Universität Ulm | 89069 Ulm | Germany

Faculty of Engineering, Computer Science and Psychology
Databases and Information Systems Department

Development of a generic concept to process questionnaire result data in different statistical applications

Bachelor's thesis at Universität Ulm

Submitted by:
Sean Duft
sean.duft@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert

Supervisor:
Johannes Schobel

2018

Version from December 28, 2018

CompositiDIe on: PDF-LaTeX 2_ε

Abstract

Up to this day, data collection in domains like medicine, psychology or education is mostly carried out manually through questionnaires. These questionnaires are often associated with a lot of effort, because each participant needs a physical copy of the questionnaire. Furthermore, the results of the survey need to be evaluated manually or have to be transferred to a statistical application by hand. This is not only time consuming, but also an opportunity for potential errors. The *QuestionsSys* project tries to solve these issues by providing a flexible framework, that can be used to create and configure questionnaires, to deploy them on smart mobile devices and to store collected data.

In order to analyze this digitally collected data, it needs to be imported into a statistical tool where it can be analyzed further. The task of importing data requires a high amount of programming knowledge, which makes it rather difficult and time consuming for someone not familiar with programming. To compensate this, this thesis introduces a generic concept to import questionnaire result data into statistical tools. Furthermore, the *QuestionSys* project will be expanded with concrete implementations for the statistical programming language *R* as well as for *Microsoft Office Excel*

Acknowledgment

I want to thank everybody who helped me during the creation of this thesis, first and foremost my family and friends for their constant support.
Particularly, I have to thank Johannes Schobel for his supervision of my work, helpful advice and fast responses to any of my questions.

Thanks.

Contents

1 Introduction

In the past, the task of collecting data was mainly performed by telephone surveys, face-to-face interviews or mailing questionnaires to applicants. In contrast to these manual forms of data collection, with the increasing advancement and distribution of electronic devices, more and more surveys are performed digitally. These digital surveys may be realized as video chat interviews, social media surveys or smart mobile device surveys [1].

Each variant of data collection comes with its own set of advantages and disadvantages. Data collection through digital questionnaires often have a higher response rate, lower costs per answer, fewer missing or forgotten answers as well as a higher flexibility in the design of questionnaires [2]. On the other hand, domain specific knowledge is needed to design a valid and meaningful questionnaire and programming knowledge is needed to actually create it. Because of this, it requires a close cooperation between domain experts and application developers to create digital questionnaires, which is often costly and creates room for misunderstandings and errors. The *QuestionSys* projects aims to enable domain experts without programming knowledge to create and maintain digital questionnaires by themselves. This is to be achieved by creating a process driven framework for creating and editing questionnaires, an application on smart mobile devices for executing questionnaires and a server that stores created questionnaires as well as collected data [3].

1.1 Problem statement

Downloading data from a server used for digital data collection is rather difficult for someone without advanced programming knowledge. Resulting from this, any project that aims to minimize the expenditure of creating digital data collection processes needs to contain a component that encapsulates the technical aspect of requesting result data from the server. Due to the fact that the server may use a data format not well suited to analyze the data efficiently, it is necessary to convert the requested data. To minimize effort and improve usability, this should be performed by the component that is requesting the data, without the user noticing it. This thesis aims at developing a generic structure that is supporting this issue, as well as providing an implementation of this structure in the statistical programming language *R* as well as the development of a plugin for *Microsoft Excel*.

1.2 Objective

The goal of this thesis is to develop a generic concept and implement a tool in the context of the *QuestionSys* project, that makes it possible for users to import questionnaire result data from the *QuestionSys* server into their preferred tool for statistical analysis, such as *Microsoft Excel* or the statistical programming language *R*. This tool should provide a simple and intuitive way for users that are not familiar with programming to request and update the data obtained from their questionnaires, import it to their selected statistical tool and display the obtained data. To ensure that every user is capable of requesting data this way, the application should provide an appropriate user interface. This interface should include options to set all required parameters such as the location of the data as well as the amount of requested data. Additionally, the structure of the application should be simple enough so that it can be easily extended with new features, as the *QuestionSys* project is still in development and bound to change as the project progresses.

1.3 Structure of the Thesis

This thesis begins by explaining the general motivation for the *QuestionSys* project in chapter 2, as well as introducing the different components of this project. Chapter 5 compares the *QuestionSys* project to other approaches at digital data collection. After that, Chapter 3 analyzes which requirements need to be fulfilled to meet the goal of this thesis. Afterwards, Section 3.2 specifies what differences and particularities occur between the implementations of the generic structure in *R* and *Excel*. Subsequently, Chapter 4 introduces the general concept of the application. Each phase of this structure will be described in detail, followed by an explanation of the implementations in *R* and *Excel*. Finally, Chapter 6 summarizes this thesis and explains what changes likely will be made to the application in the future.

2 Background

Data collection is an integral part in every domain of scientific research. Without raising and collecting data, validating any scientific thesis is impossible. In economics, medicine and psychology, surveys are the most common form of large scale data collection. Following the technological progress, survey methods are getting more and more refined, and even new survey methods develop, as the distribution of new technological devices increases. In the 1960s, telephone surveys became more and more popular, internet surveys started appearing in the 1990s. Finally, in this decade, mobile surveys are increasingly used to collect data on a large scale [4].

Each existing form of survey comes with its own advantages, as well as with its own disadvantages. Conventional forms of data collection, such as surveys, paper based questionnaires or telephone surveys, are already proven valid, there is lots of experience in the design of these types of surveys. Through this large amount of knowledge, domain experts know what affects response rates of their surveys positively and whats affects them negatively, as well as how to design a survey that achieves a high response rate. There are standard procedures to perform a representative survey, like the method of random number dialing in telephone surveys [5]. On the downside, in comparison, manual data collection is expensive per data set. Raised data has to be gathered and evaluated. In this process, it takes a lot of caution to digitalize this data without errors, so that it can be digitally analyzed and evaluated. Also, depending on the extend of the survey or questionnaire, a significant amount of time is taken up by this process [2].

On the other hand, digital data collection provides more freedom in the design and structure of surveys. Question descriptions are not limited in length as they are in traditional media. Multimedia files may be embedded in surveys, such as links to web pages, audio, image or video files [6]. The order of questions or even the text of the questions may change based on answers to previous questions. Digital data collection is much more efficient for receiving and evaluating data, furthermore, the risk of errors while transcribing collected data to electronic worksheets is reduced to a minimum. Manual and digital data collections both share similar response rates and reliability, but digital data collection comes with the benefit of significantly lower number of missing answers. Contrary to these benefits, digital data collection is in need of more preparation time compared to manual data collections. The software for the digital survey needs to be created and adapted to the special tasks for the specific situation, in addition to the expenditure of creating the questions. Testing and correcting errors in digital questionnaires is harder than in printed questionnaires, because a more complex structure leads to more special cases that need to be considered [2]. To this day, creating digital surveys often requires the cooperation of domain experts, providing the knowledge, structure and questions of the survey, and programming experts, implementing the survey. This is often times very expensive and missing communication and misunderstandings are a huge cause for errors in the finished product as well as delays and even higher costs [3].

2.1 The QuestionSys Framework

The *QuestionSys* projects aims to improve some of the disadvantages that data collection through digital questionnaires on smart mobile devices face. The main goal is to enable domain experts from domains like medicine and psychology to design, create and modify digital questionnaires for smart mobile devices, without the need for experts in programming. To achieve this, the projects aims to create a generic framework that is able to support the entire lifecycle that questionnaires for smart mobile devices undergo. This lifecycle can be divided into the following five distinct phases:

Design and Modeling In first phase, domain experts create the data collection model, including navigation logic that determines what question is shown depending on answers to previous questions.

Deployment In the second phase, the created model is deployed to a mobile device capable of executing the questionnaire.

Enactment and Execution In the third phase, different instances of the data collection model are created and the individual questionnaires are executed to collect data.

Monitoring and Analysis This data is analyzed in the next phase, both on the mobile device as well as on the back-end system.

Archiving and Versioning In this last phase, the collected data is stored as well as versioned depending on the corresponding release cycle.

So far, if a questionnaire should be created and evaluated in digital form, a lot of technical knowledge is needed. Programming skills are required in the design and modeling phase, the deployment phase as well as in the archiving and versioning phase. This comes with a lot of effort, because of which paper based questionnaires are often chosen over digital questionnaires [7].

2.1.1 Components

To support all of these phases, the *QuestionSys* project consists of different components:

Questionnaire Configurator A program for creating questions and possible answers, defining rules after which the questionnaire is executed and set other possible options, such as possible languages for the questionnaire. Finished questionnaires are identified with a process model that represents the execution rules of the questionnaire.

Server The server is the key component of the *QuestionSys* application. Finished questionnaires are deployed to the server where they are stored and versioned. The results of executed questionnaires are stored on the server as well.

Mobile Client The mobile client is used to provide an easy way to execute questionnaires, collect data and upload it to the server. In the *QuestionSys* project, the mobile client is realized as an application for smart mobile devices such as smartphones and tablets.

while the user creates and edits a questionnaire using the *Questionnaire Configurator*, a process driven flow model is generated that represents the sequence of questions in the questionnaire. In this process model, each question of the questionnaire is represented as a data element, and the pages of the questionnaire correspond with process activities. After the user finished the questionnaire in the *Questionnaire Configurator*, it is deployed to a process management system, the *QuestionSys* server. The server allows it to create instances of the questionnaire on the *Mobile Client*, which can be executed there to collect data. After the questionnaire is finished, the collected questionnaire data is stored locally until there exists an internet connection and the collected data is exported to the *QuestionSys* server [8].

The component described in this thesis should be able to import this collected server data into statistical tools to conduct further analysis on the data. It is not a part of any of the listed components, but rather an independent *R*-library as well as an plugin for *Excel*, but has to communicate with the server via the *JSONAPI* protocol to request and update data.

3 Analysis

3.1 Requirements

As the goal of this thesis is to create an application that enables domain experts without programming knowledge to import data into a tool for statistical analysis, there a certain requirements that this application needs to support to fulfill its purpose. These include both the functional requirements described in Section 3.1.1 as well as the non functional requirements described in 3.1.2.

3.1.1 Functional Requirements

FR1 (Server Request)
: The application should be able to create a *HTTP* connection to the *QuestionSys* server and send *GET* request to receive resources from it. The *URL* address required for the connection as well as the location of requested data should be generated from a single *URL* address entered by the user, without the need to make multiple requests by hand.

FR2 (Server Response)
: The response of requesting data from the server as specified in *FR1* should be evaluated by the client. The included data should be decoded from the *JSON-API* protocol response and parsed into an appropriate representation of the data.

FR3 (Content of Requests)
: The client should request, parse and save data concerning the question structure,

the set of responses, and possibly a collection of additional information about the answers of the questionnaire.

FR4 (Pagination)

While requesting data from the *QuestionSys* server, the client should use the pagination that is specified in the *JSON-API* standard. This should be hidden from the user, meaning that the client will split and execute a request into multiple requests for the different pages by itself, without the user noticing this process. Afterwards, the client should add all requested data pages back to a single data collection.

FR5 (Avoid Spam Protection)

If a request is split into multiple requests, for example through pagination, or multiple different requests should be performed in a short amount of time, the client has to ensure that the spam protection of the *QuestionSys* server is avoided. To achieve this, the client should keep track of how many requests he sent in the last period of time, and how many requests are still available. If the request limit per time is to be exceeded, the client should wait until further requests are permitted by the server, before any further requests are performed.

FR6 (Loading Bar)

When performing a request with a duration of over a second, the application should display a loading bar, indicating the user that his request is still handled, and displaying the current progress.

FR7 (Preview)

The user should be able to specify, how many data sets he wants to request at maximum from the server with his request. This makes it possible to use the request feature as a preview feature, as well as well as a more flexible way to update data.

FR8 (Data Caching)

Downloaded data sets should be saved locally by the application. This includes the question data, the result data as well as additional information such as the date on which the data was last modified. When requesting data from the server, the application should check whether the data is already stored locally, and if this is the case, limit the requested data to result sets that changed after the last update to the old data. This process shall be hidden from the user.

FR9 (Interface)

To enable the user to use the component, the application should be equipped with an suitable user interface. In *R*, this interface should be realized as an object oriented console interface. In *Excel*, this interface should be realized as a plugin integrated into the graphical user interface.

3.1.2 Non Functional Requirements

NFR1 (Installation)

It should be possible to install the component as a user that is familiar with *R* and *Excel*. To allow this, the *R* program should be installable via the *install_github* function provided by the `devtools` library [9]. The *Excel* plugin should be installable through the use of an installer, that opens a configuration wizard for the installation.

NFR2 (Maintainability)

To allow for the later updating of the software because of changes in the server, protocol or data format, as well as fixing any found issues, the application should follow the structure defined in this document.

NFR3 (Stability)

The application should not crash through any occurring error, such as the sudden end of a connection to the *QuestionsSys* server or invalid user input. In case of an error, the application should handle it as far as possible, such as saving all obtained data, and display a meaningful and expressive error message to the user.

NFR4 (Usability)

A user familiar with either *R* or *Excel* should be able to use the application without any further help other than the *in-program* help tools and descriptions, as well as a written manual describing the functionality of the product.

NFR5 (Expandability)

The components structure should be designed to allow a developer familiar with the programming languages *R* and *C#* to expand the application by using this document and the code comments and documentation provided in the component.

NFR6 (Security)

The application should not allow any way to execute custom code that may be inserted through answers or questions. Any possible code should be escaped in a way that it is not executed.

NFR7 (Testability)

To ensure that the application is working properly, every function should be covered by automated tests that are verifying if the component is producing correct results in regular operation as well as ensuring correct error handling in case of network failure, wrong data formats or similar situations.

3.2 Implementation Specific Requirements

Because each of the implementations is written in another programming language as well as dependent on the statistical tool it was written for, it is clear that some implementations of the component may have other requirements, such as the installation process or the user interface.

3.2.1 R

R1 (Installation via GitHub)

To make it as simple as possible to install this package, it should be possible to install it with as little experience in *R* as possible. To ensure this, the source code to this component should be placed in a *GitHub* repository, and installable by using the existing *install_github()* function.

R2 (Object Orientation)

To simplify the usage of the component, the component should be developed using the *R6* classes framework. *R6* is an implementation of the concept of object oriented programming in *R* that is faster and simpler to use than the default implementation of classes in *R* [10]. This should provide an intuitive and easy approach to handle connection parameters as well as requested data and its transformation.

R3 (Data Frame Conversion)

To ensure a generic and flexible approach, requested data should be stored in a *data frame*.

3.2.2 Excel

E1 (Installer Program)

To simplify the installation process of the *Excel* plugin, the component should provide an installation program. This installation program starts a wizard, requiring as little user inputs as possible.

E2 (Local User Installation)

Because some users may not have the required rights to install software globally, the installation wizard should offer the option to install the plugin locally, only for the current user.

E3 (Graphic User Interface)

The user interface for the *Excel* plugin should be realized as a graphical user interface integrated in the default *Excel* user interface. All methods required to request or update data should be accessible through a new ribbon. This ribbon should include buttons for all required functions, and function parameters for these functions should be requested from the user in a popup dialog.

E4 (Data Conversion)

Requested data should be directly inserted into a worksheet. The user can choose if he wants to include it in the currently selected worksheet or create a new worksheet for the data. If the new data is to override existing values, the user is notified and asked for a confirmation if he really wants to continue.

4 Structure

Any application that aims at providing domain experts with a tool to import questionnaire result data from the *QuestionSys* server into any statistical tool to further analyze the data, especially *R* and *Microsoft Excel*, needs to support two different operations. First, the application should be able to open a connection to the *QuestionSys* server and request the data that the user specified. Furthermore, the application should also be able to handle the obtained data in a useful manner, enabling the user to further analyze the results in a meaningful and efficient way.

4.1 General Structure

In the generic concept for the goals presented in this thesis, each of these two operations is divided into two separate phases, as shown in Figure 4.1. Requesting data is divided into the specification of all parameters for the request, such as the *URL* of the server, as well as the location of the data on the server. Furthermore, the amount of data sets to be requested needs to be specified. The second phase is to actually request the data based on the specified parameters. This includes creating a connection to the *QuestionSys* server, sending a request for the data and questions, as well as receiving and processing the response. On the other hand, handling data is divided into data conversion and data storage. The server uses the *JsonAPI* specification to communicate with clients, which is a specification designed for client-server communication using the *JavaScript Object Notation (JSON)* [11]. The data included in these messages needs to be decoded, converted into a fitting data type and possibly put back together if multiple requests were needed to import the data, and combined with existing data. Finally, in the last phase, the data should be saved locally, to enable later usage of this data, even without an internet connection.

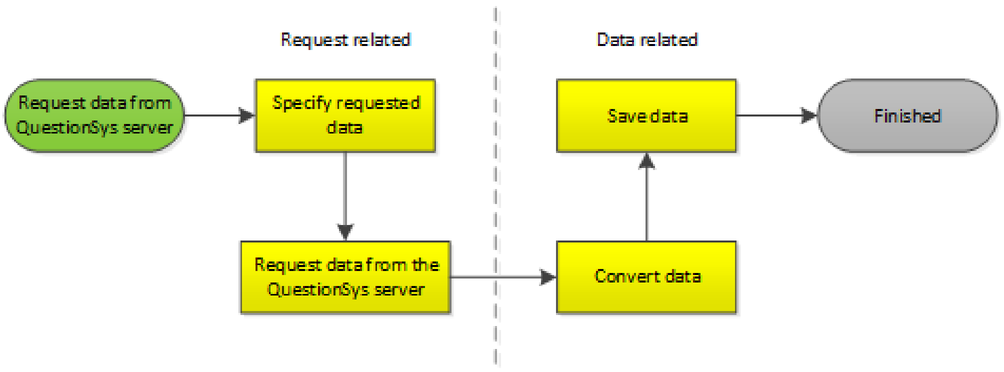


Figure 4.1: The general phases of importing data from the *QuestionSys* server

4.1.1 R

The implementation in *R* is written in an object oriented manner. This is achieved through the usage of the `R6` library, providing a simple, intuitive and fast gateway to object oriented programming in *R* [10]. The only public method that the *QuestionSys* page provides is the `questionSys_create()` method. If called, this method creates a new instance of the `QuestionSys` class, which provides further methods to specify and execute the request.
The first phase is realized as different `setter`-methods provided by the `QuestionSys` class, that allow the user define the request to his needs. After everything is defined, the method `requestData()` needs to be called. This method signalizes the end of the user input, and calls the three private methods `specifyConnection()`, `sendRequests()` and `convertData()`. Each of these methods handle one of the phases in the generic model. `specifyConnection()` specifies the request and connection parameter, as well as handling user inputs. The method `sendRequests()` starts the *HTTP* connection with the specified parameters, and executes the data requests. Finally, `convertData()` converts the obtained result data into a `data.frame` and sets the public variable `questionSysData` of the current `QuestionSys` object to this `data.frame`. The last

phase, storing data, is only executed if the user calls the public method `saveData()` and saves obtained result data as well as additional information into a file specified by the user.

4.1.2 Excel

The *Excel* implementation is integrated into the graphical user interface of *Excel*. When installed, it provides its own *Ribbon*, as seen in Figure 4.2. This menu contains a section for importing data as well as a section for updating data. When the user clicks a button in this menu, a dialog window is opened that provides a form for further required user input, such as the *URL* address of the *QuestionSys* server. After all parameters are

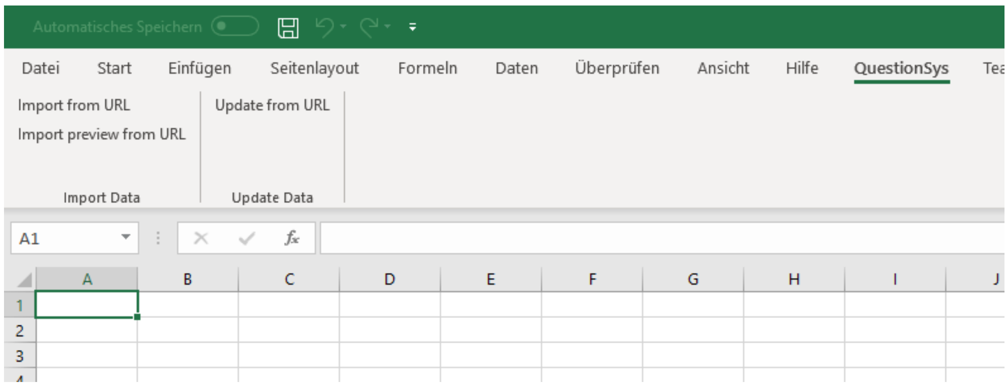


Figure 4.2: A Screenshot of the *QuestionSys* Ribbon in the Excel plugin

set correctly, the application creates a new instance of the `DataManager` class. This class serves as a controller for the different steps performed when requesting data from the *QuestionSys* server. The second phase is handled by the `NetworkHandler` class, that provides *static* methods to connect to a server and request questions and answer data from it. The `DataManager` instance invokes these methods, and passes these results on to the next phase. The third phase of converting data is handles by the `QuestionSysData` class, as well as by the subordinate classes `ResultData`, `Answer` and `Question`. These classes provide a constructor that accepts the results from the second phase as input and convert it into a single `QuestionSys` object. Furthermore, it is possible to include another already existing instance of a `QuestionSys` object when creating a new `QuestionSys` object, which will be updated with the new data contained in the request results. Saving data is realized by filling a range of the current worksheet or a newly created worksheet with the result data. This is handled by an instance of the `DataSaveManager` class, that takes a `QuestionSysData` object and a location inside the *Excel* workbook, and stores the information of this `QuestionSysData` object at the given location. Each answer to a question is stored in the same column, and answers of the same set, sharing the same `answerId`, are stored in a row. Because it is not possible in Excel to store multiple values inside a single cell, it may be required to create multiple rows with the same `answerId` to display all result data. Additional information, such as the timestamp of the last update and the questionnaire id are stored as comments to a cell at the beginning of the cell range.

4.2 Request Specification

Specifying requests consists out of different steps, as shown in Figure 4.3. First of all, the user needs to specify the *URL* address of the questionnaire on the *QuestionSys* server. This URL, as shown in Listing 4.1, is generated by the server and available for a user that created or edited this questionnaire. It is the base address and the location of both the questions and the answers to this questionnaire can be generated out of this address. It consist out of the *IP* address of the *questionSys* server, the protocol version that is used by the server, the identifying number of the selected questionnaire as well as an authentication token that ensures that the user has access to his own questionnaires.

```
http://[questionSys server IP]/[version]/questionnaires/
    [questionnaireID]?authentication=[AuthenticationID]
```

Listing 4.1: Format of the server generated *URL* address

Next, the user selects if he wants to import all data sets in the corresponding questionnaire result data set, or if he wants to limit the maximum amount of data sets to be downloaded from the server, for example as a method to preview the data. After this, the application needs to determine if there already exists local data that needs to be completed with new data entries, or updated with changed data sets, and determine when the local data was last changed. Finally, the request has to be generated out of the *URL* given by the user, and modified depending on the stated conditions to match the users settings.

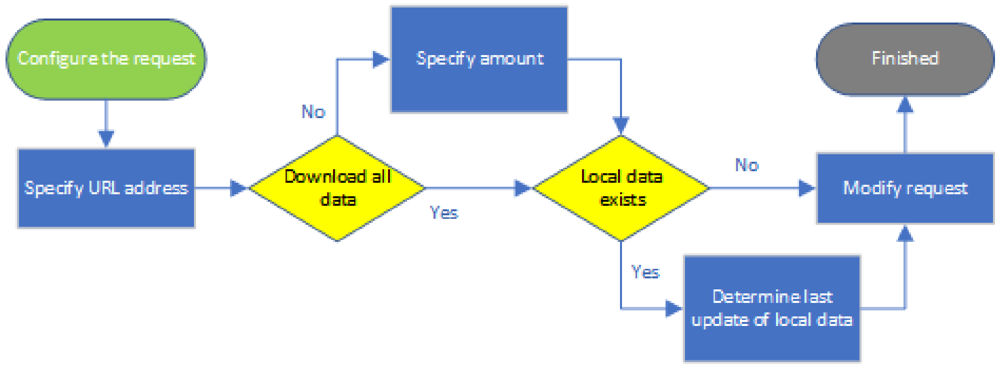


Figure 4.3: The steps of specifying request parameters

4.2.1 R

The specification of user input in the *R* application is realized through the public methods `setUrl()`, `setAmount()` and `setInputFile()` (see Listing A.1). `setUrl()` stores the given URL in the private variable `serverUrl` of the current `QuestionSys` object. `setAmount()` sets the private variable `amount` to the given value. If the parameter `all` of this method is set to true, the function resets the amount back to all possible data sets. `setInputFile()` sets the private variable `inputFile` of the current object to the given file. If the given file is not a path to a file containing the data but `null`, the application opens a graphical file chooser where the user can select a file to input data from. All three methods return the current `questionSys` object, so that it is possible to chain calls to these methods behind each other, to allow setting all parameters in one line and a faster workflow.

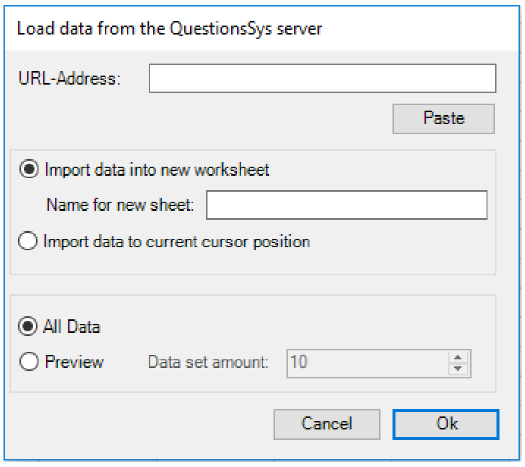
After the user called the `requestData()` function, the `specifyConnection()` function separates the important parts of the `URL`. The helping methods for this are stored in the `specifyRequest` file, providing a better testability for these supporting methods. If the user decided to load existing data from a file, the method `.loadData()` (see A.2) uses the built in `load` function to load existing data, a list of `questionIds`, the time when the data was last updated, as well as the id of the questionnaire belonging to this data. If the old `questionnaireId` does not match the `questionnaireId` extracted from the *URL* address, loading old data is canceled. Also, if the selected file is not a valid *QuestionSys* save, the user gets an error message, and nothing is returned.

4.2.2 Excel

User input in the *Excel* implementation is handled through two different dialog windows, one for specifying a request of new data, and one for updating existing data. The dialog for requesting data from a *URL*, as seen in figure 4.4, is shown after the user presses either the `Import from URL` button or the `Import preview from URL` button. It contains a *textfield* for the *URL* address, as well as a button that copies the last entry from the clipboard to this *textfield*. Furthermore, a radio button indicates whether the user wants to import the data at the currently selected cell, or if the data should be imported into a newly created worksheet. If creating a new worksheet is selected, a *textfield* is enabled where the user can enter a name for the new worksheet. If no name is entered, a name containing the current unix timestamp is generated. Additionally, the user can decide through another radio button, if he wants to import all relevant data sets, or limit the result sets to a certain amount, for example as a preview. If this menu was accessed through the `Import preview from URL` button, the preview option is selected per default. If this preview radio button is selected, a field is enabled that contains the number of data sets that should be requested. The user can either input a number between zero and the maximum integer value directly, or use the small arrow buttons on the side of this input field to increase or decrease this count by one. Finally, the `Cancel` button allows to cancel this process, and the `Ok` button checks if all inputs are valid and starts the request as specified.
The dialog for updating data, as seen in Figure 4.5, is shown after the user clicks the `Update from URL` button in the `Update Data` section. This dialog also contains a

textfield to input the *URL* address of the *QuestionSys* server, as well as a button to paste the last entry from the clipboard into this *textfield*. In addition, a radio button lets the user select if he wants to update data at the currently selected position, or in another worksheet. If this radio button is enabled, a dropdown list of all existing worksheets is enabled, allowing the user to select one. If the current selection is chosen, the application tries to locate questionnaire data with the top left cell of the selection as the top left cell of the questionnaire data. If the user wants to update data in another worksheet, the top left cell of this worksheet is selected as the top left cell of the questionnaire data.

To ensure that all data specified in the `Import data` dialog by the user is valid, the


Load data from the QuestionsSys server

URL-Address:

Paste

(●) Import data into new worksheet

Name for new sheet:

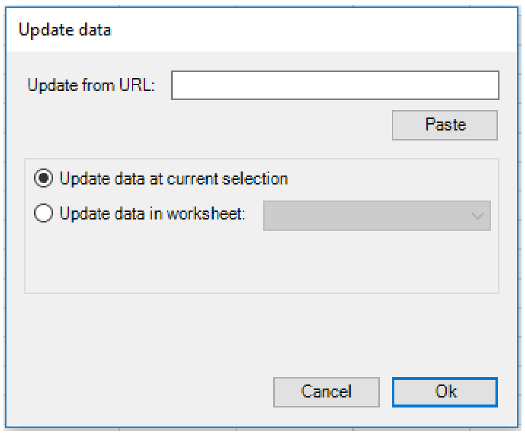
( ) Import data to current cursor position

(●) All Data

( ) Preview   Data set amount: 10

Cancel   Ok

Figure 4.4: A Screenshot of the import data dialog in the Excel plugin


Update data

Update from URL:

Paste

(●) Update data at current selection

( ) Update data in worksheet:

Cancel   Ok

Figure 4.5: A Screenshot of the update data dialog in the Excel plugin

application checks all inputs when the `Ok` button is pressed (see Figure 4.6). If the user presses the `Cancel` button, the dialog is closed and no data is imported. If the `Ok` button is pressed, the application checks if the user specified a *URL* address. If no *URL* address was specified, a warning is shown that informs the user that a *URL* address is required to request data, and the dialog stays open. If a *URL* address was specified, the application checks if the user wants to import the data into a new worksheet or if he wants to include it at the current selection. If the current selection was chosen, a warning is displayed that informs the user that importing data into an existing worksheet may override existing data. If he accepts this warning through the *Ok* button of this message box, the data is imported and the dialog closed. If he rejects it through the *Cancel* button of the message box, the application returns to the import data dialog. If the user decided to import the data into a new worksheet, the application tries to create a new worksheet with the given name, or an automatically generated name if the user did not specify a name by himself. If this fails because there already exists a worksheet with the selected name, a warning is shown that informs the user that the creation of a new worksheet failed, and the dialog is closed. If a worksheet could be successfully created, the data is imported into it and the dialog is closed. Validation in the `Update data` dialog is

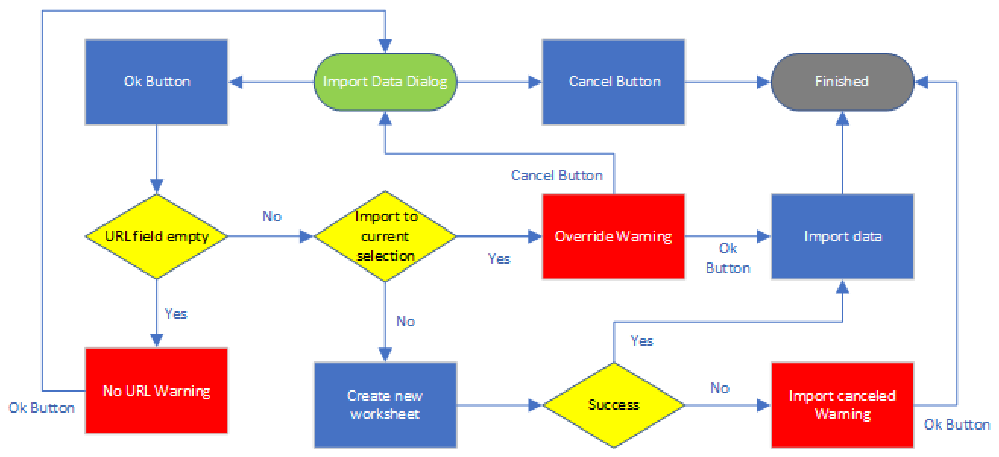


Figure 4.6: The steps performed by the Excel application to validate user input

fairly similar to the validation of the *Import data* dialog, as shown in Figure 4.7. With the `Cancel` button, the user can close die dialog without updating data. After the `Ok` button was pressed, the application checks if a *URL* was specified, if there is none, a warning is displayed and the application returns to the dialog. If a *URL* was specified, the application checks if the data should be inserted at the current location. If this is the case, a warning message is displayed informing the user that the process might override existing data. If he accepts this warning with the `Ok` button of the displayed message box, the data is updated and the dialog closed. If the user presses the `Cancel` button instead, the application returns to the `Update data` dialog. If the user wants to update data in an existing worksheet instead, the application checks if an existing worksheet was selected. If none was selected, a warning is displayed that the user is required to select a worksheet, and after this message box is closed, the application returns to the dialog. If a worksheet was selected, the data is updated in this worksheet and the dialog is closed.

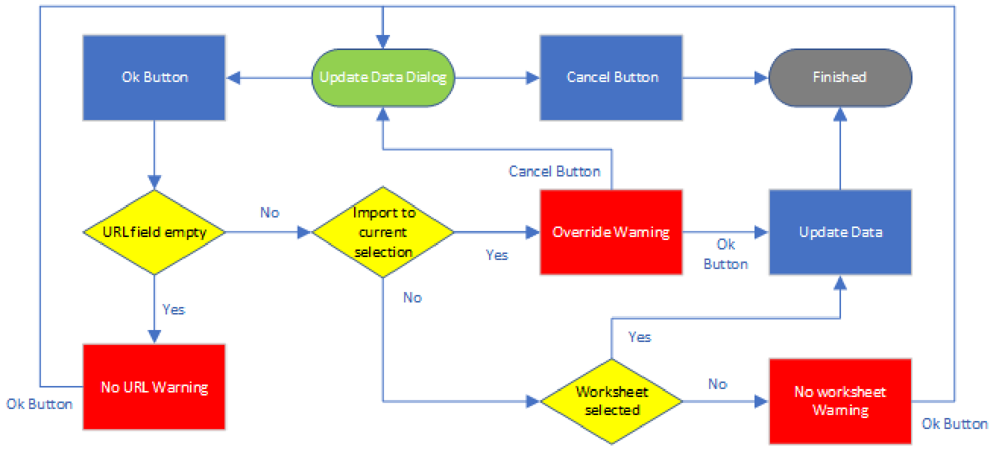


Figure 4.7: The steps performed by the Excel application to validate user input

4.3 Data Request

The process of sending a data request, as shown in Figure 4.8, takes multiple steps. First, a *HTTP* connection to the *QuestionSys* server is established and a *GET* request for the questions of the questionnaire data is sent. This request follows the schema as seen in Listing 4.2. The parameter `[questionnaireId]` needs to be replaced with the id of the questionnaire of which the result data should to be imported. Additionally, the `[authenticationID]` token is included in the `Authorization` header field in the *HTTP* request, ensuring that the user requesting data is granted access to the data he wants to import.

The response from the server is read from the network connection and stored, so that it can be decoded in the next phase. Next, a second *HTTP* request is sent to get the questionnaires answer data, using the request that was generated in the request specification phase. If the server uses pagination, and not all requested data is included

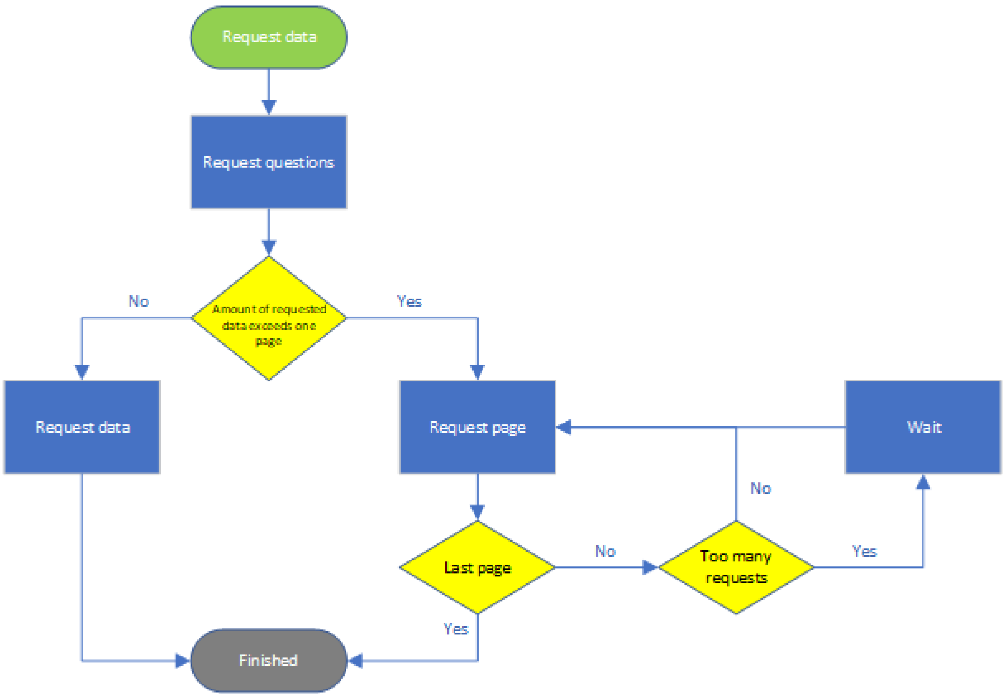


Figure 4.8: The steps of sending a data request to the *QuestionSys* server

```
GET questionnaires/[questionnaireID]/questions
  HTTP/1.1
Accept: application/vnd.api+json
Authorization: [AuthenticationID]
```

Listing 4.2: HTTP Request for fetching questions

in the page returned to the request, a new request is posted until all demanded data is obtained. A request to fetch questionnaire result data from the *QuestionSys* server follows the scheme shown in Listing 4.3. Again, the `authenticationID` token is included in the `Authorization` header field. The parameter `questionnaireID` is handled in the same way as in requesting the questions. Furthermore, `pageNumber` indicates the number of the page that is requested in the current step. If the user decided to limit the amount of results that he wants to request, the parameter `pageSize` is used to set the result page size to the amount of requested data. Else, this line is not included in the request and the default page size defined by the server is used. The `filter` query parameter is optional too, and only included if there already is questionnaire result

```
GET questionnaires/[questionnaireID]/results?
    filter= [lastModified] &
   page[number]= [pageNumber] &
   page[size]= [pageSize]
   HTTP/1.1
Accept: application/vnd.api+json
Authorization: [AuthenticationID]
```

Listing 4.3: HTTP Request for fetching a page of questionnaire result data

data stored locally. The parameter `lastModified` is set to the timestamp when the local data was last updated, enabling the *QuestionSys* server to limit the response to new data sets or result sets, that changed since the last update.

Each request response is read from the network and stored in a list, so that the data can be decoded in the next phase as well as added back together. After each request, the client needs to check if the maximum amount of allowed requests per time unit, that the spam protection of the server allows, has been reached. This limit is transmitted to the client via the *HTTP* response header field `X-RateLimit-Limit`. This field contains the maximum allowed requests. The field `X-RateLimit-Remaining` contains the amount of allowed requests left for the current user, and `X-RateLimit-Reset` contains a unix timestamp of the moment, when the request limit will be set back to the default request limit. If this limit has been reached, the application needs to wait until this reset moment has passed and the request is valid again, then continue with requesting data. Else, the next request is executed right away.

4.3.1 R

The *R* implementation utilizes the library `crul` to handle the *HTTP* connection to the *QuestionSys* server. Although there already exists `rjsonapi`, a library dedicated to handling *JsonAPI* connections in *R*, `crul` was selected because it provides a better flexibility when sending requests to the server that may not be `JsonAPI` compliant. Furthermore, it provides an easier access to request and response header fields, such as `Authorization` or `X-RateLimit-Limit`.
Requesting data is handled in the method `sendRequests()` (see Listing A.5). In

the first line, a new connection to the *QuestionSys* server is established, using the method `.startConnection()` (see Listing A.3). This methods establishes a new *HTTP* connection to the given *URL* address. If the parameter `authenticationToken` is used, this token is included as the value of the `Authorization` header field. Next, in line five, the questions of the questionnaire are requested using the `.getResource()` method (see Listing A.4). This method sends a *HTTP-GET* request to the resource specified in the `path` parameter, using the connection in the `connection` parameter. If the parameter `absolutPath` is `true`, the application uses the path as given, after removing the *URL* address in front of it. If `absolutePath` is false, the requested path is combined out of the *JsonAPI* `version`, the `questionnaireID` and the amount of data sets that should be requested. The *HTTP* response obtained from this request is stored, and a list is returned, containing the response text and the content of the header fields `X-RateLimit-Remaining` and `X-RateLimit-Reset`. The content is parsed to an *UTF-8* string, and then decoded to a `R6` object, using the method `fromJSON()` of the `rjson` library.
After the question data has been requested, the application requests the first page of answer result data in line nine, and the result list is split into its separate values. In line 16, the application checks if the server is using pagination, and if the first page contains all required data, by comparing the link to the last page with the link of the current page. Additionally, if the user specified a limit for the total data amount, it is not necessary to request more pages, because the page size was overwritten to request all data sets at once. Otherwise, the next page needs to be requested, too. Furthermore, in line 20, it is checked whether a request to the server would exceed the servers request limit. If it does not, the next page is requested in line 21. The location of this page is taken from the `links` object of the last request. In line 26, the result of this request is added to the `answerResult` list. The number of allowed requests left is stored in the `requestsLeft` attribute, as well as the timestamp when the request limit resets is stored in `requestLimitExpires`. If no valid request would have been possible, in lines 28-30, the application waits until the moment specified in `requestLimitExpires` passed. After this, the request counter is reset, and the requests are continued. After each request, it is checked, if the last requested page is the last page of the data set, by

comparing their links in line 35. If it is, the loop is ended, the current time is stored in line 40 in the private attribute `lastUpdate`, and a list containing the question results as well as the answer results is returned in line 41.

4.3.2 Excel

Creating a *HTTP* connection to the *QuestionSys* server as well as requesting data from it is handled by the `NetworkHandler` class. It provides a static method `SetUp()` (see Listing A.9). This method takes the *URL* address of the server as well as an `authorizationToken` as input. First, it creates a new `HttpClient` object in line three, sets its base address to the given address in line four, changes the accepted response media type to *JsonApi* in line six, and in line eight, the `Authorization` request header is added and set to the given `authorizationToken`.
Requesting the question data is handled by the method `RequestQuestions()` (see Listing A.10), that takes the *JsonApi* `version` and the `questionnaireID` as its input. It sends a *GET* request to the server, and returnes the content of the response as a string.
Requesting the answer result data is a bit more complicated, and handled by the method `RequestAnswers()` in the `ConnectionHandler` class (see Listing A.11). As input, it requires the *JsonApi* `version` used by the server, the `questionnaireID` of the data to be requested, as well as optionally, the `amount` of data sets that should be requested, and the timestamp of the last update, if there already exists local data. First, the correct path to the required results needs to be created, depending if the user specified an amount and local data exists. Line seven requests the data, if the user specified such amount, line 14 requests this data if no data was specified. In line 17, the content of the response content is extracted and stored in the variable `Content`. Furthermore, the application checks in line 21 and line 25, if the response to this request contains the response header values `X-RateLimit-Remaining` and `X-RateLimit-Reset`. If they do, their corresponding values are stored in the variables `requestLeft` and `requestReset`. To check whether it is required to request further pages, the `Content` string needs to be partially decoded using the *Json.NET* library [12]. Therefore, a

`JToken` object `lastPageToken` is generated, representing the `last` attribute in the `links` object of the response. If this object does not exist, because the server does not use pagination, the already obtained response is all there is to request, and the current `Result` list is returned in line 35. If a request limit was specified, or the first page is already the last page of data, as checked in line 40, the first result already contains all relevant data sets. If this is not the case, then the next page needs to be requested. The next page is requested in line 48, with the requested path taken from the `next` link in the `links` object of the previous response. Next, the `requestsLeft` and `requestReset` values are updated in line 55 and line 59. In line 63, the content of the last response is added to the `Result` list. Afterwards, in line 67, the application checks if this page is the last page that should be requested, and if it is, the loop is terminated. Before every request, the application checks in line 45, if the server uses a spam protection, and if the next request is allowed. If it is not, the application waits in lines 74-78 until the moment in `requestReset` is reached, and then resets the `requestsLeft` count. Finally, the list of results is returned in line 83.
Because these methods are written asynchronously, the return type of these methods is `Task<String>` and `Task<List<String>>`, these `Tasks` need to properly executed by the `DataManager` class, as seen in Listing 4.4:

```
NetworkHandler.SetUp(new System.Uri(BaseUrl), Token);
QuestionResponse = NetworkHandler.RequestQuestions(Version,
    QuestionnaireId).GetAwaiter().GetResult();
AnswerResponseList = NetworkHandler.RequestAnswers(Version, QuestionnaireId,
    Amount, LastUpdate).GetAwaiter().GetResult();
```

Listing 4.4: Requesting data in the Excel application

4.4 Data Conversion

When converting data, as shown in Figure 4.9, the application first needs to convert the response the application received in the second phase.

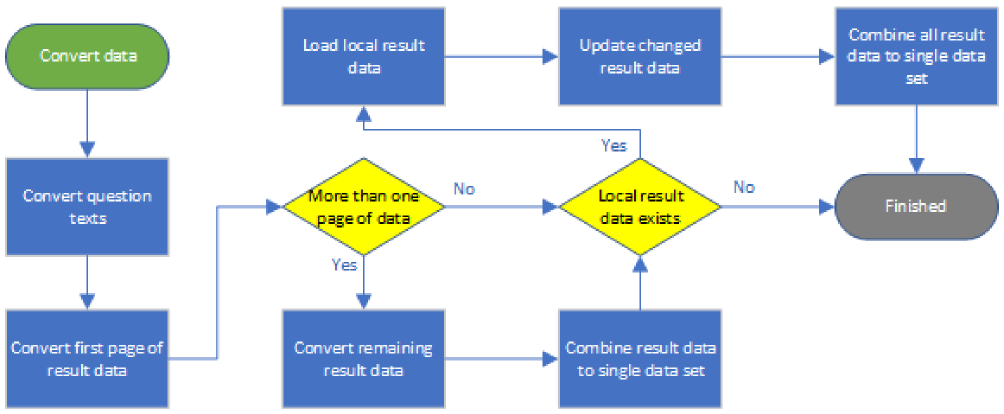

Convert data
Convert question texts
Convert first page of result data
More than one page of data
Yes
No
Convert remaining result data
Combine result data to single data set
Local result data exists
Yes
No
Load local result data
Update changed result data
Combine all result data to single data set
Finished

Figure 4.9: The steps of converting the response of the *QuestionSys* server

```
{
  "data": [
    {
      "type": "questions",
      "id": String
      "attributes": {
        "questionText": String
      }
  }, ...
  ],
  "links": {
    "self":    (String) Link to these questions
    "related": (String) Link to the questionnaire
  }
}
```

Listing 4.5: Response format for questions returned by the *QuestionSys* server

The body of a response may look similar as shown in Listing 4.5. The `data` array contains a list of objects that contains information about the individual questions. The field `id` is a string used to uniquely identify all questions in the questionnaire. The important and for the client relevant information is stored in the `attributes` object. Here, the `questionText` attribute contains the text of the question. This text needs to be extracted and stored together with the respective `id`.

After that, the client needs to decode the set of answers. As the server may use pagination, the application first needs to decode the first page of the result data, and if there are more than one page of result data, the other pages of results need to be

```
{
  "data": [
  {
    "type": "answers",
    "id": String
    "attributes": {
      "answers": [
      {
        "questionID": String
        "answerText": [String,...]
      }, ...
      ]
    }
  }, ...
  ],
  "links": {
    "related": (String) Link to the questionnaire
    "self":    (String) Link to this page
    "first":   (String) Link to first page
    "prev":    (String) Link to previous page
    "next":    (String) Link to next page
    "last":    (String) Link to last page
  }
}
```

Listing 4.6: Response format for answers returned by the *QuestionSys* server

decoded as well. Each page of results may look similar to the schema shown in Listing 4.6. The *JSON*-array `data` includes a list of *JSON* objects that each represent one set of answers. The field `id` is a server wide unique identifier of this answer set. The attributes object contains a list of answers, each containing a `questionID`, identifying the answer with the corresponding question. The attribute `answerText` contains the given answer of the current user for this question. It is a list, so it is possible to support multiple answers per question. The application can traverse all pages, beginning with the first page, using the `next` link in the `links` object.
After all data sets have been decoded, the application needs to combine the different data sets to a single object, containing all the data. Next, the application needs to check whether there already exists local data. If this is the case, the data needs to be loaded, and it needs to be checked, what part of the new data is a newer version of already existing local data, by matching the `id` fields of the answer result data. These data sets are then replaced with the new data sets. After this, all new data needs to be added to the already existing and updated data into a single set.

4.4.1 R

The process of converting data in the *R* implementation of this concept is managed in the `convertData()` method (see Listing A.6). It takes the `questionResultData` and `answerResultData` received in the previous phase as input, as well as an optional parameter `replaceQuestionIds` which is set to `true` per default, and indicates if in the final `data.frame` the column titles should be named after the `questionIds` or instead be replaced with the text of these questions. First, in line ten, the application determines how many sets of result data there are in total, and creates a list containing the `id` of all answers in line twelve. The variable `questionIds` is created as an `environment` to match all `questionIds` to their corresponding `questionText`. Here, an `environment` is used as a `HashMap`, because *R* lacks a default implementation of `HashMaps`. For each question, the `questionId` is stored in the list `questionIdList` in line 24, and every `questionText` is stored in the list `questionList` in line 25. Next, a matrix is created in line 31, where each row is a set of result data, and each column is all answers to one question. In line 33, the method `dimnames()` sets the names of each column and row of the given matrix to the values contained in the assigned lists, which makes it possible to address a cell directly using the respective `questionId` and `answerId`. After this, the `answerResultData` is traversed and the list of answers for each question is inserted into the matrix in line 42. If `replaceQuestionIds` has been set to true, in line 51, the name of each column is replaced with the actual text of this question, using the `questionList` created earlier. Finally, the public attribute `questionSysData` is set to a new `data.frame` generated out of this matrix, and the attribute `questions` is set to the list of questions.
After that, the method `insertOldData()` is called (see Listing A.7), to combine the existing data with the new data obtained in the last phase. First, the application checks in line twelve for all data sets in the old data set, if they are already part of the new data set. If a result set is not, a new row with empty values is added to new `data.frame` in line 15. Because it is possible that the questionnaire was updated between this request and the last time the data was requested, there may be more questions in the current data than in the old data. Hence, the application checks at what position all questions in the

`oldData` are in the new `data.frame`, and place them at the right position. Therefore, the position texttttcurrentQuestionId, where the data is inserted into the `data.frame`, is determined by finding the index of the element in the `questionIdList`, that equals the current old question id, as seen in line 18. Then, the element of the new question list with this index is selected as the new index for this response. Finally, the public `questionSysData` attribute is set to this modified `data.frame`.

4.4.2 Excel

Converting data in the *Excel* application is handled in an object oriented manner. More precisely, the conversion from a *JsonApi* compliant string into a useful data structure is handled by the class `QuestionSysData`. This class provides a constructor, as seen in Listing A.12, that takes the result from the question request as well as a list of results from requesting answers as well as the `questionnaireId` as input. On top of that, it is possible to include another `QuestionSysData` object that represents local existing data. To represent all data related to a questionnaire, the class `QuestionSysData` contains an attribute `questionnaireId`, a list of `Question` objects, a list of `ResultSet` objects and a `Dictionary` used to map `questionIds` to the number of this question. When the constructor is called, first a list of `JToken` is generated from the question response, each representing one element of the `data` object in this response. In line 17, from each of these tokens, a `Question` object is generated using the constructor as seen in Listing 4.7, and added to the `Questions` list. This constructor takes a `JToken`, and sets the variable `Id` according to the `id` value in the token, as well as the field `QuestionText` to the `questionText` value of the `attributes` object in the token.

```
public string Id;
public string QuestionText;

public Question(JToken jsonToken)
{
  Id = jsonToken["id"].ToString();
  QuestionText = jsonToken["attributes"]["questionText"].ToString();
}
```

Listing 4.7: A constructor of the Question class to convert data in the Excel application

After this, from each answer response in the `AnswerResponse` list, a list of `JToken` object is generated out of the `data` objects in this list. In line 27, from each token in this list, a new `ResultSet` object is generated, using the constructor seen in Listing 4.8. This constructor takes a `JToken` object, and sets the variable `Id` to the value in the `id` field of the token. A list of `JToken` objects is created out of all objects contained in the `answers` list of the `attributes` object. Out of each of these tokens, a `Answer` object is created and added to the answer list.

```
public string Id;
public List<Answer> Answers;

public ResultSet(JToken jsonToken)
{
  Answers = new List<Answer>();
  Id = jsonToken["id"].ToString();
  List<JToken> AnswerToken = jsonToken["attributes"]["answers"]
      .Children().ToList();
  foreach (JToken answerToken in AnswerToken)
  {
    Answers.Add(new Answer(answerToken));
  }
}
```

Listing 4.8: A constructor of the ResultSet class to convert data in the excel application

The constructor for an `Answer` object, as seen in Listing 4.9, takes a `JToken` object as its input. The value of the variable `QuestionId` is set to the value of the `questionId` field, and in line 10, the list `AnswerText` is filled with each entry in the `answerText` list of the token.

```
1 public string QuestionId;
2 public List<string> AnswerText;
```

```

public Answer(JToken answerToken)
{
  AnswerText = new List<string>();
  QuestionId = answerToken["questionID"].ToString();
  foreach (JToken answerStringToken in answerToken["answerText"]
      .Children().ToList())
  {
    AnswerText.Add(answerStringToken.ToString());
  }
}
```

Listing 4.9: A constructor of the Answer class to convert data in the excel application

After all `ResultSets` are generated in Listing A.12, the `QuestionNumbers` dictionary is created in line 32, and each question id is added. Finally, if the variable `oldData` was specified, this `QuestionSysData` object needs to be merged with the new `QuestionSysData` object. To accomplish this, for each `ResultSet` in the `oldData` object it is checked if the `resultId` of this `ResultSet` is already contained in the list of `ResultSets` in the new object. If it is not, this resultSet is added to the `Answers` list.
In order to update data, it is necessary to generate a `QuestionSysData` object from existing values in an *Excel* file. To allow this, the `QuestionSysData` class contains a second constructor, as seen in Listing A.13. This constructor takes a cell in the current *Excel* worksheet as its input, and tries to read the questionnaire data from this position. First, in line eight, the application extracts the `QuestionnaireId` from the comment at the given `StartPosition`. Next, the application needs to determine the questions of the given questionnaire. Therefore, each cell right to the starting cell is read in line 13, until an empty cell is reached. Out of each non empty cell read, in line 17 a `Question` object is created, with the read value as the `QuestionText` and the value of the cell below as the `QuestionId`. After this, the answers need to be read. Thus, in line 25 every cell below the starting cell is read, until an empty cell is reached. If the cell value is not empty, the entire row is read and out of each cell, that is not empty, an `Answer` object is created in line 34, and added to the `ResultSet` created out of each row. Finally, all

`ResultSets` need to be combined into a single list, while answers to the same questions in different `ResultSet` objects with the same `answerId` need to be combined into a single `ResultSet`. To achieve this, the method `AddResultSet()` (see Listing 4.10) is used. This method checks in line five for each `ResultSet` in the `Answers` list, if their `answerId` is identical to the `Id` value of the given `ResultSet`. If no match is found, the `resultSet` is added to the `Answers` list in line 16. If a match is found, in line nine, each `AnswerText` of the given `resultSet` is added to the `AnswerText` list of the correct `Answer`, using the dictionary to find the right answer index.

```
1  public void AddResultSet(ResultSet resultSet, Dictionary<string, int> dic)
2  {
3    foreach (ResultSet results in Answers)
4    {
5      if (results.Id == resultSet.Id)
6      {
7        //Add resultSet answers to results
8        foreach (Answer answer in resultSet.Answers)
9        {
10         results.Answers[dic[answer.QuestionId] - 1]
11             .AnswerText.AddRange(answer.AnswerText);
12       }
13       return;
14     }
15   }
16   Answers.Add(resultSet);
17 }
```

Listing 4.10: Method to add a result set to a QuestionSysData object in the Excel application

4.5 Save Data

The last step, that the application needs to perform, is to save the data to the file, as shown in Figure 4.10. For this, the user first needs to decide where the data should be saved. This is either a newly created file or an already existing file selected by the user. If the user already selected a file to load data from, the application can offer this file as the default location for the user, enabling a more efficient workflow. After this, the program needs to store additional information of the questionnaire, such as the time when the data was last updated. After this, the application needs to save the questions and answers in a meaningful and efficient format, depending on the platform of the application.

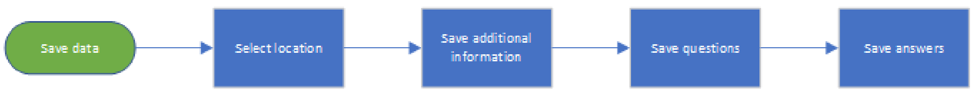


Figure 4.10: The steps of saving questionnaire result data

4.5.1 R

For the *R* implementation of this application, saving data is handled through the default data storage function `save()`. Although there are functions dedicated to storing `data.frames`, these methods are only working with data frames of singular values. But because it is possible to answer multiple distinguished answers to a single question, for example to a multiple choice question, the final result `data.frame` is a `data.frame` where every value is a list of different length on its own. The possibility of converting every list of answers to a single answer before saving the data would arise a lot of problems, like escaping any delimiter character in the actual results to allow for a definite representation of the list as a single string as well as decoding of these strings when reading the data. Instead, the default *R* object storage is used. Therefore, the method `storeData()` (see Listing A.8), that is responsible for handling data, uses this `save()` function to save the `questionSysData`, `questionIdList` to identify the questions if the `questionIds` in the `data.frame` were replaced with the actual question text, the `lastUpdate` timestamp as well as the `questionnaireId`, so that save files can be matched to the respective questionnaires. In this method, each variable is stored in a local variable instead of the normal storage location as an attribute of the `questionSys` object, enabling it to directly address these attributes after loading the file.

4.5.2 Excel

Saving data in the Excel implementation is realized through writing the data into an open worksheet. After this, the user may work with this data as it fits his needs, and actually saving this worksheet is left to the user trough the default *Excel* means. Writing the questionnaire data to a worksheet is handled by the class `DataSaveManager`. This class provides a static method `InsertData()` (see Listing A.14), that takes a cell in an *Excel* worksheet and a `QuestionSysData` object as input and writes the content of this `newData` object to the appropriate file.
The method starts by writing additional information to the file in line four. The method `WriteMetaInformation()` adds a comment containing the `questionnaireId` to the given cell, as well as the timestamp of the last update to the cell right next to it. In line eleven, the `QuestionText` of each `Question` object in the `Questions` list is written to its own cell to the right of the start position. In line twelve, the respective `QuestionId` is written in the cell below each `QuestionText`. After this, the answers need to be written to the worksheet. Because it is not possible to store multiple values in a single cell in Excel, it is necessary to split results with multiple answers to a single question into multiple lines. Each line belonging to the same `ResultSet` is marked with the same `AnswerId`. For each `Answer` object in a `ResultSet`, line 23 stores the number of answers of the `Answer` object with the most elements in its `AnswerText` list into the variable `maxLength`. Line 26 writes each answer text into the column belonging to the `QuestionId` of this `Answer` object, and if there are more than one answer, the other elements of the `AnswerText` list are written below the first answer in the same column. After all `AnswerText` elements are written to the worksheet, line 34 writes the `AnswerId` into the first column of all rows that contain answers of this `ResultSet`, namely the number stored in `maxLength`. After this, some formatting is executed, to improve the readability of the imported data, such as auto adjusting the width of all columns with questionnaire data in line 43, and creating a border around the questions and the answers to distinguish them from each other and surrounding data.

5 Related Works

As digital data collection offers advantages to data collection compared to manual data collection, there are already a number of tools dedicated to creating digital surveys. Most existing tools, such as *SurveyMonkey* [13] or *LimeSurvey* [14] focus on creating web surveys, but lack the advantage of mobile execution of the created questionnaires and are reliant on web browsers to gather data. Exporting data to statistical tools in *SurveyMonkey* enables the user to export the results to either a format optimized for *Microsoft Excel* or a *SAV* file for *SPSS* [15]. Exporting data in *LimeSurvey* allows a wider range of options, such as exporting data to a *CSV* file, an *Excel* file. Further there are dedicated tools to export data to *R*, *SPSS*, *Stata* [16].
CheckMarket [17] and *QuestionPro* [18] are a frameworks that support the creation of mobile data surveys, and support the export of collected data to *SPSS*, to *Excel* and to a *CSV* file.
In contrast to the approach presented in this thesis, these export tools completely rely on a server sided export of the data, resulting in a file that the corresponding software needs to import. This concept provides the advantage of an easier extension of the export process, as only the conversion to the new data format needs to created to add an export functionality. On top of that, the process of exporting data is simplified by the fact that the phase of specifying and creating a connection to the server is not necessary in this format. On the downside, this server sided approach does not provide the full range of possibilities that the client sided approach presented in this thesis provides, such as updating existing data in a worksheet. These data exports can not adapt to existing local data and it is necessary to export all data sets at once.
In this context, the *QuestionSys* project follows a rather unique concept to provide tools

to import data into the statistical tool instead of exporting it through the server to a fixed data format and enabling the user to download this file.

6 Summary

In this thesis, collecting questionnaire data through digital means was compared to traditional data collection techniques. The *QuestionSys* project was introduced as a solution to the costly creation and execution of digital questionnaires. The difficulties for domain experts to import collected questionnaire result data from the central *QuestionSys* server dedicated to the storage of questionnaire result data into a tool dedicated to statistical analysis were discussed. To improve this situation, a generic concept to import this data was introduced. Four distinct phases of this process were identified as specifying the request, sending data requests to the server, converting the response and storing the data. Each of these phases was described in detail, and a list of steps for each phase that an implementation of this generic concept would need to perform was given. Furthermore, the requirements needed to be fulfilled by an implementation of this generic concept were described. Finally, an instance of this concept was implemented in the statistical programming language *R*, as well as an plugin for the *Microsoft Office Excel* program that fulfills these requirements. For each phase of this concept, the implementation was described for both of these implementations, and differences to the generic concept were pointed out.

6.1 Outlook

Because the *QuestionSys* project is still in development and bound to change as the project is developed further, it is very likely that changes need to be made to the application in the future. The concept of the server is likely to undergo a rework as the development progresses. Because of this, the response format for requests as shown in

Section 4.4 is likely to change. Thus, the step of converting the response from the *JSON* response needs to be adapted. Furthermore, it is planned to extend the functionality of the application. First of all, the exported data by the server can be extended to support further information, such as meta data that was collected when the user answered the questionnaire, such as changed or corrected answers, or data collected by the mobile device, such as through a pulse sensor. Furthermore, it is possible that not all answers are answered by a user, and the current format does not include information whether not answered questions were not answered because the answer did not appear to the user because of answers to previous questions, or because of the users choice. Furthermore, the question result does not include any information concerning the type of questions or the possible answers to this question, meaning it is not possible to see from the results alone if one existing answer was not picked at all. The current implementation allows for the addition of new data sets and new data questions, as well as changing the `questionText` while keeping the same `questionId`, but does not support the removal of either result sets or even entire questions. In future versions of this application, it is planned to support these operations, too. All these changes affect the phases of data conversion and storage, which possibly leads to further adaptions in these phases. Furthermore, it is planned to create more implementations of the generic concept presented in this thesis, to cover the most widely used statistical tools. This possibly includes the programming language *Python* and tools such as *SPSS*, *SAS* and *Stata* [19]. Another feature that may be included is an uniform data save format making it possible to load existing local data with any implementation of the generic concept presented in this thesis. On top of that, a further very important change that needs to be implemented in the future is the usage of the *HTTPS* protocol on top of the already used *HTTP* protocol. Data security and encryption is a crucial aspect of data collection in fields such as medicine and psychology, as these fields deal with sensitive information [3]. The *HTTPS* protocol ensures that this information is encrypted and can't be read by someone who is not authorized to do so.

As the *QuestionSys* project is bound to change and evolve, and more and more features are integrated into the application, it is possible that new steps need to be added to a phase, however the general structure of the four phases presented in this thesis will likely stay the same.

Bibliography

[1] Szolnoki, G., Hoffmann, D.: Online, face-to-face and telephone surveys—Comparing different sampling methods in wine consumer research. Wine Economics and Policy **2** (2013) 57 – 66

[2] Boyer, K., Olson, J., Calantone, R., Jackson, E.: Print versus electronic surveys: a comparison of two data collection methodologies. Journal of Operations Management **20** (2002) 357 – 373

[3] Schobel, J., Schickler, M., Pryss, R., Reichert, M.: Process-Driven Data Collection with Smart Mobile Devices. In: 10th International Conference on Web Information Systems and Technologies (Revised Selected Papers). Number 226 in LNBIP. Springer (2015) 347–362

[4] Vehovar, V., Lozar Manfreda, K.: Overview: Online Surveys. In Fielding, N., Lee, R., Blank, G., eds.: The SAGE Handbook of Online Research Methods. Sage reference. SAGE Publications (2016) 143–161

[5] Dillman, D.: Mail and Internet Surveys: The Tailored Design Method – 2007 Update with New Internet, Visual, and Mixed-Mode Guide. Wiley (2011)

[6] Boyer, K., Olson, J., Jackson, E.: Electronic Surveys: Advantages and Disadvantages Over Traditional Print Surveys. Decision Line **32** (2001) 4–7

[7] Schobel, J., Pryss, R., Schickler, M., Ruf-Leuschner, M., Elbert, T., Reichert, M.: End-User Programming of Mobile Services: Empowering Domain Experts to Implement Mobile Data Collection Applications. In: 5th IEEE International Conference on Mobile Services (MS 2016), IEEE Computer Society Press (2016) 1–8

[8] Schobel, J., Schickler, M., Pryss, R., Maier, F., Reichert, M.: 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) 371–382

[9] Wickham, H., Hester, J., Chang, W.: devtools. `https://devtools.r-lib.org/` (2018) [Online; accessed 26-December-2018].

[10] Chang, W.: R6: Encapsulated object-oriented programming for R. `https://r6.r-lib.org/` (2018) [Online; accessed 6-December-2018].

[11] Klabnik, S., Katz, Y., Gebhardt, D., Tyler, K., Resnick, E.: {jspon:api} A SPECIFICATION FOR BUILDING APIS IN JSON. `https://jsonapi.org/` (2018) [Online; accessed 7-December-2018].

[12] Newtonsoft: Json.NET Popular high-performance JSON framework for .NET. (2018) [Online; accessed 25-December-2018].

[13] SurveyMonkey: The World's Most Popular Free Online Survey Tool. `https://www.surveymonkey.com/` (2018) [Online; accessed 28-December-2018].

[14] LimeSurvey: Professional online surveys with LimeSurvey. `https://www.limesurvey.org/` (2018) [Online; accessed 28-December-2018].

[15] SurveyMonkey: Exporting Survey Results. `https://help.surveymonkey.com/articles/en_US/kb/Exports` (2018) [Online; accessed 28-December-2018].

[16] LimeSurvey: Exporting results. `https://manual.limesurvey.org/Exporting_results` (2018) [Online; accessed 28-December-2018].

[17] CheckMarket: Create fully responsive mobile surveys. `https://www.checkmarket.com/mobile-surveys/` (2018) [Online; accessed 28-December-2018].

[18] QuestionPro: Exports. `https://www.questionpro.com/features/reports/` (2018) [Online; accessed 28-December-2018].

[19] Muenchen, R.: The Popularity of Data Science Software. `http://r4stats.com/articles/popularity/` (2017) [Online; accessed 26-December-2018].

A Sources

A.1 R

This appendix contains important source code snippets from the *R* implementation.

```
setURL = function(url,...){
      private$serverUrl <- url
      invisible(self)
    }

    setAmount = function(amount, all = FALSE, ...){
      if(all){
        private$amount <- NULL
      } else {
        private$amount <- amount
      }
      invisible(self)
    }

    setInputFile = function(file = file.choose(), ...){
      private$inputFile <- file
      invisible(self)
    }
```

Listing A.1: Input specification methods of the R application

```
.loadFile <- function(file, ...){
  if(!is.null(file)){
    tryCatch({
      load(file)
      return(list(data, questionIdList, lastUpdate, questionnaireID))
    }, error = function(e){
      print("Not a valid questionSys file")
      return(NULL)
    })
  }
}
```

Listing A.2: Loading existing data in the R application

```
.startConnection<- function(url, authenticationToken = NULL, ...){
  if(is.null(authenticationToken)){
    HttpClient$new(url)
  } else {
  HttpClient$new(url, headers = list(
    Authorization = authenticationToken
  ))}
}
```

Listing A.3: Starting a url connection in the R application

```
.getResource <- function(connection, path, questionnaireID = NULL,
    version = "v1", absolutePath = FALSE, amount = NULL, lastUpdate = NULL){
  if(absolutePath){
   result <- connection$get(.removeURL(path))
  } else{
    result <- connection$get(
      .generateRequest(version, questionnaireID, path, amount, lastUpdate))
  }
  return(list(fromJSON(result$parse("UTF-8")),
          result$response_headers$`X-RateLimit-Remaining`,
          result$response_headers$`X-RateLimit-Reset`))
}
```

Listing A.4: Requesting a resource in the R application

```
sendRequests = function(){
  connection <- .startConnection(private$baseURL,
      version = private$jsonApiVersion, private$authorizationToken)
  # request questions
  questionResult <- .getResource(connection, version = private$jsonApiVersion,
      path = "questions", questionnaireID = private$questionnaireID)[[1]]

  # Request first page
  resultList <- .getResource(connection, version = private$jsonApiVersion,
      path = "results", questionnaireID = private$questionnaireID)
  currentAnswerResult <- resultList[[1]]
  requestsLeft <- resultList[[2]]
  requestLimitExpires <- resultList[[3]]
  answerResults <- list(currentAnswerResult)
  # Check if there are more pages and request these too
  if(!is.null(currentAnswerResult$links$last) && is.null(private$amount)
      && currentAnswerResult$links$last != currentAnswerResult$links$self) {
    while(TRUE){
      # check if request limit expired
      if(is.null(requestsLeft) || requestsLeft>0){
        resultList <- .getResource(connection,
            path = currentAnswerResult$links$"next", absolutePath = TRUE)
        currentAnswerResult <- resultList[[1]]
        requestsLeft <- resultList[[2]]
        requestLimitExpires <- resultList[[3]]
        answerResults <- c(answerResults, list(currentAnswerResult))
      } else {
        while(as.numeric(Sys.time())<=requestLimitExpires){
          #wait a second
          Sys.sleep(1)
        }
        # there should be at least one request now ^^
        requestsLeft <- 1
      }
      if(currentAnswerResult$links$self == currentAnswerResult$links$last){
        break;
      }
    }
  }
  private$lastUpdate <- as.numeric(Sys.time())
  return(list(questionResult, answerResults))
}
```

Listing A.5: Requesting question and result data in the R application

```
convertData = function(questionResultData,
    answerResultData, replaceQuestionIds = TRUE){
  # determine total number of answer sets
  totalResponses <- 0
  self$answerData <- answerResultData
  answerIdList <- list()
  questionIdList <- list()
  questionList <- list()
  for(page in 1:length(answerResultData)){
    totalResponses <- totalResponses + length(answerResultData[[page]]$data)
    for(i in 1:length(answerResultData[[page]]$data)){
      answerIdList <- c(answerIdList, answerResultData[[page]]$data[[i]]$id)
    }
  }
  totalQuestions <- length(questionResultData$data)

  # create table to match ids and names
  questionIds <- new.env()
  for(i in 1:totalQuestions){
    questionID <- questionResultData$data[[i]]$id
    if(!is.null(questionID)){assign(questionID,
        value = questionResultData$data[[i]]$attributes$questionText,
        envir = questionIds)
      questionIdList <- c(questionIdList, questionID)
      questionList <- c(questionList,
          questionResultData$data[[i]]$attributes$questionText)
    }
  }

  #Create new matrix for data
  dataMatrix <- matrix(rep(list(), totalQuestions * totalResponses),
      ncol = totalQuestions, nrow = totalResponses)
  dimnames(dataMatrix) <- list(answerIdList,questionIdList)
  for(page in 1:length(answerResultData)){
    for(dataSet in 1:length(answerResultData[[page]]$data)){
      currentDataSet <- answerResultData[[page]]$data[[dataSet]]
      if(!is.null(currentDataSet)){
        currentId <- currentDataSet$id
        for(i in 1:length(currentDataSet$attributes$answers)){
          currentAnswer <- currentDataSet$attributes$answers[[i]]
          if(!is.null(currentAnswer)){
            dataMatrix[[currentId,currentAnswer$questionID]]
                <- unlist(currentAnswer$answerText)
          }
        }
      }
    }
  }
  # rename the collums to the actual questions
  if(replaceQuestionIds){
    dimnames(dataMatrix) <- list(answerIdList, questionList)
  }
  self$questions <- questionList
  self$questionSysData <- data.frame(dataMatrix)
  private$questionIdList <- questionIdList
  private$insertOldData()
}
```

Listing A.6: Converting data in the R application

```
insertOldData = function(){
  if(!is.null(private$oldData)){
    data <- self$questionSysData
    oldData <- private$oldData
    questionIdList <- private$questionIdList
    oldQuestionIds <- private$oldQuestionIdList
    oldAnswerIds <- row.names(oldData)
    # All data sets from the old data
    for(i in 1:length(oldAnswerIds)){
      currentAnswerId <- oldAnswerIds[[i]]
      #check if this data set is relevant
      if(!currentAnswerId %in% row.names(data)){
        #add this data set
        #add a new row
        data[currentAnswerId,]<-rep(list(),length(questionIdList))
        #add all answers
        for(j in 1:length(oldQuestionIds)){
          currentQuestionId
              <- questionIdList[[questionIdList==oldQuestionIds[j]]]
          data[currentAnswerId, currentQuestionId] <- oldData[i, j]
        }
      }
    }
    self$questionSysData <- data
  }
}
```

Listing A.7: Merging old and new data in the R application

```
saveData = function(file = file.choose(), ...){
  if(!is.null(file)){
    questionnaireID <- private$questionnaireID
    lastUpdate <- private$lastUpdate
    questionIdList <- private$questionIdList
    data <- self$questionSysData
    save(questionnaireID, lastUpdate, questionList, data, file = file)
  }
}
```

Listing A.8: Saving data in the R application

A.2 Excel

This appendix contains important source code snippets from the *Excel* implementation.

```
public static void SetUp(Uri baseAddress, String authorizationToken)
{
  client = new HttpClient();
  client.BaseAddress = baseAddress;
  client.DefaultRequestHeaders.Accept.Clear();
  client.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue("application/vnd.api+json"));
  client.DefaultRequestHeaders.Authorization =
      new AuthenticationHeaderValue(authorizationToken);
}
```

Listing A.9: Configuring a connection in the *Excel* application

```
public static async Task<String> RequestQuestions(
    String version, String questionnaireID)
{
  HttpResponseMessage Response = await client.GetAsync("/"
      + version + "/questionnaires/" + questionnaireID + "/questions");
  return await Response.Content.ReadAsStringAsync();
}
```

Listing A.10: Requesting the question data of a questionnaire in the *Excel* application

```
public static async Task<List<String>> RequestAnswers(String version,
    String questionnaireID, String amount, String lastUpdate)
{
  HttpResponseMessage Response;
  if (amount != null)
  {
    Response = await client.GetAsync("/" + version + "/questionnaires/"
        + questionnaireID + "/results?page[Size]=" + amount
        + (lastUpdate == null ? "" : ",filter=" + lastUpdate));
  }
  else
  {
    Response = await client.GetAsync("/" + version + "/questionnaires/"
        + questionnaireID + "/results"
        + ( lastUpdate == null ? "" : "?filter=" + lastUpdate));
  }
  string Content = await Response.Content.ReadAsStringAsync();
  int requestsLeft = -1;
  long requestReset = 0;
  IEnumerable<string> values;
  if (Response.Headers.TryGetValues("X-RateLimit-Remaining", out values))
  {
    requestsLeft = Int32.Parse(values.First());
  }
  if (Response.Headers.TryGetValues("X-RateLimit-Reset", out values))
  {
    requestReset = long.Parse(values.First());
  }

  List<string> Results = new List<string>() { Content };
  JObject Page = JObject.Parse(Content);
  JToken lastPageToken = Page["links"]["last"];
  if (lastPageToken == null)
  {
    return Results;
  }
  string LastPageLink = lastPageToken.ToString();

  //Check if there is more than one page of data
  if (amount == null
      && !LastPageLink.Equals(Page["links"]["self"].ToString()))
  {
    while (true)
    {
      if (requestsLeft > 0 || requestsLeft == -1)
      {
        //request the other pages
        Response = await
            client.GetAsync(RemoveURL(Page["links"]["next"].ToString()));

        //update reset timer
        if (Response.Headers.TryGetValues("X-RateLimit-Remaining",
            out values))
        {
          requestsLeft = Int32.Parse(values.First());
        }
        if (Response.Headers.TryGetValues("X-RateLimit-Reset", out values))
        {
          requestReset = long.Parse(values.First());
        }

        Content = await Response.Content.ReadAsStringAsync();
        Results.Add(Content);
        Page = JObject.Parse(Content);

        //Breaking condition
        if (LastPageLink.Equals(Page["links"]["self"].ToString()))
        {
          break;
        }
      }
      else
      {
        //Wait until the request is valid again
        while((new DateTimeOffset(DateTime.Now).ToUnixTimeMilliseconds())
            < requestReset){
          System.Threading.Thread.Sleep(1000);
        }
        requestsLeft = 1;
      }
    }
  }
  return Results;
}
```

Listing A.11: Requesting answer result data of a questionnaire in the *Excel* application

```
public string QuestionnaireId;
public List<Question> Questions;
public List<ResultSet> Answers;
public Dictionary<string, int> QuestionNumbers { get; }

public QuestionSysData(string questionResponse, List<string> AnswerResponse,
    string id, QuestionSysData oldData)
{
  QuestionnaireId = id;
  Questions = new List<Question>();
  Answers = new List<ResultSet>();
  //create questions
  List<JToken> QuestionTokenList = JObject.Parse
      (questionResponse)["data"].Children().ToList();
  foreach (JToken Token in QuestionTokenList)
  {
    Questions.Add(new Question(Token));
  }

  //set up answers
  foreach (string AnswerResponseString in AnswerResponse)
  {
    List<JToken> AnswerTokenList = JObject.Parse
        (AnswerResponseString)["data"].Children().ToList();
    foreach (JToken Token in AnswerTokenList)
    {
      Answers.Add(new ResultSet(Token));
    }
  }

  //Set up dictionary
  QuestionNumbers = new Dictionary<string, int>();
  for (int i = 1; i <= Questions.Count; i++)
  {
    QuestionNumbers.Add(Questions[i - 1].Id, i);
  }

  //Insert old data here
  if (oldData != null && oldData.QuestionnaireId == QuestionnaireId)
  {
    List<string> answerIdList = getAnswerIdList();
    foreach (ResultSet result in oldData.Answers)
    {
      if (!answerIdList.Contains(result.Id))
      {
        Answers.Add(result);
      }
    }
  }
}
```

Listing A.12: A constructor of the QuestionSysData class to convert data in the Excel application

```
public QuestionSysData(Excel.Range StartPosition)
{
  //initalize lists
  Answers = new List<ResultSet>();
  Questions = new List<Question>();
  QuestionNumbers = new Dictionary<string, int>();

  QuestionnaireId = StartPosition.Comment.Text().Substring(18);
  //read questions
  int questionCount = 1;
  while (true)
  {
    string CellValue = StartPosition
        .Offset[0, questionCount].Value?.ToString();
    if (null == CellValue) break;
    string CellId = StartPosition.Offset[1, questionCount].Value?.ToString();
    Questions.Add(new Question(CellId, CellValue));
    QuestionNumbers.Add(CellId, questionCount);
    questionCount++;
  }
  //read answers
  int answerNumber = 2;
  while (true)
  {
    string AnswerId = StartPosition.Offset[answerNumber, 0].Value?.ToString();
    if (null == AnswerId) break;
    ResultSet CurrentResultSet = new ResultSet(AnswerId);
    for (int i = 1; i < questionCount; i++)
    {
      string AnswerText = StartPosition
          .Offset[answerNumber, i].Value?.ToString();
      if (AnswerText == null) continue;
      CurrentResultSet.Answers
          .Add(new Answer(Questions[i - 1].Id, AnswerText));
    }
    AddResultSet(CurrentResultSet, QuestionNumbers);
    answerNumber++;
  }
}
```

Listing A.13: A constructor of the QuestionSysData class from existing data in the Excel application

```
public static void InsertData(Excel.Range StartPosition,
    QuestionSysData newData)
{
  //storing meta information
  WriteMetaInformation(StartPosition, newData.QuestionnaireId);

  //convertingQuestions
  //Input questions
  StartPosition.Value = "AnswerID";
  for (int i = 1; i <= newData.Questions.Count; i++)
  {
    StartPosition.Offset[0, i].Value = newData.Questions[i - 1].QuestionText;
    StartPosition.Offset[1, i].Value = newData.Questions[i - 1].Id;
  }

  int CurrentYOffset = 2;
  //Input answers
  foreach (ResultSet resultSet in newData.Answers)
  {
    int maxLength = 1;
    //write all answers
    foreach (Answer answer in resultSet.Answers)
    {
      if (answer.AnswerText.Count > maxLength) maxLength
          = answer.AnswerText.Count;
      for (int answerNumber = 0; answerNumber
          < answer.AnswerText.Count; answerNumber++)
      {
        StartPosition.Offset[CurrentYOffset + answerNumber,
            newData.QuestionNumbers[answer.QuestionId]].Value
            = answer.AnswerText[answerNumber];
      }
    }
    //write answer ids
    while (maxLength > 0)
    {
      StartPosition.Offset[CurrentYOffset, 0].Value = resultSet.Id;
      maxLength--;
      CurrentYOffset++;
    }
  }

  //Formating stuff (optional, looks finer)
  for (int i = 0; i <= newData.Questions.Count; i++)
  {
    StartPosition.Offset[0, i].Columns.AutoFit();
  }
  //Remove Borders inside the data
  StartPosition.Cells.Range[StartPosition, StartPosition.Offset
```

```
        [CurrentYOffset - 1, newData.Questions.Count]].Borders.LineStyle
        = Excel.XlLineStyle.xlLineStyleNone;
    //Border around everything
    StartPosition.Cells.Range[StartPosition, StartPosition.Offset
        [CurrentYOffset - 1, newData.Questions.Count]].BorderAround2();
    //Border around questions and questionIds
    StartPosition.Cells.Range[StartPosition, StartPosition.Offset
         [1, newData.Questions.Count]].BorderAround2();
}
```

Listing A.14: Saving data in the Excel application

List of Figures

Name: Sean Duft Matriculation number: 905163

Honesty disclaimer

I hereby affirm that I wrote this thesis independently and that I did not use any other sources or tools than the ones specified.

Ulm, ...

Sean Duft