QSYS - System Documentation

Version 0.8

`http://survey4all.org`

1 Introduction

Here, a short description of its functionality and technical background is given, which should serve as a short introduction to how the software can be extended (e.g. add question types and access modes for the respondents or modify the styling of the webpage). To take a look at a running instance of the actual version and get additional information, visit `http://www.survey4all.org`.

1.1 General Software Description

QSYS is a Web based survey software intentionally developed for running experiments dealing with visual design effects to respondent's behavior. In the current version, the software includes an easy to use online-editor supporting the online survey's creation, accomplishment and administration. The users can develop an online-questionnaire without any previous knowledge of HTML.

As online survey software constitutes a competitive market, one has to ask, why do we need another product in this field? Conversely to other commercial and non-commercial products, QSYS gives users and developers access to the source code and the permission to create derivative works from the original survey software. This option allows the adaptation and extension of the software to customers' needs and the installation of the complete package without any license fees. Additionally, there is no software with good support for experimenting with visual design in web surveys. QSYS attempts to bridge this gap. Of course it would have been much easier to use an existing package (preferentially an open source package) and customize it according to the special needs for the experiments. But all the existing packages have one major drawback: separation between content, style and design of the questionnaire respectively is not strictly implemented. The conclusion would have been to copy all questionnaires as often as different styling's are needed and assign the styles to them, which is not an optimal approach. The second motivation was simply to write a well structured and extendable open source survey development tool, publish it as open source, and see what happens.

The development process focused on the design of reliable and extendable software architecture. Therefore, new question types, extensions to existing question types, style and appearance of questions can be created and adapted quite easily without the necessity of breaking with existing software design concepts. Moreover, questions' visualization is strictly separated from the questionnaire's content, so each survey can be presented in an individual style with low demand on resources. QSYS supports all common question types together with a few innovative ones (like image map questions), which are customizable as well, and a diverse range of participation modes. Additionally, features like PDF and XML export are implemented. The elaborate software architecture allows a rapid extension, customization and embedding into a proprietary infrastructure.

1.2 Overview of Features

This section provides an overview of all the QSYS features:

- Several different question types are supported with various configuration possibilities.
- For all question and alternative texts, a WYSIWYG-editor is offered to the questionnaire designer to easily assign the desired styling, so changing font types, adding links, tables images and all other controls supported by HTML can be added without any web-technology knowledge necessary. [1].
- Questions can be displayed in a paging (one page per question) or in a scrolling (multiple questions on one page) mode. When applying the scrolling mode, page separators support the question's categorization as logical units.
- Branching on behalf of answers to closedended questions is possible.
- Questions can be marked as mandatory to give feedback to the respondent, when the question is not answered sufficiently.
- PDF export creates an offline version of the questionnaire (e.g. for mixed mode studies).
- XML export serves as an exchange and archiving possibility (answers of respondents can also be exported as XML). When a survey creator is familiar with the XML schema used for questionnaires, survey creation or e.g. repeating modifications, these can be done very efficiently by directly editing the XML-document. It is even possible to write a custom editor tailored to your needs.
- Paradata tracking is implemented e.g. for exporting the time needed for filling out one question together with browser and operating system information. In addition the export of the IP address for each participant can be enabled. Storage can be turned off or masked (to export only parts necessary to identify e.g. an institution rather than an individual person, which would for example like this: *138.232.xxx.xxx*) to assure privacy, because this feature has to be used with caution: "Organizational researchers need to be sensitive to the confidentiality and security issues associated with the use of Internet survey tools, as participants may be

[1]see http://tinymce.moxiecode.com/ for further information.

unwilling to provide candid responses to a survey if their anonymity is not ensured" [Tru03, p.35].

- Language independency is secured, as language tokens are stored externally. As a result, a simple translation of these tokens adapts the software to a further language.
- An optional summary of all answers for one respondent can be offered at the end.
- The questionnaire's completion can be limited to a certain period.
- Advanced interviewee restriction modes (all can participate, common PIN-code, user/password list, members of certain LDAP-groups, ...).
- Entire software is published under an Open Source License allowing extension and customization to distinct needs.
- A common Servlet engine is sufficient to install QSYS on a server. Not even a database is needed (but it is possible to use one).
- A status (*DEVELOPMENT*, *PRETEST*, *OPERATIONAL*, *DISABLED*) can support the user to distinguish between different phases of the survey stored together with the given answers.
- Data can be exported as CSV (which can be imported into all statistical analysis tools including Excel), SPSS (currently *sps*-files are generated) and native XML.

All these tasks are described in more detail in section 1.4.

1.3 Supported Question Types

QSYS supports a wide range of question types (all allow several variations):

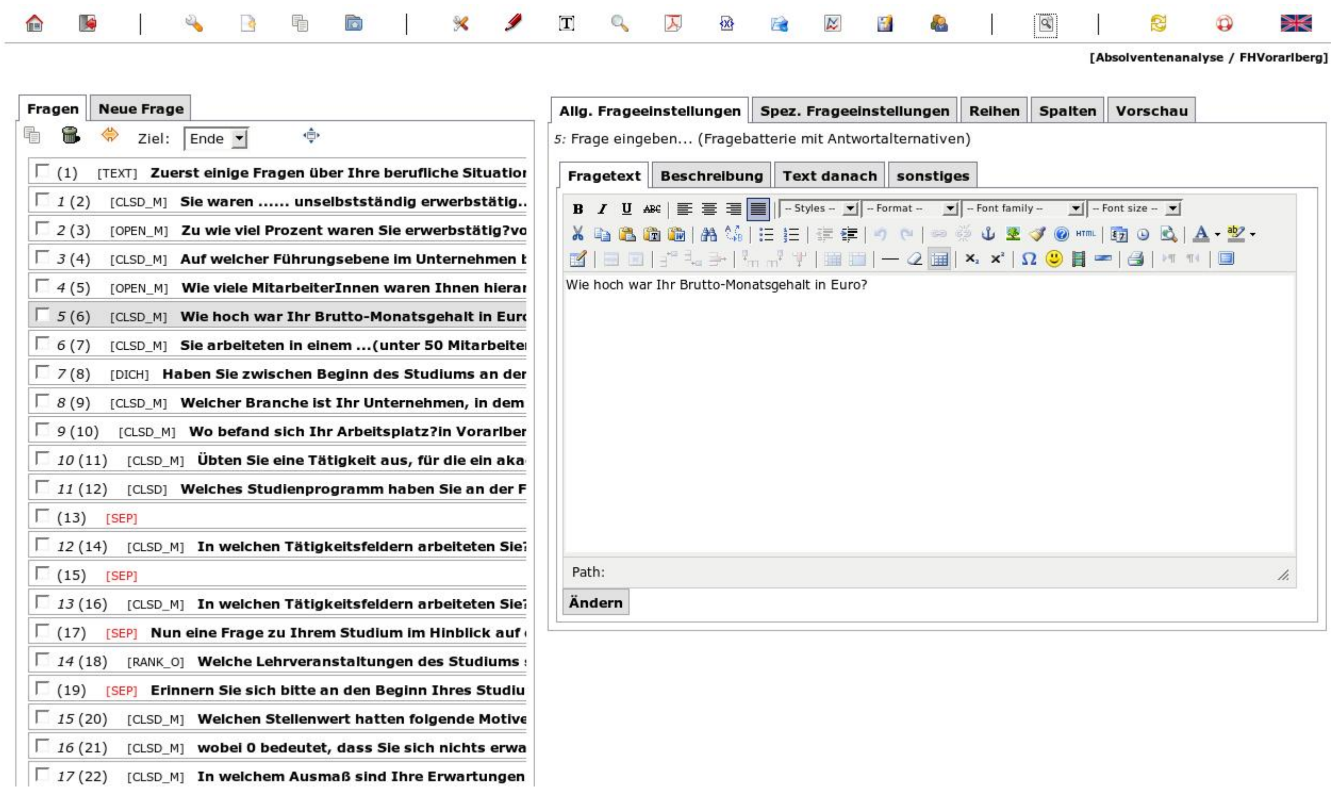
- Closedended questions.
- Dichotomous questions: similar to closedended questions. Here, the respondent can select one of two alternatives.
- Pictogram questions: similar to closedended questions. Here with pictures instead of textual alternatives.
- Closedended ranking questions: alternatives have to be brought in the right order.
- Question matrix: multiple questions with the same alternatives to select.
- Question matrix with column grouping: integrating multiple matrices with the same sub questions in one view.
- Semantic differential or VAS: Visual Analogue Scale (VAS) constitutes a measurement instrument measuring a characteristic or attitude believed to range across a continuum of values. VAS is verbally anchored on each end, e.g. *very good* vs. *very bad* (support for a couple of sub questions and the anchor points can be set for each subquestion individually.

- Interval question matrix: just like a semantic differential, except for this type, labelling of anchor points is the same for all subquestions.
- Interval questions: similar to interval question matrix, but without sub questions (the main question itself is rated)
- Openended questions. For this type, size of input field can be varied, also number and date fields are supported.
- Openended ranking questions: presenting multiple openended input fields for one question to the user.
- Openended question matrix: presenting an openended input field for each subquestion of the matrix to the user.
- Image map questions: respondents can select a certain region of an image map (e.g. a geographical map).
- Text blocks and page separators: these are workflow elements, not questions, for the purpose of separating parts of the questionnaire and to add HTML code between questions.

1.4 Editor

Generally, the CMS area, of which the editor is one part, is split up into groups. Each group has a group administrator with a password used for logging in. One group can contain multiple questionnaires, which are all listed and ready for editing on the group administrators overview page.

The editor for creating and customizing the survey is easy to use and featured with *AJAX* technology. This has the positive effect that pressing a *submit* button and waiting for the browser to reload the whole page becomes unnecessary. A simple click on e.g. a checkbox or button is sufficient to send edited data to the server for storage (although feedback is given if the action was performed successfully or if an error occurred). This drastically speeds up entering and editing questionnaire content.


[Absolventenanalyse / FHVorarlberg]

(1) [TEXT] Zuerst einige Fragen über Ihre berufliche Situatior
1 (2) [CLSD_M] Sie waren ...... unselbstständig erwerbstätig..
2 (3) [OPEN_M] Zu wie viel Prozent waren Sie erwerbstätig?vo
3 (4) [CLSD_M] Auf welcher Führungsebene im Unternehmen t
4 (5) [OPEN_M] Wie viele MitarbeiterInnen waren Ihnen hierar
5 (6) [CLSD_M] Wie hoch war Ihr Brutto-Monatsgehalt in Eurc
6 (7) [CLSD_M] Sie arbeiteten in einem ...(unter 50 Mitarbeite
7 (8) [DICH] Haben Sie zwischen Beginn des Studiums an der
8 (9) [CLSD_M] Welcher Branche ist Ihr Unternehmen, in dem
9 (10) [CLSD_M] Wo befand sich Ihr Arbeitsplatz?in Vorarlber
10 (11) [CLSD_M] Übten Sie eine Tätigkeit aus, für die ein aka
11 (12) [CLSD] Welches Studienprogramm haben Sie an der F
(13) [SEP]
12 (14) [CLSD_M] In welchen Tätigkeitsfeldern arbeiteten Sie?
(15) [SEP]
13 (16) [CLSD_M] In welchen Tätigkeitsfeldern arbeiteten Sie?
(17) [SEP] Nun eine Frage zu Ihrem Studium im Hinblick auf
14 (18) [RANK_O] Welche Lehrveranstaltungen des Studiums
(19) [SEP] Erinnern Sie sich bitte an den Beginn Ihres Studiu
15 (20) [CLSD_M] Welchen Stellenwert hatten folgende Motive
16 (21) [CLSD_M] wobei 0 bedeutet, dass Sie sich nichts erwa
17 (22) [CLSD_M] In welchem Ausmaß sind Ihre Erwartungen

5: Frage eingeben... (Fragebatterie mit Antwortalternativen)

Wie hoch war Ihr Brutto-Monatsgehalt in Euro?

Figure 1.1: A sample view on the questionnaire editor

In figure 1.1, the main editor window is shown: on the left, a navigation bar listing all questions together with an abbreviation of the question type to give an overview over the whole questionnaire and to enable easy navigation is given. A full view mode for the navigation bar is offered, which means that this bar occupies the whole screen, so full question texts are visible. This should ease and speed up exchanging, copying, and removing one or more questions. To add a new question, simply select the desired type and the position from the tab on the left, where the question should be added. In addition, page separators can also be added this way.

The main section of the editor view is split up into different tabs. The tab *common question settings* is the same for all question types and is used to enter the general question text as well as a description text (which should not be part of the question itself, but should give hints how the question should be filled out. This text is usually written under the question text itself, in smaller letters). Also, a text to be displayed after the question can be specified. As additional settings for each question, some attributes can be set, like mandatory status of the question and general successor of the question, which is used for branching.

All other tabs depend on the question type which is currently edited. There is a tab *specific questionnaire settings*, where question type specific editing can be done (e.g. for question batteries, column width and alignment can be set). All tabs to the right are also question-dependent. For example, when a question battery is edited, additional tabs like *row* and *column* come up, where sub questions and alternatives for the battery can be added. As last tab, preview is given, which is also available for all question types. The currently edited question is displayed is it would have been displayed to the interviewee.

The following *basic settings* for can be applied the whole questionnaire:

- Short link: when the link to the questionnaire is published (in invitation letters, e-mails or news groups), it is beneficial to have a link with only a few and short parameters (to avoid e.g. line break problems with some e-mail clients).
- Should the IP-address be stored: when anonymity plays a major role for a survey, IP-addresses can be made anonymous (which is the default setting), or masked.
- Number of questions per page: it is possible to distinguish between *one question per page*, *all questions on one page* and *using separators* (which can be added within the main editor view as described above).
- Should a progress bar be offered to the respondent or not.
- Should a summary of entered data be offered to the respondent after filling out.
- A logo can be uploaded which is displayed on each page of the questionnaire.
- A data range can be specified where the questionnaire is set as active. If the current date is outside this interval, an appropriate message is displayed when accessing the questionnaire and participation is impeded.

1.5 Conventions

It was attempted to abide by a few general conventions in regard to the software:

- Only open source tools are used for development, running, administering, documenting and all other tasks which have to do with the software itself.
- The software is published as open source with all its components and subprojects.
- The software development process should be driven by the end users, which means, that the needs of the end users (those who are using this tool to run surveys) should be considered. This is a good test of the extensibility of the software architecture and is an interesting experiment in which direction the development will go.
- But nevertheless QSYS will always focus on being an online survey tool, there is no intention to integrate e-learning capabilities or the possibility of using data analysis. Of course it is desirable to create such functionality, but the focus should be kept on separating this functionality in external projects , where QSYS serves as a core system offering basic functionality via Web services.

The software is already published as open source (*GNU General Public License (GPL)* [2] or see [St.04] for basic principles of open source software licensing in general and a detailed explanation of GPL on pages 35-48), visit http://www.survey4all.org or qsys.sourceforge.net for further information. A sociological approach to open source strongly related to the shift in labor in general is given in [HB07].

1.6 Technical Background

In this chapter, an overview on the technical background and technologies used should be given. Also technical preconditions necessary to install and run the software are listed.

The application is based on Java Servlet technology. For the creation of the application *Jakarta Struts* framework[3] was applied as controller (in a slightly unconventional way). XML plays a major role on all layers of the software architecture. All questionnaires and additional information is stored as native XML documents. To render these questionnaires as HTML or PDF, *XSLT* and *XSL-FO* is used, which results in a complete separation of content and view. The decision which style sheet is used for which content file (which means e.g. for which questionnaire) is administered in an external configuration file. It is even possible to assign a certain style sheet at random or based on certain conditions (in case of the experiments, technical preconditions fetched from the browser settings (e.g. if Javascript was enabled) determined the necessary style sheet). An oracle database, eXist (open source database) or even the file system alone (which is currently the preferred option) can be used for data storage.

It should also be mentioned that during the development of the whole project, a standalone Struts XSLT framework was created, which meets the demands of both, struts and XML/XSLT (with language independency and other useful features). Some features of the framework are specially created for the needs of the whole project, such as selecting an XSLT style sheet for one view by chance.

1.6.1 Technical Preconditions

For installing and running the software, only a Servlet engine like e.g. *Tomcat* (all tests have been run on this engine) or *Jetty* is necessary. Data is stored (at least in the default configuration) directly within the file system, which means not even a database system has to be installed or connected to. Because of these few technical necessities, it is even possible to run the software on the local machine to prepare a questionnaire offline and upload it later to the productive system or use QSYS for small surveys (like evaluation sheets in the field of education) directly on your local machine or laptop (which would work as a small server in this case).

[2]see: `http://www.gnu.org/licenses/`
[3]http://struts.apache.org/

2 Software Architecture

In this chapter, a brief overview of the software architecture of the whole system and its components is given.

The software is split up into subprojects to ensure reusability of the particular components in other projects. Figure 2.1 shows a component model of the whole QSYS-system. All parts are described in the following sections:

- *QSYS-core*: the core functionality (like creating, storing and managing questionnaires) of the whole system.
- *QSYS-web*: the Web interface for the online version of QSYS with all XSLT style sheets and mapping information.
- *QSYS-tools*: console based tools for automated processing are implemented accessing QSYS-core.
- *struXSLT*: a standalone XSLT-extension for the Struts framework. Amongst other projects, *QSYS-web* is based on this framework.
- *q-utils*: simple utility classes used by all components.

2.1 QSYS-core

This is the core component of the QSYS system. It was separated from the Web frontend to be open to a possible branch which would lead to a standalone (and not web-)

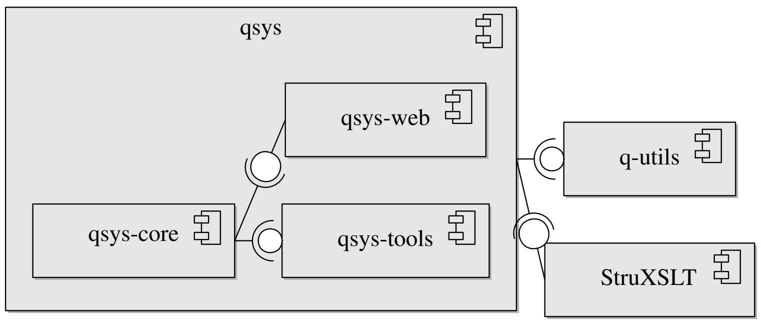


Figure 2.1: Component model of the whole QSYS-system

application for creating questionnaires. Currently, there is only a Web version available for creating the surveys.

One of the main tasks this component is responsible for is the storage of questionnaires. A questionnaire consists of a list of questionnaire items. Each questionnaire item has two representations, one as object and the other one as XML fragment. Mapping between these two manifestations is done within each class, which means each question type has its own XML-(de-)serialization methods. There are tools who could do this in an automated way (like e.g. *JAXB*[1]), but most of them have drawbacks which in some cases influence software architecture (e.g. concerning visibility of member-variables). Because of strong similarity of the questions, inheritance is heavily used. For one simple question type, an UML class diagram is shown in figure 2.2 to demonstrate the hierarchy and organization of the questionnaire item classes:

2.1.1 Questionnaire Items

Subsequently a short overview of the class hierarchy of questionnaire items (a questionnaire item is the base class for questions and workflow elements) is given. For a graphical representation see figure 2.2, where just examples of the concrete question classes are given: an interval question (giving a rating on a scale between two anchor points) and a simple openended question. For the concrete questions, the *isDataComplete* method checks if the question was completely filled out by the respondent (this method is called by *canDataBeStored*). All other question types are organized the same way (some with multiple inheritance hierarchies). For a complete list of supported question types, see section 1.3. For all questions holding sub questions, an interface with an implementation exists managing all the tasks necessary for holding multiple questions.

In both, the textual description and UML class diagram only the core concepts are illustrated, for a more detailed view consider to take a look at the source code which can be browsed and downloaded at `www.survey4all.org`.

A *QQuestionnaireItem*-class has two member variables, the id, which is unique for the whole questionnaire and used for referencing the questions, and *dispId*, which is the id to be displayed on the questionnaire. The *item-type* of each item is set via annotations[2]. It is possible to get a list of all concrete questions (or questionnaire items respectively) with their class and the *type*. This mechanism is also used to generate an instance of a class only by knowing the type (such a type could be e.g. *questionMatrix*, the corresponding annotation would be *@QuestionA(name = "questionMatrix", order = 5)*, where *order* is the position of the question when presenting all questions within a list. Here another example of an annotation for the page separator: *@QuestionA(name = "pageSeparator", order=2, isQuestion=false))*. Class *QFactory* is responsible for retrieving concrete ques-

[1]https://jaxb.dev.java.net/

[2]Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate. (taken from the documentation-pages of `http://java.sun.com`)

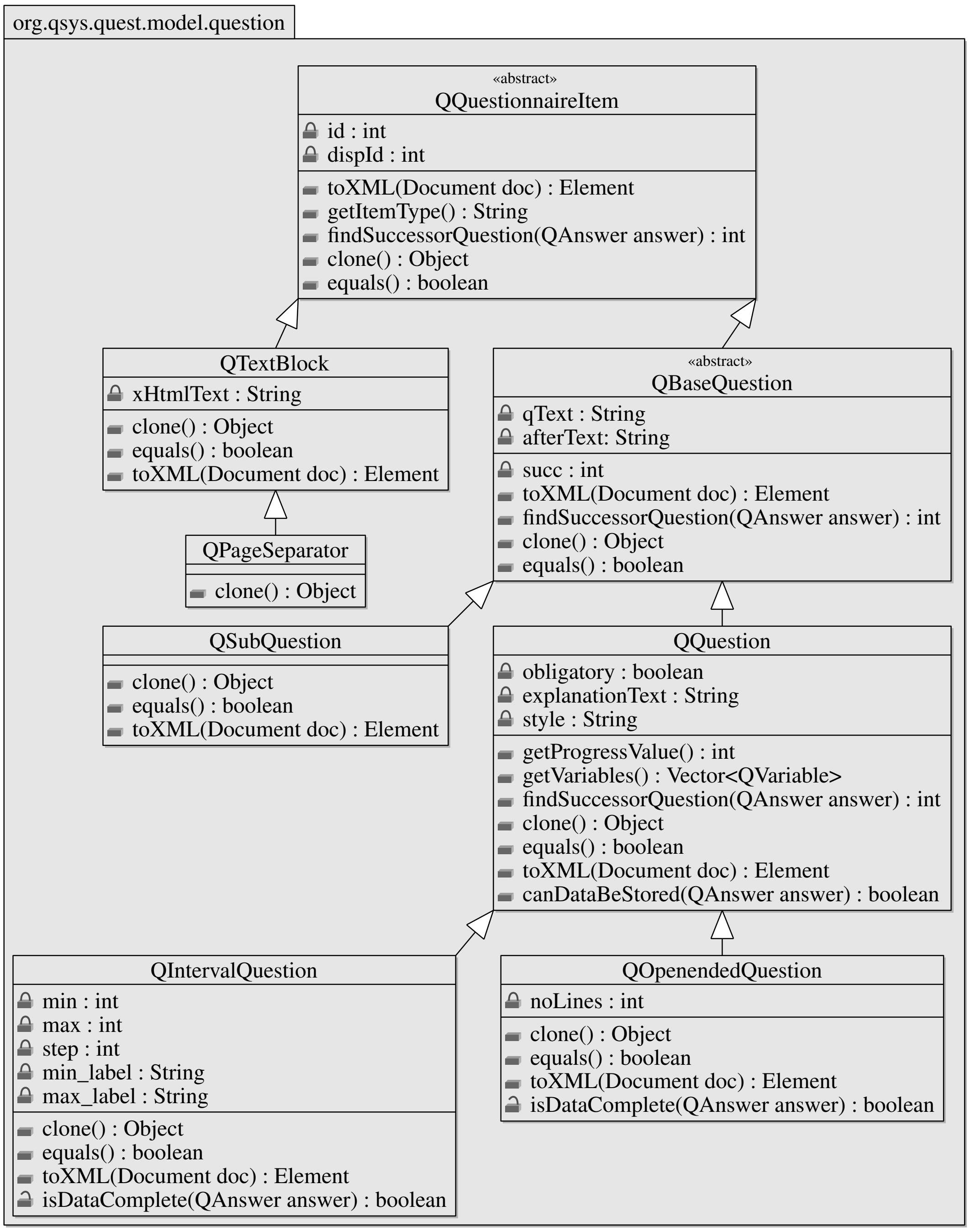


Figure 2.2: Question classes diagram showcase

tionnaire item instances by question type. To achieve this, the reflection-technique is heavily used. Suitable constructors are invoked dynamically.

For all question types, the methods of class *Object* (like *clone* (to create deep copies of question objects), *equals* (for comparing questions, mainly used within the test classes)) are overridden. Each question has a default successor question, which is used for branching (which will be described in a later section). All of these implement the *Cloneable* interface, and all subclasses do implement a *toXML* method used for serialization to XML, and a constructor taking an XML element used for generating a concrete questionnaire item based on the content of this element. In addition each questionnaire item knows best how to find out the successor question according to the answer given (for a description of the *Answer*-class hierarchy see section 2.1.2).

On each position of the questionnaire, *text blocks* can be integrated, where all capabilities of HTML can be used. These blocks are also created with a WYSIWYG-editor within the Web presentation layer. A *page separator* has the same functionality, but if manual page separation is enabled, these elements act as page separators and are displayed as the first entry of a section containing multiple questions. The items have to be strictly separated from questions, because they have no corresponding answers, which affects different central parts of the software.

All question types inherit *QBaseQuestion*. Each question must contain a question text (*qText*), a text after the question (*afterText*) and a default successor question (*succ*). Again, all texts stored can contain HTML. When a question consists of multiple sub questions, these classes contain a list of *QSubQuestion*.

All standalone questions (which means all but sub questions) inherit from *QQuestion*. Therein the information is stored whether a question is obligatory and an additional explanation text can be set, which is usually displayed under the question text itself and should guide the respondent through the filling out process, as well as whether a *style* attribute can be set per question. This should give the presentation layer a directive as tohow this question should be displayed. An example of different styles would be the display of radio buttons instead of a text input field for an interval question. The method *getProgressValue* calculates some kind of *expected duration weights* for one question. These values e.g. depend on the number of sub questions and the question type itself, but these strategies can be extended according to the own needs. The method *canDataBeStored* determines if the answer given by the respondent is sufficient if the question is marked as mandatory. The method *getVariables* returns a list of variables which are generated for one question, which is primarily used for data exporting.

The whole questionnaire is held as an instance of *QQuestionnaireObj*, where a list of *QQuestionnaireItem* objects are stored together with a *QHeader* instance, which holds

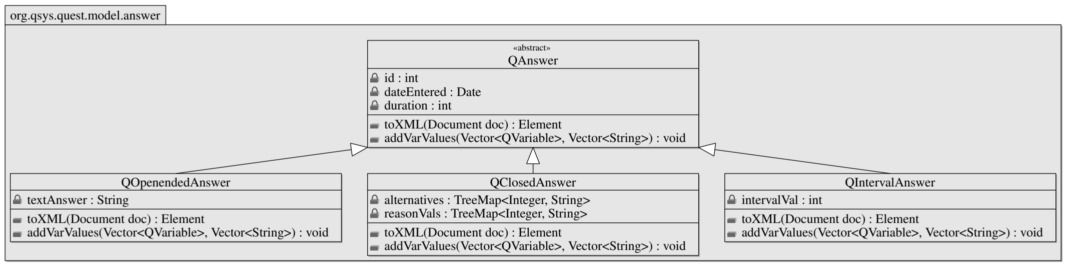

org.qsys.quest.model.answer
«abstract»
QAnswer
id : int
dateEntered : Date
duration : int
toXML(Document doc) : Element
addVarValues(Vector<QVariable>, Vector<String>) : void
QOpenendedAnswer
textAnswer : String
toXML(Document doc) : Element
addVarValues(Vector<QVariable>, Vector<String>) : void
QClosedAnswer
alternatives : TreeMap<Integer, String>
reasonVals : TreeMap<Integer, String>
toXML(Document doc) : Element
addVarValues(Vector<QVariable>, Vector<String>) : void
QIntervalAnswer
intervalVal : int
toXML(Document doc) : Element
addVarValues(Vector<QVariable>, Vector<String>) : void

Figure 2.3: Answer class diagram showcase

administrative data regarding the questionnaire itself (e.g begin- and end-date, status, creator, creation date). All modifications on such a questionnaire object are done by *QuestionManager*. This layer is necessary to retrieve and store the modifications done on the questionnaire object on the preferred medium (see section 2.1.4).

2.1.2 Answers

One strategical consideration was to strictly separate storing the questionnaire and the answers given for a questionnaire, which has the consequence of a class hierarchy for answer classes parallel to the questionnaire items hierarchy. To weave question and answer objects together (which means to determine which answer class is responsible for storing data for which question type), once again annotations are used. For example, the annotation *@QuestionsA(names={"questionMatrix", "questionMatrixMult"})* when placed for *QClosedMatrixAnswer* would mean that this answer type is responsible for processing answers for question types *questionMatrix* and *questionMatrixMult*. There are less answer classes than question classes, because some questions generate the same answer data, and so, some answer classes can be used twice.

In figure 2.3, same as for the questionnaire-items, an UML class-diagram together with a short description of the classes and their methods, is shown. Again, only a few concrete answer classes are added to the diagram just to show the basic class structure. All classes inherit directly from QAnswer and simply contain the answered values, which was fetched out from *QAnswerDictionary*. E.g. in case of the openended answer class, this is a simple string, in case of the closed answer, a list of possible selected alternatives and reason values (which mean additional texts which can be entered next to a selected alternative) and in case of the interval questions a position on the scale is given.

Similarly as for questionnaire items, the *toXML* method exists together with a constructor including an element used for XML-serialization. Additionally, a constructor exists for each concrete *QAnswer*-object with *QAnswerDictionary* as the parameter for stuffing the answer objects with content. This class inherits from *Hashtable<String, String>* and is used to transfer the answers from the frontend to the model in a generic way. Each answer class then knows best how to interpret the content of this hashtable and reads the appropriate information. In addition a method *addVarValues(Vector<QVariable> var, Vector<String> vals)* exists, where concrete variable values are generated, which is used for exporting. All values are appended to variable *vals* (call-by reference) according to the variable settings given in *var*. The list of variables is the only communication point which should show the concrete answer how and which variable values should be exported. Additionally, for each answer, the duration (in milliseconds) is stored. In the current Web implementation, this duration is tracked on client side, which means, measurement starts when the page is completely loaded, and ends, when the submit button is pressed.

Class *QAnswersObj* holds all answers given for a questionnaire together with a header, which holds some metadata about the filling out process, such as: the time when filling out started, an unique interviewee id, a flag whether the respondent finished filling out or not, which style was assigned (this was essential for the experiments) and paradata (see section 2.1.3) collected amongst others directly within the client's browser.

2.1.3 Paradata Tracking

Online-surveys give the additional opportunity to track information about the filling out process and the instrument used by the respondent. Here a list of paradata tracked by the software and what can possibly be implied from this information:

- Session identifier: identify answers filled out immediately one after the other on the same computer.
- Screen resolution (width and height)
- User agent (the web-browser used)
- Operating system
- Cookies enabled
- Java enabled
- Javascript enabled: this and previous paradata within the list can be used to possibly identify side effects or technical problems e.g. with certain browsers having special configuration set (e.g. when Javascript is disabled or low screen resolution).
- First referrer: in case when recruitment is done on different Web pages, it can be determined from which websites the respondents came from.
- Remote address (the IP-address of the respondent). Storing the IP address unmasked is turned off by default because of possible problems concerning the anonymity of the respondent. If those who run the survey are aware of these problems and turn IP-tracking on, additional possibilities come up: in some cases it is possible to determine which questionnaires have been filled out on the same machine. This statement is qualified, because in some cases (in general within big institutions) different computers appear with the same IP-address on the internet (which is in

most cases the IP-address of the proxy-server of the institution). But in this case at least the institution can be found out. In addition some providers use dynamic IP-addresses, which means that when the respondent's PC is connected to the internet sequentially multiple times, always a different IP-address is assigned and there is no chance to identify pc's individually. In some cases it is possible e.g. if the respondent filled out the questionnaire at home or at work, if this information is of any interest for the survey.

Technically, tracking is directly done within the client's browser via Javascript. An AJAX call delivers all the parameters to the server whilst the webpage is loading in the backgroun of the respondent's browser. The connection to the answers given by the respondent is done via session-id. If Javascript is turned off, most of these parameters cannot be accessed. Some parameters like operating system, user agent, session-id and first referrer can also be fetched from the *HTTP header* on the server side.

2.1.4 Storage

Basically, three types of storage are currently supported; information is held...

1. ...directly within the file system as native XML-documents.
2. ...within an oracle database, where XML-documents are stored as *XMLType.* A free version of an oracle database (Express Edition[3]) is available for free use, which fulfills the requirements of QSYS.
3. ...within the open source native XML-database eXist[4].

To make QSYS work with one of these options, setting the entry *DB* within *qsys.properties* (which is located in the QSYS-project's WEB-INF/classes) is sufficient. Possible entries are: *FILESYSTEM, ORACLE, EXIST*.

The first option is preferred for several reasons: no preconditions (which means in this case no installation or access to a running database is necessary) concerning storage have to be fulfilled. It is also the most frequently used version for all currently installed versions of QSYS, so it's the most stable variant In addition some features are currently not implemented for the other two alternatives. This mode is also very fast and does not need much space on the server. Furthermore, backup of simple XML-files can also be done very easily. If any urgent modifications have to be done, simple direct changes on the files are sufficient.

For these reasons, this storage mode is described here in more detail (but e.g. storage for the other 2 alternatives works very similarly):

[3]`http://www.oracle.com/technology/products/database/xe`
[4]`http://exist.sourceforge.net/`

2.1.4.1 File Organization

To set the root folder for XML-storage, one should alter the file *WEB-INF/classes/fsdb.properties* within the QSYS-project and set the parameter *PATH* to the desired storage location. The files are hierarchically organized as follows: within folder *conf*, two files are located: *groups.xml* (which contains basic general and status information for each group) and *shortlinks.xml* (a mapping of group id and questionnaire id to a tiny link is provided to shorten the link to a questionnaire). All other information is stored in folder groups: Each group has a folder (the id of the group is the folder name) with a document *questionnaires.xml* which contains basic overview information of each questionnaire and a subfolder for each questionnaire. The organization of files within a group can be seen in listing 2.1. In this example, all terms within angle brackets should symbolize a variable for any string. The following files are directly located within the group's root directory:

- *basicSettings.xml* holds basic settings about how questioning should be done (e.g how many questions should be displayed on one page, should the IP-address be stored, should a summary be displayed at the end and should a progress bar be displayed).
- *interviewees.xml* stores the mode of interviewee recruitment and basic settings for the selected mode.
- *questionnaire.xml* is the questionnaire itself.

The subfolder *answer* contains all answers given by the respondents, one document for each. The name of the file consists of the unique interviewee id together with a timestamp when the interview was made to ensure uniqueness. When taking a look at the questionnaire during creation, the results of the test runs are stored in *groupadmin.xml* in order be able to exclude this document in the further analysis.

```
$ tree <group_1>
<group_1>
|-- questionnaires
|   |-- <quest_1>
|   |   |-- answer
|   |   |   |-- ADUDOIIO_1197533982151.xml
|   |   |   |-- AIQTONZY_1196676852825.xml
|   |   |   |-- ALFQWJMO_1196595314843.xml
|   |   |   |-- AVJRVCGO_1196589478675.xml
|   |   |   `-- groupadmin.xml
|   |   |-- basicSettings.xml
|   |   |-- interviewees.xml
|   |   `-- questionnaire.xml
|   `-- <quest_2>
|       |-- answer
|       |   |-- TPQDDYSJ_1195763591505.xml
|       |   `-- groupadmin.xml
|       |-- basicSettings.xml
|       |-- interviewees.xml
|       `-- questionnaire.xml
`-- questionnaires.xml
```

Listing 2.1: A simplified example of a tree-view for files stored for one group in QSYS

2.1.4.2 XML Queries within the File System

Even if there is no database available and all answers are stored as simple XML documents, easy queries are possible to get an impression of the current response using standard *UNIX* tools such as *xmlstarlet* (which supports even XPath 2.0 functions like *min*, *max*,...), *uniq* and *sort*. As an example, a simplified answer document is given in listing 2.2 whcih is used to demonstrate the query examples in the following listings. All examples run very fast, at least not appreciably slower as if a database would have been used.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<response>
    <header finished="true">
        <tracker remoteaddress="130.82.1.40" />
        <creDate>28/01/2008 03:36:59 PM</creDate>
    </header>
    <answer dateEntered="28/01/2008 03:39:21 PM" duration="135967"
        id="2" type="QClosedMatrixAnswer">
        <subquestion id="1">
            <alternative value="1"/>
        </subquestion>
        <subquestion id="2">
            <alternative value="2"/>
        </subquestion>
    </answer>
    <answer dateEntered="28/01/2008 03:56:09 PM" duration="33251"
        id="46" type="QOpenendedAnswer">
        <answer>1962</answer>
    </answer>
    <answer dateEntered="28/01/2008 03:56:09 PM" duration="35142"
        id="48" type="QClosedAnswer">
        <alternative value="5"/>
    </answer>
</response>
```

Listing 2.2: A simplified example of an answer document

To perform queries on a folder containing such XML documents, the combination of simple UNIX tools (as mentioned above) is sufficient. This should be illustrated with the following commands: to find out how many respondents have currently filled out a questionnaire (with a distribution of how many finished and how many dropped out), run the following command:

```
$ xmlstarlet sel -t -v "//header/@finished" * | sort | uniq -c
156 false
 73 true
```

Listing 2.3: Command for respondent's overview of one questionnaire

Here (and also in the following script example) an *XPath* query is applied on all documents of the *answer* folder. All resulting values are sorted and counted by the *uniq*-command. The result of the example shows that 73 completed filling out the questionnaire and 156 dropped out.

Another example script, where a distribution of the last filled out id's is given (*NaN* means not a single question was filled out):

```
$ xmlstarlet sel -t -v "math:max(//answer/@id)" * | sort -n | uniq -c
93 NaN
 5 2
31 12
 1 13
 2 47
 1 49
71 50
```

Listing 2.4: Command for finding out the distribution of the last filled out question

The same can be done with concrete results of single questions, e.g. for openended questions (see question with id=46 in listing 2.2):

```
$ xmlstarlet sel -t -v "//answer[@id=46]/answer/text()"  * | sort | uniq -c
14 1968
15 1969
11 1970
 5 1971
 1 1972
```

Listing 2.5: Command for querying the results of an openended question

The same procedure is valid for closedended questions:

```
$ xmlstarlet sel -t -v "//answer[@id=48]/alternative/@value"  * | sort | uniq -c
57 1
 1 2
 6 4
 5 5
```

Listing 2.6: Command for quering the results of a closedended question

These are just small examples. The possibilities of these tools are huge, when combining these scripts, integrate more logic and existing information (like the question texts), a whole reporting system could be implemented quite easily.

But of course also some drawbacks of storing native XML documents within the file system have to be mentioned:

- Data integrity cannot be assured as it could be done within a database when using constraints.
- Central storage and access control is not as easily possible.

2.1.5 Exporting

When implementing the exporting package, it was attempted to generate a common framework for writing tabular data to file, because this task is, regardless which content is written, always the same. Furthermore, introducing a new export format should be possible without much effort. So the strategy was to build a clear class hierarchy where methods in the base classes do all common tasks and only assigning different content should be done within the sub classes. *BaseExporter* and *BaseDataExporter* implement already common tasks for writing (or abstract methods are defined to leave concrete implementations open) a header line and multiple content lines together with the whole exporting process. Concrete implementations are then located within the two sub classes. It is sufficient to request a concrete data-exporter (either *CVS* or *SPSS*) from class *ExporterFactory* to get the desired output format.

Currently *CSV*[5] is used as an exporting format. As a second format, *SPSS* is supported. Because the format for *SPSS* data files (*.sav*) is relatively complex and only commercial libraries exist (like Java *SPSS* Writer from pmStation[6]), and these are solutions which do not fit into the overall concept, data is written into an *SPSS* syntax file (which has the same effect, all necessary information can be set, and after running the syntax, data is written as a common *SPSS* data file). Exporting can also be done in a separate thread to enable immediate return when requesting an export from the frontend. *ParadataTimeExporter* exports either client or server sided duration per question. An overview of the classes within the export package can be seen in figure 2.4.

2.2 StruXSLT

Because the Web frontend of QSYS is based on this framework and most of it's concepts are adopted, it is described here before *QSYS-web* is covered.

2.2.1 Basic Functionality

StruXSLT sits on top of *Struts*, extending its existing functionality to allow *Action* classes to return XML that will be transformed by technologies like *XSLT* and *XSL-FO*. One motivation to use *StruXSLT* is to remove the need to use *JSP* and tag libraries for the presentation layer of the *Struts* framework. However, *StruXSLT* does not necessarily force exclusively the XML way, both technologies will work side by side. The basic idea was taken from an article published at `www.javaworld.com`[7], see figure 2.5 to get an idea of this concept. Here no *JSP* or *Taglibs* are used for visualizing data, but *XSLT* style sheets. Within the control layer, an XML document is created instead of storing multiple variables to the session scope. When using XML/XSLT instead of the conventional struts-approach, separation between view and the other layers of MVC-model is stricter. Furthermore, *XSLT* is standardized at the *W3C*[8] and is vendor and

[5]Character Separated Values, as field separator character, TAB is used
[6]http://spss.pmstation.com/
[7][MB02]
[8]`http://www.w3.org/TR/xslt`

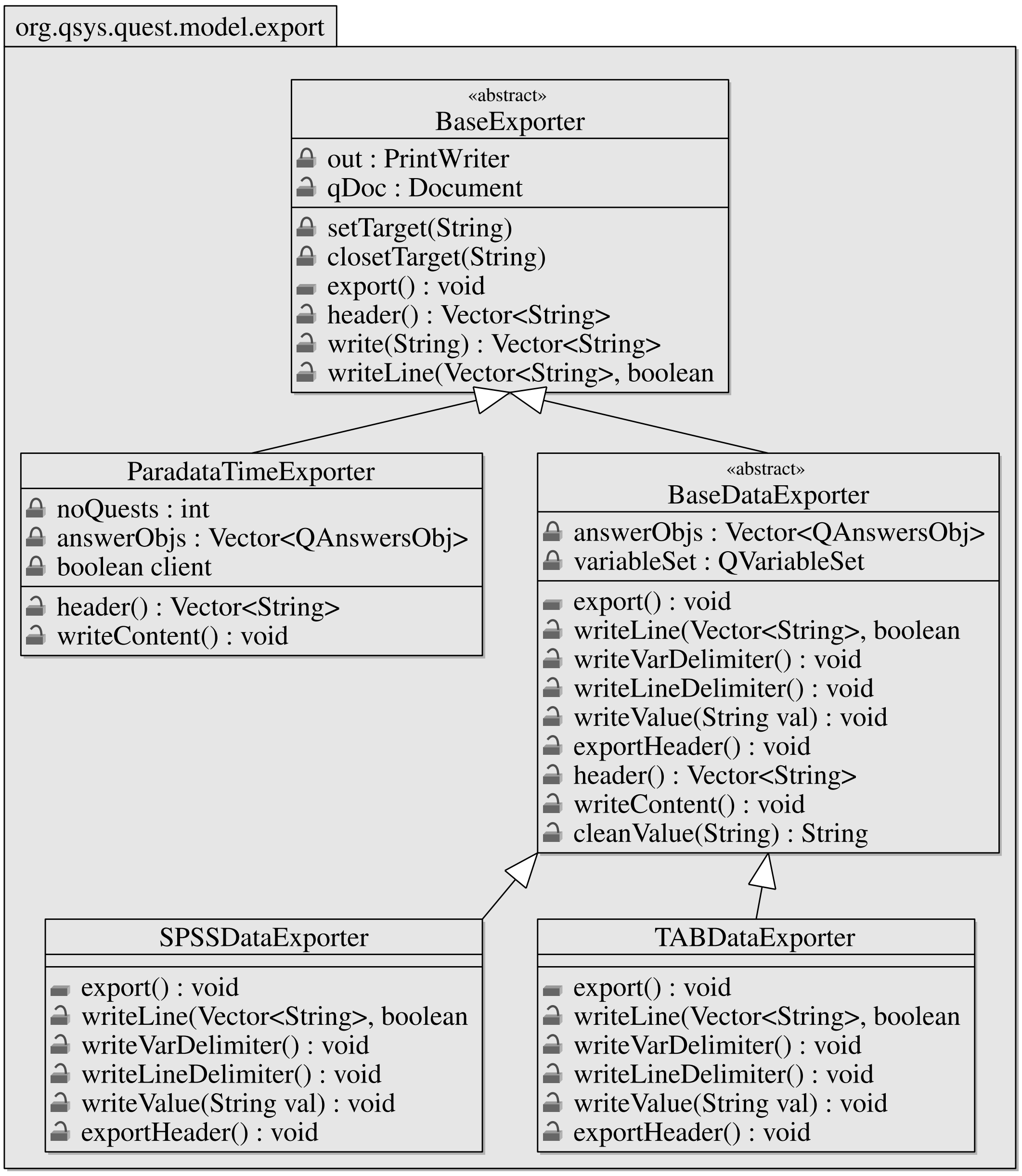

org.qsys.quest.model.export
«abstract»
BaseExporter
out : PrintWriter
qDoc : Document
setTarget(String)
closetTarget(String)
export() : void
header() : Vector<String>
write(String) : Vector<String>
writeLine(Vector<String>, boolean
ParadataTimeExporter
noQuests : int
answerObjs : Vector<QAnswersObj>
boolean client
header() : Vector<String>
writeContent() : void
«abstract»
BaseDataExporter
answerObjs : Vector<QAnswersObj>
variableSet : QVariableSet
export() : void
writeLine(Vector<String>, boolean
writeVarDelimiter() : void
writeLineDelimiter() : void
writeValue(String val) : void
exportHeader() : void
header() : Vector<String>
writeContent() : void
cleanValue(String) : String
SPSSDataExporter
export() : void
writeLine(Vector<String>, boolean
writeVarDelimiter() : void
writeLineDelimiter() : void
writeValue(String val) : void
exportHeader() : void
TABDataExporter
export() : void
writeLine(Vector<String>, boolean
writeVarDelimiter() : void
writeLineDelimiter() : void
writeValue(String val) : void
exportHeader() : void

Figure 2.4: Export class diagram showcase

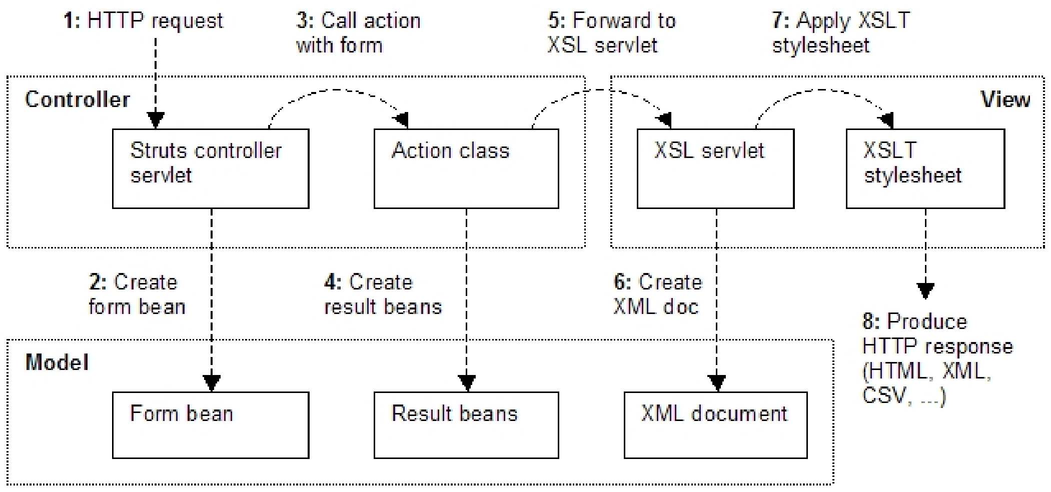


Figure 2.5: MVC-2.x Model as taken from `www.javaworld.com`

technology independent, so all style sheets generated can be reused in other projects, even in those who use a completely different technology (like e.g. .NET). These and other advantages are also mentioned in [KK08].

The framework described here is an essential part of *QSYS* and the concept behind it (even though it has become a standalone framework) and therefore it is necessary to give a brief description of this component here.

Several other solutions within the field of open source exist (like *StrutsCX*[9] and *stxx*[10]), but none of these packages implemented random or conditional assignment of style sheets which was necessary for the experiments of this thesis (how this can be done is described below). For this reason, this new framework has been implemented, which is also employed in other projects independent from *QSYS*. *StruXSLT* is also published as open source and can be downloaded from sourceforge[11]

The basic features of this framework are listed here:

- **Language independency**: all language tokens are set within external XML files, which are woven together with the content of the page to be visualized via *XSLT* afterwards. The settings for linking language files to actions are described in section 2.2.4, where the *XSLT mapper* is described in detail.
- **No JSP necessary**: The main concept of the framework is to simply use *XSLT* for

[9]http://it.cappuccinonet.com/strutscx/doc/v08/en/intro/index.html
[10]http://stxx.sourceforge.net/
[11]`http://struxslt.sourceforge.net`

rendering content. Nevertheless usual *Actions* working with e.g. Struts Taglibs[12] can be used.

- ***AJAX*-support**: data generation and forwarding can be influenced by an action parameter delivered when actions in the view-layer are requested.
- **Support for debugging**: several functions e.g. showing the generated XML which is used for rendering directly within the browser have been implemented.
- **Predefined methods** for preparing information for actions in the view-layer, e.g. *addAdditionalMenueEntries*, where e.g. menu entries, which should be offered on the Web page are listed in the XML to be rendered.
- **Feedback functions**: from every position within the action classes, message and error codes can be set and transferred to the view-layer. Only error and message codes are set, the textual version is automatically read from the language files to assure language independency also for error and warning messages.
- **Central configuration**: one main configuration file exists where all mapping of struts-actions (which do generate XML content in this case), *XSLT* style sheets and language token XML files, is carried out.
- Style sheets can be **assigned by chance**. The probability for selecting a certain style can be set within the main configuration file.
- Style sheets can be **assigned following certain conditions**. A concrete example: it is sometimes useful to assign different style sheets depending on whether Javascript is turned on or off within the client's browser. Another application would be offering different styles for different client types, e.g. to offer an *iPhone* version for a Web page.
- The way *XSLT* style sheets are selected can easily be customized and extended. To do so, simply a new *XsltMapEntry* class has to be implemented (see section 2.2.4 below)
- The framework also clearly differentiates between *Action* classes for processing data and actions for generating content for visualization, which improves the architecture and gives a better overview.

2.2.2 Usage

To use this package, a new base class has to be used instead of Struts' *Action* class, but it is not necessary to exchange the *ActionServlet*, which would be the case with other packages such as *stxx*, so registering Struts within *web.xml* can be done the customary way. It is simply necessary to copy *q-struxsl.jar* (which is the archive containing StruXSLT) to the application's *WEB-INF/lib* folder and extend from the *Action* classes described below.

[12]`http://struts.apache.org/1.x/struts-taglib/index.html`

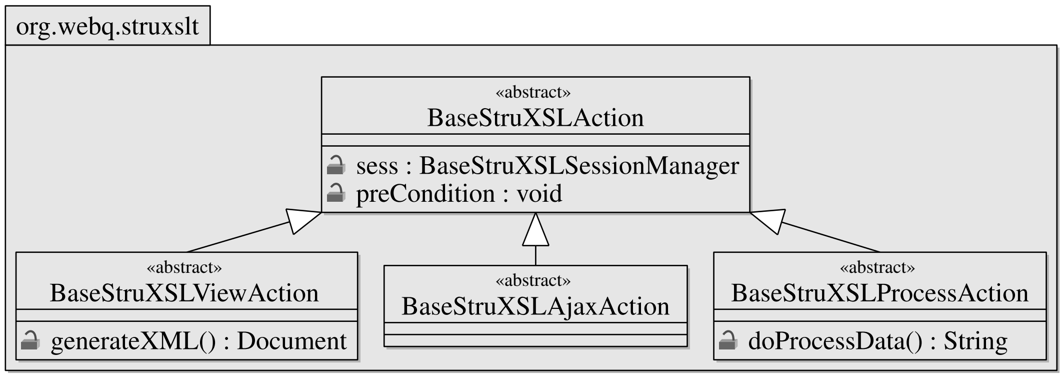


Figure 2.6: Base Action classes of the *StruXSLT*-framework

2.2.3 Action Classes

StruXSLT strictly separates viewactions from process actions, so two main base classes exist within the framework. This diagram should give an overview of the methods to be overridden from the perspective of the framework user.

2.2.3.1 BaseStruXSLAction

This class serves as the base class for the *Action* classes within the whole framework. The main purpose of this common base class is to band together core functionality necessary for all concrete Actions described below. For example, an instance of a session manager is stored as well as the basic settings for the user's session is stored, which are common for all subclasses. In general, messages, errors and other information, which can be either used for communication between the concrete *Action* classes and for the presentation layer (all set messages are also written to the XML used for rendering), can be set from any *Action*. So basic managing (like adding errors and messages) of all these messages is implemented here. All messages to be set are language independent, which means that simply an error or message code is set. This code is afterwards substituted by the definite message text taken from the XML language tokens.

Furthermore, enabling debug mode is implemented here, which is simply a flag that's also written to the view-XML. This can be used to allow the output of additional helpful information during development. Additionally, the retrieval of language tokens and logging also resides here. This class works in the background, when using this framework, the developer is more confronted with the base classes described below.

2.2.3.2 BaseStruXSLViewAction

This class serves as the base class for all *Actions* generating (XML-) data to be used for rendering via *XSLT* or *XSL-FO*. All information specific for a certain *Action* is generated within the *generateXML*-method, so one must implement this abstract method in the project specific view class when overriding *BaseStruXSLViewAction* and the document is returned and can be used for rendering. Within this *Action*, transformation from XML into the desired output format (e.g. *HTML* or *PDF*) according to the stylesheet assigned is done. Style sheet assignment is done within the mapper classes described below, but storing and managing preconditions (e.g. if Javascript is enabled) is handled here.

Language tokens are added to the XML document according to the desired language automatically. If debug modus is enabled, parameter *out* can be used to directly see the XML document used for rendering when adding parameter *out=xml*. The same goes for *xsl* which shows the XSL-document used for rendering. To uniquely add a stylesheet to an application-dependent key (which is used to assign the same stylesheet for the whole session to a certain identifier, the method *getXsltKey* has to be overridden in the concrete subclasses)

2.2.3.3 BaseStruXSLProcessAction

This is the base class for all actions which should process any data (which in general means writing information to the database, file system or to the session). The abstract method *doProcessData* has to be overridden within the concrete subclass to do so. As a return value, we receive an *ActionForward*, where as normally a forward to a concrete *BaseStruXSLViewAction* is received. Within this method, error and warning messages can be generated (which are stored within the session) and used by the view-layer to give feedback to the user as to whether data processing was successful or not.

Two parameters can be set to influence the given output:

- **ajax**: if this parameter is set to true, just errors and feedback messages are delivered to the requesting page. These are used to give the user feedback after performing an *AJAX*-request.
- **tfb**: if this parameter is set to true, a simple text message is delivered to the requesting page. This is used in the case of automated data processing or when certain functions are called from external systems, e.g. in case everything worked fine, the return value is just *OK* in mimetype *text/plain*.

2.2.3.4 BaseStruXSLAjaxAction

This class is integrated for future usage and should handle simple AJAX requests where no response is delivered.

2.2.4 Mapper Classes

The central configuration file for the whole framework is *xslt-map.xml*, where the XML-generating action is woven together with the stylesheet and language token documents to ensure language independency. Place this document directly into your web-application's *WEB-INF* folder. Internally, one *XML* document is generated per request, which is rendered by the *XSLT*-document specified.

The main settings within this document have the following structure:

```
<xsltEntries debug="false">
     <supportedLangs langs="EN, DE" />
     <messageCodes lang_file="global/msgs.xml"/>
     <global_langs>
          <global_lang lang_file="global/menue.xml"/>
          <global_lang lang_file="global/head.xml"/>
     </global_langs>
</xsltEntries>
```

Listing 2.7: An example of a simple XSLT map header

Here with the attribute *debug* the debug mode can be turned on by default. All supported languages can be listed within the second element. Mapping from message codes to the language dependent messages itself is done within element *messageCodes*. Global language token files which are needed by the view-layer can be specified within *global_ lang* elements. These and the message-code documents are automatically copied to the XML used for rendering. In the following, the two currently implemented mapping modes are described. It is easy to write an additional mapping.

simple A simple entry which weaves together an Action, a language token file and an XSLT file can be defined as shown in the following example:

```
<entry path="/basicsettingsv" type="simple"
     lang_file="admin/basicsettings.xml">
     <xslt path="admin/basicsettings.xsl" />
</entry>
```

Listing 2.8: An example of a simple XSLT map entry

Each simple entry-element consists of the following attributes and child-elements:

- **path**: specify the path of the *Struts* action which is responsible for generating XML content used for rendering.

- **lang_file** give a comma separated list of language token files. These are dynamically selected according to the language currently set within the running system.
- also the **mimetype** can be set (which is not shown in the example above because default response mimetype is set to *text/html*), which would make sense in the case of employing *XSL-FO* for *PDF*-generation (which would need a *application/pdf* mimetype). No additional tasks have to be done within the subclassed *Actions*. The correct transforming engine depending on the mimetype being HTML, XML, PDF or plain text, is automatically invoked.
- **xslt** contains an attribute **path**, where the relative path to the *XSLT*-stylesheet is specified.

Such an entry is necessary for all views of the Web application. These entries have to be placed directly as child elements of the root element *xsltEntries*.

random Another more complex mapping type called *random* can be configured in the following way:

```
<entry path="/doqv" type="random" group="univ_uibk" questionnaire="webpage"
     lang_file="do/do_main.xml">
     <xslt path="/to/any.xsl" name="personal_1" prob="34"
          conditions="java, javascript"/>
     <xslt path="/to/any.xsl" name="personal_2" prob="33" conditions="javascript"/>
     <xslt path="/to/any.xsl" name="personal_3" prob="33"/>
</entry>
```

Listing 2.9: An example of a random XSLT map entry

These settings have the following meanings (those who are equal to the *simple*-example are not explained here):

- **keys**: for the attributes *group* and *questionnaire*, concrete groups and questionnaires can be set and only if these two values are equal to those when requesting the map entry, will this map entry be selected. This allows the distinction between different groups and questionnaires and assigns different style sheets to each without any necessary code manipulation. Consequently, it is possible that two entries ore more entries with the same path exist in the configuration file. The one to be chosen for rendering is selected via these two additional attributes. Of course, the terms *group* and *questionnaire* are just examples (it was used that way within QSYS when running surveys with different styling in parallel) and can be specified within the application. In fact, these two attributes could also be named *key1* and *key2*.
- **prob**: define the likelihood for a certain style to be selected. The sum of all chances must be 100.
- **conditions**: some preconditions can be set for a style to be selected (e.g. if Javascript and/or Java is enabled or not). Again, the naming for these conditions is not fixed and can be freely chosen for each application. If a certain *xslt-entry*

was selected, but conditions are not satisfied, random selection is repeated until an XSLT-entry is selected where all conditions are fulfilled (therefore it is necessary to pay attention to these conditions, because no measures have been made yet to avoid infinite loops).

- **name**: the name of the selected style can be used for branching within an *XSLT*-document. In contrast to the example above, the *path*-attribute values of course can differ.

2.2.5 Language Independence

All language token files have to be located within the following path (the root for all language files is *WEB-INF/lang*):
<language_ abbreviation>/<path_ to_ lang_file>, e.g. *EN/admin/login.xml.*

A normal language token file has the following structure: because these language documents are directly copied to the XML document used for rendering, also hierarchies of elements or attributes can be set and used within the style sheets, one simply adds all child-elements of the root-element <*lang*>. E.g. within the *QSYS* project, language key is the element name and language value is stored within the text node.

Introducing a new language could easily be done when following these steps (currently the support is limited on languages using *ISO-latin* character set, but this will be changed in future releases):

1. Add an abbreviation for the language in *xslt-map.xml* at *supportedlangs/langs*, e.g. IT for Italian.
2. Copy all language tokens from an existing language (e.g. EN) to the new language folder and translate the text nodes.

2.3 QSYS-Web

QSYS-web is the Web frontend for the *QSYS* system. It is based on the *StruXSLT* framework and uses the functionality of *QSYS-core*, so sections 2.1 and 2.2 must be read before attention to this section can be given. For rendering XML content to other output formats, two open source projects from the Apache group are used, namely *Xalan*[13] for generating *HTML* and *FOP*[14] for PDF output. Both work very well, style sheets simply need to be written and the desired output is generated. The *FOP* developers plan to

[13]Xalan-Java is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. It implements XSL Transformations (*XSLT*) Version 1.0 and XML Path Language (*XPath*) Version 1.0 and can be used from the command line, in an Applet or a Servlet, or as a module in another program [http://xml.apache.org/xalan-j/]

[14]Apache FOP (Formatting Objects Processor) is a print formatter driven by *XSL* formatting objects (XSL-FO) and an output independent formatter (`http://xmlgraphics.apache.org/fop/`

improve the *RTF*, which would be used as an additional output format for questionnaires withi QSYS. The advantage of *RTF* over *PDF* is that it can be modified after generation by the user using e.g. Open Office or Microsoft Word.

2.3.1 Class Hierarchy

The concept of separating between *Actions* responsible for generating content for the view-layer on the one hand and on the other hand *Actions* for processing requests as predetermined by *StruXSLT* will be retained and extended with another differentiation, namely between public accessible *Actions* and those only accessible by the group or system administrators. Because core functionality is already implemented within *StruXSLT*, it is sufficient to concentrate on project specific classes and methods without any distraction from basic and workflow functionality.

In the following the main structure of these *Action* classes should be shown exemplarily together with some concrete derivations.

2.3.1.1 View

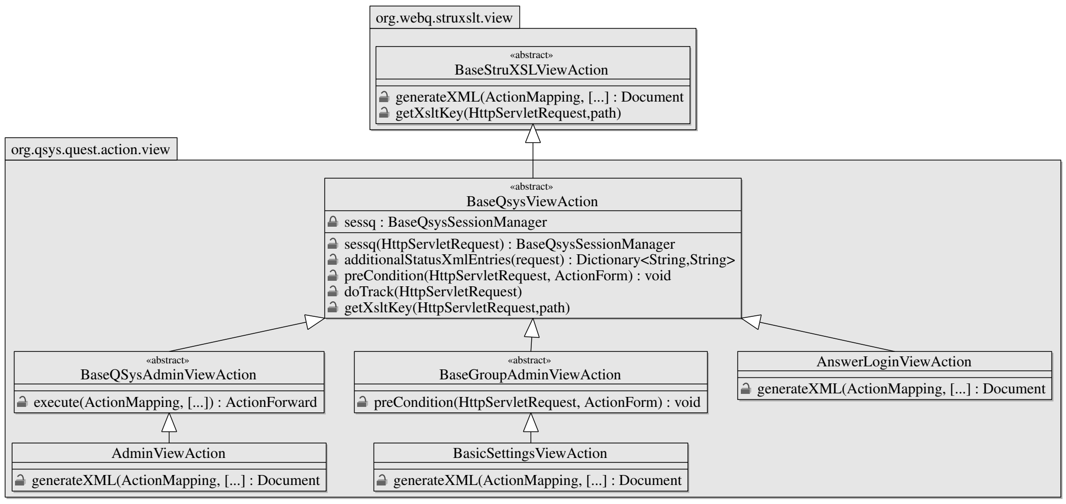

org.webq.struxslt.view
«abstract»
BaseStruXSLViewAction
generateXML(ActionMapping, [...] : Document
getXsltKey(HttpServletRequest,path)
org.qsys.quest.action.view
«abstract»
BaseQsysViewAction
sessq : BaseQsysSessionManager
sessq(HttpServletRequest) : BaseQsysSessionManager
additionalStatusXmlEntries(request) : Dictionary<String,String>
preCondition(HttpServletRequest, ActionForm) : void
doTrack(HttpServletRequest)
getXsltKey(HttpServletRequest,path)
«abstract»
BaseQSysAdminViewAction
execute(ActionMapping, [...]) : ActionForward
«abstract»
BaseGroupAdminViewAction
preCondition(HttpServletRequest, ActionForm) : void
AnswerLoginViewAction
generateXML(ActionMapping, [...] : Document
AdminViewAction
generateXML(ActionMapping, [...] : Document
BasicSettingsViewAction
generateXML(ActionMapping, [...] : Document

Figure 2.7: *QSYS*-web view class diagram showcase

In figure 2.7, a sample showcase is described for the view-classes hierarchy within the *QSYS*-web component. *BaseQsysViewAction* serves as a base class for all view-actions used within *QSYS*. Here *getXsltKey* generates a key consisting of the group-id, the questionnaire-id and the interviewee-id to assure proper assigning of the style sheets. Within the overridden method *precondition*, necessary environment variables are set.

doTrack is responsible for tracking user paradata (gathered either from client the or from the server side).

2.3.1.2 Process

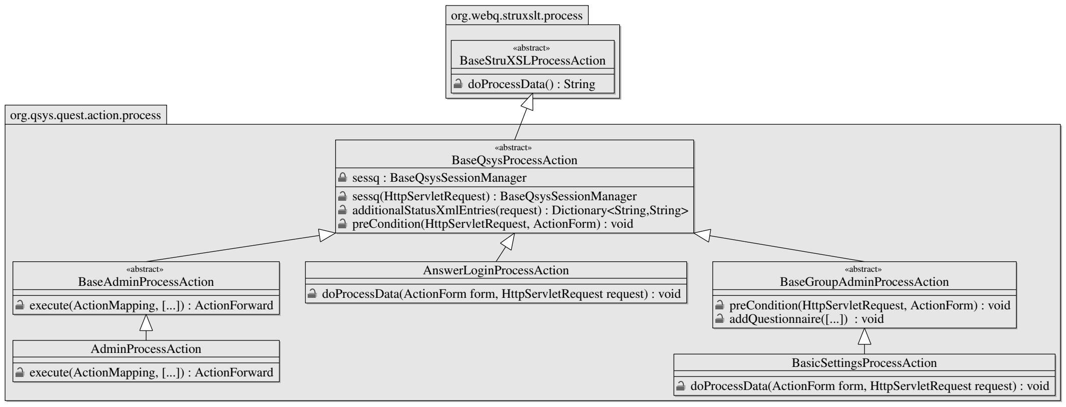

org.webq.struxslt.process
«abstract»
BaseStruXSLProcessAction
doProcessData() : String
org.qsys.quest.action.process
«abstract»
BaseQsysProcessAction
sessq : BaseQsysSessionManager
sessq(HttpServletRequest) : BaseQsysSessionManager
additionalStatusXmlEntries(request) : Dictionary<String,String>
preCondition(HttpServletRequest, ActionForm) : void
«abstract»
BaseAdminProcessAction
execute(ActionMapping, [...]) : ActionForward
AdminProcessAction
execute(ActionMapping, [...]) : ActionForward
AnswerLoginProcessAction
doProcessData(ActionForm form, HttpServletRequest request) : void
«abstract»
BaseGroupAdminProcessAction
preCondition(HttpServletRequest, ActionForm) : void
addQuestionnaire([...]) : void
BasicSettingsProcessAction
doProcessData(ActionForm form, HttpServletRequest request) : void

Figure 2.8: *QSYS*-web Process class diagram showcase

Figure 2.8 shows a sample showcase for the process-classes hierarchy. *BaseQsysProcess-Action* serves as a base class for all process-actions.

2.3.2 Configuration and Installation

2.3.2.1 Preconditions

To run *QSYS* on your server or local machine, a Servlet engine like Tomcat[15] (the software was tested with version 6) with Java Runtime Environment (version >= 1.5) is sufficient, which is pre-installed in most cases. Not even a database is necessary when running the *file system storage-mode*. Because of this, it is also possible to install *QSYS* locally (e.g. on the laptop) so QSYS can also be used offline (questionnaires can then easily be imported into the online-system). To install, simply download the *.war* (web archive)-document from the project's homepage and deploy (which means either copying the archive to tomcat's *webapps* directory or deploying via tomcat manager)

2.3.2.2 Property Files

Two files have to be created and located on the class path:

qsys.properties, which stores general settings used within the whole system.

[15]http://tomcat.apache.org/

```
DB=FILESYSTEM
MAIL_ADDRESS=
SMTP_SERVER=
DATA_EXPORT_DIR=[data_export_dir] #change
ADMIN_PW=[admin_pw] #change
LDAP_URL=[path_to_ldap] #optional
LDAP_GROUP_PREFIX=[ldap_group_prefix] #optional
```

Listing 2.10: Settings within qsys.properties

fsdb.properties, which defines the location where all information is stored when storing on the file system is the selected storage method.

```
PATH=[path_to_root_storage_dir] #change
```

Listing 2.11: Settings within fsdb.properties

2.3.2.3 Compile from Scratch

To compile the sources, currently a shell script (for Linux and Mac-users; *qsys-web/build/build.sh*) exists which completely builds all projects necessary to create the whole web-archive file. This script calls *ANT* scripts and copies created libraries to the *WEB-INF/lib* directory. Each project contains a build-folder, where the appropriate *ANT*-script (which is always named *build.xml*) is located. The resulting *qsys.war* file will be located according to the webapp-property as set within *qsys-web/build/build.xml.*

2.3.3 Additional Tools

Several tools have been used to ease the development process, here only the most important ones are described:

- **ANT**[16] (which is an acronym for *Another Neat Tool*) is a Java-based build tool. In the current version, many extensions are available enabling the use of ANT for a lot of other tasks (e.g. integrating *JUnit*-tests within the build process). Within the *QSYS*-project, *ANT* is used for automatically building all subprojects, generating a Web archive for deploying on the Servlet engine, generating *JavaDoc* and running *XDoclet*-statements.
- **XDoclet** [17]: The generation of the *struts-config.xml* is done via *XDoclet*-tasks, which has several advantages: it is error-prone to edit *struts-config.xml* directly, because for bigger projects this file becomes very complex and hard to view as

[16]`http://ant.apache.org`

[17]*XDoclet* is an open source code generation engine. It enables Attribute-Oriented Programming for java. In short, this means that you can add more significance to your code by adding meta data (attributes) to your Java sources. This is done in special JavaDoc tags. `http://xdoclet.sourceforge.net/xdoclet/index.html`

a whole. Additionally, it is beneficial when the whole configuration for *Struts* is directly written next to the concerned classes. See listing 2.12 for a sample of an XDoclet statement for a *Struts Action* (here for the login process action). The instructions are directly placed over the class declaration integrated into a *JavaDoc* comment. This information directly goes to the *struts-config.xml* when running the corresponding *ANT* task.

In line 2 of the sample, an *ActionForm*-class and a Web path is assigned to the *Action*-class. In the subsequent lines, the action forwards are defined with a name and a path where the forward should go to. If redirect is set to true, the URL within the navigation bar of the browser changes to this new URL.

- **Eclipse** with *Lomboz*-plugins: As IDE[18], Eclipse was used with certain plug-ins (the most essential ones are already bundled within the Lomboz distribution). To run the Tomcat Servlet engine within Eclipse, a special plugin was used[19] which even enables debugging of Servlets.

```
1 /**
2  * @struts.action name="answerloginform" path="/answerlogin" scope="request"
3  * @struts.action-forward name="success" path="/answerstartv.qsys" redirect="true"
4  * @struts.action-forward name="toquestion" path="/doqv.qsys" redirect="true"
5  * @struts.action-forward name="interviewee_exists" path="/intervieweeexists.qsys"
6          redirect="true"
7  * @struts.action-forward name="error" path="/answerloginv.qsys" redirect="true"
8  */
9 public class AnswerLoginProcessAction extends BaseQsysProcessAction { ... }
```

Listing 2.12: An example of XDoclet metadata attributes (for the login-process)

2.4 Utility Classes

The utility classes package is called *q-utils*. It simply contains some classes with handy functions which are in constant requisition. For example, classes are provided to ease XML processing as well as accessing databases efficiently, working with generics and reflection or connecting to an *LDAP* server. This should just serve as a toolkit for all Java projects to ensure reusability of common functionality. This has several benefits: The functionality grows with every new project, and stability of these functions increases because they are heavily used, so errors or strange behavior is revealed very early. Unit tests are implemented (using *JUnit*[20]) for most of these utility classes to protect against side effects with regression testing after modification.

Of course also for this package source code is available when downloading the actual *QSYS* or *struXSLT* release, but making an extra release for these classes is not worth

[18]Integrated Development Environment
[19]taken from `http://www.eclipsetotale.com/tomcatPlugin.html`
[20]`http://www.junit.org/`

the effort. There are better solutions like e.g. Jakarta commons[21].

2.5 Additional Notes

2.5.1 Quality Control

To assure software quality is 1as good as possible, testing of the software became a major point during development. As tool for unit and regression testing, JUnit was used.

2.5.2 Software Metrics

Here just a few figures to describe the complexity of the generated software: The number of classes for *QSYS-core* is 152 (12.514 LOC), *QSYS-web* consists of 98 (4834 LOC + all XSLT-documents), StruXSLT of 20 (1581 LOC) and the utility package of 86 classes (5785 LOC)[22]

2.5.3 Schema RFC for Questionnaires (and Answer Documents)

A schema to describe XML documents which hold questionnaires and answers has been implemented within the scope of the thesis and will be enrolled at the *W3C* in the form of a request for comments (RFC). Standardization would be one big step forward to enable exchangeability of questionnaire definitions between surveys. Furthermore, archiving of questionnaires and answer documents would be simplified and generalized. The goal is not to create the standard but more to put it up for discussion so possible further steps are set from other survey software developers. The schema definitions can be found at `http://www.survey4all.org/qsys-xsd`.

[21]`http://commons.apache.org/`

[22]lines of code have been determined with `http://www.dwheeler.com/sloccount/`

3 Additional Tasks to be Implemented

3.0.4 Federated Identity Based Authentication and Authorization

A first step towards integrating QSYS into the infrastructure of companies and institutions is given through the support of LDAP. This support for example enables the limitation of respondents for a certain questionnaire to members of one or more LDAP groups. Further steps in this direction could be the integration of or offering interface to technologies like *shibboleth*[1] and *OpenId*[2] to benefit e.g. from *single-sign-on*. Additionally, it is planned to offer most of the functionality of *QSYS* via Web services.

3.0.5 R Reporting Server

Most commercial and open source online survey tools contain basic (or in some cases advanced) reporting functionality. The strategy for *QSYS* is that reporting should be implemented outside of the system, the software itself should concentrate on it's core competence and further jobs should be excluded (the same is true for e.g. *panel management tasks*) but communication with these external components should be done over clearly defined interfaces. One idea was to use the R environment for statistical computing[3] and it's capabilities for reporting. R has several advantages compared to usual reporting packages:

- The possibilities of statistical computing with R are enormous (also for generating certain graphics like charts). Of course, in the first version, simple descriptive statistics like frequency distributions and graphics like histograms will be part of the report, but when generating the infrastructure for using R as a reporting tool, improvement and extending the reports can be done with less effort
- R supports several output formats, like PDF, HTML, RTF and even Latex.
- R contains a fully object oriented language, which enables well structured development. Even wrappers exist for all common programming languages.
- R is also open source, which fits well into the license strategy *QSYS* has.
- R has a big community behind it, which means masses of packages and functionality from different areas exist. Furthermore, a very active mailing list exists, where

[1]`http://shibboleth.internet2.edu/`

[2]`http://openid.net/`

[3]More information about the huge functionality of R can be found at `http://www.r-project.org/` and [R D06]

support on concrete questions is given.

Because R was also used for statistical analysis of the experiments, some code fragments generated for this purpose can be reused, which is also a positive side effect. A lot of literature has been published on different levels of R usage, e.g.: introduction to R: [Lig07], [EH07]; graphics with R [Mur06]; linear modelling with R [Far05], [Woo06]

3.0.6 Observation Features

Because of the introduction of Web 2.0 technologies like e.g. *AJAX*, observation of the user is not limited to information gathered when the submit button is pressed, but it is also possible to track user behavior DURING filling out (which means *what is done on the webpage*). Concrete scientific questions would be: which text has been written in the text box before pressing the submit button (possibly the first statement written was deleted and substituted by a statement which fits more to the social desirability for openended questions; the same is applicable for closedended questions: did the respondent select another alternative than the one which was selected when submitting the form). Concrete experiments could be made in the future for which these features will be implemented.

3.0.7 Accessibility

More effort will be invested in designing barrier-free questionnaires. The goal is to support all *should*-criteria (*AA*) from `http://www.w3.org/TR/WCAG20/`.

4 Evaluation of the Software

In this chapter, a short evaluation of *QSYS* with external criteria is given. E.g. [Kok07] already evaluated eight commercial software solutions, criteria are taken from this article and applied to *QSYS*. Additionally, some ideas for evaluating survey software especially for academics have been taken from [PR02]. [Bat03, p.10] identifies the following main requirements for Web based survey tools (evaluation concerning *QSYS* is directly given next to these points):

- Progress bar (graphical or simply the current and number of pages of the questionnaire). *Yes.*
- Question filters used for branching. *Yes, but actually branching can only be done depending on the direct antecessor question.*
- Randomized assignment of respondents to different conditions. *Yes, this feature is extensively supported, all possible variations are possible (but HTML knowledge is necessary for implementation, because XSLT documents must be created).*
- Item rotation to avoid ranking effects. *No, this will be done in the next release.*
- Plausibility checks to automatically identify data inconsistency (ideally during entering data). *Yes, both, on client and on server side.*
- Possibility to send invitation letters and reminders. *No, these tasks should be kept outside, a tool is planned which has these capabilities and communicates with QSYS.*
- Access limitation (e.g. the usage of passwords). *Yes, there are several different access modes.*
- (Real time) report statistics. *No, again this feature should be implemented in an additional tool communicating with QSYS (see section 3.0.5 for further details).*
- Integration of multimedia (pictures, films,...) possible. *Yes, everything which can be done with HTML is possible.*
- Data export in common statistic-software (like SPSS, SAS, etc.). *Yes, CSV, which can be imported in all common statistics packages, is supported*
- Multiple project managers can create questionnaires concurrently. *Yes*
- Secure data transmission (SSL encryption). *This depends on the server, but it is no problem to run QSYS e.g. on Tomcat using https.*

In addition, [?, p.281f] give a list of features a professional survey software package should support (again, QSYS evaluation is given next to the criteria):

- Sample management allowing the researcher to send out prenotifications, initial invitations and follow ups for nonrespondents. *Here again, this functionality should (possibly in combination with a (open source) panel management tool, one candidate could be phpPanelAdmin[1]) be implemented externally with interfaces to QSYS.*
- User-friendly interface for questionnaire design, with several features. For instance, manuals, online help, tutorials, but also question/questionnaire libraries, and export from other software packages. *Yes, everything mentioned here exists within QSYS (e.g. a detailed documentation of the editor exists in both English and German)*
- Flexible questionnaire design regarding layout (e.g. background color/pattern, fonts, multimedia, progress indicator), question forms (e.g. open, close, grid, semantic differential, yes/no questions), and features of computer-assisted survey information collection (e.g. complex branching, variable stimuli based on previous respondent's selections, range controls, missing data and consistency checks. *As already mentioned for [Bat03, p.10] criteria, all these points are implemented, except complex branching and related features.*
- Reliable and secure transfer and storage of data. *Storage of data has been approved with several surveys run with QSYS, secure data transfer can be assured with https, if the Servlet container is configured this way.*

Here a few additional criteria are given:

- User, developer and administrator documentation: *documentation exists for all of these three roles.*
- Printable survey version: *it is possible to export the survey as PDF and print out this document. The way the questionnaire should look is fully customizable in editing the appropriate XSL-FO document. It is also possible to assign different style sheets generating PDF for different questionnaires.*
- Sufficient offer of question types: *as can be seen in section 1.3*
- Software should not only be suitable for market and social research, but also for other scientific fields. [BB05] discusses special needs for epidemiological research, where e.g. the need of supporting Visual Analogue Scales is mentioned. *VAS are fully supported in different graphical representations.*

[1] `http://www.goeritz.net/panelware/`

Bibliography

[Bat03] BATINIC, Bernad: Internetbasierte Befragungsverfahren. In: *Österreichische Zeitschrift für Soziologie* (2003), Nr. 4, S. 6–18

[BB05] BÄLTER, Olle ; BÄLTER, Katarina A.: Demands on Web survey tools for epidemiological research. In: *European Journal of Epidemiology* 20 (2005), S. 137–139

[Bir00] BIRNBAUM, Michael H.: SurveyWiz and FactorWiz: JavaScript Web pages that make HTML forms for research on the Internet. In: *Behavior Research Methods, Instruments, & Computers* 32 (2000), Nr. 2, S. 339–346

[EH07] EVERIT, Brian S. ; HOTHORN, Torsten: *A Handbook fo Statistical Analyses using R*. Boka Raton : Chapman & Hall/CRC, 2007

[Far05] FARAWAY, Julian J.: *Linear Models with R*. Boca Raton, Florida : Chapman & Hall/CRC texts in statistical science series, 2005

[GB05] GÖRITZ, Anja S. ; BIRNBAUM, Michael H.: Generic HTML Form Processor: A versatile PHP script to save Web-collected data into a MySQL database. In: *Behavior Research Methods* 37 (2005), Nr. 4, S. 703–710

[HB07] HOLTGREWE, Ursula ; BRAND, Andreas: Die Projektpolis bei der Arbeit. Open-Source Softwareentwicklung und der "neue Geist des Kapitalismus". In: *österreichische Zeitschrift für Soziologie* (2007), Nr. 3, S. 25–45

[KK08] KARG, Markus ; KREBS, Sebastian: Der saubere Weg. Herstellerunabhängiges Reporting mit XSL und Co. In: *iX. Magazin für professionelle Informationstechnik* (2008), Nr. 6, S. 106–109

[Kok07] KOKEMÜLLER, Jochen: Online-Umfragen: Acht Softwarelösungen. In: *iX. Magazin für professionelle Informationstechnik* (2007), Nr. 11, S. 110–115

[Lig07] LIGGES, Uwe: *Programmieren mit R*. 2. Auflage. Berlin, Heidelberg, New York : Springer Verlag, 2007

[MB02] MERCAY, Julien ; BOUZEID, Gilbert: *Boost Struts with XSLT and XML*. `http://www.javaworld.com/javaworld/jw-02-2002/jw-0201-strutsxslt.html`. Version: 2002. – [Online; accessed 10-September-2008]

[McC03] MCCALLA, Rachel A.: Getting results from online surveys - Reflections on a personal journey. In: *Electronic Journal of Business Research Methods* 2 (2003), Nr. 1, S. 55–62

[Mur06] MURELL, Paul: *R Graphics*. Boka Raton : Chapman & Hall/CRC, 2006

[PR02] POCKNEE, Catherine ; ROBBIE, Diane: *Surveyor: A Case Study of a Web-Based Survey Tool for Academics.* Paper presentation at the ASCILITE 2002, December 8-11, 2002; Auckland, New Zealand, December 2002

[R D06] R DEVELOPMENT CORE TEAM ; R FOUNDATION FOR STATISTICAL COMPUTING (Hrsg.): *R: A Language and Environment for Statistical Computing.* Vienna, Austria: R Foundation for Statistical Computing, 2006. `http://www.R-project.org`. – ISBN 3-900051-07-0

[RN02] REIPS, Ulf-Dietrich ; NEUHAUS, Christoph: WEXTOR: A Web-based Tool for Generating and Visualizing Experimental Designs and Procedures. In: *Behavior Research Methods, Instruments, & Computers* 34 (2002), Nr. 2, S. 234–240

[RS04] REIPS, Ulf-Dietrich ; STIEGER, Stefan: Scientific LogAnalyzer: A Web-based Tool for Analyses of Server Log Files in Psychological Research. In: *Behavior Research Methods, Instruments, & Computers* 36 (2004), Nr. 2, S. 304–311

[St.04] ST.LAURENT, Andrew M.: *Unterstanding Open Source and Free Software Licensing.* Gravenstein Highway North, Sebastopol : O Reilly, 2004

[Tru03] TRUELL, Allen D.: Use of Internet Tools for Survey Research. In: *Information Technology, Learning, and Performance Journal* 21 (2003), Nr. 1, S. 31–37

[Woo06] WOOD, Simon N.: *Generalized Additive Models. An Introduction with R.* Boca Raton, Florida : Chapman & Hall/CRC texts in statistical science series, 2006

List of Figures

List of Tables

Listings