs for the three types of features introduced in this section.

Chapter 3

Architecture

Quby is a web application in which domain experts can define questionnaires, and patients can fill out questionnaires. With the exception of the questionnaire editor, it is not designed to work stand-alone. Instead it provides a web service API for integration with other web applications. An overview of Quby and the systems it interfaces with is shown in Figure 3.1.

Quby has two types of users, both with their own use case: nonprogrammer domain experts that define questionnaires, and both patients and medical professionals that fill out questionnaires.

There are two places where data is stored: questionnaire definitions are stored in a version control system (VCS), and filled out questionnaires (answers) are stored in a database.

Quby exposes an API to communicate with RoQua. From RoQua it receives the age and gender of a patient, which are used to calculated scores from a filled out questionnaire. These scores are then communicated back to RoQua.

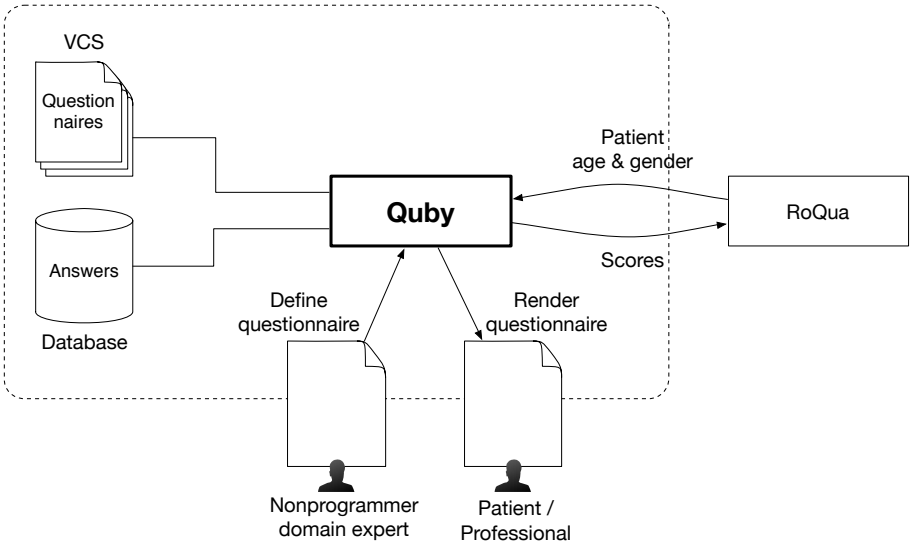


Figure 3.1: An overview of the Quby questionnaire engine

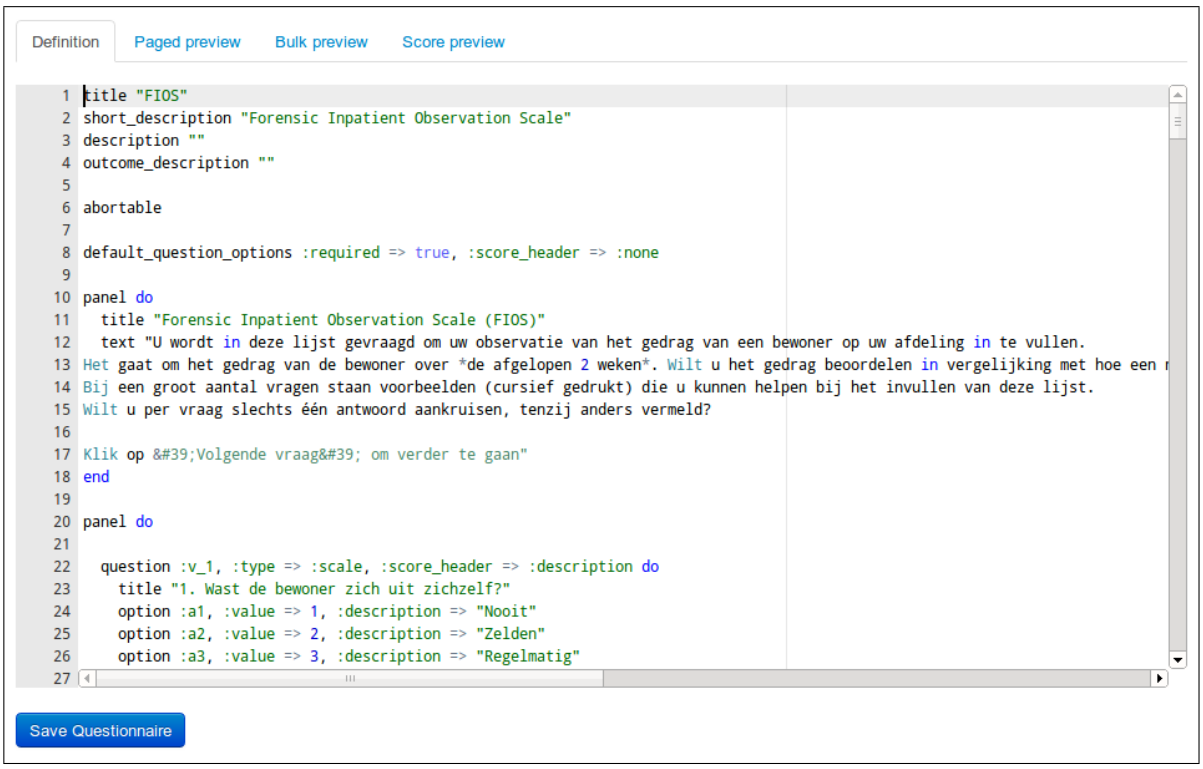


Figure 3.2: The web based questionnaire definition editor called QubyAdmin

3.1 Creating and editting questionnaires

To let nonprogrammer domain experts create and edit questionnaires in Quby, we have developed a web-based editing environment called QubyAdmin, shown in Figure 3.2. Within this editor it is possible to write questionnaires in the DSL, preview how they render, and test the score calculations.

Because questionnaires are written in a DSL, it is possible to save them as text files in the same version control system that is also used by the development team to manage the source code for Quby and RoQua. This allows questionnaire definitions to be part of the same code review process that is used for the application code. Code reviews on the questionnaire definitions are used to ensure that no accidental changes get made to questionnaires that are already being used in a production environment.

Upon saving a definition, the editor checks that the definition is valid, both syntactically and semantically. In case of an error, the error is reported back to the domain expert with a line number. If the definition is valid, the definition gets saved to the version control system.

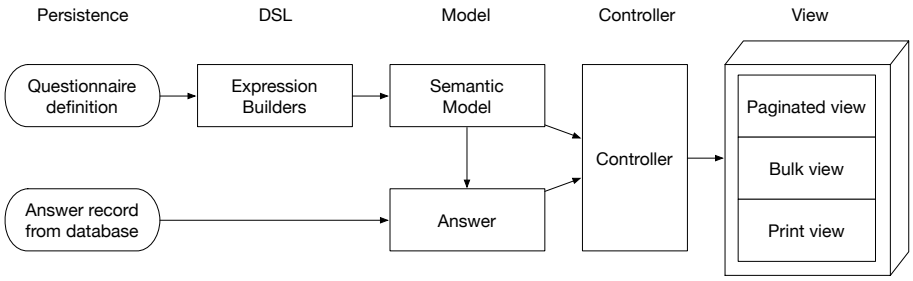


Figure 3.3: Rendering an HTML form for filling out a questionnaire

3.2 Filling out a questionnaire

Once a questionnaire has been defined by a domain expert it can be filled out by patients and medical professionals. For this to be possible, a form must be rendered for the given questionnaire, and in the case of editing an already filled out questionnaire, the form must be repopulated with the existing answers. In addition, the form can be rendered in three different ways. The complete pipeline to render a questionnaire can be seen in Figure 3.3.

Semantic Model and Expression Builders

A clean pattern for implementing domain-specific languages is to split them into two parts: the semantic model and the expression builders. The semantic model is a modeling of the domain of the DSL. Expression builders take the input text written in the DSL, and transform it into objects in the semantic model.

There are a number of reasons why this division of labor is beneficial. Firstly, it makes it easy to test the parts independently (in isolation). Secondly, it means that the rest of the application only has to see the semantic model, and that you can reuse existing knowledge on domain modeling. Finally, it is easier to switch to an external DSL if needed later on, because the semantic model is decoupled from the way the DSL instantiates the models.

The semantic model is often a subset of the domain model, although it does not have to be. Consider if the domain model is a modeling of a questionnaire, the domain model would consist of classes like `Questionnaire` and `Question`. If the domain-specific language was about questionnaires as well, then the semantic model and the domain model would overlap greatly. However, when dealing, within the same application, with a DSL that makes database usage easier, that DSL would probably have a semantic model that is in terms of `Table` and `Association`, and be completely separate from the domain model.

Controllers

The controller layer coordinates communication between the model layer and the view layer. In Quby, this means that controllers receive web requests from

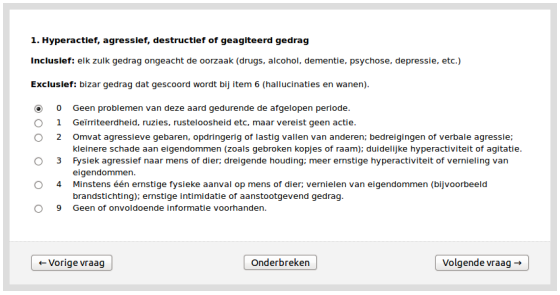


Figure 3.4: Paged view

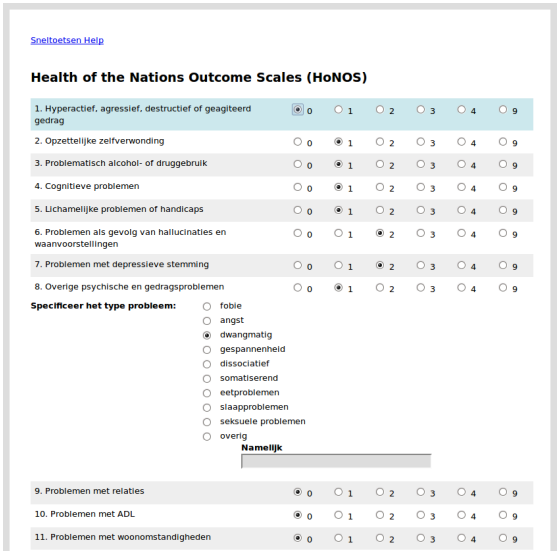


Figure 3.5: Bulk view

a browser. They fetch the correct questionnaire and answer for the request, determine which view should be rendered, and tell that view to render the questionnaire and answer. The controller then returns the rendered view to the browser.

Output formats

Given a questionnaire Quby can render three views: paginated, bulk and print.

The paginated view (Figure 3.4) shows questions with their full texts, and splits questionnaires into multiple pages with previous and next navigational buttons. The pages are determined by the domain expert in the questionnaire definition.

The bulk view (Figure 3.5) is targeted at quick data-entry, mostly for copying over the answers on a paper form into Quby. To this end, the division of the questionnaire into pages is ignored, and questions are shown in a compact form.

Finally, the print view only shows the questions and their chosen options or entered texts.

All three views are automatically determined from a single questionnaire definition, which is an improvement over GlobalPark where they had to be created as separate questionnaires.

In the next chapter we will go into more detail regarding the semantic model.

Chapter 4

Underlying Model

In this chapter, we will describe the structure of psychiatric questionnaires, and the semantic model we built around it. In the next chapter we will dive into details on how we implemented the expression builders for the DSL we show here.

4.1 Basic structure of a questionnaire

Questionnaires are the top-level elements in our system. Questionnaires have a title and a description, and have panels and scores. Panels are pages of questions, which are used to group related questions together, and additionally serve to not overwhelm the patient with a single large list of questions. Scores are calculations that are performed after the patient has completed the questionnaire, the result of which is presented to the doctor treating the patient afterwards.

Each panel can have a title, and then contains a series of items: pieces of informational text and questions (see Figure 4.1).

Informational texts are simply strings, although they go through a markup processor that generates HTML from intuitive markup formatting (Markdown

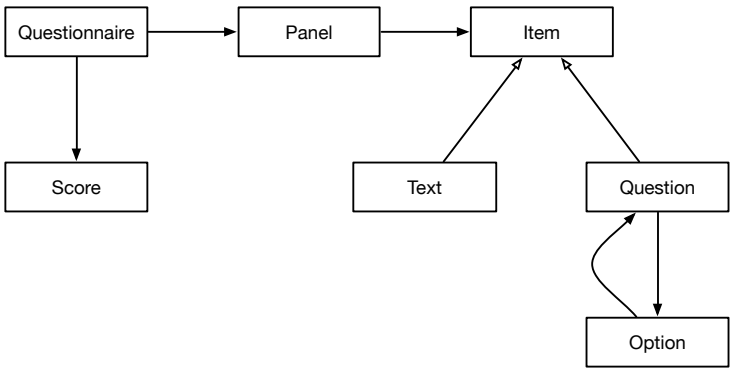


Figure 4.1: Semantic Model

[13]), e.g. `*bold* text`, or `1. A list item`.

Every question has a few pieces of required information: an identifier, a type and a title. The identifier is used to look up the value of an answer later. The type determines the type of data the question asks for and can be one of `radio`, `check_box`, `scale`, `string`, `date`, `integer`, `float`, `textarea` and `select`. The last required attribute is the title of a question, which is a string of text that is displayed with the input field.

Radio buttons, check boxes and select questions have a limited number of options that the patient can choose from, and these options are given as a list to the question. Options have identifiers of their own, a value and a description. The value is used by score calculations that are performed after filled out questionnaires are saved, and the description is what gets shown to the patient for that radio or check box. Some radio buttons or check boxes have one or more subquestions that are applicable only if that option is chosen.

4.2 Example: the "Phamous Algemeen" questionnaire

With all these properties in mind, we can have a look at a real questionnaire. The "Phamous Algemeen", shown in Figure 4.2, is an introductory questionnaire. Besides questionnaires that are self-reports, designed to be filled out by the patients themselves, there are some questionnaires that are designed to be filled out by staff members of the health-care facility as well, and this is one of them. The questionnaire belongs to a set of Phamous questionnaires, which are part of a global research initiative in the Netherlands.

While as a questionnaire it is atypical in its form, as an example it serves one goal nicely: it exercises a lot of the features needed in a questionnaire system.

It has a single page that asks for a date, the name of the professional who performed the observations, and a reason for this particular moment of observation. The first two questions require an answer. It then continues with three questions, all of which are optional, and record the year of the first psychotic episode, the year the patient first went to see psychiatric help, and the patient's ethnic background.

The full domain-specific definition for this questionnaire is listed in the appendices, in listing A.1. The "Phamous Algemeen" questionnaire will be used as the running example in the rest of this chapter.

4.3 The Quby DSL

Having established the basic structure of a questionnaire, and what the semantic model looks like, we can go into further detail regarding how to structure the domain-specific language around the semantic model. An overview of the DSL is shown in Figure 4.3.

Every questionnaire has one required attribute: a title. There are some additional attributes like `description` or `outcome_description` that define metadata

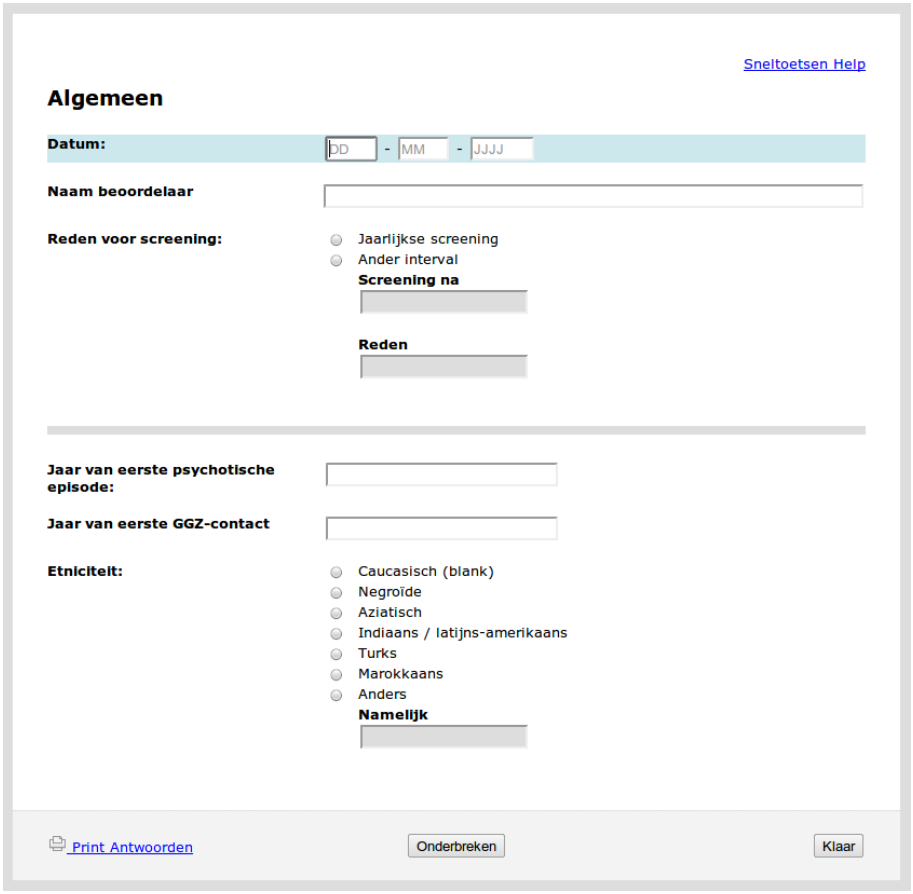


Figure 4.2: The "Phamous Algemeen" questionnaire as rendered by Quby

that is not used within Quby, but are specified so that other applications can read and use them. listing 4.1 lists the lines of DSL text that deal with the attributes for the "Phamous Algemeen" questionnaire.

Listing 4.1: Attributes of the Phamous Algemeen questionnaire

```
title "Phamous Algemeen"
short_description ""
description ""
outcome_description ""
```

Besides these pieces of textual information about a questionnaire, Quby also has a couple of behavioral options. For example, the Phamous Algemeen questionnaire should be abortable, which is to say that it should be possible to save the answers in the state they are currently in, regardless of whether the answers are in a state that pass validating constraints. This can be done by specifying the `abortable` keyword in the top level of the questionnaire definition.

In addition, this questionnaire needs to have hotkeys enabled, which enables the use of e.g. the arrow keys for jumping from question to question, and the

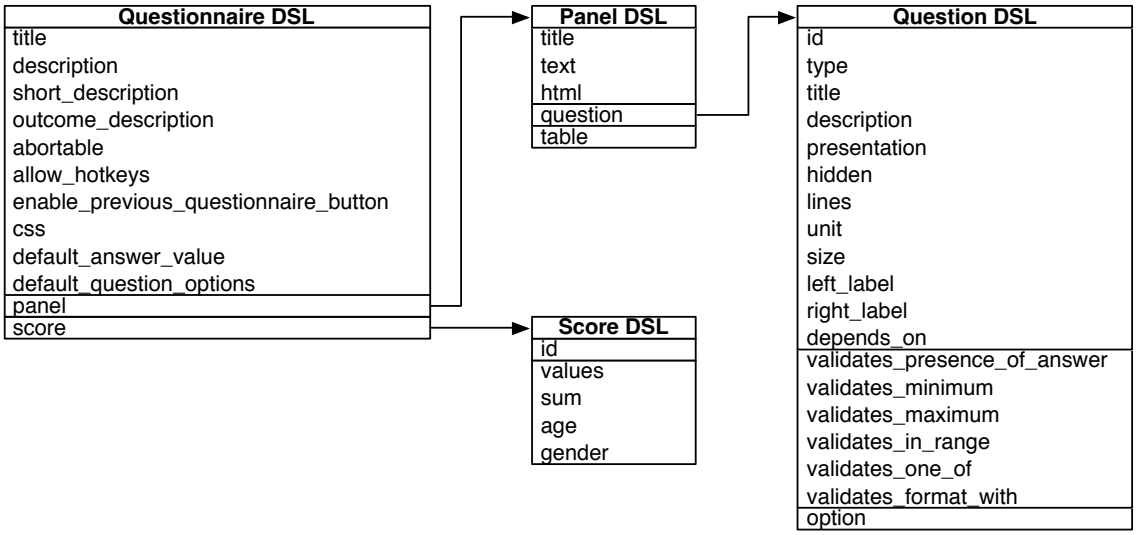


Figure 4.3: A slightly simplified version of Quby DSL. The inner workings of table and options are not shown.

number keys for selecting options in radio-type questions. This is done with the `allow_hotkeys :all` statement. Configuring both these behaviors is done by the code in listing 4.2.

Listing 4.2: Behavioral configuration for the Phamous Algemeen questionnaire

```
abortable
allow_hotkeys :all
```

After the questionnaire configuration, the DSL text for a questionnaire follows with a series of panels. Panels are containers for items, and thus introduce a scope for the definitions of the items they contain. We considered two possibilities in the DSL for managing this scope: implicit (listing 4.3) and explicit (listing 4.4).

Listing 4.3: Implicit scope

```
panel
# question here
# question here

panel
# question here
# question here
```

Listing 4.4: Explicit scope

```
panel do
  # question here
  # question here
end

panel do
  # question here
  # question here
end
```

The advantage of implicit scoping is that the domain expert does not need to ensure that the `do`'s and `end`s are balanced. However, we opted to use the explicit scoping style because implicit scoping would be cumbersome for some of the other places we need scoping, as we will see in section 4.4.

Within the scope of a panel, the items of that panel are defined. We described the items as either textual strings, or questions. In the DSL we have three ways of adding textual items that are formatted differently: `title`, `text` and `html`. The `title` method that adds the given string formatted as a heading (large font, bold), shown in listing 4.5. The `text` method processes the given string through a converter that transforms plain text into formatted HTML, and finally `html` simply passes any given string as-is into the rendered HTML-versions of the questionnaire.

Listing 4.5: A panel with a textual heading item

```
panel do
  title "Algemeen"
  # questions here
end
```

The first two questions of the Phamous Algemeen questionnaire ask for a date and a string (listing 4.6). Questions are defined with the same scoping style as panels are, however they have two properties that need to be specified outside of the scope.[1] The first is the identifying key for the question, and the other is the type of question being defined.

Within the block for the question, a lot of attributes are available, although not all attributes are applicable to all types of questions. All question types have attributes like a title and a description, but for instance the unit attribute is only applicable to integer and float typed questions.

Listing 4.6: First two questions of the Phamous Algemeen

```
  question :v_date, type: date do
    title "Datum:"
    validates_presence_of_answer
  end

  question :v_143, type: string do
    title "Naam beoordelaar"
    validates_presence_of_answer
  end
```

4.4 Managing dependencies between questions

The third question in the Phamous Algemeen questionnaire is a multiple choice question (listing 4.7). Possible choices for a multiple choice question are defined with `option`.

[1] This is because the `Question` model expects these properties to be given to the constructor, and the implementation of the expression builder calls its constructor before evaluating the block. Work-arounds for this issue are possible, but domain expert comprehension did not seem to suffer from this small abnormality.

Occasionally, choosing a specific option will need to require further questions to be answered. This is the case for the third question of the Phamous Algemeen questionnaire.

In the Quby DSL, we made it possible to pass in these further questions directly to the option they should fall under, using the same block syntax that the domain experts already use for describing the scope of panels and questions.

Listing 4.7: The third question from the Phamous Algemeen questionnaire

```
question :v_144, type: radio do
  title "Reden voor screening:"
  option :a, value: 1, description: "Jaarlijkse screening"
  option :b, value: 2, description: "Ander interval" do
    question :v_145, type: string do
      title "Screening na"
    end
    question :v_146, type: string do
      title "Reden"
    end
  end
end
```

The method described above can be used when questions are only applicable when a specific option is chosen. In other cases, the relation is the other way around and questions are only applicable unless a specific option is chosen. For these cases, it is possible to specify a set of questions that need to be disabled and hidden when a given option is selected. Defining this relation can be done by specifying which questions should be hidden when a given option is selected, as shown in listing 4.8.

At other times, more complicated relations between questions exist. For instance, it might be required that at least some number of questions of a larger set of questions have an answer. For this, we support adding questions to a group, and specifying the minimum and maximum number of questions that need to have an answer for each group (listing 4.9).

Listing 4.9: Defining groups for questions, and setting limitations on them

```
question :v9, type: radio, question_group: :eetlust,
                           group_minimum_answered => 1,
                           group_maximum_answered => 2 do
  # title and options here
end

question :v10, type: radio, question_group: :eetlust do
  # title and options here
end
```

At the moment, due to time constraints this last type of relation is not yet nicely implemented in the DSL, and is hence managed by the development team instead of the domain experts. This is possible because any additional arguments passed to the `question` method that do not belong to the DSL get passed on to the semantic model directly. Once added, a domain expert can

Listing 4.8: An example of hiding questions, taken from the "FIOS" questionnaire

```
panel do
  question :v_13, type: radio do
    title "Is the patient sociably present in the group?"
    option :a1, value: 1, description: "Never"
    option :a2, value: 2, description: "Occasionally"
    option :a3, value: 3, description: "Sometimes",
        hides_questions: [:v_13a, :v_13b]
    option :a4, value: 4, description: "Often",
        hides_questions: [:v_13a, :v_13b]
    option :a5, value: 5, description: "Always",
        hides_questions: [:v_13a, :v_13b]
  end
end

panel do
  text "In what manner is the patient present in the group?"

  question :v_13a, type: radio do
    title "Dominantly present?"
    option :a1, value: 1, description: "Never"
    option :a2, value: 2, description: "Occasionally"
    option :a3, value: 3, description: "Sometimes"
    option :a4, value: 4, description: "Often"
    option :a5, value: 5, description: "Always"
  end

  question :v_13b, type: radio do
    title "Solitairily present?"
    option :a1, value: 1, description: "Never"
    option :a2, value: 2, description: "Occasionally"
    option :a3, value: 3, description: "Sometimes"
    option :a4, value: 4, description: "Often"
    option :a5, value: 5, description: "Always"
  end
end
```

understand it, but it is not implemented in an intuitive enough manner for a domain expert to be able to work with it on his or her own.

It is possible to imagine far more complicated relations between questions, but in practice we have found that for psychiatric questionnaires the relations we described are enough to support the questionnaires we have seen so far.

4.5 Working with validations

In order to perform validations, we chose to add a number of different methods to the question builder, for different types of questions and validations. These validations are within the scope of a question. The first two questions (shown in listing 4.6) required an answer, and the `validates_presence_of_answer` adds the needed data validations in that case.

Another question asks for a year of the patients first psychotic episode, and needs to ensure that this year makes some logical sense. For integer and float-typed questions, Quby can validate that the value is between some upper and lower bound. For this, the DSL has `validates_minimum`, `validates_maximum` and a shorthand that defines both `validates_in_range`. The last method is used by one of the questions of the Phamous Algemeen questionnaire to ensure that a year entered is between 1900 and the year 3000 (listing 4.5)[2]. This question also uses the `size` method to specify that the width of the text box shown should adjust to have room for only 4 digits, instead of the default of taking up as much width as is available.

```
question :v_1274, type: integer do
  title "Jaar van eerste psychotische episode:"
  size 4
  validates_in_range  1900..3000
end
```

For string-type questions, the DSL also supports specifying a regular expression with `validates_format_with` to validate the string that was entered against, although obviously this is not something a domain expert is expected to be able to add. At the moment, there are only two questionnaires that use this type of validation, since most data format validations are automatically added by choosing the correct type of question.

4.6 Calculating scores

When answers are saved, a set of scores can be calculated. These scores are defined in the questionnaire definition. An example of a score can be seen in listing 4.10. This score calculates the sum of the value of 6 radio-typed questions, where the value of a radio-typed question is the value of the selected option. It also specifies the norm value for this score, which is to say that if the value is greater than 5, the score should be considered higher than normal.

[2]This questionnaire changes yearly so we are not to worried about a Y3K problem.

Listing 4.10: An example of a score definition

```
score :total, label: "Total" do
  {
    value: sum(values(:v1, :v2, :v3, :v4, :v5, :v6)),
    norm: 5
  }
end
```

Defining multiple scores for a questionnaire can be done by simply repeating the given example, albeit with a different key, label and calculation.

A large proportion of the scores that are used in psychiatric questionnaires are as simple as a summation or average of some values. In those cases, the domain experts are able to define the score calculations on their own. In a small portion of the cases however, calculations are more intricate, and in those cases a developer will define the calculation. The body of a score definition supports the full usage of Ruby, which is a benefit we have for creating our DSL as an internal DSL in Ruby, and a developer can use that to write complicated procedural calculations when needed.

In the next chapter we will look at the implementation of the domain-specific language we have shown.

Chapter 5

Expression Builders

In this chapter, we will describe how we implemented expression builders in the Ruby language. In order to explain the usefulness of separating the DSL away from the domain models into expression builders, we will first look at a simpler way of creating domain-specific languages that does not use this separation.

We will call methods that are part of the DSL *fluent methods*, and methods that are part of the regular domain model *command-query methods*. The name for fluent methods comes from fluent interfaces, which is a common name for the interfaces exposed by DSLs. Command-query comes from the Command-Query Separation Principle [14] which describes that methods on domain models should either modify state, or return data, but not both.

5.1 A questionnaire DSL without expression builders

The problem with implementing a DSL directly (as opposed to using expression builders) is that fluent methods then live in the same namespace as command-query methods. This easily leads to name conflicts. To illustrate why these name conflicts happen, we will start implementing a part of the DSL we described in the previous chapter. The pattern for implementing a DSL in this section is described in Design Patterns in Ruby [15].

For example, we could give the `Questionnaire` class a method called `question` that takes some parameters, constructs an instance of a `Question` object from them, and appends the question to an array of questions (appending to an array can be done with `array << element` in Ruby).

```
class Questionnaire
  def question(id, title)
    @questions << Question.new(id, title)
  end
end
```

In order to use this as a DSL, we can construct a `Questionnaire` object, and tell it to evaluate some DSL text. Ruby, being a dynamically interpreted language, has various methods to evaluate a string as code. The `instance_eval` method

takes a string (or a block, which will be explained in section 5.2), and evaluates it within the context of the object it was called on.

```
questionnaire = Questionnaire.new
questionnaire.instance_eval "question :v1, 'What is your
    name?'"
```

After evaluating that string of DSL text, the questionnaire object would have a question object with a title of "What is your name?".

However, the lack of explicit separation between the fluent methods and the command-query methods quickly becomes an issue once a command-query method must be added to the `Questionnaire` class that retrieves the question for a given key. A logical name for this method would be `question`, however that name is already in use by the fluent interface. This leaves two options: either change the fluent interface to use e.g. `add_question`, or use a less desirable name in the command-query interface, e.g. `get_question`. Neither option is particularly desirable, since both solve the problem by muddying up the interface with method names that are less intuitive in their respective contexts.

5.2 Nested closures

The solution is to use separate classes for the fluent methods. These types of classes are called expression builders. An expression builder is a class that translates the fluent interface into method calls on the command-query interface.

Various design patterns exist for the implementation of expression builders, for instance function sequences, method chaining or nested functions. Each pattern supports a specific style of DSL. The style of DSL as described in chapter 4 can be implemented by a pattern that is called a nested closure [9].

A closure is inline code stored in an object, to be evaluated at a later time. In the process of wrapping it up into an object, the scope of the block of code gets captured, so that any references to variables that are defined outside of the block of code will remain intact.

Ruby has three different types of closures: blocks, procs and lambdas [15]. There are some differences in the exact workings between the three types [18], but for the purposes of this thesis they can be considered synonyms.

To demonstrate how closures can be used in a DSL, we can have a look at the example in listing 5.1. To construct a block the syntax `methodname do ... end` is used, where the block itself is whatever code is between the `do` and `end`. The example defines a class called `Example`. Example has getters and setters for a variable called `base`, and a method `calculate` that takes a block as an argument.

The `calculate` method then uses `instance_eval` to evaluate the block within a different scope than the one where the block was originally defined. For this reason, the code in listing 5.1 will ignore the global variable `base` with a value of 200, and instead use the identically named attribute with a value of 100, which results in the value `105` being printed by the last line.

Listing 5.1: Using closures in Ruby

```
class Example
  attr_accessor :base

  def calculate(&block)
    instance_eval(&block)
  end
end

example = Example.new
example.base = 100

base = 200
result = example.calculate do
  base + 5
end

puts result # prints 105
```

5.3 Expression builders in Ruby

Expression builders can be implemented by defining classes for each builder, defining methods to build the desired fluent interface.

The nested closure style of DSLs can then be implemented by taking a block of DSL text, and using `instance_eval` to evaluate that block in the scope of the expression builder, which makes the fluent interface available to the block.

An expression builder for the nested closure is implemented by defining a class method `build` that instantiates a new builder, evaluates the given definition string or block within the scope of that builder, and finally returns the object the builder built (listing 5.2, lines 2-7). A instance of an expression builder then has an attribute to hold the object it is building (line 9), and its constructor fills that attribute with a new object (line 12).

Expression builder classes inherit from the `BasicObject` class (line 1). This is done so that they contain as few methods of their own as possible, so that almost only the fluent interface is available. This reduces the chance that a typo would cause a method that is built into the Ruby language to be called, and therefore reduces the chance of confusion when the DSL text gets evaluated.

This expression builder can then be used by calling the `QuestionnaireBuilder.build` class method with either a string of DSL text, or a block containing DSL text (listing 5.3).

Implementing the DSL as shown in Figure 4.3 is then done by adding methods to `QuestionnaireBuilder` that call command-query methods on the `Questionnaire` instance. A few examples are shown in listing 5.4.

Listing 5.2: Empty expression builder for questionnaires

```
class QuestionnaireBuilder < BasicObject
  def self.build(definition = nil, &block)
    builder = self.new
    builder.instance_eval definition if definition
    builder.instance_eval &block     if block_given?
    return builder.questionnaire
  end

  attr_accessor :questionnaire

  def initialize
    @questionnaire = Questionnaire.new
  end
end
```

Listing 5.3: Two ways of using an expression builder

```
QuestionnaireBuilder.build("# DSL text")

QuestionnaireBuilder.build do
  # DSL text
end
```

Listing 5.4: Examples of how to implement the fluent interface

```
class QuestionnaireBuilder < BasicObject
  def title(string)
    @questionnaire.title = string
  end

  def abortable
    @questionnaire.abortable = true
  end

  def allow_hotkeys(value = :all)
    @questionnaire.hotkeys = value
  end
end
```

5.4 Delegating to other expression builders

Fluent methods that introduce a new scope, e.g. `panel` or `question`, are implemented as methods that take a block, and pass it on to another builder. For instance, the `panel` fluent method delegates to `PanelBuilder.build` to construct a panel from the definition within the block given to the `panel` method (listing 5.5), and appends that panel to the array of panels that the questionnaire has.

Listing 5.5: The `panel` fluent method delegates to `PanelBuilder`

```
class QuestionnaireBuilder < BasicObject
  def panel(&block)
    panel = PanelBuilder.build(&block)
    @questionnaire.panels << panel
  end
end
```

The fluent method `question` takes some arguments, as discussed in section 4.3. In the fluent method in the `PanelBuilder` class, these options are received and passed on to the `QuestionBuilder` (listing 5.6).

Listing 5.6: The `question` fluent method

```
class PanelBuilder < BasicObject
  def question(id, options = {}, &block)
    @panel.items << QuestionBuilder.build(id, options, &
        block)
  end
end
```

Within the `QuestionBuilder` (listing 5.7), the type parameter is retrieved from the options. The class for the given type is automatically determined. This works by taking the type (e.g. `:radio`), transforming it to a strings (`"radio"`), concatenating it with `"_question"` (`"radio_question"`). After that, the classify method turns that string into a camel-cased version (`"RadioQuestion"`), and the constantize method turns that into an actual class (`RadioQuestion`).

After this trick to find the correct class for the given type of question, the code is similar to other expression builders.

Listing 5.7: The `QuestionBuilder`

```
class QuestionBuilder < BasicObject
  def self.build(id, options = {}, &block)
    type  = options.delete(:type)
    klass = (type.to_s + "_question").classify.constantize
    @question = klass.new(id, options)
    @question.instance_eval &block
    return @question
  end
end
```

5.5 Convenience methods

The type parameter for a question in the semantic model is a symbol. However, in the DSL we wanted to minimize the amount of syntax needed. In order to make it possible to write `question :v1, type: radio` in DSL text, a convenience method for `radio` can be added that returns a `:radio` symbol (listing 5.8).[1]

Listing 5.8: Convenience methods added to `QuestionBuilder`

```
class QuestionBuilder < BasicObject
  def radio
    :radio
  end

  def check_box
    :check_box
  end

  # same for scale, string, date, integer,
  # float, textarea and select
end
```

5.6 Score definitions

For score calculations, the block of the definition needs to be stored to be evaluated against a specific filled out questionnaire when an answer gets saved. To do this, we store the block in a `Scorer` object. Upon saving an answer, the block stored in the scorer is run against expression builder that also knows the values of the answer and the age and gender of the patient.

In order to do this, instead of using `instance_eval`, we pass the block directly into the `Scorer` class to be stored.

```
class QuestionnaireBuilder < BasicObject
  def score(id, options = {}, &block)
    scorer = Scorer.new(id, options, &block)
    @questionnaire.scores << scorer
  end
end
```

Before an answer is saved, its scores are calculated. To this end, the answer model loops over all scores defined in the questionnaire, and calls the `ScoreCalculator` class.

It is only at this point that the block from the questionnaire definition get executed. This is done by the `ScoreCalculator` class. Its structure is similar to the rest of the expression builders. To calculate to scores, it is given the answer values for the filled out questionnaire, patient data and the block that was originally defined in the questionnaire definition.

[1] Note to Rubyists: The implementation in this paper is written in this manner for pedagogical reasons. In the actual code we loop over an array and dynamically define methods for each element.

Listing 5.9: Score calculator

```
class ScoreCalculator
  def self.calculate(values, patient, &block)
    calculator = self.new(values, patient)
    calculator.instance_eval(&block)
    return calculator.score
  end
end
```

Chapter 6

Results

We have shown how to create domain-specific languages in the Ruby language using expression builders, in a manner that is extensible and testable.

Quby's DSL is currently working to satisfaction at the RGOc. Our questionnaire database currently contains over 190 different questionnaires, and all of these have been defined by two of our domain experts with almost no help from the development team. The domain experts frequently mention how much easier their work is with Quby than it was with the GlobalPark system.

Quby is also used by RoQua as its questionnaire engine to deliver all questionnaires to patients. We are currently working on expanding the DSL to include the calculation of scores from the answered questionnaires.

So clearly, in terms of actual usage, Quby has not been unsuccessful. However, in order to see how well Quby's DSL approach performs (in terms of usability by domain experts) when compared to GlobalPark, we have also performed a small trial.

6.1 Trial setup

In this trial, we asked three users of GlobalPark to try and build two questionnaires (that we provided). We asked them to record how long it took them for each questionnaire/system-combination and at the end we asked them to fill out a short survey.

These users are not developers, but do have some experience working with scripting in programs like SPSS, which they use to analyze datasets for their day jobs.

None of these users had ever worked with Quby or our DSL before. They did have experience with building psychiatric questionnaires in the GlobalPark system.

We realize that a small trial with only three participants is not enough to provide any definitive answers, but it does help to provide some insights into the effectiveness and understandability of the DSL and editor for new users.

The questionnaires we asked the participants to create were the "Phamous Algemeen" (PHAM) and the "ROM AZM" (AZM). The PHAM is the questionnaire as seen in Figure 4.2, a questionnaire with a few different types of questions: a few text fields and a few radio buttons. The AZM is a more repetitive questionnaire, consisting of 12 radio questions where the questions are all similar except for the question text itself. We chose these in order to see if there is some difference in working with repetitive questionnaires versus working with a questionnaire where each question is unique in form.

Both are types of questionnaires that we often have to build although in practice we tend to find that 90% of the 140 questionnaires we currently have implemented in Quby are almost only repetitive of nature.

While they worked on building the questionnaires, the screen of the participants was captured for later analysis.

At the end their session, the participants were given a set of questions to answer about their experiences.

In the following sections, the names of the participants are fictional.

6.2 Timings

One of the things we measured where how long it took the participants to build each of the questionnaires. First of all, in table 6.1 are the raw durations that the participants took to build.

Table 6.1: Time taken per participant, per questionnaire

Who	Questionnaire	System	Time	Order
Andrew	phamalg	globalpark	10m	(fourth)
Andrew	phamalg	quby	13m	(second)
Andrew	romazm	globalpark	13m	(third)
Andrew	romazm	quby	11m	(first)
Barbara	phamalg	globalpark	13m	(first)
Barbara	phamalg	quby	44m	(third)
Barbara	romazm	globalpark	18m	(second)
Barbara	romazm	quby	21m	(fourth)
Claire	phamalg	globalpark	15m	(second)
Claire	phamalg	quby	20m	(third)
Claire	romazm	globalpark	27m	(first)
Claire	romazm	quby	7m	(fourth)

Another way of looking at these times is by tallying them up for each system, and measuring the difference between the two (table 6.2).

We can see that there is no conclusive answer arising from these times. One participant took about twice as long in Quby as she did in GlobalPark, while

Table 6.2: Summarized times per participant per engine

Who	GlobalPark	Quby	Difference
Andrew	23m	24m	104%
Barbara	31m	65m	209%
Claire	42m	27m	64%

another spent about the same time in either, and a third was about one-thirds quicker in Quby. In the discussion section we will explain these findings.

6.3 Ratings

Upon completion of the tasks set to the participants, we asked them to give their opinions on a few areas, rated on a scale of 1 to 9. These ratings are shown in table 6.3. On the scale used, a 1 represents a strong preference to GlobalPark, a 9 represents a strong preference to Quby, and thus a 5 represents an indifference between the two.

Table 6.3: Ratings given by participants

Area	Andrew	Barbara	Claire	**Average**
General impression	6	3	7	**5.3**
Workflow	7	3	7	**5.6**
Intuitiveness	7	4	7	**6.0**
Usage for ROM AZM	7	2	3	**4.0**
Usage for PhamAlg	7	4	3	**4.6**

6.4 Answers to interview questions

After the participants had built the questionnaires, we also asked them a few open-ended questions.

If you had to create 20 questionnaires, where a large number would be similar in terms of structure, which system would you prefer to use, and why?
Andrew: Quby. Copying and pasting is much faster.
Barbara: GlobalPark, because I know the most about it. But in order to make a good judgement I would have to work a little more with Quby.
Claire: I think GlobalPark. The advantage of GlobalPark is that you can easily copy a question, or an entire questionnaire. You can do that in Quby too, but in GlobalPark it is a little easier because there is a button for it.

Would you recommend GlobalPark or Quby to a collegue?
Andrew: Both, depending on their computer knowledge. With larger questionnaires rather Quby.

Barbara: I don't really have any experience with Quby, which I do have with GlobalPark, so I would sooner recommend GlobalPark.
Claire: I would recommend Quby because it is easier to enter different datatypes (in GlobalPark I could not find the 'date' type). I also found the layout easier to customize.

What level of programming knowledge do you estimate someone needs in order to be able to work with Quby? More or less than your own level?
Andrew: A little more. I have a basic knowledge of HTML, but no more.
Barbara: It is probably about my level, but having time to learn about the questionnaire to be built, and learn about existing examples in the editor instead of having work with it immediately would make it easier.
Claire: About the same, so 5.

Do you have any recommendations or things that need improvement for either of the questionnaire systems?
Andrew: Make error messages clearer.
Barbara: -
Claire: With both of the systems I had to look at the manual frequently, because it has been over a year since I last used GlobalPark and Quby was completely new to me. Before I started answering these questions, I had the impression that Quby was easier than GlobalPark. Yet I did take longer in Quby, because I had to keep looking for (typing) errors. Quby does give an error message, but it is very unclear. It would be easier if it clearly stated what error it was (and not just what line). Or even better: if while typing it would immediately indicate that you are doing something wrong (e.g. Word when the spell checker is enabled).

Chapter 7

Discussion and Conclusions

We set out to determine if it is possible to develop a Ruby-based internal DSL that is suitable for use by non-programmer domain experts. To this end, we designed and implemented a DSL for questionnaires, and evaluated its effectiveness through a trial. The trial results are somewhat inconclusive: out of three domain experts one as twice as fast in Quby, while another was twice as slow.

Apart from the trial, Quby has been in commercial use and nearly 200 questionnaires have been defined by non-programmers. This is evidence that the DSL is comprehensible.

Overall, we have found DSLs to be simpler to use than graphical user interfaces. The benefits of a DSL include that it's easier to work with repetitive structures by copy/pasting previous sections of DSL text, and that it is easier to maintain overview of a questionnaire because no unused GUI elements are shown, but only relevant information.

The biggest problem is that non-programmers are not used to the precision required by parsers. In our case this manifested itself in string quoting and block delimiters. This problem is exacerbated by the fact that parsers are usually bad at determining the line of these types of parse errors.

7.1 Trial

In the previous chapter we described the results from a trial we took to find out whether domain experts would be able to effectively work with Quby's DSL. To this end, we asked three domain experts to build two different questionnaires in both GlobalPark, which is a web-based questionnaire service they were accustomed to, and in Quby, the system we built.

We realize that doing a trial with only three people can in no way be considered statistically relevant, however finding a large group of domain experts was unfeasible. It is also important to remember that the participants were GlobalPark users, and that we are thus comparing a known system (GlobalPark) to an unknown system (Quby).

In addition to the results we measured in our trial, we also made observations while working with domain experts to develop the Quby questionnaire engine.

7.2 Successes

In working with domain experts, the most often heard benefit of using a DSL-based system over using GUI-based tools is that plain text is much easier to copy and paste. This means that it is easy to provide examples on how to approach certain more complicated questionnaire designs, and it is also easier to communicate about those examples, because you can also paste them into an e-mail. However, it also confused at least one participant because there was no graphical button to click to copy/paste questions.

Looking at the time results from the trial (table 6.2), we conclude that working with the DSL can be at least as fast as working with a GUI tool, especially when we take into consideration that the users were already familiar with GlobalPark, but were completely new to Quby. For this reason, we feel we can believe the claims of the domain experts with whom we worked while building Quby, that they indeed can work more efficiently with the DSL-based approach than they were with the GUI-based approach used by GlobalPark.

There are also secondary benefits gotten from using a DSL. It meant that questionnaire definitions could be treated more like regular source code, and allowed us to manage them in a version control system. This means that we get the benefits that such a system provides, like easily getting the differences between two versions, and making sure that the version that is currently in active use doesn't change between releases of RoQua.

7.3 Problems

Early on in the project, the editor in the web interface was a very simple text area with a submit and a preview button. This was functional, but in actual usage we noticed that syntax errors were a frequent occurrence. Most of these were missing opening or closing double quotes. Once we integrated syntax highlighting into the editor, we noticed a very large drop in these types of errors.

However, when they do occur, the error messages are still far to cryptic. We have not had the time yet to have a look at how to get clear understandable errors with good line number detection. Currently we are simply outputting the stack trace Ruby gives us. From this, we can tell the user the line number, but as is often the case in parsers, this number can be off by a few lines. Programmers get used to this, and know how to look around for the source of an error. For non-programmers, it is extremely confusing to get a line number for a line that is in itself perfectly fine.

On the other hand, we saw in Figure 2.4 that while GlobalPark presents a graphical editing interface, its errors are confusing as well.

Listing 7.1: An example of using implicit scoping

```
panel
question :q1, type: string
title "What is your name"

question :q2, type: string
title "What is your favorite colour?"

panel
question :q3, type: radio
title "A third question on a new panel"
option :a1, description: "Yes"
subquestion :q3a, type: radio
subtitle "A subquestion"
suboption :a1 "Option A"
suboption :a2 "Option "B
option :a2, description: "No"
```

With Quby we solve this issue by letting domain experts call in the help of a developer when they get stuck, cannot quickly figure out some error, or simply do not know how to do whatever they want to accomplish. In our case developers are located in the same hallway as the domain experts. Domain experts can visit one of the developers who can then provide the assistance needed. This support takes about 5 minutes of developer time per week per domain expert. Obviously, this is not a solution that will work for everybody, or a solution that will scale to a large number of domain experts.

This explains what happened with Barbara, who took twice as long in Quby in comparison to GlobalPark. Participants were instructed to contact a RoQua developer in case they got stuck, in the same manner that we do for the domain experts that we worked with to build Quby. However, this participant spent a long time trying to figure out an error message before finally contacting us.

The types of syntax errors we saw participants make primarily were string quoting (unescaped double quotes within a string), and unbalanced `do [...] end` pairs.

It might be possible to change the DSL in such a way that these pairs would not be needed, changing from explicit scoping to implicit scoping (e.g. listing 7.1). For panels and top-level questions, this would probably work fine, but it causes problems with nested questions (as discussed in section 4.4) because it would be impossible to know if `question` should define a new top-level question, or if it should be a subquestion for the last given `option`. One option would be to add `subquestion` and `suboption`, but that might grow unwieldy once you get to `subsubsubsubquestion`[1].

[1] Reminiscent of Common Lisp's `cdadar` function to retrieve the value of the first of the rest of the first of the rest of a given list.

7.4 Future work

In designing our DSL we made various trade-offs in terms of the syntax supported by the DSL, like the choice between implicit and explicit scoping (section 4.3). In general, we have found little research that surveys the landscape of DSL syntaxes. Such research could find common features and design decisions of DSLs, and determine what types of DSLs are the easiest to comprehend.

For the implementation of DSLs, one idea we had was to develop a Ruby DSL for developing DSLs. We have started implementing this in an open-source library called ActiveDSL [19]. At the moment this library is able to simplify the creation of DSLs in Ruby, but unfortunately it does not yet have enough features to be able to use it for the DSL used in Quby.

Easier implementation of DSLs is one of the features of a language workbench, and this might be a good direction to take ActiveDSL. A future version of ActiveDSL could provide not only simplified implementation, but also provide editing tools that are aware of the structure of the DSL.

A further study into the effect of the editing environment on DSL comprehension could provide valuable insights. The editing environment we built for Quby is pretty sparse, and such a study would be helpful to determine what types of features we should add to the editor to improve the usability for domain experts.

One possible editor feature would be to highlight lines in case of errors. For instance, lines could be grayed out if it is certain the error is not in those lines, colored orange if it is possible that the error is on one of them, and finally the line that the syntax checker thinks the error is on could be highlighted red. Highlighting lines in this manner could make it clear to non-programmers that where the error occurred can not be determined precisely.

Another feature we intend on adding in the near future is to perform syntax checking live in the background, as the user is typing. Currently this check is performed only upon saving, and doing this live might be a better idea. One issue might be that it could be too annoying if the editor complained constantly until the final `end` for a panel was typed. This is something we will examine in practice.

To help domain experts be more productive, a feature like text snippets could be added. For instance, typing `question` could automatically expand to a complete basic question, which the tab key moving between areas that need to be changed in. Autocompletion could help with discoverability, typing `question :v1, type:` could result in a drop down appearing, showing the various types of question supported.

A final feature might be to have an importer that reads in questionnaires from Word files and converts it to a DSL script, probably guided by the domain expert. If the domain expert were to mark some pieces of text as questions and options, the importer might be able to figure out the common format and suggest a conversion for the rest of the document.

We also wish to instill some development practices onto the domain experts, particularly unit testing. It would be useful to be able to automatically verify

that score calculations are correctly written, and an extension of the Quby DSL could make that possible. The questionnaire editor currently does provide the ability to try out a calculation by filling out the questionnaire and looking at the resulting scores. For automatic verification the known good value for a given filled out questionnaire should be recorded somewhere.

Appendix A

Full questionnaire definitions

This appendix contains the full questionnaire definitions for the "Phamous Algemeen" questionnaire that is used a a running example throughout this thesis, and the "ROM AZM" questionnaire that was used in the trial we conducted.

A.1 Phamous Algemeen

```
title "Phamous Algemeen"
short_description ""
description ""
outcome_description ""

abortable
allow_hotkeys :all

panel do
  title "Algemeen"

  question :v_date, type: date do
    title "Datum:"
    validates_presence_of_answer
  end

  question :v_143, type: string do
    title "Naam beoordelaar"
    validates_presence_of_answer
  end

  question :v_144, type: radio do
    title "Reden voor screening:"
    option :a, value: 1, description: "Jaarlijkse screening"
    option :b, value: 2, description: "Ander interval" do
      question :v_145, type: string do
        title "Screening na"
      end
```

```
      question :v_146, type: string do
        title "Reden"
      end
    end
  end

  html "<hr/>"

  question :v_1274, type: integer do
    title "Jaar van eerste psychotische episode:"
    size 4
    validates_in_range  1900..3000
  end

  question :v_1275, type: integer do
    title "Jaar van eerste GGZ-contact"
    size 4
    validates_in_range  1900..3000
  end

  question :v_178, type: radio do
    title "Etniciteit:"
    option :a1, value: 1, description: "Caucasisch (blank)"
    option :a2, value: 2, description: "Negroide"
    option :a3, value: 3, description: "Aziatisch"
    option :a4, value: 4, description: "Indiaans / latijns-
        amerikaans"
    option :a5, value: 5, description: "Turks"
    option :a6, value: 6, description: "Marokkaans"
    option :a7, value: 7, description: "Anders" do
      question :v_179, type: string do
        title "Namelijk"
      end
    end
  end
end
```

A.2 ROM AZM

```
title "ROM AZM"
short_description ""

panel do
  text "Heel kort! Help ons begrijpen hoe u zich voelt....."
  text "Kunt u ons vertellen hoe u zich de afgelopen week,
      tot en met vandaag, hebt gevoeld? We vragen u een '
      rapportcijfer' te geven op een aantal gebieden."

  text "Lees elke vraag goed door en omcirkel het getal dat
      uw huidige situatie het best beschrijft."

  text "Hieronder wordt 'werk' gedefinieerd als baan, school
      , huishoudelijk werk, vrijwilligerswerk, enz."

  text "Bij 'belangrijke relaties met andere mensen' kunt u
      denken aan echtgenoot/echtgenote, levensgezel/
      levensgezellin, ouders/kinderen, vrienden,
      gezinsrelaties - wat voor u het belangrijkste is."
end

panel do
  question :v_1, type: radio do
    title "1. Hoeveel last heeft u van psychische klachten?"
    option :a1, value: 0, description: "Niet of nauwelijks"
    option :a2, value: 1, description: "Nogal"
    option :a3, value: 2, description: "Zeer veel"
  end

  question :v_2, type: radio do
    title "2. Hoeveel last heeft u van lichamelijke klachten
        ?"
    option :a1, value: 0, description: "Niet of nauwelijks"
    option :a2, value: 1, description: "Nogal"
    option :a3, value: 2, description: "Zeer veel"
  end
end

panel do
  question :v_3, type: radio do
    title "3. Hoeveel interfereren de psychische klachten
        met uw (zoeken naar) werk?"
    option :a1, value: 0, description: "Niet of nauwelijks"
    option :a2, value: 1, description: "Nogal"
    option :a3, value: 2, description: "Zeer veel"
  end

  question :v_4, type: radio do
    title "4. Hoeveel interfereren de lichamelijke klachten
        met uw (zoeken naar) werk?"
    option :a1, value: 0, description: "Niet of nauwelijks"
    option :a2, value: 1, description: "Nogal"
```

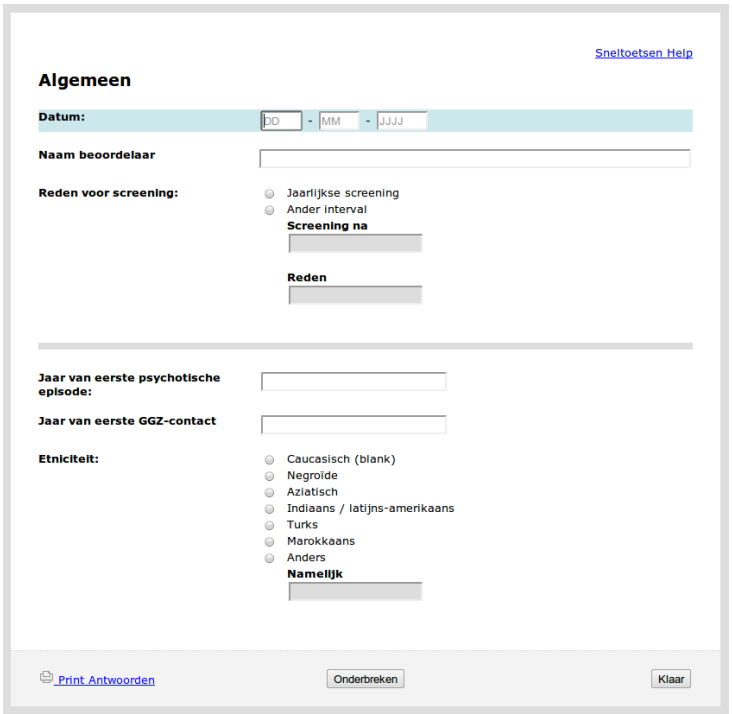
```
    option :a3, value: 2, description: "Zeer veel"
  end

  question :v_5, type: radio do
    title "5. Hoeveel interfereren de psychische klachten
        met belangrijke relaties met andere mensen?"
    option :a1, value: 0, description: "Niet of nauwelijks"
    option :a2, value: 1, description: "Nogal"
    option :a3, value: 2, description: "Zeer veel"
  end

  question :v_6, type: radio do
    title "6. Hoeveel interfereren de lichamelijke klachten
        met belangrijke relaties met andere mensen?"
    option :a1, value: 0, description: "Niet of nauwelijks"
    option :a2, value: 1, description: "Nogal"
    option :a3, value: 2, description: "Zeer veel"
  end
end

panel do
  question :v_7, type: radio do
    title "7. Hoe tevreden bent u momenteel met uw leven in
        zijn geheel?"
    option :a1, value: 0, description: "Zeer Ontevreden"
    option :a2, value: 1, description: "Gaat wel"
    option :a3, value: 2, description: "Zeer Tevreden"
  end

  question :v_8, type: radio do
    title "8. Hoe tevreden bent u met de zorg op deze
        afdeling tot nu toe?"
    option :a1, value: 0, description: "Zeer Ontevreden"
    option :a2, value: 1, description: "Gaat wel"
    option :a3, value: 2, description: "Zeer Tevreden"
  end
end
```

Appendix B

Documentation provided to trial participants

We provided the following documentation with regard to the structure of the questionnaires that needed to be built during the trial.

B.1 Phamous Algemeen



B.2 ROM AZM

Heel kort! Help ons begrijpen hoe u zich voelt.....
Kunt u ons vertellen hoe u zich de afgelopen week, tot en met vandaag, hebt gevoeld? We vragen u een "rapportcijfer" te geven op een aantal gebieden.

Lees elke vraag goed door en omcirkel het getal dat uw huidige situatie het best beschrijft.

Hieronder wordt "werk" gedefinieerd als baan, school, huishoudelijk werk, vrijwilligerswerk, enz.

Bij "belangrijke relaties met andere mensen" kunt u denken aan echtgenoot/echtgenote, levensgezel/levensgezellin, ouders/kinderen, vrienden, gezinsrelaties - wat voor u het belangrijkste is.

1. Hoeveel last heeft u van psychische klachten?

Niet of nauwelijks Nogal Zeer veel

2. Hoeveel last heeft u van lichamelijke klachten?

Niet of nauwelijks Nogal Zeer veel

3. Hoeveel interfereren de psychische klachten met uw (zoeken naar) werk?

Niet of nauwelijks Nogal Zeer veel

4. Hoeveel interfereren de lichamelijke klachten met uw (zoeken naar) werk?

Niet of nauwelijks Nogal Zeer veel

5. Hoeveel interfereren de psychische klachten met belangrijke relaties met andere mensen?

Niet of nauwelijks Nogal Zeer veel

6. Hoeveel interfereren de lichamelijke klachten met belangrijke relaties met andere mensen?

Niet of nauwelijks Nogal Zeer veel

7. Hoe tevreden bent u momenteel met uw leven in zijn geheel?

Zeer ONtevreden Gaat wel Zeer tevreden

8. Hoe tevreden bent u met de zorg op deze afdeling tot nu toe?

Niet of nauwelijks Nogal Zeer veel

Appendix C

Ruby Syntax Guide

For those who have no or little prior knowledge of the Ruby programming language, in this appendix we will provide some details on some elements of the language whose workings might not be immediately apparent. We do this by listing a snippet of Ruby code, and showing the equivalent Java code.

Ruby	Java

`attr_accessor :foo`	`private type foo;`
	
	`public type getFoo() {`
	`  return foo;`
	`}`
	
	`public void setFoo(type foo) {`
	`  this.foo  = foo;`
	`}`

`def self.methodname`
`  # code`
`end`
	`public static type methodname() {`
	`  // code`
	`}`

`something if condition`	`if (condition) { something; }`
`something unless condition`	`if (!condition) { something; }`

`def initialize`
`  # code`
`end`
	`public ClassName() {`
	`  // code`
	`}`

References

[1] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[2] K. Beck and M. Fowler. *Planning extreme programming*. Addison-Wesley Professional, 2001.

[3] M. Van den Brand et al. "Industrial applications of ASF+ SDF". In: *Algebraic Methodology and Software Technology* (1996), pp. 9–18.

[4] *Cucumber*. URL: `http://cukes.info/`.

[5] E. De Beurs et al. "Routine outcome monitoring in the Netherlands: practical experiences with a web-based strategy for the assessment of treatment outcome in clinical practice". In: *Clinical psychology & psychotherapy* 18.1 (2011), pp. 1–12.

[6] Arie Van Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages : An Annotated Bibliography *". In: *ACM Sigplan Notices* 35.June (2000), pp. 26–36.

[7] Aran Donohue. "Debugging Domain-Specific Languages". PhD thesis. University of Toronto, 2010. URL: `http://www.arandonohue.com/writing/MSc/ut-thesis.pdf`.

[8] M. Fowler and J. Highsmith. "The agile manifesto". In: *Software Development* 9.8 (2001), pp. 28–35.

[9] Martin Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321712943, 9780321712943.

[10] Martin Fowler. "Language workbenches: The killer-app for domain specific languages". In: *Accessed online from: http://www. martinfowler. com/articles/languageWorkbench. html* (2005), pp. 1–27. URL: `http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf`.

[11] J. Gray et al. "DSLs: the good, the bad, and the ugly". In: *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM. 2008, pp. 791–794.

[12] J. Greenfield et al. *Software factories*. Citeseer, 2004.

[13] John Gruber. *Markdown*. URL: `http://daringfireball.net/projects/markdown/`.

[14] B. Meyer. *Object-oriented software construction*. Prentice Hall PTR, 1997. ISBN: 9780136291558.

[15] Russ Olsen. *Design Patterns in Ruby*. 1st. Addison-Wesley Professional, 2007. ISBN: 0321490452, 9780321490452.

[16] *RSpec*. URL: https://www.relishapp.com/rspec.

[17] M. Slade. "Routine outcome assessment in mental health services". In: *Psychological medicine* 32.08 (2002), pp. 1339–1343.

[18] Robert Sosinski. *Understanding Ruby Blocks, Procs and Lambdas*. URL: http://www.robertsosinski.com/2008/12/21/understanding-ruby-blocks-procs-and-lambdas/.

[19] M Veldthuis. *ActiveDSL*. 2011. URL: https://github.com/marten/active_dsl.

[20] M.P. Ward. "Language-oriented programming". In: *Software - Concepts and Tools* 15.4 (1994), pp. 147–161. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.5062\&rep=rep1\&type=pdf.