Form Tools

Developer's Guide

Jan Bulánek
Zbyněk Falt
Lukáš Ježek
Jaroslav Kezniki

Form Tools: Developer's Guide

by Jan Bulánek, Zbyněk Falt, Lukáš Ježek, and Jaroslav Kezniki

Table of Contents

1. Overview 1
    1.1. Application overview 1
    1.2. Modules 1
    1.3. Coding style 2
2. Lib module 3
    2.1. Overall architecture 3
    2.2. Form document 3
        2.2.1. Form item 4
        2.2.2. Form property container 5
        2.2.3. Form document 5
        2.2.4. Form control 6
        2.2.5. Form control group 6
        2.2.6. Form control container 6
        2.2.7. Form page 6
        2.2.8. Form area 7
        2.2.9. Input control 7
        2.2.10. Input control container 8
        2.2.11. Saving and loading the document 8
    2.3. Program manipulation 9
        2.3.1. Program generation 10
        2.3.2. Program evaluation 11
        2.3.3. Program manipulation 13
    2.4. Printing 14
        2.4.1. Form printer 14
        2.4.2. Area printer 14
        2.4.3. Area printer item 14
        2.4.4. Text printer 14
    2.5. Exporting the document 15
        2.5.1. HTML export 15
        2.5.2. PHP export 15
        2.5.3. Export for Filler 15
    2.6. Image acquisition 15
        2.6.1. Image sources 16
3. Script module 17
4. OCR module 18
    4.1. Overall architecture 18
    4.2. Image denoising 18
    4.3. Edge detection 18
    4.4. Image aligning 19
        4.4.1. Aligning images from the scanner 19
        4.4.2. Aligning images from photos 19
    4.5. Classes for areas recognition 20
        4.5.1. Rectangle detection 20
        4.5.2. Dotted lines detection 20
        4.5.3. Detection of types and groups of detected areas 21
    4.6. Gui 21
    4.7. Performance 21
5. GuiCommon module 22
    5.1. API 22
    5.2. Form holder 22
        5.2.1. Table layout form holder 23
    5.3. Database input helper classes 23
    5.4. Flow layout in a scroll area 23
    5.5. Snapping to lines 24
6. Form Editor 25

6.1. Overall architecture 25
6.2. Data models 25
6.2.1. Basic overview 26
6.2.2. Control model 26
6.2.3. Input models 26
6.3. Paper form editor 27
6.3.1. Paper graphics view overview 27
6.3.2. Form area item 28
6.3.3. Paper graphics view 28
6.3.4. Synchronizing selection 29
6.4. Electronic form and datastore editors 29
6.4.1. Electronic form designer 29
6.5. Toolbar action management 31
6.6. Error handling 31
6.7. Property handling 32
6.7.1. Properties model 32
6.7.2. Property view and item delegate 33
6.7.3. Custom property models and views 33
6.8. Script Editor 34
6.9. Script Debugger 34
7. Filler 36
7.1. Overall architecture 36
7.2. Data storage 36
7.2.1. CSV data adapter 36
7.2.2. Database data adapter 37
7.3. Data evaluation and conversion 37
7.3.1. Data flag map 37
7.3.2. Data validator 37
7.3.3. Input convertor holder 38
7.4. Data model 38
7.4.1. Overview 38
7.4.2. Data conversion 38
7.4.3. Data validation and evaluation 38
7.4.4. Copy and paste 39
7.5. Data view classes 40
7.5.1. Form view 40
7.5.2. Table delegate 40
8. Final notes 41
8.1. Comparison to other tools 41
8.2. Development process 41
8.3. Development distribution 42
8.4. Remarks to the chosen solutions 43
8.4.1. Development tools 43
8.4.2. Architecture decisions 43
8.5. Future work 43
9. Contacts 44
Used software components 45

List of Figures

1.1. Overall architecture of the Form Tools project 1
2.1. The structure of the form document 4
2.2. The inheritance hierarchy of form items 4
2.3. Program generation and evaluation 9
8.1. Number of lines during the development 42

Chapter 1. Overview

1.1. Application overview

The Form Tools project tries to ease the process of digital form creation, its distribution to the user, collecting the filled in data, managing and printing them back to the paper form. A form consist of three parts, its layout on a paper, any number of electronic form definitions (inputs) and any number of storage definitions (CSV or DB).

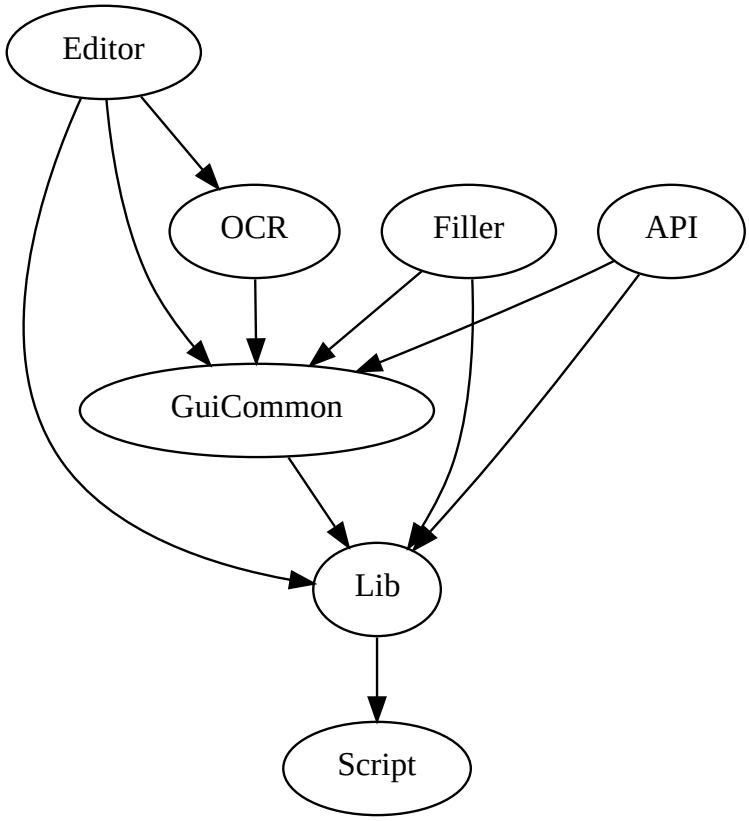
We created a powerful and easy-to-use form designer with support for paper form digitalization (there is an area detector which helps the user fit the areas to the correct positions, calibrate the image accordingly). Then both the storage and input form definitions can be created automatically based on the paper definition. There is a scripting support for the form controls which enables quick and easy creation of dynamic forms.

The created form can then be exported for filling in via web or in a dedicated application from the package, Form Filler. Form Filler can work with many data sources at once, can be used for managing the collected data and offers previewing and (batch) printing of the selected data to a preprinted paper form.

1.2. Modules

The whole project is composed of several modules. Each module provides specific services and interface for using these services. The overall architecture of the modules is depicted on the figure Figure 1.1, “Overall architecture of the Form Tools project”.

Figure 1.1. Overall architecture of the Form Tools project



The most important module is the Lib module. This module provides classes for manipulating with the form document, evaluation of the control programs, printing the filled form, exporting the document to the various formats and for the image acquisition.

The Script module is helper module which provides API for manipulation with the source codes of *ECMAScript*. It is not integrated with the Lib module, because it contains modified source codes of the Qt library.

The GuiCommon module contains common graphic user interfaces, which are used by other modules. For example a dialog for printing is shared by other modules. This module also provides base classes for views and some tools for setting different kind of behaviour.

The OCR module is responsible for area detection and image processing. The name of the module is not accurate, because in the fact it does not perform optical character recognition. It recognizes only areas and dotted lines in the scanned image. It is also responsible for some simple image settings and detecting areas type. The name remains because of historical reasons.

The Editor is the application in which the document can be designed. It provides a user interface for complete creation of the electronic form — the user can scan the image, recognize areas and transform the detected areas into the form controls. The properties and behavior of these controls can be also edited in Editor. Finally the user can define the various inputs of the data for the form.

The Filler is part of the project, which is responsible for filling the documents, which were designed in the Editor. It also provides the functionality for saving and loading the filled data.

The API module provides public interface for the mentioned modules, so any developer can use the designed documents and the electronic form in his own application. The interface also provides methods for printing the document.

1.3. Coding style

If you want to understand source code of the project well, it is good to be familiar with the used naming and formatting conventions in the sources. Coding style of the source codes is influenced by the coding style of the Qt library[QtLib]. The main reason for this decision is that the source codes are homogenous and much more readable. The main rules are following:

- In project there are used medial caps for compound identifiers. Names of all types and classes begin with the upper case letter and all other identifiers begin with the lower case letter.
- Classes have no public member variables. These variables can be accessed through the setters or getters. All members variables has a name like this `m_nameOfVariable`.
- Getters always have a name like `nameOfVariable` and setters always have a name like `setNameOfVariable`
- For public enumeration types we use our own way of their declaration, because it provides name encapsulation and more comfort. More details can be found in the file `Lib/enum.h`.

Chapter 2. Lib module

2.1. Overall architecture

The Lib module is a core module of the FormTools project. It provides almost all the non-GUI functionalities except the optical object recognizing. The main services which this module provides are:

- Manipulation with the form document, realtime checking of validities and availabilities of its properties and its items, saving the document to a file and loading the instance of the document from a file.
- Manipulation with the programs of form items, program transformations, generation and evaluation. The module also converts data between various data types.
- Printing and drawing the content of the document.
- Exporting the form document into various formats.
- Image acquisition from a scanner.

All classes in this module are in the namespace `FTLib`.

2.2. Form document

The form document is quite complex structure of many different classes. Because manipulation with the document should be easy and flexible, there is an inheritance structure among the classes. So a client of the module has an uniform way how to work with the structure of the document and uniform way how to work with properties of each item in the document. The hierarchy of the objects in the document corresponds to the GUI, so there is a system of form control groups and form controls. Each form control has list of paper areas. There is also a list of all pages in the form which contains detected areas and paper areas of the form controls, a list of input container contains all input controls etc. The structure of one document is depicted in the Figure 2.1, “The structure of the form document” and the the inheritance hierarchy is shown in the figure Figure 2.2, “The inheritance hierarchy of form items”. The detailed description of all classes is given in the rest of this section.

Figure 2.1. The structure of the form document

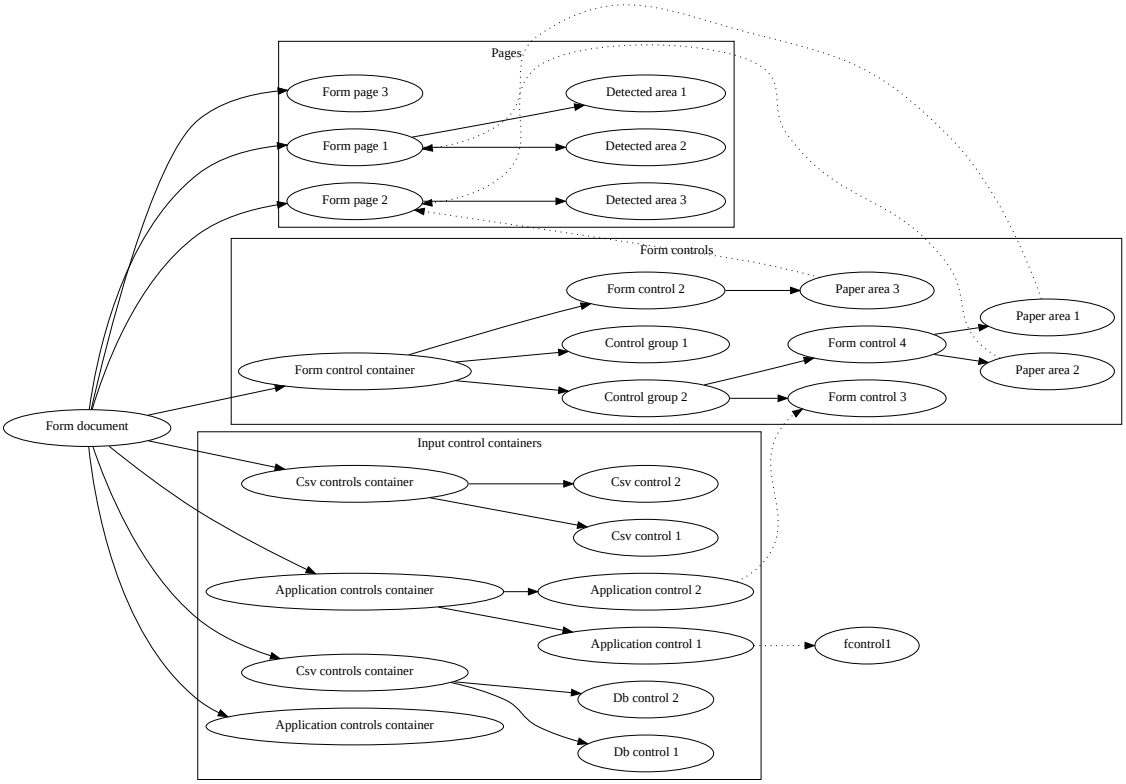
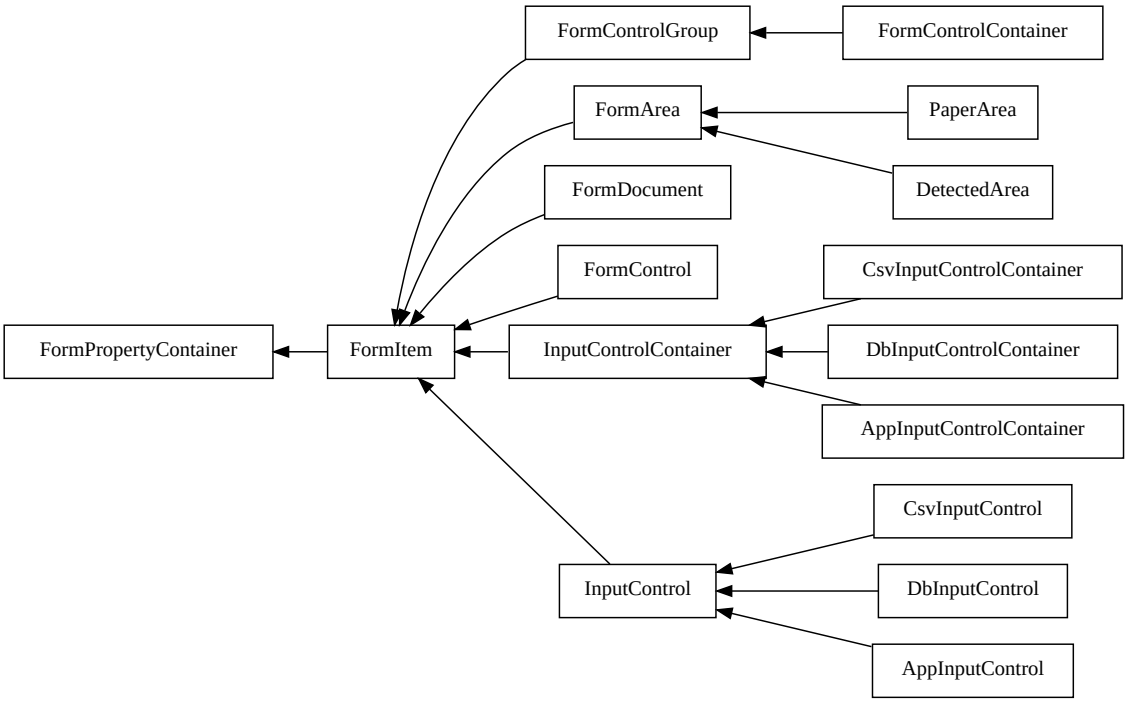


Figure 2.2. The inheritance hierarchy of form items



2.2.1. Form item

The base class of each item in the document is the class `FormItem`. The method `itemType` returns the real type of the item, e.g. `FormControl`, `PaperArea` etc. An instance of this class can be part of the form document or can be standalone. The method `document` returns a pointer to the owning

document or 0 if the item is standalone. There are also signals `bound` and `unbound` that notify whether the item is bound to the document item or unbound from the document. The slot `validate` causes revalidation of the item. This method should be called whenever some external event can cause, that validity of the item has changed.

Except the public interface there are some helper methods which can be used by derived classes. The most important methods are `validateImpl` and `validatePropertyImpl`. The validation is quite complicated process, because the item must notify the owning document about its errors. When the item is unbound it must clear all its error from the document and when it is bound it must send its errors to the document. The item does all these things itself. Derived classes should implement only those methods and validate the given property or optionally the whole object. From the mentioned methods some of `appendError`, `FormProperty::setError`, `FormProperty::setWarning` can be called. The form item then collects all errors which have occurred during validation and propagates them into the owning form document.

2.2.2. Form property container

The form item is base class of each item in the form document, but it is not the root in the inheritance hierarchy. The root is a class `FormPropertyContainer`. This class provides interface for enumerating all public properties of the object. The uniform way for enumeration all the properties is important for example for the GUI, ,because any kind of properties editors can be developed and it can be used for any item in the form document.

The most important method is the method `properties` which returns a list of all possible properties of the container. The properties can be enumerated also through the `propertiesMap`, which returns also unique identifiers for each property. There are also signals for notifying about changes in the properties. The signal `propertyAvailabilityChanged` is emitted when availability of some property has changed, `propertyValidityChanged` is emitted when some property has become valid or invalid and finally `propertyValueChanged` is emitted when the value of some property has changed.

For the derived classes there are two very important methods. The method `load` which loads values of properties from the given element [1] and `save` which stores values of the properties into the given element. Form items do not have to take care of storing and loading its values, they can easily delegate this operation to these methods.

Form property

The form property container provides API for enumeration all its properties. This enumeration is returned as the list of instances of the class `FormProperty`. This class provides interface for obtaining the description (methods `label` which returns short description of the property and `help` which returns more information about the property), state of the property (the method `state`) which can be one of `Valid`, `Warning` and `Error`. If the property is not `Valid`, then `message` returns information about what is wrong with the property. The method `isAvailable` determines availability of the property. When the property is unavailable, user should not be able to read its value or set a new value of the property. Finally `value` returns value of the property as an instance of `Property`.

For the form items there are useful methods `setError`, `setWarning` and `clearError` that are usually called in the `FormItem::validatePropertyImpl`. Method `isMandatory` determines if the property must be stored in the file when loading the document. If the property is not mandatory, then `defaultValue` is used. Mechanism of mandatory and optional properties exists because of the backward compatibility of the form document file. New optional properties can be easily added to the document and the new version of program can still load the files from the previous versions.

2.2.3. Form document

An instance of the `FormDocument` holds the content of the whole document. It provides methods for accessing the form controls (the method `controlContainer`), a list of its pages (the method `pages`)

[1] See the section called “Storage element”

and all its inputs (the method `inputsByType` and the method `inputsById`). The method `errors` provides information about the invalid form items in the document (each item takes care of its own validity and must pass its errors to the owning document). The signal `errorChanged` notifies, that some item became invalid or valid. Methods `save` and `load` save or load the complete structure of the document to or from the file. The more information about this process can be found in the section the section called “Implementation of saving and loading document”.

2.2.4. Form control

Form control is encapsulated by the class `FormControl`. It has properties which holds its programs, data type, identifier etc. Most of the methods are only getters and setters to its properties.

Besides these methods there are some methods for managing the form control. For traversing the form document structure there is the method `paperArea` which returns a list of all paper areas of the form control. The method `parent` returns pointer to the owning form control group.

For changing the structure of the document, there is the method `moveTo` which moves the control into the destination group, `unbindFromControlGroup` which unbinds the control out of the owning group and makes it standalone. The method `unbindPaperAreas` removes all paper areas from the control and returns the list of all of them.

For easy implementation of the control programs debugger there are methods `fillSnippets` which gets a list of instances of `CodeSnippet` and `loadSnippets` which loads the control programs from the given snippets.

2.2.5. Form control group

The class `FormControlGroup` encapsulates the list of form controls and the list of other form control groups. Form controls and the form control groups can create hierarchy such as a filesystem. And form control group is like folder in the filesystem analogy. The only property is its identifier. The method `formControls` returns list of all form controls in the group and the method `subgroups` a list of all subgroups. The method `parent` returns a parent group in the hierarchy or 0 if the group is the root of the hierarchy.

It also provides methods for finding a subgroup or a form control by the given identifier (methods `controlById` and `groupById`), moving a group into another (a method `moveTo`), unbinding a group (a method `unindFromControlGroup`) and unbinding all its subgroups and form controls (methods `unbindFormControls` and `unbindSubGroups`.

For the GUI there are methods `defaultControlName` and `defaultGroupName` which return a new unique name for a new control or a new subgroup.

2.2.6. Form control container

It is the root in the form control groups and form controls hierarchy. It is logically derived from the class `FormControlGroup`. It adds methods `controlByJid` which finds form control by the given *jid*, `controlByPath` which finds form control with the given path and method `listOfControls` which returns list of all controls in the container.

2.2.7. Form page

This class encapsulates one page of the form document. It contains properties like a background image, its size, resolution, size of the page and most of the methods are only setters and getters of this properties. The page can return list of all paper and detected areas which are placed on the page (methods `paperAreas` and `detectedAreas`. These areas could be easily removed from the page (methods `removeAll...Areas`). When the offset of the background image has changed then the method `moveAreas` could be called, when the ratio of the page has changed the method `scaleAreas` should be called.

For changing the order of the pages in the document there is the method `moveTo`.

2.2.8. Form area

A class form area is a base class of detected and paper area. It extracts common methods of both type of areas into one interface, so GUI can work with these areas in the similar way. It provides method for getting and setting coordinates and size of the area on the page (method `rect` and `setRect`). Because GUI should sometimes draw the content of the area (e.g. static paper area) there are methods `update` which prepares the area for drawing, `boundingRect` which returns bounding rect of the graphical content and `paint` which paints the content to the given painter. These methods are designed to be easily called by the implementation of the `QGrahpicsItems` which encapsulates the area in the GUI. The method `clone` should make easier copy-paste operation in the GUI.

Paper area

A class `PaperArea` corresponds to the area of the graphical output of the form control. It is just container of various properties which influence the type of the output (selection or text) and the style of the output. Almost all methods access only this properties.

Methods `moveTo` can move area from one form control to another, the method `setPage` can set the page where the area should be placed on. Methods `unbindFromControl` and `unbindFromPage` makes the area standalone or independent on the page.

Detected area

The class `DetectedArea` corresponds to the area which was detected by the OCR module. The detected area is almost only the container of properties. The only interesting method is `setPage` which sets the page where the detected area is placed on.

2.2.9. Input control

The class `InputControl` provides the data to the evaluator when the document is being filled. Each input control should be associated with one form control and provides data to the form control. Input control has its own data type, which is independent on the data type of the associated form control. The only restriction is, that conversion between these types must exist. This technique enables for example the text edit to be the input for the the form control with the type Date and time. Because dates, times etc. can be inserted in the various textual format each input control has properties which describes the format.

The important method is `dataType` which returns the type of the data which the control provides. There are also methods `moveTo` and `unbindFromContainer` for changing the structure of the input controls. The method `setFormControl` associates the input control with the given form control. The method `type` returns the type of the concrete implementation of the input control.

This class is meant to be derived by the concrete input controls such as application input control, csv input control etc.

Application input control

The class `AppInputControl` encapsulates the application input. For example a dialog or a HTML form. The input control is enhanced by the type of the application input control such as a text edit, a checkbox or a combobox.

Csv input control

The class `CsvInputControl` encapsulates one item in the Csv file. The situation is very simple, because all csv controls have only one data type — the string.

Database input control

The class `DbInputControl` encapsulates one item in the SQL Lite database. The implementation is almost the same as the implementation of the `CsvInputControl`.

2.2.10. Input control container

The class `InputControlContainer` contains a list of the input controls. The container contains only the controls of the same type and this type is returned by the method `type`. The container also contains properties which contains formatting string for the textual representation of the nontextual data such as date and time. The input controls in the container can inherit these formatting strings from its container. The method `moveTo` and `unbindFromDocument` can be used for changing the structure of the containers in the document.

Also this class is meant to be derived by the concrete implementations.

Application, csv and database input control container

These classes inherits from the class `InputControlContainer`. They only enhance the properties about technology specific settings. In the csv there is important a separator and a quote character. For the database the name of the table and the name of the column with the primary key is important.

2.2.11. Saving and loading the document

The whole document can be stored and loaded from the disk which is very important. There are some classes which provides methods for making this process easier.

File storage

The class `FileStorage` provides functionality for storing multiple chunks of the binary data to one file. The format of the file is the same as format of uncompressed ZIP file. The chunks are numbered from zero to the count of the chunks minus one. Each chunk can have its own suffix.

The most important methods are `append` which appends the given chunk to the file and `dataAt` which returns chunk at the given index. The method `save` saves all chunks to the file and the method `load` loads the chunks from the file.

Storage element

The instance of the class `StorageElement` can contain set of named and typed values and the named list of other instances of this class. In fact, it is almost same as the API for working with XML document, where this class corresponds to the XML element and the instance stores and loads its content to the or from the XML element. But this class is designed to check types of the values and their existence. This is useful because during loading the document almost none additional checks have to be done.

The next enhancement is, that binary data can be easily stored to the element. It is thanks to cooperation of this class with the class `FileStorage`. The binary data are stored as a chunk to the file storage and to the XML element only the number of the chunk is stored.

The most important methods are `...Property` (`intProperty`, `imageProperty` etc.) which return the value which is stored under the given name and methods `set...Property` which save the new value to the element under the given name.

The method `save` saves the whole content recursively to the given XML element and the given file storage. The method `load` loads the content recursively from the given XML element and the file storage.

The DTD of the XML which is created by the method `save` can be found in the file `ftd.dtd`.

Implementation of saving and loading document

All item in the document has pair of methods `save` and `load`. These methods have one parameter — the instance of `StorageElement`, where the content of the value should be stored or from which the data should be loaded.

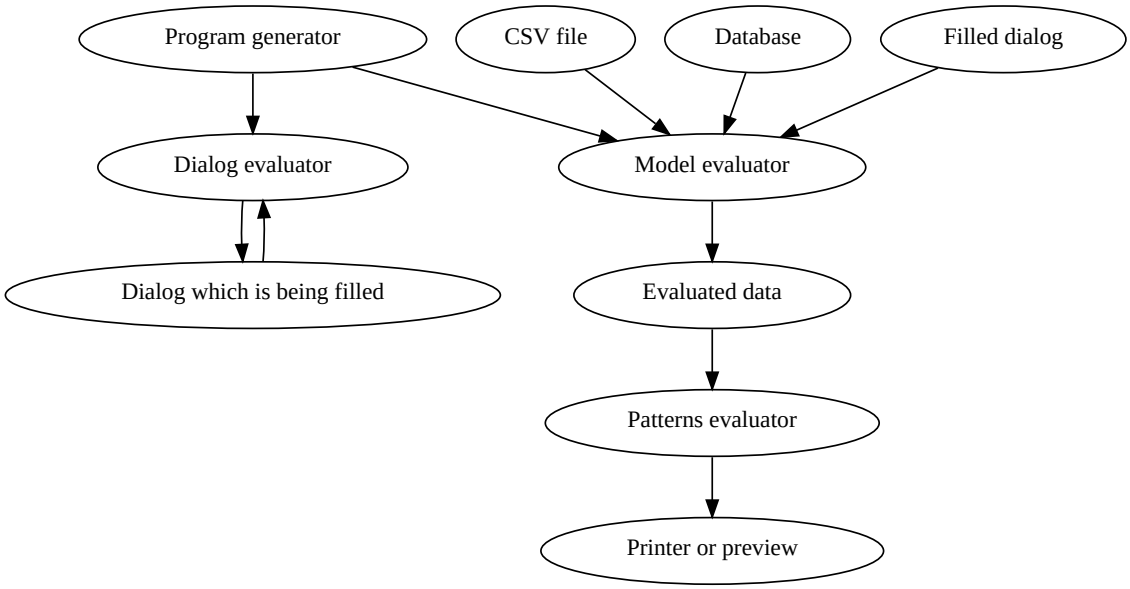
When the document should be saved, it creates the instance of a root storage element, then it calls the mentioned method `save` on its child elements and they save their content and recursively content of their children to the given storage element. When this process is done one instance of the root element of the XML document and one instance of `FileStorage` is created and the root storage element is stored to this pair. See the section called “Storage element”. In the end the XML document is serialized to the chunk number 0 of the file storage and the file storage is saved to the file.

Loading is the same process as saving but reverted.

2.3. Program manipulation

One of the features is that user can customize the behavior of the controls. User can write programs which determine values, validities, availabilities etc. of the controls. The programming language which is used for this customization is *QtScript* which is supported by the Qt library. *QtScript* is compatible with the *ECMAScript* which is compatible with *JavaScript*. Thanks to this our documents can be easily used in the web environment.

Figure 2.3. Program generation and evaluation



The process of the evaluation of the data in the document is depicted in the Figure 2.3, “Program generation and evaluation”. First one complex program which aggregates all partial programs of the form controls is generated. The source code of the program is passed to the evaluator. The evaluator reads data from the data source and writes the evaluated data to the data consumer. The first variant is that the producer and also the consumer of the data is dialog, which also processes notifications about changes of validities and availabilities of the controls. The second variant is, that the producer and consumer are separated. In this case the values of all controls are passed to the consumer for the next processing.

There are many classes that provide methods for manipulation with the form control programs. These classes could be divided into this groups:

- Classes for program generation
- Classes for evaluating programs

- Classes for program transformations. These classes are heavily used by the Editor

2.3.1. Program generation

Each form control has many different programs. For example program which calculates a value of the control, program which checks validity of the control, its availability etc. These programs should be merged with programs of other controls into the one big program which can be easily evaluated. But this main program can differ when it is used in the HTML page or when it is used for the filling dialog. So it must be generated in the different ways.

Although the main program can have various forms, some properties of the program are same in all representations and each representation must satisfies some requirements. The common requirements are these:

- An object model of the controls which corresponds to the hierarchy of form controls and form controls group must be created. Thanks to this control programs can access results of the other form controls.
- The program must ensure that partial programs of the controls will be evaluated in the right order. When there is a cyclic dependency it must be correctly detected.
- The program must keep graph of a data dependency among the form controls. So when a value of some control has changed only such controls that directly or indirectly depends on this value should be reevaluated.
- Programs must correctly detect an usage of the invalid value in the program of the form control and marks the result as invalid. It is important because the result of the form control program which depends on the invalid value is also invalid and user should be notified about it.
- The program must provide methods for interacting with the source of data (for example a dialog or a HTML form).

There is a template program which implements almost all of these features and only some parts — such as the object model or the reaction on some events during the evaluation — must be inserted into the template. `program.js` contains this template and all relevant information are written in comments in this file.

Program generators

The class `ProgramPreprocessor` is a simple implementation of the preprocessor. The syntax is inspired by the C# preprocessor with some restrictions. It also supports one new command `#tr` which is used for translation of the strings in the source program. Thanks to this preprocessor the properties of the program template can be changed. It is also used for translating the builtin packages with the global functions.

The base class of all program generators is `ProgramFiller` which is application of Template method design pattern. It reads the template program (which is preprocessed by the `ProgramPreprocessor` and when it found any incomplete part in it, it calls a corresponding method which should generate the missing content. It also translates language dependent parts of the template program. All program generators should inherit this class and implements the abstract methods for filling the incomplete parts.

Intermediate program generator

Very important class is `IntermediateProgramGenerator` which generates parts which are common for all destination technologies. So the intermediate program fills only these parts and left parts which depends on the destination technology untouched. The intermediate program can be a little bit customized. For example if the `setValue` callback should be called for all controls or only for non-input controls. These customizations are given in the constructor of the object and are passed to the program preprocessor.

C++ program generator

The class `CxxProgramGenerator` is derived from the intermediate program generator and fills the technology depended parts. This generator generates program which could be easily integrated to the C++ environment. It passes all event callbacks to methods which are implemented in the C++ so communication with the *QtScript* program is very easy.

Input program generator

The class `InputProgramGenerator` generates a program for usage when the data are read from the input. The conversion functions which convert data types of the input control to the data type of the form control are generated. The `setValue` callback is not called for the input form controls. And default values of the form controls are calculated.

Model program generator

The class `ModelProgramGenerator` generates program for the situation when the data are read from the input and written to the given instance of the `FormData`. It also supports generation of the program which performs no conversions and assumes that the input data are in the correct format. The `setValue` is always called, so all data are written to the output form data.

Html program generator

The class `HtmlProgramGenerator` is derived from the intermediate program generator and generates program which could be used in the web environment. It fills the technology dependent parts to ensure correct behavior. The resulting program can easily cooperate with the HTML form.

Package management

When the user writes the custom programs, he can use prepared packages of functions. The class `PackageManager` manages these packages and can append their content to the program which is generated. So the functions in the package can be easily called. There are two kinds of packages. Internal which contains functions for internal purposes and user cannot influence their usage and user packages which can be imported and used by user. The important methods are `importPackage` and `importInternalPackage` which write the content of the given package to the given program writer.

Convertor management

The class `ConvertorManager` contains database of all conversion functions. For each conversion it provides the name of the function and required internal packages for the conversion.

2.3.2. Program evaluation

When the program is generated it should be evaluated. Fortunately the Qt library provides the interpreter of the *QtScript* so binding the program written in the *QtScript* to the code written in the C++ language is quite easy. Unfortunately some problems remain to be solved.

Function evaluator

The first thing which must be solved is that program can contain an infinite loop. So some time limit for the evaluation must exists. These functionality provides the class `FunctionEvaluator`. This class is wrapper for the `QScriptEngine` and adds methods for evaluating functions and programs in the given time limit. If the evaluation does not finish during the limit an exception is thrown. If some time-consuming operation must be done during reaction on some callback from the script, method `pause` should be called. This method stops the measuring of the script evaluation time so the operation will not influence the limit for the the script evaluation. After the operation and before returning the process back to the script the method `resume` should be called. This method starts measuring the time for the evaluation again.

Program evaluator

One step higher in the class hierarchy is a class `ProgramEvaluator`. This class encapsulates communication between the generated program and the the rest of the program written in C++. It inherits methods for evaluation from `FunctionEvaluator` so it does not have to take care about time limits. The class initializes the `QScriptEngine` and registers functions in C++ that will be called in the *QtScript* program callbacks. This class must obtain in its constructor an instance of `ProgramEventHandler` to which all callbacks are passed. Thus the evaluator does not take care how to handle the events it only forwards them to the handler. The class also provides methods `evaluateAll` which calls the function `evaluateAll` in the script and the method `valueChanged` which passes its the argument to the `valueChanged` in the script and calls it.

Program event handler

The class `ProgramEventHandler` is the abstract template class which must be implemented by the client of `ProgramEvaluator`. Through the handler the evaluator reads the form controls data, sets the evaluated data of the form controls and notifies about changes of availability, validity or of a set of possible values of the form controls. Data are exchanged through instances of the template parameter of the class. So it can be used for reading `QVariant` from the dialog or `FormValue` when the whole model is evaluated.

Utils for program event handler

Because the interface `ProgramEventHandler` is quite important, there exists some prepared implementations of it. The class `ProgramHandlerAdapaterForVariant` adapts the interface so the communication is through instances of `QVariant`s instead of `QScriptValue`s. The variants must be of same type as the corresponding input controls.

The class `EventHandlerValidatingAdapter` is partial implementation of the `ProgramEventHandler`. It remembers all errors which occurred during evaluation and after evaluation it is easy to find out if some errors have occurred and enumerate them. This class could be used when the whole data model of the form document is evaluated. Other methods are left unimplemented.

The class `EventHandlerValidatingWriterFormValue` enhances implementation of the class `EventHandlerValidatingAdapter`. The class remembers all values which where evaluated by the evaluator in the given form data.

The classes `EventHandlerReaderVariantToFormValue` and `EventHandlerReaderFormValues` are very simple implementations of the mentioned interface. They read data from the instance of the class `FormData` or from the map from *jid* to the `QVariant` with the value of the corresponding form control and saves the evaluated values to the output instance of the `FormData`. They inherit the `EventHandlerValidatingWriterFormValue` so list of errors is remembered. These classes are used for evaluating the data model of the document, for example when printing the document.

Data conversions

When filling the form in the conversions between various data types must be performed. For example when data are read from the storage (for example from a CSV file) and the data are in other format than the format which is required for the filling. The interface `InputConvertor` provides methods (`toFormValue` and `fromFormValue`) for converting the data of the input control to the instance of the `FormValue` and vice versa. One concrete implementation is the class `ScriptInputConvertor` which uses the convertors written in the *ECMAScript*. There is no support for direct conversion of data in one input control to the other, but two convertors can be created one from the source data to a form value and second for converting the form value to the destination data.

High-level evaluators

The class `ProgramEvaluator` is quite low-level. The user of the class must generate the appropriate program, pass it to the evaluator and choose the right implementation of the interface `ProgramEven-`

tHandler. Because of this, there is class HighLevelEvaluator which is the base class for easy-to-use evaluators. It provides a minimalistic API and wraps both a program evaluator and a program generator. This class is inherited by the DialogEvaluator and the ModelEvaluator. The dialog evaluator should be used for evaluating dialogs because it uses instances of QVariant as the communication medium and calls the method setValue only for non-input form controls. It also returns the list of default values of the form controls and also calls setValues for the enumeration form controls.

The class ModelEvaluator should be used for evaluating complete data of the model. It gets the input form data and reference to the output form data where the result will be stored.

These classes contain the method init which initialized internal evaluator with the appropriate program. After calling this method, evaluator is ready to use.

Patterns evaluation

The class PatternsEvaluator is used for evaluating the pattern programs of the form controls. These programs are not a part of the data model and they are used only when printing the form document. It also inherits the class FunctionEvaluator because the programs are also written in *QtScript* so there is also the problem with infinite loop. The main method is evaluate which evaluates the pattern program of the form control with the given *jid*. The evaluator evaluates all expressions and methods in the program except the evaluation of the method format which is passed to the PatternEvaluator which can evaluate arguments of this method.

2.3.3. Program manipulation

In the GUI of the form document editor the user can edit the source codes of the form controls program. The user can edit them as the properties of the form controls. There is only one problem to solve. Programs in the form = expression; must be transformed into the return expression;. But there are more features when editing the source codes.

Program utils

The class ProgramUtils provides some methods which do common operation with source codes. The most important method is the method makeProgramSafe which reads the source code and optionally overwrite comments and strings in quotes with the spaces. This is very useful, because the source codes can be then processed with regular expressions. For example when extracting arguments of the method is needed, it could be found as all characters between (and) because all possible character) in the string arguments which could break this rule are removed. The “safe program” has the same layout, so when the arguments are matched and their position and length is found in the safe program, the same numbers (position and length) can be used in the original program for extracting the real values.

Program writer

The class ProgramWriter provides methods for writing prettier source codes. In fact it wraps QTextStream and has methods for indenting and unindenting lines. All program generators use this class so the textual output of the generators could be easily read by the man.

Program representation

One of the features in the form document editor is that user can view all the programs of form controls in one editor. This editor has some features which were not easy to implement. The order of the programs, indentation and coding style should be kept between editations. User can write global variables and global function in this editor. So there must be a mechanism which reads all programs of form controls, merges them with the global code and makes one string representation of all these things. The user can edit this representation then and when is finished, the modified string is parsed, the bodies of form controls programs are extracted and saved back to the owning form controls.

All these operations performs the class ProgramRepresentation. As the input it expect a *program template*. It is a string with the special marks. These marks have the form like /

`*__@ProgramName(uid) @code*/` or `/*__@ProgramName(uid) = function { @code }*/`. These marks represents one form control program. `ProgramName` is name of the program (valid, value, available etc.) `uid` is the *uid* of the control and `@code` is the body of the program. The first variant is for programs in the form `= expression;` and the other for programs like `commands; return expression;`. When processing template these marks are replaced by the real values of the form controls, so this mechanism handles correctly when the form control is renamed. It must be also handled when the user changes form of the program in the properties of the form control. For example from form `= expression;` to the other form, but this is only technical detail.

The opposite problem is making the program template from the edited representation. In this case the abstract syntax tree of the source code is constructed. In the tree the bodies of the programs are found and replaced by the marks. Other content such as global functions and comments lefts untouched.

Communication with this class is performed through the string with the edited representation or the program template. Bodies of the programs are exchanged through instances of the class `CodeSnippet` which is designed for the purpose.

2.4. Printing

The printing of the content of the form document is in fact the most important thing, because it is the final output from the whole program. This part takes care only about the graphical output. Evaluation of the data which should be printed is described in the section Program evaluation. The only problem which remains to solve is formating the output. Area has it own rectangle on the paper and its content should be printed to this rectangle.

2.4.1. Form printer

The class `FormPrinter` provides all services related to the printing. It gets instances of the form document, its data and a list of instances of `QGraphicsScenes`. One scene for each page of the document. Printer creates one instance of `AreaPrinterItem` for each paper area and add them to the scenes. The area printer item inherits the class `QGraphicsItem` and is responsible for painting the content of the corresponding area to the scene. This design is flexible enough to draw the document to the screen and show the preview of the result or print the document to the printer or to the PDF.

Main method of the class is `init` which initializes the printer and the method `draw`. This method reads the data from the input and distributes it to the area printer items. It must be also called when the input data has changed to update the content of the items.

2.4.2. Area printer

The class `AreaPrinter` is responsible for printing content of one paper area. The method `pushValue` passes the content to the printer. According to the capacity of the paper area and to the overflow policy this method can return excess of the content. This content can be passed to the next area. This method also prepares internal structures for painting. Because of performance reasons the content is formatted only once. After the formatting, the positions of the words are stored and thus we do not have to perform formatting again during next repaint.

2.4.3. Area printer item

The class `AreaPrinterItem` only adapts interface of the `AreaPrinter` to the `QGraphicsItem` so it can be added to the `QGraphicsScene`.

2.4.4. Text printer

Whole textual output is created by the class `TextPrinter`. This class gets the string and formats it to the given rectangle. It supports features like overflow policy which influences the behavior when the

string is to long. The string can be truncated and the excess is returned or the font size can be made smaller or the bottom border (if the area is multiline) or right and left border (if the area is singleline) of the rectangle can be just ignored.

Because there are no methods in the Qt library which can be directly used, all the formatting functions implements the class itself.

2.5. Exporting the document

The finished document can be exported to the various formats. The basic export is to the lite version of the file. The suffix of the file is `.ftx` and it is almost the same as the `.ftd` file, but it does not contain images and detected areas. So the resulting file is much smaller.

These `.ftx` files can be used for example with the FormTools API, because images and also detected areas are useless for filling or printing forms.

2.5.1. HTML export

The form document can be exported to the HTML file. This HTML file should provide the same functionality for filling as the electronic form. The application input controls are mapped to the corresponding controls which are allowed in the HTML language. There is also included Javascript code, which evaluates the programs and responds to the user input.

The process of exporting is performed by the class `HtmlExport`. The process is divided into the separate parts. The generation of the HTML header, the script and the form. Thanks to this, the resulting HTML file can have separated Javascript code and the HTML data. The Javascript is generated by the class `HtmlProgramGenerator`. The generated file is XHTML 1.1 valid.

2.5.2. PHP export

Generation of the HTML file is quite unusable, because it cannot be easily customized. That is why there is also possibility to generate file with functions written in PHP, that manage to generate separately Javascript code and also the HTML form. These functions have parameters which influence the basic properties of the result. The usage is very easy. The developer of a web page just include the generated file and calls the provided function. So the form can be placed anywhere on the page. There is also function, which makes loading the content of the form easy. Developer must only fetch the array of the form data for example from the database and pass them to this function.

The generation of the PHP file is performed also by the class `HtmlExport`.

2.5.3. Export for Filler

Documents described by the form document can be used to draw an Qt form which will allow to fill in the data of the input form controls and to store these data to one of the defined data storages. In Form Tools project there is a dedicated application called Filler which allows to do this.

The main goal of the Filler is to be as simple as possible. Therefore, the author of the document has to specify some details regarding filling and saving of the data. All the required information is stored in the `FormToolsProject` instance. The project contains list of data storages. Each data storage is described by the class `ProjectItem`. The project instance is serialized to the xml file.

2.6. Image acquisition

On some platforms the scanning is supported. On the Windows the TWAIN[TWAIN] and on the Unix the SANE[SANE] library are used. The implementations is inspired by the documentation of the TWAIN and by the documentation for the SANE. Because these standards are very different there is class `ImageSources` which provides one interface for both technologies.

2.6.1. Image sources

The class `ImageSources` provides easy-to-use interface for manipulation with scanners. The method `isScannerSupported` returns `true` when the scanning on the platform is supported and `false` if it is not. Methods `selectDialog` and `scanDialog` returns pointers to single instances of `SelectDialog` respectively `ScanDialog`.

Select dialog

The class `SelectDialog` has only one method `exec`. This method shows the dialog for selecting the source of the images.

Scan dialog

The class `ScanDialog` encapsulates the scanning process. The method `exec` shows the dialog for setting up and run the scanning. The method `image` returns the scanned image if scanning was successful and the process was not canceled by the user.

Chapter 3. Script module

The Form Tools project heavily use manipulation with the *ECMAScript* source codes. The Qt library provides functions only for evaluating them and no for manipulation with the source codes. The evaluation of the scripts is implemented in this way: First the abstract syntax tree of the program is created and then this tree is evaluated. The syntax tree holds the structure of the source code so it can be be traversed so the transformation of the source can be performed much more easier. The problem is that the syntax tree which is created by the *Qt library* is not accessible with the official interface.

Fortunately Qt library is open source and are released under the LGPL license. So the source codes can be used and modified for the purposes of the Form Tools project with the restriction, that these changes must be also under the LGPL license.

The only part which is needed from the library is that which creates the syntax tree from the given source. But the sources must have been a little bit modified. The first modification is, that nodes of the tree stores their real position in the source code as an offset from the begin of the input and their length. Originally the nodes contains only the number of line and number of column where they are. There was also a bug which caused, that position of some nodes in the program were set to wrong values. So this error had to be fixed in this module.

Second modification is that other memory management is used. The problem is, that during construction of the tree some error can occur. At this moment the tree is not complete and can not be easily deleted from the memory. The solution is, that every instance of the node is registered in the `SyntaxTree::NodesHolder` and all the nodes are deleted together this instance of this class. So tree do not have to be complete.

The next important change is, that the lexical and syntactic parser are modified to support the import of packages. The only change in the lexical parser is addition of new keyword import. In the syntactic parser the gramatic had to be enhanced.

The last modification is that the syntax checking was improved. The original syntax checking returns the error messages only in the english. These messages can be easily translated now and they are more accurate. Originally the messages were too common and sometimes even wrong.

Chapter 4. OCR module

4.1. Overall architecture

The OCR module is responsible for object recognition and providing additional information about them. Its core job is to recognize shapes which could be form areas, detect rotation of the given image and finally to do some simple transformation of the image. This provided functionality is implemented by separated modules where every module typically corresponds to some part of the image processing. Since those modules are almost independent, it is possible to replace any algorithm of processing by another one. This architecture was done for future implementations, because we expect to have more than one algorithm for some parts of the recognition and thus better results can be provided.

The main modules are following:

- Image denoising
- Edge detection
- Image straightening
- Areas recognition
- Detection of type and group of detected areas

4.2. Image denoising

For image denoise are responsible classes which implement the interface `DenoiseFilter`. So far the only implementation is Gaussian filter, because of its speed and good results in denoise. In addition it preserves edges which is very important for further analyzes. Our special feature is that we support denoise in one direction (horizontal or vertical). Thus edges in one direction are preserved while in the other one the noise is reduced. All calculations are parallelized. The filter behavior is determined by one number (intensity) which is converted to Gaussian filter parameters, so the filter setting is very user friendly.

4.3. Edge detection

The interface `EdgeDetector` provides abstract methods for edge detection. The only implementation it is the class `CannyEdgeDetector`. The name is a little bit confusing since it is not Canny edge detector. This class is only inspired by its idea. We made many tests with all edge detectors we know but the results were not sufficient at all. In fact we need precision about 2 pixels per line, but those detectors are not created for such tasks. They are good when copying shapes, however you cannot tell them that you expect shapes to be a line. In addition, if you know that you are looking for lines you can do additional heuristics. So we decided to write our detector using the fact that we know that we are looking for horizontal and vertical edges.

First let's assume that we want to find all horizontal edges, so we perform denoise in horizontal direction. This is very important for the next step. Then we calculate gradients in the image using Sobel operator. Because we used changed denoise filter, gradients in vertical direction are preserved (no denoise in that direction was performed), while all noise (in this case, we assume that noise is everything which is not horizontal line) is made smaller. This is very important step since it improves results significantly. The main benefit is smaller sensitivity for edge detection settings. Analogously the detection is performed for vertical edges.

We still assume that we have just calculated Sobel operator (the method `calcGradients`). There is a little interesting thing — it is much faster to unroll the for-cycle for calculation of the gradient of the one point. Maybe it is too technical detail but we had to made many decisions during the implementation

of this library like this, and this is one example of what possibilities we had to check for almost every line of the code. After that we use quite usual technique — the points which have the maximal value of Sobel operator are preserved and all other are set to zero. This calculation is parallelized again. Then we go through the field with gradients (`findEdgesVertical` and `findEdgesHorizontal`) until we find a point with sufficiently high gradient (according to the settings) and we start to search for a line. For the given point we look to its neighbourhood whether it contains point with sufficient gradient (we prefer vertical or horizontal direction) and if so we move to this point (in DFS manner). Then we search neighbourhood for the new starting point and we continue until we cannot find next point with sufficiently high gradient. Notice that since we go only to the right (or down), we cannot change direction of the line and it will be one pixel thick. In addition we do not search only in neighbourhood but also 2 or 3 pixels far thus we can rule out one pixel big noise. Finally we take all points found by this algorithm and using linear regression we calculate the most probable position of the line. As you can see the time complexity of all those operations is linear to the size of the image, except the image denoise. However this operation is usually the fastest one (for reasonable values of intensity) because only simple operations are performed in it. It is not edge detection responsibility, but if we use these edges for creating areas, we adjust them to be strictly horizontal or vertical (we simply rotate them). As you can see this algorithm will have the best results for the strictly horizontal or vertical edges, although can be used for detecting all lines (but the improvement of the denoise filter is not used). That is the reason why we need aligned image.

4.4. Image aligning

For image aligning classes implementing the interface `TransformDetector` are responsible. The interface provides methods for finding the most probable transformation to get the "original" image from the image which is being processed. Currently there are two detectors which implements this interface. One for the images from a scanner and another one for images from cameras. The second one is implemented, however it is not used now. The reason why is explained later.

4.4.1. Aligning images from the scanner

We use a following observation — the only possible transformation is rotation. Thus if we find the angle of this transformation we are finished. And this is exactly what the class `RotationTransformDetector` does. This algorithm takes all edges detected in the image and calculates the most probable angle for the rotation. We assume, that in the original image the most probable edges are horizontal or vertical. We calculate average angle and remove an edge which differs most. Then we recalculate average angle and we continue, until difference between upper and lower bound is small. And this is our resulting angle. Our implementation does it in O(N log N) time. We sorts lines by angle. Then we can remove the most diverging angle in the constant time because it will be the first or the last angle and thus removing part needs O(N) time. The results are so good, that we remove possibility of manual angle correction. We find out that automatic correction is always better than you can do by hand because forms are very specific images. In addition we can scale down image twice and still obtain good results — which enables us improve performance.

4.4.2. Aligning images from photos

In the second detector (`GreatestRectangleTransformDetector`) we in fact try to rule out with arbitrary transformation. We try to approximate it by linear transformation. There are too many unknowns thus we have to estimate size of the original image — but it does not have to be a problem, since you have to measure it to get right calibration. This is necessary in any case because we do not have enough information and one picture can be mapped to infinitely many images. Results were not terrible, but they were not accurate enough, because we need less than one millimeter accuracy. In addition the results for images from scanner are very bad — the detected transformation is not accurate enough and sometimes it goes totally wrong. The main problem is to detect important point for detecting transformation. Even if we detect them correctly (for almost 90% of forms we really did), we were not accurate enough — even 2 or 3 pixels (which means something like 1 mm) inaccuracy caused that the results were not sufficient. In addition even a small deformation of the source image or deformation caused by camera or shaking hands cannot be automatically detected from one image and thus cannot be corrected. That is way it is in the most cases better to correct just rotation than use this more sophisticated algorithm.

4.5. Classes for areas recognition

The main idea is following — take all edges, find rectangles which they produce, and finally try to recognize type of areas. In addition this classes are responsible for detecting dotted lines. We start with searching rectangles created by edges.

4.5.1. Rectangle detection

For rectangle detection the interface `RectangleDetector` and its implementations are responsible. One implementation provides the class `SweepLineRectangleDetector`. First we define what we are looking for. We are searching for such rectangles that does not contain another rectangle and are big enough. Such definition does not ensure that there will be only one result for one image and in such case, it is impossible to decide which one is better. Thus we take first we find. We divide algorithm into two steps. First we find intersections between edges. Then we use this intersections to determine areas. During this analyze we also check whether the found area contains boxes for letters.

First we find intersections between edges in such a way, that for every horizontal edge we find a set of vertical edges that intersect it. Searching for intersections is performed by `drawRemoveConnection` and `findNeighboursAndConnections`. The searching itself is done in the following way: We create bitmap of the same size as the original image. Then we "paint" horizontal edges into this bitmap (first method). It also enables us to connect horizontal edges which overlap into one. Finally we take vertical edges one by one and we find intersections (second method). The intersections are stored to structures. There are only two such structures. For horizontal(`HorizontalLine`) and vertical edges (`VerticalLine`). There are few technical problems, but they are not important (for example when the corners of the rectangles are rounded). It is usually faster (linear in lines length) than trying to intersect all pairs of edges (which has square time complexity to the number of edges).

With such precalculated data we can find areas using the method `detect`. We use some kind of sweep line. We sort horizontal edges from the top to the bottom and from the left to the right. We take these edges one by one and for each horizontal edge we do the following. We take all vertical edges intersecting with this horizontal line from left to right. For each such line we try to take vertical edges left to this one (we take them form right to left, so we try to create smaller rectangles first) until we find rectangle which does not contain another rectangle or the horizontal line ends. If we find such rectangle, we sign that there is the rectangle and we continue. When searching for the rectangles, we use intersections stored by every vertical line. We also have to know, where is the bottom of the last rectangle for every image point. We do this by one array which represents last rectangle position for every image column. Although this seems to have O(N^4) (where N is the number of edges) time complexity, we cannot set up any example for more than O(N^2). But we cannot prove that it cannot happen. However for all our test data it runs very fast. We do some additional tests to recognize, that area consists of one letter edits, for example the edit with those small lines inside it. We do it in the following way — we take all lines intersecting the bottom line of just detected rectangle and convert their position to line of 1 and 0 where 1 means that there is a line, 0 there is not (in fact we do "bitmap" of the bottom line of the detected rectangle). We perform fourrier transformation for this array and according to its results we find letter boxes. It is done by `SweepLineRectangleDetector::BoxCellDetector`.

4.5.2. Dotted lines detection

The interface for dotted line detection is `DotLineDetector` and the only implementation is the class `CenterLinesDotLineDetector`. This task was really an issue and unfortunately it is not solve at all. For pictures with low resolution (very low) and very low quality, we can find false dotted edges because of noise and on the other hand we can miss existing lines because we cannot precisely distinguish between noise and dots. Fortunately this is not the case for images from scanner in reasonable DPI. Our algorithm consists from two parts — in the first part we find candidates for dots and in the second part we try to create lines from them.

In the first part we use a bitmap, which is created by edge detector. It tries to recognize areas where could be dots. It is done using gradients and denoised images from previous calculations — thus we

can improve speed significantly. Having those borders we use BFS to determines areas of single dots. However this is the part, which caused malfunction and even if the detection of dots is quite sophisticated it sucks for bad images. After this part we have list of `DotCandidate`. Before we create lines, we try to filter out dots with non circle shapes and those which are too big or small. Lines are created from dots in the following way. Every line candidate is represented by the class `LineCandidate` . For every dot we try to add it to an existing line candidate and if we cannot do that because all lines are too far from this point we create new line candidate. Finally we chose those line candidates, which are long enough (in sense of number of dots in it).

4.5.3. Detection of types and groups of detected areas

Finally we want to find types and groups of the areas. The type is estimated by the class `GroupTypeDetector` . The main clue for determining type is the shape of the area. This heuristic has reasonable results if connected with group detection. The first problem are areas which are rectangular and which are not checkboxes. However then there are usually more than one in the group and we can recognize them. Groups are done according to common bottom lines — again this heuristic has very good results. The second problem is to recognize multiline areas. We try to estimate row height (we take the most frequent height of the line edits) and using this we decide for every text area.

4.6. Gui

Every part of this algorithm needs its settings. Thus we created interface `ToolGui`. By implementing this interface detector can provide its GUI which can be used by owner of this tool. Every widget can ask for GUI of this tool and can use it. The responsibility of deleting this GUI is on the tool side.

4.7. Performance

We spent a lot of time by improving speed. Finally for image of a common size (which means about 3000x2500 points) we can do the analysis for reasonable settings in less than 5 seconds. The first thing we did was parallelization. We used OpenMP which is very efficient but still it is not too hard to integrate it to the existing source code and in addition it does not make it less readable. Another improvement is done by writing algorithms which are cache oblivious. Also testing for practical data sometimes results in using asymptotically worse algorithm because of its practical results. Finally we obtain very good improvement by reusing memory for images and allocated images as one block — and thus minimize the need of calling new which is expensive.

Chapter 5. GuiCommon module

GuiCommon is a library of classes which are used in multiple applications of the Form Tools project so that they can be easily reused. The module contains various classes mostly gui oriented, like common dialogs and widgets.

The GuiCommon module does not have its own namespace but the individual classes are part of the namespace of the application that these were originally designed for.

5.1. API

The GuiCommon module also contains the private implementation of the Form Tools API interface (which is independent of the Qt SDK). The private implementation is similar to the non-Qt interface with standard types replaced by the Qt types.

The main class representing the logic of the API is the `FormDialogPrivate` class. This class contains all the required objects to show, fill in, evaluate and print the data of a form document (even the QApplication instance if required). The document can be loaded by calling the method `loadForm`. It will create all the required objects for this form (such as data evaluator) and initialize the GUI. The GUI can be shown by calling `exec`. It also allows setting and getting the data filled into the opened form document using the methods `data` and `setData`. The form dialog private also offers methods for showing a preview of the current data and printing it to a printer or a PDF.

The data can be edited in the form (which is the same form as used in the Filler) with help of the class `FormDialogGui`. This class contains the `FormHolder` instance that represents the form and handles evaluation of the filled data along with tracking of the form field changes. The current evaluator is passed by the `setEvaluator` method. The gui implementation also contains resources for previewing and printing of the filled data.

5.2. Form holder

The `FormHolder` class is an implementation of a `QWidget` that creates and maintains the form corresponding to the given `AppInputControlContainer`. The created form allows the user to edit the data of the associated form document in the corresponding editor widgets and display the optional computed controls as static texts. The form holder also allows to set the validity and availability of each form field. The `FormHolder` is an abstract class, the final child classes have to reimplement the `reloadContainer` method and fill the inner containers according to the given input container and its controls. The child form holder classes are responsible for creating the jid-widget mapping which is then used by the inherited implementation.

The form holder basically creates an interface above the form widgets. This interface corresponds to the structure of the given input container and maps the jid of the form control to the corresponding form widget and its value. The error handling uses also the jid-widget mapping. The values of the associated form widgets can be accessed by methods `value` and `setValue`. The non input control values are read only since they are computed by an evaluator. That means that the `value` method returns only the input control values and for non input controls the null `QVariant` is returned. The data adapter also contains methods for setting and getting the whole data at once and methods for setting the available values for enumerable form controls. The validity of a control is displayed as an icon signalizing the invalid state and containing the error message as its tool tip and status tip. The availability if a form control is mapped to the enabled state of the corresponding form widget.

The form widgets are created according to the type of the associated input control in the method `createWidget`. Because different widgets have different methods for getting and setting their values, the special proxy object called `FormHolderItem` is created for each form widget. Using this proxy object the value of every form widget can be accessed via a single interface. These holder items are created in the `createHolderItem` method. For each used form widget there is a child implementation of the `FormHolderItem` interface.

The form holder emits the `fieldValueChanged` signal every time the value of some widget in the dialog is edited (e.g. `textEdited` signal of a `QLineEdit`). Changes of inner widgets are tracked in the `fieldChanged` slot where the changed item is identified and the `fieldValueChanged` signal is emitted.

The form holder also supports external undo/redo handling. If any of the widgets, which are supporting undo/redo, requests an undo action, the request is further passed using the signal `undoRequested` if the widget can not perform the undo itself. The undo operation on a form widget can be enforced using the `tryFieldUndo` method (if the target widget supports undo). The same thing is with the redo action with the exception that the `redoRequested` signal is emitted regardless of the availability of a redo in the widget. The redo can the be enforced using `tryFieldRedo` method. This behavior is disabled by default but can be enabled by calling the `setUseExternalUndoRedo` method.

Individual form widgets can be accessed using the `widgetForJid` and `jidForWidget` methods.

5.2.1. Table layout form holder

The `TableLayoutFormHolder` is a particular implementation of the `FormHolder` abstract class which inserts all the form widgets into a `QGridLayout` according to the layout defined in the given `AppInputControlContainer`. The tab focus order of the individual widgets in the inner containers corresponds to the order of the associated controls in the input container. All the information necessary to set up the grid layout (including the stretch factors of the individual columns) is stored in the given input container and its controls.

The table form holder uses a new type of form widget `QTextEdit` which is used for multiline text controls. All corresponding methods are modified according to this enhancement. Worth mentioning is that the `QTextEdit` can not return whether it's enabled for undo so the table form holder has to track this information by itself. The same problem is with capturing the end of the editing (this is simulated by catching the `focusOut` event of the widget using he `eventFilter` method).

5.3. Database input helper classes

The database datastore of the form document requires some common functionality across both Form Editor and Filler. The shared classes handle for example encoding and decoding of the database connection strings and the generation of SQL commands which should be used to modify the datastore.

The encoding and decoding of the database connection strings is implemented in the `DbInputHelper` static class. The connection information is represented by the `ConnectionInfo` class. The connection string can be encoded using the `encodeConnString` static method and decoded using the `decodeConnString` static method.

The generation of SQL commands is implemented in the `DbInputSqlGenerator` class. This class creates SQL statements according to the given `DbInputControlContainer` instance. The commands are created using text templates into which the passed arguments are filled in. This class is suitable for batch generation of SQL commands. The SQL commands can be created using methods `insertCommand`, `updateCommand`, `deleteCommand` and `createTable`.

5.4. Flow layout in a scroll area

Multiple pages on a screen are shown in a single scroll area to enable zooming and just one pair of scrollbars for the whole viewport. The pages are arranged in a flow layout (from top to bottom in rows flowing from left to right).

`PaperScrollArea` is a simple extension of a `QScrollArea` which is able to return its relative `scrollBarPositions()` and restore the relative positions later within the `viewportResized()` method. It also emits a signal when a wheel scroll event was caught in the scroll area. This enables zooming pages by mouse on the whole viewport rather than only above the inner widgets.

`FlowLayout` is a basic implementation of layout which tries to position its items next to each other within the available width, if the next item does not fit to the current row, a new row is started below any of the above items. The available space needs to be explicitly set by the `setAvailableSpace()` method. It supports horizontal alignment of items within the available space and margins. The main method performing the layout is `doLayout()`. The `sizeHint()` returns the minimum space required by the items for the given available width.

5.5. Snapping to lines

The class responsible for snapping is called `Clipping`. This class is somewhere between OCR and the GuiCommon since it directly uses results from OCR analyze. This class basically takes a set of horizontal and vertical edges and then it quickly finds for the given point the closest point of any detected line and in the horizontal and vertical direction (`clips`). And that is what we need for snapping. In addition it supports merging of the existing clipping structure with the newly obtained one (`mergeWith`).

The main structure that contains the clipping is quite simple. We realized that it is enough to store for every line of the image the sorted list of the coordinates representing the lines they intersect. And the very same we can do for columns. Thus for the given point's coordinates we can find the closest lines in the horizontal and vertical direction in at most log N time (where N is the width or the height of the image). Notice that usually it will be much faster, because number of edges in the line is rather small. When you want to snap a line you use method `clipLine` which finds the closest line for the whole line — it is done per every point of line thus obtaining N log N time complexity in the worst case — which is fast enough.

Small problem is a calibration. All calculations are performed in image coordinates and you have to take care about that.

Finally we want to write something about merging two clippings together. Imagine that you made additional detection, thus you obtain new clipping for some area and you want to replace this area in the original clipping. This is done very simply. We take every row contained in the new clipping and we remove those indexes which correspond to the new clipping. Then we insert newly obtained lines to this rows and we are finished. We can do exactly the same form columns.

Chapter 6. Form Editor

Form Editor is the most important and the most extended part of the Form Tools project. It allows user to create digital forms from scans or images of paper forms and also to specify, how the data filled in the digital form should be printed into the original paper form. Besides, Form Editor allows to define data storage for the data filled in the digital form.

6.1. Overall architecture

Form Editor is a Qt application with a main window represented by the class `FormToolsEditorWindow`. It allows to edit multiple files at one time. Each file is edited in one subwindow (which is in fact MDI window). Each of the opened documents is represented by the `EditorsWidget` class instance which contains the corresponindg instance of the `FormDocument` class from Lib module. Each part of the edited document (paper layout, electronic layout, storage) is then edited in its own editor which is child class of the `FormEditor` (e.g. `PaperFormEditor`, `AppInputFormEditor` etc.). The `EditorsWidget` instance is responsible for creating and connecting these individual editors as well as redirecting global actions (from the main window) to the specific form editor.

Form Editor consists of several modules:

- Data models
- Paper form editor
- Image analyzer
- Electronic form and datastore editors
- Toolbar action management
- Error handling
- Property handling
- Script editor
- Script debugger

All these modules (except the image analyzer which is described in the Chapter 4, *OCR module*) will be described in more detail in the following sections.

All classes in this module are enclosed in the `FTGui` namespace.

6.2. Data models

All the data of a form are stored in a library object `FormDocument`. Form Editor provides ways of displaying contents of this complex structure to the user and also allows the user to change the document. Most parts of Form Editor are based upon the Qt model/view framework, where the models are proxies above specific parts of the form document. All data which can be edited in Form Editor are stored in some data model (defined by children of the `QAbstractItemModel` class). Data stored in models can be displayed to the user and edited via views (defined by subclasses of the `QAbstractItemView` class). This allows loose coupling between storing and displaying data. Moreover, the same data model can be displayed by multiple views simultaneously. Using model/view architecture also makes easier the support of undo in the application.

The only exception to this architecture is the definition of the layout of the paper form, where we need to display many items (areas) at once in a graphical way. For this part of Form Editor, the Qt Graphics View framework is used.

6.2.1. Basic overview

The data models are used to create an adapter object over the `FormDocument` class instance (which holds the actual data) via the standard Qt interface `Qt::AbstractItemModel` from the Qt Model/View framework. All data models are based on the `TreeModel` class which is the basic implementation of the `QAbstractItemModel` interface. This model provides mapping of the indexes to objects (`TreeItem` and subclasses), which enables different types of items to coexist in a simple model. The model is a tree, allowing each of its items to have subitems.

The `TreeModel` also provides functions for dynamic changes of the model (as `insertRows()`, `removeRows()`, `moveRows()`) and an interface for easier implementation of drag and drop (`mimeEncodeItems()`, `mimeDecodeAll()`). The model supports saving all the actions performed to a `QUndoStack` which makes the incorporation of undo and redo into the application relatively easy since almost all changes of data are passed through some child class of this model. The model also works fine without the undo stack.

Subclasses of the `TreeItem` class have to implement access to the actual model data via `data()` and `setData()` methods. All subclasses of the `TreeItem` class store a pointer to a document control (eg. `AppInputControl` for `AppInputModelItem`) and allow access to its properties via the `Qt::AbstractItemModel` interface. They also provide additional functionality connected to individual cells of the model (`flags()`, `headerData()` etc.)

The model items also handle the external changes in their associated controls and inform the views displaying the model's contents about the changes being made by emiting the `dataChanged()` signal.

6.2.2. Control model

This model is implemented in `ControlModel` class (which is the child class of `TreeModel`) with `ControlItem` or `ControlGroupItem` classes for items. The model allows to acces the `FormControl` and `FormControlGroup` hierarchy of the `FormDocument` instance. This model uses the hierarchy of items as stored in form document. The model items do not manage their child items themselves but they ensure the undo/redo functionality for the underlying control hierarchy.

The model items store the `FormControl` pointer (for `ControlItem`) or the `FormControlGroup` pointer (for `ControlGroupItem`). For the `ControlGroupItem` model item class the list of child items and methods for their management are provided. The child item manipulation is performed directly on the associated form control group. In the `ControlItem` item class the methods concerning child items manipulation are empty.

The responsibility of the model items is to track changes of their associated controls and their errors. The error changes are handled in the `errorsChanged` method and can lead to the `dataChanged` signal if the state of the stored control has changed.

6.2.3. Input models

Input models allow to modify the controls of the given `InputControlContainer`. All the input models are flat (compared to the `ControlModel` which is hierarchical). The common funtionality of input models is implemented in the `InputModel` class (which is the child class of `TreeModel`) with `InputModelItem` class for items.

The basic extension of the `TreeModel` is the drag and drop handling. Each input model supports three different mime types for dragging and droping its items. The first is the mime type for internal move of items which is defined in the `TreeModel` class. Both drag and drop actions are enabled for this mime type and the implementation is similar to the `TreeModel`. The second supported mime type is the `InsertMimeType` which is used to create new input controls for the form controls specified in the encoded data. This mime type is supported only for drop action. The third mime type `MoveMimeType` is encoded along with the `InsertMimeType` and serves to determine the origin of the drag action. This

has to be done because dragging data which have the `InsertMimeType` between documents is not allowed (each input control is bound to one particular form control of the same document).

App input model

This model allows to acces and modify the associated `AppInputContainer` instance and its controls. The properties of the input controls are accessed through model items (`AppInputModelItem`) and their methods. The properties of the container are accessible by specific methods (`layout()`, `setLayout()`). The model also allows extended handling of undo and redo for its actions by providing additional methods `setMultipleData()`, `setMultipleDataCommand()` and signal `multipleDataChanged()` to ease the operations in the associated designer view as these operations require more functionality than the common interface offers (this will be discussed later in this chapter).

Csv and database input models

These models (implemented by `CsvInputModel` and `DbInputModel` classes) manage access to the associated `InputControlContainer` class isntance. They only extend the access to the stored input control data according to the particular type of the model via overriding appropriate methods in the model item classes (`CsvInputModelItem` and `DbInputModelItem`).

Input filter model

The `InputFilterModel` allows access only to those controls from the given `ControlContainer` which can be used to create new input controls in the given `InputControlContainer`. This model is readonly and flat (compared to the control container which is hierarchical) with just one column. The structure of the model is rebuilt when the source control container or the source input control container has changed. The input filter model is a standalone class and doesn't use any of the functionality provided by previously mentioned model classes.

`FormControl` instances are availible for new input controls only if they don't have any corresponding input control in current input container (acording to the form control pointer). The model distinguishes two types of form controls: mandatory and optional. Mandatory form controls are the controls which have the `FTib::FormControl::isInput()` set to true, the other ones are optional. The types are distinguished using model roles affecting the items' appearance (eg. `Qt::DecorationRole`).

6.3. Paper form editor

For editing of the paper form layout of the form document the paper form editor module is used. It consists of two parts. The first part is the editor of the structure of the form controls, groups and form areas and their properties. This part is based on the Qt model/view framework. Form cotnrols and groups are displayed and edited using the `ControlModel` class along with the `ControlView`. For editing of the form control and form area properties the classes `ControlPropertiesModel` and `ControlPropertiesView` are used.

The second part is the graphics designer of the paper form layout. For this part of Form Editor, the Qt Graphics View framework is used. There is a `QGraphicsScene` containing the information about the layout of the graphical items and a subclass of the standard `QGraphicsView` called `PaperGraphicsView` which shows the scene's contents to the user and handles the modifications of the items. The rest of this chapter will be dedicated to this paper graphic view class.

6.3.1. Paper graphics view overview

Each page in a document (`FormPage`) is represented by the `PaperPage` class in Form Editor. Each of the areas on the page is represented by `FormAreaItem`, all the areas of the page along with its background are placed on the page's `scene()`. The scene is displayed to the user via the `PaperGraphicsView`, which handles moving, copying, converting, removing of the areas and is responsible for saving all the actions into an undo stack (it can also be used without it).

6.3.2. Form area item

`FormAreaItem` is a QGraphicsItem representing single `FormArea` on a graphics scene. Method `paint()`, used for displaying the contents of the graphics item, draws a frame around the area's bounding rectangle and then lets the area to paint itself (the static areas' contents are painted directly even while designing the paper layout). Item's borders are always painted with the same line width in view pixels, this effect is accomplished by associating `ViewInfo` class with the item. This info class knows the current scale of the view. Info class directly associated with the item disables its posibility to be shown by multiple graphics views, but we do not need this (each area can only be on one page).

The form area item is also responsible for resizing itself. It draws resize corners while being hovered and handles all the mouse events during the resizing. While being resized, the item uses detected clipping lines (it uses the info to access the view's `clipLine()` method) and snaps the corner being resized to the nearest detected line.

The `boundingRect()` method returns the whole rectangle occupied by the item (along with its resize corners and possibly overflowing contents of a static paper area), on the other hand the `shape()` method returns only the rectangular frame around the area (without the overflowed contents). The item's `zValue()` is set by the `GraphicsScene::setItemLayer()` method according to the area type, the current application tool and the size of the area (the smaller are positioned above the bigger, so that even the smallest one can be selected in case of overlapping areas).

6.3.3. Paper graphics view

This view graphically presents the contents of a single page (its scene, to be precise) to the user. It is based on the `GraphicsView` class from the GuiCommon package which will be also described in this section. The base view implemetation is used whenever some areas (including the detected ones) need to be shown. All the mouse events are processed by the base view which then calls virtual functions that have simple, usually dummy implementation. These are overriden in the paper view, which has more information about the document, it is aware of controls etc.

The paper view is aware of the fact, that there might be more views showing different pages of one document at the same time. All these views need to be zoomed to the same level, the selection needs to be synchronized etc. The `performZoom()` method is called when the view finds out that the user wants to zoom it in (by scrolling the wheel while holding CTRL), the `zoom()` actually performs the zoom of the single view. The `addItem()` and `convertItem()` create new form controls, `selectOneItem()` and `areaGroup()` are used for handling the selection.

Moving (and copying) the items on the view is implemented by drag&drop, because it is impossible to detect end of an internal item move. Another reason for this was a need for cooperation of more scenes within one document (moving areas from one page to another). The "`application/ft-areas-selection`" mime type is used. Moving the areas is simulated by using the pixmap with all the areas as the drag pixmap and hiding the areas. The areas are set visible (and placed to new positions) after they are dropped.

Paper view also draws its context menu with cursor position (and selection) sensitive actions. These are usually not handled by the view itself but are rather forwarded to the `PaperFormEditor`, which can operate with form controls better.

When moving, inserting or copying items it is useful to allow the scroll area containing the view to be scrolled. The method `scrollTo()` is repeatably called when the user is dragging an item and the cursor position should be visible within the view. The base class just delegates the call to the standard QGraphicsView's function `ensureVisible()`, while the paper graphics view uses the form editor to scroll itself (since it does not have its own scrollbars).

Both the parent and the child view also support unmovable items which are always drawn relatively to the visible region of the view. Those items are used as rulers in calibration etc. The only thing you have to do to obtain unmovable item is to register your item via method `addUnmovableItem`. Now you can

set item pos only by method `setUnmovableItemPos`. This pos is relative to visible part of the view. You can also set, that only one direction will be related to the view and another will be not if you set unmovableOrientations. Finally it is your responsibility to unregister item by `removeUnmovableItem`. The core for this items is method `justifyUnmovableItems`. It has to be called whenever the visible part of the scene changed (notice that it contains not only geometry changes to view but also moving scene).

6.3.4. Synchronizing selection

The selection on the whole `PaperFormEditor` is synchronized by the `PaperSelectionModel`. It ensures, that each of the selected areas' control is selected within the `ControlView`, all the selected controls in the ControlView are present in the `ControlPropertiesModel` and that the selection of areas on the pages, in the `ControlPropertiesView` and the set of areas in the `AreaProperties-Model` are the same.

The class uses a simple internal locking by `lockSelection()`. While it is locked, no signals are processed until the `unlockSelection()` is called. Most of the selections are not updated immediately, the changes are rather kept in a buffer and the changes take place after a timer finishes. This prevents the synchronization from overwhelming the CPU.

6.4. Electronic form and datastore editors

The editors for the individual electronic forms and datastores are also based on the model/view architecture. Each editor is represented by the child class of the `InputFormEditor` class. The editor allows modification of the given `InputControlContainer` using graphical designers or property editors (see Section 6.7, “Property handling”). The individual editors are very simmilar so the parent class which contains almost all the required functionality. The particular editors should just provide the main designer view, the target input container and the main data model. Therefore the addition of another input editor type is very simple.

Each editor consinsts of four parts. The main part is the designer of the container controls. It allows arranging of the target controls in a container with respect to the type of the container. The `InputModel` child classes are used as models for this designer. As for views, the standard Qt view classes are used. However, there is one custom designer view for arranging the application input form called Electronics form designer which will be described in more detail in the following subsection.

The second part is the source list of the avalible form controls that can be used to create a new input control. For this the `InputSourceModel` and `InputSourceView` are used.

The third part is the list of all created input controls along with some basic properties and associated form control indentifiers. This is implemented by the `InputModel` child classes for model and standard Qt view classes.

The last part is the editor of properties of the current container and the selected input control. For this the classes `PropertiesModel` and `PropertyItemDelegate` are used.

6.4.1. Electronic form designer

The implementation of this designer is split into two parts represented by `AppInputViewTableHolder` and `AppInputTableView`. The first one holds the layout of the table, which means that it remembers the merged cells, etc. The latter one is responsible for interaction with the user, undo and also creating paper-like layout. We will start with first class which is the core class for table manipulations.

Table manipulation

As we proposed in the previous paragraph, the class which is responsible for table manipulations is `AppInputViewTableHolder`. The class itself stores the information about cells in `Cell` objects. However its interface enables the caller to operato on areas consisting of merged cells. Such interface is

more comfortable to work with and hides the manipulation with the cells which might be potential source of errors. The main invariant which holds during all the operations is that all areas (merged cells) need to be rectangular. Although this class provides a powerful interface, it is caller's responsibility to ensure that none of the operations break this condition. Later, we will show a way how to check whether a sequence of operations is correct in this sense.

Implementation details

As mentioned before, we internaly store a table of cells. Every cell (class `Cell`) remembers whether its' bottom and right border is a break (methods `isRight`, `isBottom`, `setRight` and `setBottom`) and the proxy widget (`QProxyWidget`) placed in this cell if any. The situation with the widgets is a little bit harder — for every merged area, the widget is stored in an arbitrary cell of the area. The main problem is, that the cell is not the owner of the widget and it cannot delete it — that means that before deleting associated proxy widget you have to set another parent to it. Another problem is when copying this class — it would not be clear who the owner of the proxy widget is and thus nobody would deallocate it. However we provide quite a nice solution to this problem.

We realized that usually we do not need the copy of proxy widgets — it is enough to copy just the layout and a sign that any area is not empty. This is done by method `dummyCopyFrom` which copies information about borders between cells and instead of copying original proxy widget it stores dummyWidget to it. This is enough for all the the cases where we need to copy the table. If we want to copy geometry from this table we do it by calling the `applyDummyGeometryTo` function. Well, the good question is, why do we need to copy the table? The answer is, that it is useful for checking whether our operation is correct or not. We do a dummy copy, try to do the proposed operation and finally we check whether the newly obtained table is correct (using e.g. method `isCellAreaRectangular`). If so we perform the proposed operation on the original table. Notice that we do not need original proxy widgets for layout operation, it is enough to know, that there should be one.

However if we need a copy of the part of the table (e.g. when moving some part of the table to another place) we can do it. But we can do it only inside the table. This is done by functions `moveCellsRectToBuffer` and `moveCellsRectFromBuffer`.

Caller interface

The work with table is encapsulated and is done via methods operating on areas, like a `mergeArea` or `clearArea`. Notice that every merged area is uniquely defined by an arbitrary cell inside it and thus all such function work either with coordinates of one such cell or by rectangle which bounds some cells inside the table. The interface also offers functions for inserting and removing rows and columns. You should be a little careful when using those for removing rows and columns (`removeCol`, `removeCol` etc.). Before you remove the cells you should ensure, that all the widgets that could disappear by removing these columns or rows are deleted before. This can be ensured by calling `removeAreaWidgets` . On the other hand removing and also inserting the row or column does not break the invariant of rectangularity and ensures you that if you insert in the middle of a merged area, this area will not be split.

Table view

There is nothing interesting about the class `AppInputTableView`. Just tons of code which can be easily understood from doxygen comments. The class handles both the undo commands, the interaction with users, writing the changes to the model (to ensure that other view can notice the changes), and finally it can do an automatic layout which will be described later. The class itself is based on the QGraphicsView, and it is responsible for both creating our own layout designer (with all the functionality) and displaying all the widgets within it. However, the price is big. We have to handle all the painting, the user interaction and the undo commands ourselves. On the other hand we have absolute freedom in the layout and the result is quite good.

Automatic paper-like layout

As we mentioned before, the `AppInputTableView` class is able to layout the controls so that these look similar to the positions of their areas on the paper form. This is done in the `createPaperLike-`

`Layout` method. The controls can be spread over cells to occupy more rows and columns (but only in a rectangular area), all the rows have the same height, and we want to create the layout using the smallest possible number of columns. To make it easier we consider only the first area from every control.

Now we can assume that we have a bounding rectangle of every control so we can try to split them into rows. First we start with estimating the row height (`estimatedRowHeight`). We do it simply by bucketing and take the height which is the most frequent. Then we start to divide the controls into rows using sweepline (`distributeControlsIntoRows`). Notice that it is independent to the distribution of to the columns. Now we take the first hypothetic row from the top of the page and we look whether there is any control which intersects with this row. If there is not such control we move to another row. If so we take all of them and for each control which has no intersection with other rows we adjust the top of the row. Then we take again all the controls and we adjust the bottom of the row. Then we go through all the controls third time and if they intersect with this row we add them to this row (which means either that they start in this row or they should be expanded to this row if these are multiline.

Now we can continue by distributing the controls into columns (`distributeControlsIntoCols`). At first we build an oriented graph from the controls in such a way that if control A is in the same row(s) as B and if the left border of A is smaller than left border of the B then there will be edge from the A to B. Notice that such a graph cannot contain cycles so we can do topological sorting. Now in this sorted graph we find the longest path — the length of this path will be the number of the columns in the table. The only task that remains is spreading the controls horizontally if possible. It is done in the `spreadColumns` method. For each control in the graph we take all its children and find the maximum from their columns in the rows and remaining columns to the right minus the longest way from the control to a leaf. Using this algorithm we can spread this control.

Finally we just split every column to two because we need to add a label to each widget and we are finished.

6.5. Toolbar action management

Form Editor has many states, not all the actions on the toolbar apply to all the states, thus there is a need to enable or disable the actions and their targets. Whenever an editor specific action is triggered it is handled by the `FormToolsEditorWindow`'s `performEditorAction()` method which calls the current `FormEditor`'s `performAction()` method. This method takes two parameters, the action type and the parameter parameter of the action. Each of the form editors then defines a set of supported action types, when the current form editor changes, the actions associated to the new editor's supported action types are enabled, the other ones are disabled. There is a singleton class handling action states of the main application window called `MainWindowState`.

There are two exceptions to this rule, the first one is the undo actions. The `MainWindowState` connects the undo/redo actions directly to the undo stack of the current form editor. The second exception is the bunch of copy/paste/cut/delete actions which are not targeted to the whole form editor but always refer to the `focusedWidget()`. The focused widget might be of any type, to handle this we define the `copyPasteActions` Q_PROPERTY in all the widgets that support these actions, reading it returns the supported actions and writing to it performs the given action.

There are also actions which do not make sense to be triggered when there is no document or when the current document is in error state, these are also handled by the `MainWindowState` object.

6.6. Error handling

Form Editor tries to handle all the errors in a soft way, allowing to edit the document with errors (even to save a document with errors) and to allow delayed solution of those errors. To be able to do that, the opened document has its own error handling logic (see chapter Section 2.2.3, “Form document”). The role of Form Editor is to track and display these errors.

Tracking of the errors is done in two ways. Firstly, all model items track errors of their form controls and display them using model roles (`Qt::DecorationRole` for example). The first solution of tracking

errors is implemented in all input models by catching the signal `changed()` of the associated `InputControl` instances. In the `ControlModel`, the changes of errors are passed from outside of the model by calling `errorsChanged()` on the target model item.

The second way of tracking errors in Form Editor is the overall list of all errors in the `FormDocument` instance which is accessible by method `errors` and can be modified or further inspected by specific methods. The changes of errors in the document are announced by the `errorsChanged` signal. The error changes are then further propagated using the `errorsChanged` signal of the `EditorsWidget` class. The list of errors is then displayed in the bottom right corner of Form Editor using the `ErrorsWidget` class. The owner of the errors widget (which is the single `FormToolsEditorWindow` instance) forwards all error changes by connecting the slot `errorsChanged` of the errors widget to the previously mentioned signal of the editors widget.

Form Editor allows user to highlight the item with errors by clicking on the appropriate link in the errors widget. This is implemented by connecting the signal `highlightItem` of the errors widget to the samenamed slot of the `FormToolsEditorWindow` which passes the call to the currently active editors widget. The editors widget then finds the target editor which contains the item to be highlighted and forwards the call. Form Editor then highlights the item according to its type (the target editor finds index of the item in the current model and enforces selection of this index in all the associated selection models). All methods in this call chain are named `highlightItem`.

6.7. Property handling

Since all the elements of the form document have their properties stored in the `FormProperty` class instances and implement the `FormPropertyContainer` interface to allow uniform access, Form Editor implements unified way of displaying and editing these properties in so called property editors. The property editors are based on the Qt model/view framework. For this purpose the `PropertiesModel` class is implemented which allows to edit these properties in common Qt views. There are some specialized child model classes of the `PropertiesModel` and some specialized views since some controls require specific handling of their properties.

6.7.1. Properties model

The `PropertiesModel` class implements the standard Qt item model interface to allow editing of form item properties in common Qt views. The model tracks property changes of the selected form controls and allows access to their values via the `QAbstractItemModel` interface. All changes of the properties are undoable. The model is based on the `TreeModel` with the `RootPropertyItem` class and `BasePropertyItem` classes for model items (the second one is further extended by child classes). The model is a single level tree with specific root item to which all the methods of the model are delegated. The subclasses of the `BasePropertyItem` are in the leaves of the tree and allow access to the associated properties.

Each properties model is only available for editing (and displaying) a specific set of properties, which needs to be expicitly set by the `setProperties` method before any other methods are called. Some of the properties can also be set as read-only, which is achieved by passing the identifiers of those properties to the `setReadOnlyProperties` method of the model. After the model is given the supported properties, it creates a `BasePropertyItem` for each of them. The model allows to display and edit properties of multiple controls at once. The current set of controls is set by the `setItems` call. Only the properties shared by all the current controls are displayed (the other properties are hidden) and if the values of some property differ then the list of availible values is displayed in the model.

To increase the usability of the property editor based on this model in the situations where the set of the selected controls changes frequently, there is an option to set the timeout for refreshing of the model. This basically delays the visibility of the item changes when the selected item set is modified by ther user. The timer is set every time a property change of some selected item occurs. The refresh timeout length is accessible via `refreshInterval` and `setRefreshInterval`. The timer can be cancelled by setting its value to 0. The refresh of the current property values can be enforced by calling `refresh`.

Property model items

Each property of the selected comtrol is edited in one row of the model via the `BasePropertyItem` item class and its subclasses. The base property item contains the implementation of data accessor methods `data` and `setData` for simple properties (eg `bool`, numeric properties, `QString` based properties or enumerations of strings). For more complex properties there is a `ExtenderPropertyItem` class which allows the property to have children items. It is inherited by two subclasses, the `RectPropertyItem` and `FontPropertyItem`, both defining simple subproperties (like width and height or point size).

6.7.2. Property view and item delegate

To display the contents of the properties model, the `PropertiesView` class is used along with the `PropertyItemDelegate` as a delegate class (using delegates is a standard way of extending a view's editing capabilities, they allow customization of drawing and editing of particular model indexes).

The properties view is based on the standard `QTreeView` view and adds the ability to preserve expanded items when they are removed and inserted back again (occurs often in the properties model as the items are selected and deselected frequently). This behavior can be suppressed by setting the `setPreserveExpanded` to `false`.

The `PropertyItemDelegate` is a standard implementation of the `QStyledItemDelegate` interface. It allows to edit the properties in the properties model in the custom widget `PropertyEditor`, which adds a dropdown list with all the distinct values of current items and an icon for editing the property value in extended editor if needed (e.g. a program). The property editor creates an inner widget for edditing the value depending on the type of the edited property. The distinct values of the property can be displayed using `showHints` method. The property editor allows to open an external editor for editing the associated property using the dedicated methods `openProgramEditor`, `openPatternEditor`, `openTextEditor` and `openFontEditor`. The last importatnt feature of the property editor is the `focusOut` event handling. By default, the editation of an item is commited when another item in the view recieves focus or ENTER is pressed. This leads to ignoring the changes when the save action was chosen from the menu or the editor looses focus. Thus the editor handles the `focusOut` of its children (the real editor) by installing an event filter to them and processing their events in the `eventFilter` method.

6.7.3. Custom property models and views

Some items of the document require special handling of their properties (eg. `PaperArea` or `FormControl`) and therefore a specialized child classes of the previously mentioned property model and view are implemented.

Area properties model

The `AreaPropertiesModel` extends the behaviour of the `PropertiesModel` in such a way, that setting the data of an area also saves their selection and restores it when the command is undone. This is implemented by overriding the `newSetDataCommand` method and adding a `PaperSelectionModel::SaveSelectionCommand` as a parent command of the default set data for the properties.

Control properties model

The `ControlPropertiesModel` is a model class for both editing the properties of the selected `FormControl`s and showing the list of areas of those controls. It is based on the properties model and handles the lines with the areas itself (opposed to the other models it does not use items to represent the areas). The areas can be dragged from one control to another and thus change the parent controls, this is also handled in the model.

The list of current controls can be modified using the `setControls` and `changeControls` methods. Dragging the areas is implemented using the Qt drag and drop mechanism in the `mimeEncode` and

mimeDecodeAll methods. The model allows to undo the move of the areas. This is done by using the ControlPropertiesModel::MoveAreaCommand undo command class. While undoing the data changes, the previous selection is restored and the properties model looks the same as before the action. The selection model used to store the current selection before making the requested change in the model data is set by the setSelectionModel method.

6.8. Script Editor

There is a JavaScriptEditor for editation of scripts to make writing the controls' code more comfortable and easy. The editor is based on the QTextEdit class which is usable for rich-text editing. It provides features like syntax highlighting, word completion and automatic indentation.

JavaScriptHighlighter class is used as a highlighter for the script editor. It is based on the QSyntaxHighlighter class, thus it highlights text based on the matched regular expressions (it does not have a parser). It uses a list of HighlightingRule objects (each of them associates a pattern with a format) which are applied in predefined order to the input text.

JavaScriptModel object can be associated with the script editor. Then the editor uses the model for word completion of the currentWord(). The model has a predefined set of javascript keywords and a pointer to a ControlModel containing the hierarchy of the form controls and groups. It maps this structure to the script (separating the hierarchical names by '.' character) and adds function names (as given by the controls) as the leaves of the model.

The editor handles each key separetely in keyEvent(). According to the key and the it decides whether to just type the key, show the completer (where the JavaScriptModel is loaded) or whether some more characters need to be inserted in case of automatical indentation. The autocompletion is a simple heuristic where indentation is increased after opening curly bracket { and it persists until next closing curly bracket }. There are also single line indented blocks after line ending on a closing parenthesis) or an else command.

The editor emits signal positionChanged() whenever the position of the cursor within the document changes and lineCountChanged() when the number of lines has changed. This allows other widgets to display line numbering. The editor itself can show the cursor position on a label if the label was previously set by the setLabel() call and a friend class JavaScriptVerifyButton can be used to verify the code within the editor. The button after being clicked shows the verification result and in case of an error sets the cursor position to the error position.

6.9. Script Debugger

To make writing of control programs more comfortable and clear, Form Editor offers a simple debugger form these programs. The main feature of the debugger is that the user can see all the programs of all defined widgets in one place. The debugger consists of two parts. The first one is the script editor for the merged program code. The second part is the designed application input form (this form is hidden when no application input is defined). The debugger is implemented by the class ScriptDebugger.

The debugger uses the methods program and setProgram of the current form document to extract all programs from all controls to a single string or store it back from a given string. The code is than loaded into a DialogEvaluator instance which evaluates the data filled in the application input form.

The application form is represented by the class FormHolder (the same class is used also in Filler). The form holder class creates editor widgets according to their definition in the given AppInputControlContainer. The programs of form controls have to be executed each time when some of their input values changes. Therefore the evaluator is called every time a value of a form field changes. The fieldValueChanged signal of the form holder is connected to the valueChanged slot of the debugger which forwards the calls to the dialog evaluator. The debugger has its own event handler class ScriptDebuggerEventHandler for handling evaluator events which forwards all calls to the form holder instance.

The field values filled in by user are accessible via methods `data` and `setData`. This allows the owner of the script debugger to restore the previously filled values when de debugger is opened.

The debugger also allows print preview of the filled values using the class `PreviewDialog`. The input data for the preview dialog is computed from input form field values using `ModelEvaluator`.

Chapter 7. Filler

Filler is a simple Qt application that allows user to fill in the data of the opened Form Tools project and store it into the defined data storage or print it (either to a printer or into a file). [1]

7.1. Overall architecture

Filler is based on the Qt model/view architecture. The main reason is that this architecture allows an uniform access to the edited data from standard (possibly multiple) Qt views. This is exactly the case of Filler where the data are edited using two views, a table view and a special view representing the digitalized form (this view is based on the `FormHolder` class). It also allows centralized evaluation of user defined programs.

At the bottom of the architecture there is a data storage allowing loading and saving of the stored data. The data storage is used by the central data model (one data storage at one time, but can be switched). The model uses evaluation of programs and data conversion provided by classes from the Lib module. The loaded and evaluated data is than displayed in the data views according to the user definition of the digital form.

Filler is an Qt based application with a main window. The main window is designed using the Qt Designer and the logic is implemented in the class `Filler`. This class contains all child widgets and objects forming the application and handles signals of the designed UI. It also connects the main modules and is responsible for loading and saving the project files and data.

Main modules are the following:

- Data storage classes.
- Classes for evaluating and converting the data.
- Data model class.
- Classes form viewing the data.

All classes in this module are in the `FTFiller` namespace.

7.2. Data storage

The base of the application architecture is the data storage. It allows to load and save data of the opened Form Tools project. The basic interface of a data storage is defined in the class `DataAdapter`. Each data adapter has a `InputControlContainer` pointer to get the detailed format of the stored data.

The data in Filler are viewed as a list of records (each item of a record contains value of the associated form control, the mapping of controls to record items is defined in the associated input container). Therefore the interface defines methods for modifying these records such as `selectRow`, `updateRow`, `insertRow` or `removeRow`. It also defines methods for single record item manipulation where the record items are identified by the `jid` of the associated form control. The methods `commit` and `loadData` should save and load the stored data.

Currently there are supported two types of data storage, the CSV data storage and database data storage, but the application can be easily extended by implementing the `DataAdapter` interface.

7.2.1. CSV data adapter

The `CsvDataAdapter` class implements the defined `DataAdapter` interface to allow storing the form document data into a CSV file. The format of the CSV file to be saved or loaded is defined in the given `CsvInputControlContainer` including delimiter, quote or encoding.

[1] The project contains path to a form definition file (.ftd), the selected application input and definition of data storages.

All the data of the CSV adapter are stored in the internal cache. So when the file is loaded, the CSV records are parsed according to the defined file format and stored into memory. On commit the cache is flushed into the given file.

7.2.2. Database data adapter

The `DbDataAdapter` class implements the defined `DataAdapter` interface to allow storing the form document data into a SQL database. It uses the `QtSql` framework to access the database. All the information needed to connect to the target database is taken from the project file and from the given `DbInputControlContainer`.

Each record of the data is represented by a single row in the database. Except the database columns for individual form controls, there is a primary key column (its name is defined in the input container). The data adapter maintains the mapping from the primary key to the index of the record in internal storage and vice versa (all the loaded and created data are stored in the memory of the data adapter). All the database columns representing the form document data should have type `string` (integer based values can have specifically typed columns) and the primary key is expected to have type `integer` with autoincrement.

The requested changes are performed on the internally stored data. Using the `commit` method the SQL commands will be generated and executed in a transaction (the commands will update the database according to the changes made in the program). To allow that, the data adapter stores the set of inserted, removed and updated data records. For generation of the SQL commands the `DbInputSqlGenerator` class is used.

The adapter tries to create the associated database table on load. If the operation fails, it is marked as error but the adapter continues in loading. This can be adjusted using the method `setCreateTable`.

7.3. Data evaluation and conversion

Filler uses evaluation of javascript scripts to validate and compute document data. It also allows to store the data in one format (eg. date format strings, bool value names etc.) and edit it in another format. Main classes for both of these operations are defined in the shared Lib module (see Section 2.3.2, “Program evaluation” and the section called “Data conversions”) but Filler has to define its own helper and adapter classes.

The data evaluation and conversion is performed in the `DataModel` class (will be described later in this chapter). This section contains helper classes of Filler that are used to do this.

7.3.1. Data flag map

The `DataFlagMap` class represents a matrix of flags. The individual flags can be accessed by their row and column position. The class offers methods to modify these flags and to change the size of the flag matrix. The flag map is used to store flags of the loaded data which is managed by the `DataModel` class.

Filler uses two basic flag types. The `enabled` flag and the `valid` flag. The map contains specific methods for handling these flags. It also contains methods for locking as it is accessed from multiple threads.

7.3.2. Data validator

The `DataValidator` class allows to run the evaluation of data rows using a `DialogEvaluator` in a dedicated thread, thus the user can still interact with the application. It receives the pointer to an evaluator and pointer to an event handler and starts a thread which repeatedly calls the evaluator.

The validator contains methods for starting and stopping the background evaluation or for adding or removing the list of rows to be evaluated. It provides methods for tracking the progress of the validation using signals `rowValidated` and `progress`.

The evaluation thread is implemented by the class `DataValidator::ComputeValidityWorker` which invokes the evaluator in its `run` method.

7.3.3. Input convertor holder

The `InputConvertorHolder` class allows to convert the value of a control from the format used in the datastore to the format used in the form and vice versa. Given the input container corresponding to the form used for editing and the input container corresponding to the adapter, the convertor holder allows to convert the given data back and forth from the format used in the form to the format used in the data adapter. In fact, the convertor holder is just an adapter over two instances of the `InputConvertor` class.

The convertor holder has to handle the situations when the conversion to the format of the form fails. The holder has to return different 'error' value for string controls than for other types of controls since the string controls with errors can in general contain an invalid value (compared to other types of controls which are edited by special editors and which cannot contain arbitrary data).

7.4. Data model

The `DataModel` class encapsulates a data adapter instance to allow uniform access to its data. It also adds the undo and redo functionality for all operations over the underlying data adapter. The another important role of the model is to evaluate and validate the stored data and perform the format conversion.

7.4.1. Overview

The data model implements the common Qt interface `QAbstractItemModel` and allows access to the data stored in the underlying `DataAdapter`. The model maps the list of data records represented by the data adapter to the table of model values. However, the order of the model columns corresponds to the order of controls in the given `AppInputControlContainer` rather than to the order of the adapter columns. The underlying data adapter can be changed using the `setAdapter` and the target application input container can be changed using `setAppContainer`.

The model supports undo and redo using `QUndoCommand` mechanism. It defines an undo command for each modification of the data. The methods for changing data are then implemented in three steps. First the public method is called. It will create the appropriate command object and push it onto the given `QUndoStack`. The command object then calls the internal implementation of the action with the stored arguments in its `redo` and `undo` methods.

7.4.2. Data conversion

Filler allows to store data in one format and edit it in another format. This is done by converting the data from one format to another upon request to the data model. The format of the values is part of the `InputControlContainer` definition (both the data storage and the form used for editing are defined by an input container). The data of the model is stored in the format defined by the input container of the associated data adapter. The data than can be accessed either in the format of the application form using the role `Qt::EditRole` and the original data in the data adapter format can be accessed using the user role `DataModel::UnconvertedRole`.

Converted values are used for editing the data in the form defined by the given `AppInputControl-Container`. For converting the values the helper class `InputConvertorHolder` is used. It receives the input container corresponding to the form used for editing and the input container corresponding to the underlying data adapter an performs conversion between these two formats.

7.4.3. Data validation and evaluation

The model also ensures the evaluation of the stored data (which means also their validation). There are two types of evaluation in the model. The first type is direct evaluation of specified model rows.

The second type is the evaluation on background in the dedicated thread. For each type of evaluation there is a dedicated data evaluator. For the direct evaluation the `ProgramEvaluatorAdaptor` class is used. For the evaluation on background the `DataValidator` class is used. Both these classes are adapters over the `DialogEvaluator` class from the shared Lib module (see Section 2.3.2, “Program evaluation”).

The validation results from both evaluators are stored in the `DataFlagMap` instance which maintains the `enabled` and `valid` flags for each value of the model. The flags are than accessed when one of the views displays some data (compared to the computed values which are displayed only by the form view). In case of the table view the order of requests is not predictable so, to achieve maximum performance, all the computed flags have to be stored in memory using the flag map.

Direct data evaluation and event handling

The direct data evaluator is used to evaluate a newly filled data. It requires the input values to be in the format specified in the current `AppInputControlContainer`. Therefore the input data for evaluation (which are read in format specified by the datastore input container) have to be converted to the required format using the `InputConvertorHolder` class.

Other objects can register themselves to the model to receive notifications from the direct evaluator. These objects have to provide their event handler which implements the interface `ModelProgramEventHandler`. The object is registered by using the method `registerEventHandler` and unregistered by calling `removeEventHandler`.

The direct evaluator uses the `DataModel::DataModelEventHandler` instance as its event handler. This event handler provides the evaluator with the input data which are converted using the given `InputConvertorHolder`. It also stores the results of the validation in the data flag map. This event handler also forwards all appropriate method calls to the registered model program event handlers.

The model itself registers a model program event handlers. It is implemented by the class `SetValuesEventHandler` and allows user to get directly the available values for an enumerable field (eg. combo box or list widget). These values can be computed using `getValues` method of the handler which invokes the evaluator and caches the data for the specified field. This is used while editing the data via table view. The available values for an enumerable field can be accessed using the user role `DataModel::AvailableValues`.

Background evaluation and event handling

The background data evaluation is used to validate the larger amounts of data while allowing the user to interact with the program. This evaluation requires the data format specified in the input container of the given data adapter. Since the data is stored in the required format, no conversions are needed.

The background evaluator runs in a separated thread and uses the `DataModel::DataValidatorEventHandler` instance as its event handler. The event handler tracks only changes of validity and availability of the evaluated items and stores the results into the given `DataFlagMap`. The access to the flag map is synchronized.

To ease the manipulation with the evaluator thread, the class `DataValidator` is used. The `rowValidated` signal of the data validator is connected to the model which emits the `dataChanged` signal.

7.4.4. Copy and paste

The model allows to copy and paste its data. This is implemented using the Qt drag and drop mechanism. The selected data are encoded into a `QDrag` instance and then dropped on the selected position. The row indexes of the dragged model items are encoded as an offset of an encoded row from the topmost encoded row. The columns are stored as absolute values. This is implemented in the `mimeData` method. The model supports two ways of dropping data.

The first option is to overwrite the existing data in the corresponding model cells. The row indexes are computed as the index of target row plus the encoded offsets of the items. The column indexes remain

the same as encoded because each column has a specific meaning and there is no use of copying data between columns. This option is implemented in the `dropMimeData` method.

The second option is to drop the data into newly created rows. The row indexes of the new rows are computed as the target row plus the sequence number of the encoded data row. The column indexes also remain the same as in previous case. This behavior is implemented in the `dropMimeDataInNewRows` method.

7.5. Data view classes

Since Filler is based on the Qt model/view framework, it also has to provide ways to display and edit the data. There are two ways to edit data. One way is to use the central table view, which uses the `TableDelegate` delegate class. The second way is to use the custom `FormView` view based on the `FormHolder` class.

7.5.1. Form view

The `FormView` class is implementation of the `QAbstractItemView` interface using the `FormHolder` class. This view (unlike in other standard views) can display and edit only one row of model data at once. The row is edited in the form represented by the form holder instance. Simply put, the form view adds the model/view functionality to the `FormHolder` class. The current row can be changed using the `setIndex` method.

The form view tracks changes of the model data and fills the changed data into the form holder. It also tracks the key events of the form holder's widgets. The view checks whether the specified shortcut for moving to the next column or row was pressed and if so than it moves focus to the next field or skips to the next row.

For handling the evaluation events from the model the view offers an implementation of the `ModelProgramEventHandler` interface called `FormViewProgramEventHandler`. This handler basically forwards all calls to the appropriate methods of the form holder.

7.5.2. Table delegate

For editing the data in the table view there is an item delegate `TableDelegate` class which is responsible for creating appropriate editors according to the target format of the data. The target type of the field is available by `DataModel::ControlType` user role. For editing the enumerable fields the available values are retrieved from the model using the `DataModel::AvailableValues` role. The textual fields are edited by a line edit and stored in the format of the datastore (ie. the format of the data adapter). Other types of fields are edited in the corresponding editor widget and their data are stored in format of the form (ie. the format of the current application input container).

Chapter 8. Final notes

In this chapter we want to compare Form Tools to existing software, describe the process of the development run and finally we want to show our future plans.

8.1. Comparison to other tools

All the tools we know about are missing some of the functionality provided by Form Tools. If we take functions of Form Tools one by one there is no other software which supports creation of digital form based on the paper original. In the other tool you can draw a form, but they do not help you in any way. Form Tools on the other hand enables you to create the form directly from the scanned picture and you do not have to measure every area in the form to obtain an acceptable result. This is done for you automatically or almost automatically.

Also the possibility of writing your own functionality is quite unique. There are of course some other programs supporting this (PerfectForms[1]), however system developed by us is better in many ways. The first advantage is that our programming language is JavaScript which is well known and also well documented. We also find better if you write the code than draw it. But it depends on the users. We want to point out that we use declarative style for the user programs. This saves you hours of coding. You do not have to write reaction on any change in your form, all values are automatically recalculated according to their program when necessary. This is the main difference to other tools. Speaking about the dynamic behavior of the form controls, there is a great thing in Form Tools called Script debugger, where you can see and edit programs of all the controls along with a live preview of a form, which you can fill in, test your code, modify the code and test it again and again.

The other important difference to other tools is that we support printing to the preprinted paper forms. Again there are some other programs (FORMstudio[2]) but they only provide a database of prepared forms and you cannot create your own. In addition, in Form Tools you can provide your own datasource and perform batch printing which can be very useful for companies, where you typically collect data for a longer period and when you are finished you can print them.

Quite unusual but useful thing is that we provide DLL with API for displaying and printing the forms created by our application. Thus you can use it in your own application.

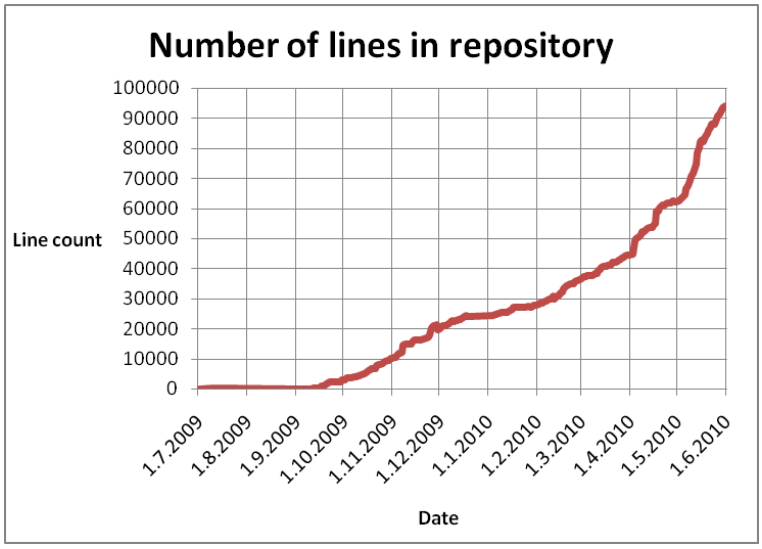
Finally, other tools commonly support export to HTML format but we can also export a PHP script or a project file for our executable application (Filler). Filler is also capable of managing and printing the collected data. Our data format is open, thus you can easily incorporate the exported PHP scripts into your web pages and add dynamic forms to them very easily, the collected data can be saved to a CSV or to a database.

8.2. Development process

The Figure 8.1, “Number of lines during the development” shows how much time we spent just thinking about the application architecture, its typical usage and potential users. It also shows that the development speed was increasing. The first reason for that is that we were not so familiar with the Qt library. The second reason is, that there were many dependencies among the modules in the beginning. So one had to wait until another member of the team finished some important part and vice versa. When the basic functionality was implemented, each developer could start working almost independently on the others, so the development speed has increased. Unfortunately there were some moments when some parts needed refactoring, because new requirements had to be solved. The big part of the project also consists of comments in the source codes and the documentation which were both developed in the last month which lead to the rapid increase in the number of lines.

[1] http://www.perfectforms.com/
[2] http://www.formstudio.cz/

Figure 8.1. Number of lines during the development



8.3. Development distribution

Very important decision was how the work on the project should be distributed among the members. The distribution which was proposed in the specification was quite good and no big changes were done. On the other side we underestimated the complexity of some parts, for example the Lib module is much more complex than we thought. Also the Editor provides more features than we planned.

Although the precise borders do not exist, tha overall distribution of work looks like this:

Jan Bulánek	Whole OCR module including its GUI and tools for image manipulation in the GuiCommon module, electronic form designer and paper layout to electronic layout convertor. Doxygen comments, both the user and the programmer documentation of the OCR and form designers.
Zbyněk Falt	Almost everything in the Lib module — classes for manipulation with form document, program manipulation (the business code of script debugger), evaluation and data conversion, printing the document to the scene, scanner support, exporting to the PHP and HTML and the pattern designer widget in the Editor. Doxygen comments, the manual in the user documentation and the whole chapter about Lib module in the programmer documentation.
Lukáš Ježek	The major parts of the Editor, including the base classes for all models, graphics view and items for the paper design, the javascript editor and overall functionality. Lukáš also developed the major part of the GuiCommon module including the preview dialog, print dialog, and a little pieces of the Lib module, like global application settings. Doxygen comments, the overall structure of the user documentation, the tutorial and advanced topics, some minor paragraphs in the programmer documentation.
Jaroslav Kezniki	The Filler application and things related to the electronic form design in Editor application — input designer support along with CSV and DB storage designers, additional works on Editor such as reporting of document errors, the script debugger allowing to fill in a form in Editor etc. Doxygen comments, the structure and the major part of the contents of the programmer documentation describing the Editor, Filler and the GuiCommon module .

8.4. Remarks to the chosen solutions

8.4.1. Development tools

The most important decision which was done in the very beginning was the choice of the programming language. We chose C++ because we all were familiar with this language and we expected some performance issues in the image processing. From this point of view it was a great choice since all the parts of our application are quite fast. The GUI things are implemented using the Qt library. It turned out that this library contains a lot of great stuff which saved us a lot of work. Also its concept of SIGNALS/SLOTS is widely used in the entire application and we appreciated it. And although we primarily developed this application for Windows, it can be run on Linux (but it is not tested so well). Finally we are developing in Microsoft Visual Studio using Git as a repository.

8.4.2. Architecture decisions

All major decisions were commented in the comparison to other tools and we think that this is the best critical ranking. There are of course some things we want to improve in the future, but after nine months we developed a tool comparable or even better than some of the professional tools. Thus we think that all of our major decisions were good.

8.5. Future work

We plan to work on the project in the future and we know that the we need to improve the data management in Form Filler a lot. We also want to add some features to the Input Layout Designer, static objects being the most crucial one. Further we want to export to more form formats (e.g. Excel sheets or Fillable PDF) and support more storage formats (e.g. more common databases). The last thing is a little bit connected to the business model we plan — we want to divide Form Tools into more applications each aiming at smaller groups of possible customers.

Chapter 9. Contacts

More information about the Form Tools project can be found at http://formtools.jezci.net/. If you have any doubts or suggestions about the functionality of our programs, do not hesitate and send us an e-mail to `formtools@jezci.net`.

Used software components

[QtLib] Copyright © 2008 — 2010 Nokia Corporation and/or its subsidiaries. *Qt — A cross-platform application and UI framework*. `http://qt.nokia.com/` .

[TWAIN] Copyright © 1992 — 2010 TWAIN Working Group. *TWAIN — Standard for image acquisition devices*. `http://www.twain.org/` .

[SANE] *Scanner Access Now Easy*. `http://www.sane-project.org/` .