University of West Bohemia

Faculty of Applied Sciences

Department of Computer Science and Engineering

Master Thesis

Tool for Automatic Generation of Graphical Templates for Mobile Devices

Pilsen, 2014

Jakub Krauz

Declaration

I hereby declare that this master thesis is completely my own work and that I used only the cited sources.

Pilsen, May 10, 2014 Jakub Krauz

....

Acknowledgement

I would like to thank to my thesis supervisor Ing. Petr Ježek, Ph.D. for his assistance, time and valuable remarks.

Abstract

Data management systems usually consist of a database and a user interface which provides access to stored data. User interfaces are mostly web-based, providing an easy access using a web browser. In addition, applications for mobile devices, which communicate with the system through web services, are sometimes offered. The common shortcoming of all such user interfaces is the strong dependence on the underlying database structure. They are designed for a specific system and cannot be easily reused for another one. The aims of this work are to solve this problem by providing a framework that is able to generate platform-independent graphical templates for an arbitrary database structure. The implemented tool is driven by an annotated data model. A use-case deployment is presented on a database for neuroscience experiments.

Keywords: graphical template, form layout, odML, neuroinformatics, EEG/ERP, web service, Java, REST

Contents

Chapter 1
Introduction

Our research group at the Department of Computer Science and Engineering takes part in the electrophysiological research of the human brain. Besides performing experiments in our laboratory we focus on building required computational infrastructure and supporting tools.

Neuroscience research involves data collection, their management and processing. Many data management systems serve this purpose. They usually offer a web-based user interface, which requires a computer connected to the Internet. But the trend of recent years are mobile applications. Nowadays they offer similar functionality to computers, being accessible almost anywhere and anytime. Most people have their mobile phones always with them, ready to use. It brings flexibility and saves time. Mobile technologies are very useful for neuroscientists as well. They bring a quick and simple way to save experimental data which increases efficiency of their work.

The problem with both web-based and mobile applications is the connectedness of the user interface with the underlying database structure. User interfaces are designed for a specific system and cannot be easily reused for another one. If we want to work with two different data management systems, we will need two different applications, each of them designed for the specific system.

The goal of this work is to propose and implement a framework which will be able to automatically generate graphical templates for an arbitrary database structure. These templates must be general and platform-independent to be compatible with web, mobile or desktop applications. It will open the way to design applications interacting with different data management systems and different database structures using a unified user interface. Such applications will be a useful tool for the neuroscience research community accessing various experimental databases. A researcher will be able to submit experimental data to various data management systems with a single application in his/her mobile phone.

This work is structured to chapters as follows. Chapter 2 describes the theoretical background. It introduces neuroscience and electrophysiology research, examines the state of the art in the field of neuroinformatics and reveals weak points. Chapter 3 clarifies the need of a framework for automatic generation of graphical templates and proposes their format. Chapter 4 deals with the design and implementation of a tool which will be able to automatically generate proposed graphical templates. Chapter 5 describes a use-case deployment of the implemented tool within a data management system called EEGBase. Chapter 6 deals with testing the implemented solution, including the tool itself as well as its sample deployment. And finally, Chapter 7 evaluates achieved results and suggests future enhancements.

Chapter 2
Theoretical Background

Neuroscience is an interdisciplinary science which applies various approaches to study the nervous system and especially the human brain. One of used methods is electrophysiology which is described in the following section. Modern neuroscience takes advantage of the cooperation with the computer science. This discipline is called neuroinformatics. It is described later in this chapter.

2.1 Electrophysiology Research

Electrophysiology studies electrical activity of biological tissues and cells. These properties are characterized by electric current and voltage changes. In the field of neuroscience, electrophysiology is used to examine brain activity by measuring electrical activity of neurons. The key terms in this area are

- Electroencephalography (EEG), and
- Event-related potentials (ERP).

Electroencephalography, introduced in [Fa05], is a non-invasive diagnostic method used to record electrical activity of a brain. It records changes in the electrical polarity of neurons through a set of electrodes placed on the surface of a head. The electrodes are placed on the scalp either one by one according to a given scheme or as a part of a special cap. The recording is called electroencephalogram. EEG is used for diagnosing seizure disorders such as epilepsy, sleep disorders, degenerative diseases such as Alzheimer's disease or brain damage such as stroke. It can also be used to monitor the brain during surgery or to determine brain death.

Classic EEG measures the brain activity globally and for that reason the electroencephalogram contains a mixture of a large amount of different signals which can be difficult to analyse [Mi13]. Event-related potentials are significant changes in the EEG signal, caused by an external stimulus, e.g. sensory one. The changes are stereotyped responses to the stimulus with typical features detected as characteristic waveforms in the electroencephalogram. Measuring ERP allows to reduce the number of required electrodes, because the measured response appears in a particular part of a brain [Je08]. The resulting record is therefore simpler and better analysable. ERP allows researchers to study responses to various stimuli and their manifestation which plays an important role in the neuroscience research [Je08].

2.1.1 EEG/ERP Experiments

EEG/ERP experiments produce large amounts of data. That includes not only the record itself but also related metadata such as used hardware configuration, description of experiment scenario, information about tested subject, weather and other external conditions. The record has a low value without this information.

According to [Mo14], researchers performing electrophysiological experiments first propose a hypothesis and design an experiment scenario. Next they perform experiments according to the defined scenario and record data and metadata. Finally, they can analyse gathered data, interpret their meaning and publish results. Now comes the problem if the data are not well-described and it is not defined how to store them. They are kept in poorly arranged storages in such cases or can even get lost. A task for neuroinformatics is to provide long-term data storages with an easy-to-use interface. If researchers use such solutions, they will have their data available in the future just as immediately after the experiment.

The complication lies in the fact that metadata produced by various research groups are heterogeneous. But it is very important to share data and knowledge among researchers in order to increase efficiency of the neuroscience research. To facilitate this process it is necessary to develop a standardized format. Although there are several initiatives which address this problem, the current state of the art lacks a widely accepted one. The most important initiatives are discussed in the following section.

2.2 Neuroinformatics

Neuroinformatics is a branch of science standing at the intersection of neuroscience and informatics. Modern computer science enables utilizing acquired knowledge, analysing data and building efficient models in any scientific discipline. The task for neuroinformatics is to support the progress in the neuroscience research by applying advanced computational tools. According to [Bj07] there are three main fields where neuroinformatics is helpful:

- shared neuroscience data and knowledge bases
- analytical and modelling tools
- computational models

2.2.1 Metadata Sharing

Neuroinformatics deals with processing, storing and sharing data and metadata from experiments. This section introduces several transport and storage formats intended to unify their structure and content.

HDF5

HDF5 is the current version of the Hierarchical Data Format[1]. It offers a data model, file format and supporting libraries and tools for storing and managing scientific data. HDF5 is a universal format, which is not connected with neuroinformatics, unlike other formats mentioned below.

HDF5 data model is organized in a hierarchical structure. It provides two basic elements – groups and datasets. A dataset is a multidimensional array of data elements. They are single units of data such as numbers or strings and the dataset object manages their storage and access to them. HDF5 groups are containers for datasets and other groups. They are analogous to file system directories. Addressing objects in HDF5 uses Unix conventions for filesystem path names, e.g. `/org/example` references `example` from the `org` group.

HDF5 file format specification defines how to write the data model to a file by individual bytes. The format is self-describing, i.e. it contains all information needed to reconstruct the original data objects.

1 HDF5 home page: http://www.hdfgroup.org/HDF5

MINI

The acronym stands for Minimum Information about a Neuroscience Investigation. It is not a data format, but rather a set of requirements to electrophysiological metadata. MINI is a standard proposed by the Carmen consortium (see section 2.2.2). It is described in [Gi09].

BrainML

BrainML[2] is an open XML-based format for sharing neuroscience data and metadata. The project started in 2004. Its purpose is to serve as an open and non-formal ontology for neuroscience. The BrainML specification[3] introduces format and set of conventions for data models for neuroscience data exchange. The project also develops a set of protocols for BrainML data transfer.

The BrainML data model is object-oriented, consisting of entities with fields. There are three types of relationships between entities – inheritance, aggregation and M:N link. Data models are described in XML Schema documents.

odML

Open Metadata Markup Language (odML), introduced in [Gr11], is another developing initiative to provide an open and easy-to-use metadata transport format. This project is the youngest from mentioned ones. However, the format is designed to be a generic and flexible format (not only) for sharing metadata in various branches, although the primary use-case was electrophysiology. Because we have chosen odML for our project, it will be discussed in more detail in chapter 3.2 odML: Open Metadata Markup Language.

Above mentioned formats belong to the most promising ones in the field of sharing data and metadata from neuroscience experiments. There is a clear tendency to use open, flexible and human readable metadata formats, often based on XML. Their significant advantage is the independence from specific software tools. The challenge for neuroinformatics is to unify existing efforts, which will increase efficiency of data exchange. A great deal of attention is currently paid to a combination of HDF5 and odML for sharing experimental data and related metadata respectively [Mo14].

2 BrainML home page: http://www.brainml.org

3 BrainML specification: http://brainml.org/xdocs/specification.pdf

2.2.2 Organizations and Databases

International Neuroinformatics Coordinating Facility (INCF)

INCF[4], described e.g. in [Bj07], is an international organization established in 2005 through the Global Science Forum of the Organization for Economic Co-operation and Development (OECD)[5]. Its purpose is to develop the neuroinformatics infrastructure and to coordinate and facilitate the global development of the field. The secretariat of INCF is located at Karolinska Institutet and the Royal Institute of Technology in Stockholm, Sweden. So-called national nodes are located worldwide in participating countries. INCF holds an annual Neuroinformatics Congress.

INCF supports collaboration among researchers through the sharing of data within the project named INCF Dataspace[6]. It is not a single database, but rather a unified interface to various neuroinformatics databases distributed worldwide. It uses iRODS[7] which is an open-source data management software providing functionality independently of storage resources. INCF Dataspace itself provides no storage, research groups link their own storages in the dataspace instead. All linked storages are exposed in a single namespace.

G-Node

The German Neuroinformatics Node[8] (G-Node), introduced in [He08], is the German national node within INCF. One of its goals is to develop infrastructure and tools for neurophysiology data and metadata storage and sharing. The above mentioned odML format is being developed here. G-Node also offers a portal[9] for neuroscientists to store and manage their data, share data with collaborators or search for data from other neuroscientists.

4 INCF home page: http://www.incf.org

5 OECD home page: http://www.oecd.org

6 INCF Dataspace: http://www.incf.org/resources/data-space

7 Integrated Rule-Oriented Data System (iRODS), http://irods.org

8 G-Node home page: http://www.g-node.org

9 G-Node Portal: https://portal.g-node.org/data

Neuroscience Information Framework (NIF)

NIF, introduced in [Ga08], is a dynamic inventory of Web-based neuroscience resources, established in 2005 by the NIH Blueprint for Neuroscience Research, which is a cooperative effort among neuroscience researchers in the USA. NIF is a member of the INCF National Node of the USA. NIF provides discovery and access to public research data worldwide through the Internet.

Besides other features NIF offers:

- searching for resources not indexed by common search engines,
- tools and standards for data interchange,
- ontologies and vocabularies for the neuroscience domain.

Carmen

The Carmen[10] project [Au11], funded by the Engineering and Physical Sciences Research Council (UK), is an effort to create a virtual laboratory for the neuroscience. The project provides a web interface called Carmen Portal[11], which allows signed users to store, analyse and share data from their experiments, search for data shared by other researchers, cooperate with other researchers etc.

CRCNS

Collaborative Research in Computational Neuroscience[12] (CRCNS) is a data-sharing initiative which provides a marketplace and discussion forum for sharing tools and data in neuroscience. Hosted data include physiological recordings from sensory and memory systems, as well as eye movement data.

EEGBase

Our department[13] is equipped with a laboratory for ERP experiments. The research group specialises in the research into attention (especially attention of drivers), children motor activity and blindness of mice. The laboratory is equipped with a number of devices and software tools, including basic EEG devices, a sound and

10 Carmen home page: http://www.carmen.org.uk

11 Carmen Portal: https://portal.carmen.org.uk

12 CRCNS data sharing: http://crcns.org

13 Department of Computer Science and Engineering, Faculty of Applied Science, University of West Bohemia, http://www.kiv.zcu.cz

electrically shielded booth, a car simulator including a car cockpit, wheel and pedals connected to the computer, projector, and software tools for the simulation of driving environment and driving itself [Mo14].

Data from experiments are managed with a custom software tool called EEGBase[14]. It comprises a database and a web-based interface. EEGBase enables researchers to store, manage and interchange data from their experiments. The project started in 2008 as Petr Ježek's master thesis [Je08] and since then it has been continuously extended and improved. Technologically, it is a Java Enterprise Edition (J2EE) application with a standard three-layer architecture. The software is developed as an open-source. More information about EEGBase can be found in [Je12].

EEGBase is intended not only for the local research but its goal is to contribute to the global neuroinformatics research as well. The project is a member of the Czech National Node of INCF.

2.2.3 Mobile Technologies

Another issue related to electrophysiological experiments is the process of collecting the metadata during experiments. Data management systems often provide a web-based user interface which allows experimenters to enter their data. But sometimes the experimenters do not have an active internet connection. In such a situation they have to note all information in a textual form and later rewrite it to the system, which is a waste of time. For that reason offline clients are required. Moreover, researchers may not have a computer available during an experiment. In such cases mobile applications can be very useful. Nowadays most of modern mobile phones are suitable for providing an offline client application, enabling experimenters to enter data when needed and synchronize them later with the system. One of such solutions is described in [Mi13].

Mobile client applications need to communicate with the server in order to synchronize data. Web services, described in chapter 5.1 Web Services, offer a modern, open, platform-independent and easy-to-use way for communication between mobile applications and a server.

14 EEGBase home page: http://eegdatabase.kiv.zcu.cz

Chapter 3
Automatically Generated Templates

Most data management systems consist of the following two components:

- **Database** to store the data.
- **Web-based interface** to enable managing the data by users.

Examples of such systems were given in chapter 2.2.2 Organizations and Databases. Most of them use relational databases to store the data. Relational databases are a time-proven and functional solution, but they also bring some disadvantages. The data model is fixed and cannot be changed easily. Applications using relational databases are tailored for the concrete database structure and cannot be easily reused. The result is that every data management system has its own proprietary application for user access.

This work addresses the described limitation and aims to propose a mechanism for creating the user interface independently of the database structure. The goal is to create a framework that will be able to generate templates for graphical user interface from an arbitrary data model. Generated templates will be used afterwards by an application running on a client device to present the graphical interface to a user. Figure 1 shows the scheme description of the proposed framework.

In order to reach this goal it is necessary to

1. Propose a format of the form templates.
2. Create a tool for generating the templates from a data model.
3. Implement a way to transfer the templates to mobile devices.
4. Create a mobile application capable of using the templates.

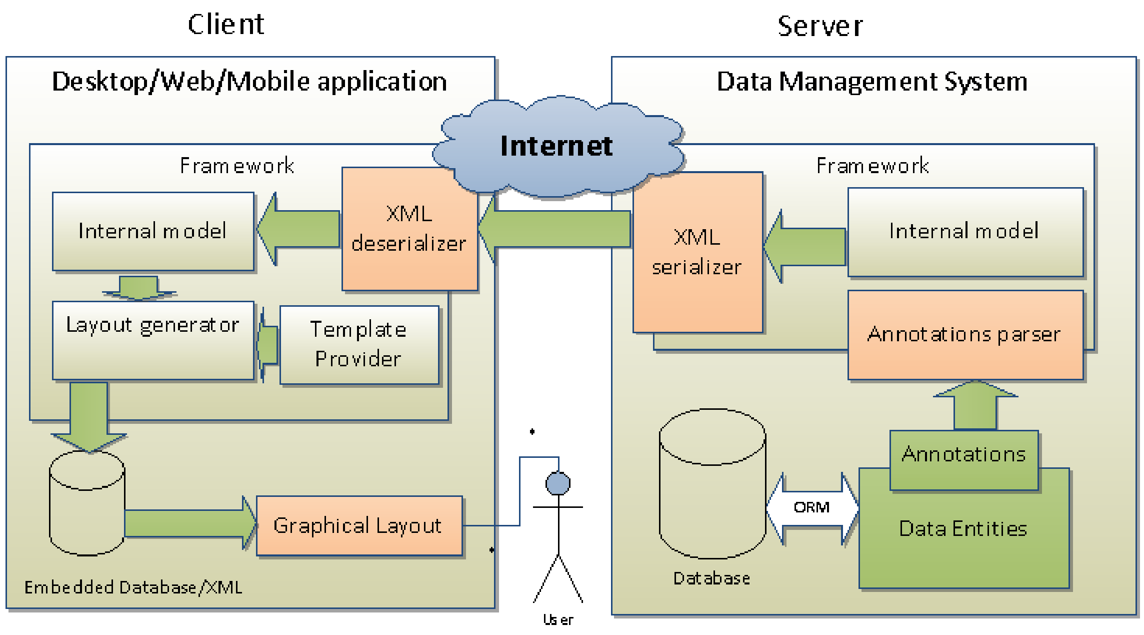


Figure 1: Scheme of the framework. [Je13]

To verify the proposed solution we have chosen the following use-case:

- **EEGBase**, described in chapter 2.2.2, as the data management system,
- client **application for Android**.

The client application is out of scope of this work. It is a topic of Jaroslav Hošek's thesis. This work focuses on the implementation of the template generation tool and its integration with EEGBase.

The rest of this chapter introduces the format of generated form templates. It was proposed in collaboration with Jaroslav Hošek, because it concerns both the mobile client and the server side tool. Chapter 4 Template Generation Tool describes implementation of the tool and Chapter 5 EEGBase: Implementing Web Services describes integration of the tool with EEGBase and implementation of the communication interface for mobile clients.

3.1 Format Selection

Various platforms, including mobile ones such as Android, iOS or Windows Phone, use different approaches for describing layout of graphical user interface (GUI). Their solutions are mostly proprietary and therefore not portable. The primary requirement for our templates is platform independence. Recently, the combination of HTML 5 and CSS 3 aims to overcome this gap by providing a platform-independent way of building GUI. However, HTML does not suit our needs and we have decided to design our own way of describing graphical layouts.

We have chosen odML as the format for our templates. OdML is currently being pushed by the neuroinformatics community. Although it was primarily intended to be a transport format for metadata, it is generic enough to describe a form and its layout. This decision brings following advantages:

- platform-independence
- simplicity and human-readability
- ability to transfer layouts as well as data with the same format

The first two points result directly from properties of odML, but the last one needs a better clarification. The basic goal of our work are templates of forms describing their layout. Purpose of such a form is to allow users to enter data. The data must be stored and transferred between a client and a server. It is very advantageous to use one transport format for both layouts and data since it simplifies the implementation of both server-side and client-side applications considerably.

It is worth mentioning that odML is an abstract model which is independent of a specific file format. However, XML is the only officially supported one so far. For that reason we will work with this implementation. Some examples in the following text will be given using the XML format because of its readability.

3.2 odML: Open Metadata Markup Language

Open Metadata Markup Language is a project within the German Neuroinformatics Node (G-Node) which is a part of INCF. The home page of the project[15] describes odML as *"an initiative to define and establish an open, flexible and easy-to-use format to transport metadata"*. The project is quite young, the odML format was specified in [Gr11] in 2011.

15 odML home page: http://www.g-node.org/projects/odml

OdML meets following requirements:

- **Ease of use** – specification is as simple as possible.
- **Human readability** – usability without specialised tools.
- **Openness** – freely available.
- **Extensibility and flexibility** – prepared for future changes.
- **Unrestricted usage** – user is not restricted by required entries.

One of key aspects of odML is an independence of its format and content. The format is defined by a general data model while the content is defined by domain specific terminologies. Next two sections describe these concepts in more detail.

3.2.1 Data Model

The data model defines the format of odML. It is specified as simple as possible, being rather general and thus widely usable and customizable. The basic idea of the model is based on key-value pairs like "name = James". The model does not define its implementation, i.e. the syntax for writing it to a document. However the odML specification [Gr11] came up with a XML implementation. The XML schema can be found on the enclosed CD (see Appendix B).

The data model is a tree-like structure with four entities depicted in Figure 2. The core entity implementing the key-value concept is *Property*. *Sections* provide means to group logically related properties and subsections together.

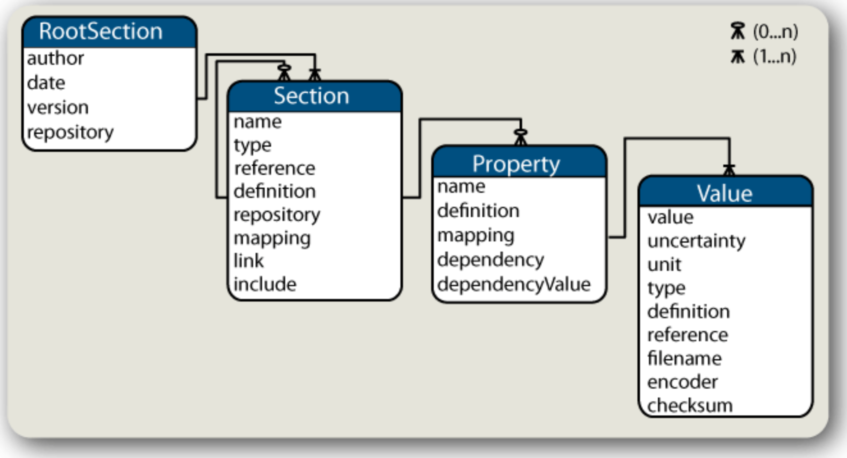


Figure 2: The odML data model. [Gr11]

As Figure 2 shows, every entity defines several specific elements. Most of them are optional, i.e. they do not need to be included in a concrete odML document. A detailed description of supported elements can be found in [Gr11].

3.2.2 Terminologies

The odML data model is very generic and thus in no wise bound to neuroscience domain. In fact it can be used to transport or store arbitrary data. This flexibility is very comfortable for users. On the other hand it does not solve the problem of sharing data using unified terms. This issue is solved with so-called *terminologies.*

An odML terminology is a set of *Section* and *Property* definitions, including their names, types and meaning. It establishes common naming conventions in a specific domain which allows collaborators to effectively share their data. The odML project provides a standardized terminology for the neuroscience domain[16], but its usage is not compulsory. Users can also define their own ones.

3.2.3 Tools

The home page of the odML project offers several tools to handle odML documents. They are all open-source and can be found in GitHub[17] repositories. The most important ones are libraries providing application programming interfaces (API) for several programming languages (Java, Python, Matlab). The Java library, called odml-java-lib, is a reference implementation. It is a small library enabling users to create the odML tree and serialize/deserialize it to/from a XML file.

Among other tools they offer a simple editor with a graphical user interface (GUI). The editor is part of the Python library. It can be used under Linux, Windows or MacOS. Finally the odML project offers XML schema to validate the XML serialization (see Appendix B) and two style-sheets for displaying metadata and terminologies using a HTML browser.

16 odML terminologies can be found on http://www.g-node.org/projects/odml/terminologies

17 GitHub is a web-based hosting service that offers repositories for the Git version control system, https://github.com.

3.3 Form Templates

A form template defines layout of a form presented to a user by a client application. A form is a component of a graphical user interface that enables users to input data. It consists of simple items, such as input text fields, and complex types. Various frameworks and technologies that are used to build graphical user interfaces name these items differently or offer specific ones, but in general there is a set of "standard form items" including textboxes, checkboxes, comboboxes, select-lists and so on. For each item there should be a description for a user, often called a label. Every item can be either compulsory for a user to fill it, or it can be blank.

There can be several layouts defined for a concrete form, each of them being described using a separate form template. They all describe the same form, i.e. each of them contains the same items, but individual items can be ordered differently, they can have different labels (thus allowing internationalization) and so on.

3.3.1 Basic Structure

A form and its items are represented using the odML *section.* The section's attribute *type* is used to indicate the type of graphical component represented by the section. We have defined several section types as listed in Table 1. The set of defined types can be later extended. The *name* attribute is used to identify individual form items, therefore unique names must be used within one form.

OdML *properties* are used to add required qualities of individual form items. A set of predefined properties is discussed below. They represent mostly layout-related properties or input value restrictions for validation purposes.

odML section type	**Description**
form	Form, can be also nested as a subform.
textbox	Input text field.
checkbox	Common checkbox.
combobox	Common combobox.
choice	Choice from a list of items.

Table 1: odML section types for form templates.

Form

A section *type* of which is set to “form” must be the root section of any form template. It represents the whole form. Its subsections represent individual items of the form. Their type can be any of the defined section types including “form”. A subsection of type “form” represents a complex input, that can be entered either using another form (defined by the subform) or chosen from a list of existing records.

The form section can have a property named *layoutName*, which contains a name of the layout. It should be used only for the root form. As there can be more than one layout for a given form, they are distinguished by the layout name assigned by this property.

Textbox

A section with the type attribute set to “textbox” represents an input field for textual values. This section can have several specific properties listed in Table 2.

Property	Description
datatype	Determines the datatype of the input. Possible values: *string*, *integer*, *number*, *date*, *email*
minLength	Restricts the minimum length of the input string.
maxLength	Restricts the maximum length of the input string.
minValue	Restricts the minimum numerical value of the input (for numerical input only)
maxValue	Restricts the maximum numerical value of the input (for numerical input only)
defaultValue	Offers a default value.

Table 2: Properties of the textbox section.

The *datatype* property restricts an input to the given datatype. If the datatype is for example integer, users are not allowed to input other characters than digits and plus and minus signs. The *minLength* and *maxLength* properties restrict the length of the input, e.g. a name of a person must not be longer than 50 characters so as database restrictions are met. *MinValue* and *maxValue* are used only for numerical input fields, i.e. with combination with the *datatype* property set to either integer or number. The *defaultValue* property is used to provide a default value, which will be used unless a user gives another one.

Combobox

Combobox offers choice among several predefined values. They are given using section's property named *values* containing set of offered values. They are presented to a user who selects one of them.

Choice

Choice is a component similar to a combobox in the meaning but with different graphical realization. It can be used to offer a choice among a bigger amount of values, which would be unsuitable for a combobox. The application could implement this component e.g. with a scrolling list.

In addition, all section types described above have several common properties listed in Table 3. However, the set of properties is not restricted to the mentioned ones. They are intended to serve as a base, but other properties can be arbitrarily added as needed. An application that uses this template should understand the described set of properties, but it may add as many own properties as required to include more information about the layout etc.

Property	Description
label	The label of the form item.
id	A unique identifier of the item within the template.
idTop	ID of an item right above.
idLeft	ID of an item on the left.
required	Determines whether the item can be left blank.
cardinality	Number of values that can be set to this item. • 1 for a single value • -1 for unlimited amount of values

Table 3: Common properties.

The *id, idTop* and *idLeft* properties are used to determine position of a form item relatively to another item. The reason is that the odML tree does not define ordering of subsections, so the corresponding form items would not have any defined order. The *required* property is used to indicate whether the item may be left blank or not. The *cardinality* property defines the number of values that a user may assign to a given form item. For common fields the value is 1, which means there can be only one value assigned to this item. But sometimes the user may be allowed to give a set of values for a given form item and the *cardinality* property determines the number of different values (-1 being chosen as “unrestricted”).

3.3.2 References to Data Entities

Every subform represents a complex input which cannot be entered as a single value, but requires a separate form. Moreover, the record may already exist in the database and a user does not want to fill the information again. Instead, it is necessary to let the user pick from the set of existing records. We can imagine e.g. a form collecting data connected to an electrophysiological experiment. One of its items is the used hardware. Because many experiments are performed with the same device, an experimenter does not need to fill all details about the device again and again, but rather pick the existing record.

But the set of existing records cannot be part of the template, because it describes static layout of a form without any dynamic content. Instead, we need to say the client application how to obtain the data from the server. For this purpose we use the *reference* attribute of a section representing a subform. Its value is used to download the existing records from the server. The application can cache the obtained records independently of the form template, for example in its local database. The transport format is described in section 3.4 Data Transport.

Listing 1 gives a sample usage of the reference. The highlighted line is the important one. An interface provided by the server should enable a way to download all records for *org.example.Hardware*. We do not define what exactly this value means, it is determined by the server-side implementation. In this case it is the fully-qualified name of the entity in an object-oriented data model.

```
<section>
    <type>form</type>
    <name>Experiment</name>

    <section>
        <type>form</type>
        <name>Hardware</name>
        <reference>org.example.Hardware</reference>

        <!-- content of the Hardware subform -->
    </section>

    <!-- other content -->
</section>
```

Listing 1: Example of the reference to existing records.

3.3.3 Data Previews

The previous section explained why it is necessary to work with existing records and introduced a way of referencing them in a layout. A client application is now able to obtain referenced data records from the server and let a user pick from them when filling in the form. It means that the application must show a kind of list containing all available records. But how to display every single item of the list? Different records have different structure and mostly contain too much information to be completely shown in the list item. Instead, it is necessary to choose some significant property which will represent the whole record. We call this a preview.

In layouts, data records are connected with sections of type *form*. We have defined two properties of a *form* section:

- *previewMajor*
- *previewMinor*

They enable to define at most two fields that will be shown in the preview. Their value must match a name of one of the fields contained in the form. Fields from subforms are not permitted.

Listing 2 demonstrates usage of these properties. A list of existing persons will display their full name and occupation for each record.

```
<section>
    <type>form</type>
    <name>Person</name>
    <reference>org.example.Person</reference>
    <property>
        <name>previewMajor</name>
        <value>
            fullName
            <type>string</type>
        </value>
    </property>
    <property>
        <name>previewMinor</name>
        <value>
            occupation
            <type>string</type>
        </value>
    </property>
    <section>
        <type>textbox</type>
        <name>fullName</name>
        <!-- definition of the fullName field omitted -->
    </section>
    <section>
        <type>textbox</type>
        <name>occupation</name>
        <!-- definition of the occupation field omitted -->
    </section>

    <!-- other items of the Person form omitted -->
</section>
```

Listing 2: Demonstration of the two preview properties.

3.3.4 Sample Template

Figure 3 depicts an odML tree of a sample form template. Individual nodes represent odML sections, the first line shown in the node describes its type and name using the “type: name” pattern. There are two forms in the tree, a root form and a subform, filled with the blue colour. Fields are filled with yellow. The further description in each node contains its odML properties as the “key = value” pair.

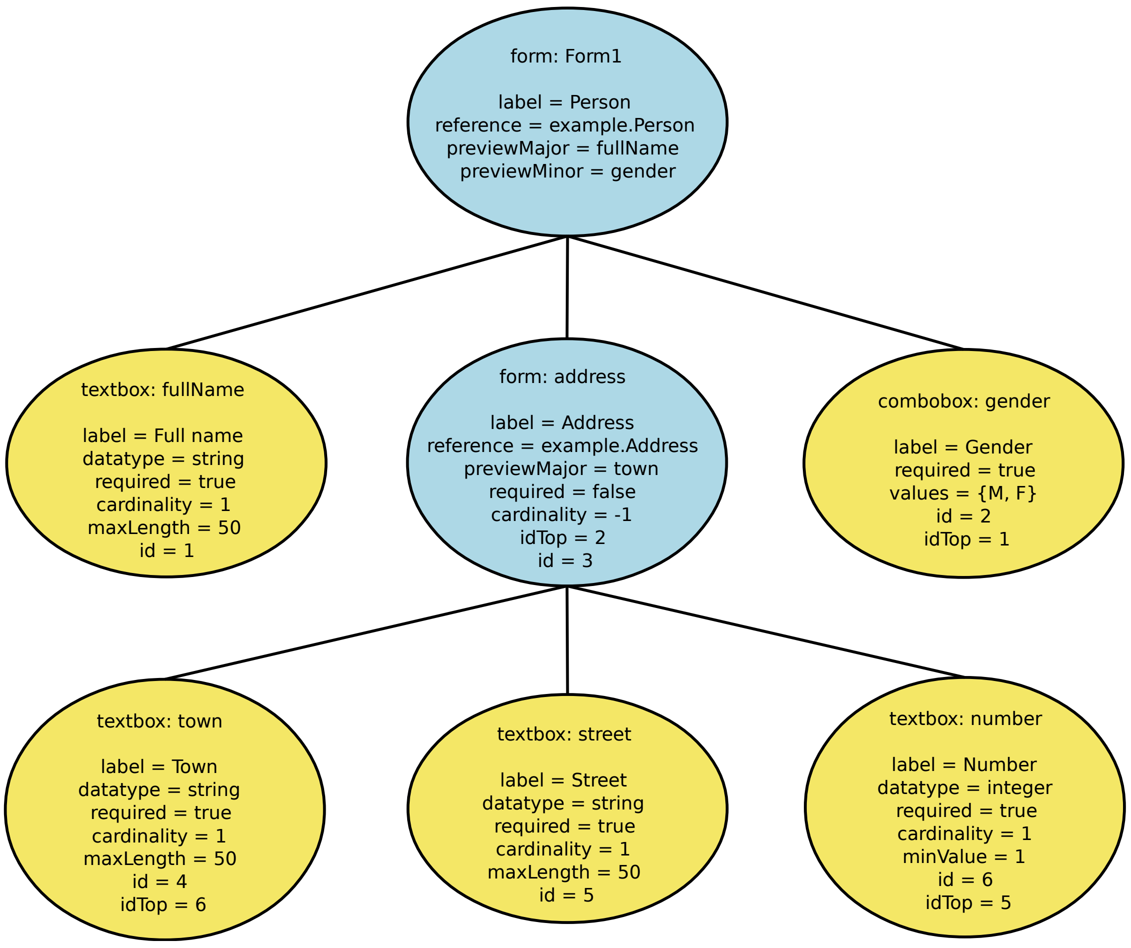


Figure 3: Sample form template.

The presented form enables a user to enter information about a person. It is a very simplified version for illustrating purposes. The root form consists of two fields, person's full name and his/her gender, and a subform for adding address. While the name and gender items are required, the address may be left blank. However, one can enter more than one address per person (which is determined by the cardinality property). An address consists of three fields – town, street and number.

An XML serialization of the form template from Figure 3 can be found in Appendix A. Figure 4 shows a possible GUI implementation of the sample form. It contains some input data just for illustration.

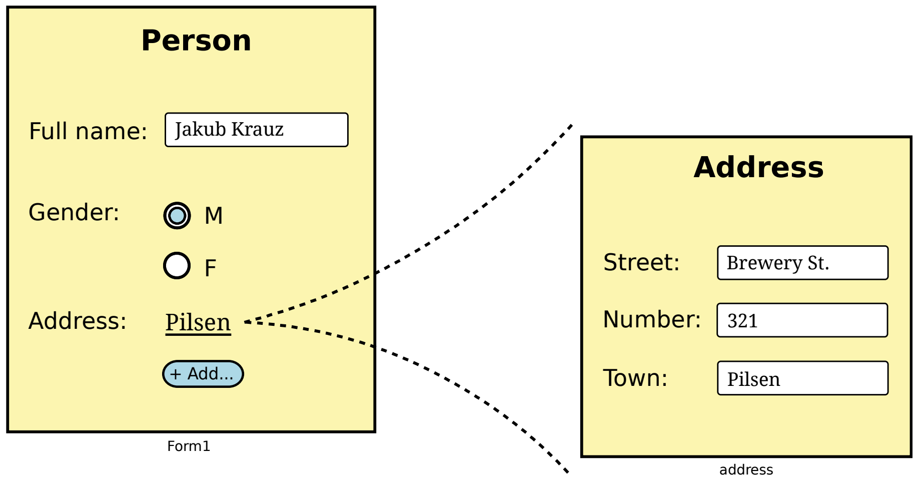


Figure 4: Possible GUI implementation of the sample form.

3.4 Data Transport

Besides form templates the framework needs to deal with data transport as well. There are two main use-cases for data transport:

1. A form template contains a complex item (subform), which in fact represents a data record. The client application should be able to download existing records from the server (see chapter 3.3.2 References to Data Entities) and offer them to a user to pick one.
2. A user has filled in a form and wants to upload the data to the server.

The main purpose of odML is to transport (meta)data, so this feature is much more straightforward than the above described templates. The only thing we have to define are rules for assigning types and names of sections and properties.

There is one important thing to realize – the structure of the odML data tree will copy the structure of the corresponding form template. If a form has ten fields, the corresponding data record will have ten properties, containing values entered in those fields. Similarly a subform in the template implies a subsection in the data tree. After this consideration it is clear, that the simpler way is to keep corresponding names of fields and subforms in the data tree. The only difference is the fact, that with templates a form field is represented by a section, but in data it is reduced to a property.

This approach cannot be used for section types, because in templates they determine graphical components of the form. For data, section types should refer to the record type. Such information is contained in the *reference* element in templates (see chapter 3.3.2 References to Data Entities). For that reason section types in data document should correspond with *reference* values in form templates.

Listing 3 contains a data record corresponding with the form template given in chapter 3.3.4 Sample Template.

```xml
<section>
    <type>example.Person</type>
    <name>Person_1</name>
    <property>
        <name>fullName</name>
        <value>Jakub Krauz<type>string</type></value>
    </property>
    <property>
        <name>gender</name>
        <value>M<type>string</type></value>
    </property>
    <section>
        <type>example.Address</type>
        <name>address</name>
        <property>
            <name>town</name>
            <value>Pilsen<type>string</type></value>
        </property>
        <property>
            <name>street</name>
            <value>Brewery St.<type>string</type></value>
        </property>
        <property>
            <name>number</name>
            <value>321<type>int</type></value>
        </property>
    </section>
</section>
```

Listing 3: Example of a data record.

Chapter 4
Template Generation Tool

In the previous chapter we have defined the basic concept of the required framework and specified format of templates for graphical user interface. Now it is desirable to create a tool responsible for the process of templates generation. The tool, hereinafter referred to as the template generator, must be able to parse the database structure and produce graphical templates of forms which will enable users to submit data to the database. The template generator should also provide means to control the generation process.

4.1 Analysis

The tool will be written in Java. It is a modern object-oriented programming language with a wide range of freely available libraries, frameworks and tools, which support an effective development. Last but not least, EEGBase, which was chosen as a use-case for its deployment, is also written in Java. This will enable an easy integration.

Maven will be used as the project's build tool. It is a popular project management tool which simplifies dependency management by the concept of public repositories.

Now it is necessary to propose an approach to templates generation. They must be based on a database structure. Generally, this information can be obtained from

a) database, or

b) object-oriented data model.

The object-oriented data model represents the database structure, but it is much more comfortable to work with native objects than a raw database. Data objects simply encapsulate data fields and provide appropriate getters and setters. This design is called Plain Old Java Object (POJO). It is a standard design of Java applications to provide POJOs representing database tables. For that reason option b) is more suitable for this project.

However, the data model itself without any additional information would be insufficient. It is necessary to propose a way of controlling the transformation process. One might want to define which fields to include in a form and which not, add some restrictions not detectable from the data model etc. Such information can be provided by

a) configuration file, or
b) Java annotations.

Configuration files represent a widely used concept, but their biggest disadvantage is that they are maintained separately from the code they control. It brings difficulties with maintenance, since changes in the controlled code may require changes in the configuration file, but it could be easily omitted. For that reason Java annotations are becoming quite popular. They bring exactly the feature that is needed since they are placed directly in the source code and do not change its semantics, which is crucial. We have chosen the annotation-based approach.

4.2 Data Model Annotations

Annotations are part of the Java language since version 5.0 released in 2004. Although it was quite a new concept without experiences from other languages, it soon became quite popular. Nowadays many popular frameworks prefer annotation-driven approach, including for example Spring, Hibernate, JUnit and many others. More information about annotations can be found in the official Java documentation or in [Pe05].

The template generator defines six annotation types to be used in data models, they are all located in the `cz.zcu.kiv.formgen.annotation` package. They are all designated for runtime processing, which is ensured by annotating their definitions with the following meta-annotation:

```
@Retention(RetentionPolicy.RUNTIME)
```

The defined annotations are used in the transformation process described in section 4.4 Parsers Implementation.

There are three annotations to be used with classes:

- **`@Form`**
  The `@Form` annotation is used to mark data entity (POJO class) for which a form should be generated. There will be generated a separate form for every annotated entity.
- **`@FormDescription`**
  This annotation adds a description to a form defined by the annotated entity. It is useful only for entities transformed to forms.
- **`@MultiForm`**
  The `@Form` annotation defines one form per annotated entity. This annotation can be used to group several entities in one form.

The other three annotations are defined for fields:

- **`@FormId`**
  This annotation is used to mark the ID field of a data entity.
- **`@FormItem`**
  The `@FormItem` annotation marks fields of an entity that should be included in the generated form. Fields without this annotations will be ignored.
- **`@FormItemRestriction`**
  This annotations provides means to add various constraints on the value of annotated data member. It is useful for validation of an input value in the form later. It should be used only for fields annotated with `@FormItem`, otherwise has no effect.

All described annotations except `@FormId` provide some optional parameters. Table 4 gives an overview of available parameters for each annotation and their meanings.

A sample usage of described annotations in a data model entity is demonstrated in Listing 4. The entity represents a person (its structure is simplified for demonstration purposes, a real application would use more complex design). The class is annotated with `@Form`, which means the tool will generate a form template for this entity. The form will be named the same as the class - *Person*.

Annotation	Parameter	Meaning
`@Form`	`label`	Label of the form.
`@FormDescription`	`value`	The description.
`@MultiForm`	`value`	Identifier of the form (string).
	`label`	Label of the form.
`@FormItem`	`label`	Label of the item.
	`required`	Must the item be filled in? (default: false)
	`preview`	Determines items used in data previews.
`@FormItemRestriction`	`minLength`	Minimum length of the input.
	`maxLength`	Maximum length of the input.
	`minValue`	Minimum value of numerical item.
	`maxValue`	Maximum value of numerical item.
	`defaultValue`	The default value.
	`values`	Enumeration of possible values.

Table 4: Overview of defined annotations and their parameters.

The form will contain 4 items – *name*, *age*, *workingGroup* and *diseases*. The *id* field will be ignored in the form template, because it is not annotated with `@FormItem`. The *name* field will have its label in the form set to "full name". This field will be the only one which cannot be left blank, because the *required* flag is set to true. The default value of this flag is false. Moreover, the length of the input for the *name* field must be 2 – 50 characters, as set with the `@FormItemRestriction` annotation. The second field of the form is *age*. Its label will be the same as its name, *age*, and the input field may be left blank. If filled in, its value must be an integer greater or equal 0.

The next item is *workingGroup* labelled with "working group". Because this field refers to another data entity, its class `WorkingGroup` will be parsed the same way as the `Person` class and the resulting form will be added as a sub-form to the *Person* form. This means that the `WorkingGroup` class should have its fields annotated with `@FormItem` as well. However, the `@Form` annotation is not required for this class. An application presenting the *Person* form to a user should be able either to let the user choose from existing working groups or to create a new group by filling in the sub-form. The mechanism for providing existing records will be discussed later.

```java
import cz.zcu.kiv.formgen.annotation.*;
import java.util.Set;

@Form
public class Person {

    @FormId
    private int id;  // this field won't be part of the form template
                     // but it will be used to reference data

    @FormItem(label = "full name", required = true)
    @FormItemRestriction(minLength = 2, maxLength = 50)
    private String name;

    @FormItem
    @FormItemRestriction(minValue = 0)
    private int age;

    @FormItem(label = "working group")
    private WorkingGroup workingGroup;

    @FormItem
    private Set<Disease> diseases;

    private String additionalInfo;  // this field will be ignored


    // other content omitted for clarity

}
```

Listing 4: Demonstration usage of defined annotations.

All the fields discussed so far may be assigned just one value (or they can be left blank, if possible). The *cardinality* of such form items is set to 1. But the last field, *diseases*, represents a collection. It means that a user may fill in 0 – N values. The *cardinality* of this form item is set to -1, which means “unrestricted”.

The *id* field is marked with `@FormId`. This annotation has no effect for the generated form template, it will be used for data processing instead. It was already mentioned that a form can offer selection from existing records. These records are usually stored in a database system and are referenced with an identifier. The `@FormId` denotes such an identifier.

4.3 Internal Model

The template generator needs to work with an object representation of generated forms. Hereinafter this representation will be referred to as the internal model. There are two basic approaches:

a) Using directly the odML model. It means that the library would internally work with the odML tree.

b) Designing own internal model used by the library. There would be an extra conversion needed between the model and the corresponding odML tree when writing the model as an odML template.

The a) option is easier to implement because the odML tree is created directly during processing the data model and no additional conversion is needed. However, the option b) is more general. In this case the library internally represents generated forms with its own model and an additional step is needed when writing a form in the odML template. Should the library later support another template format, the only required change to implement is the conversion between the internal model and this new format. For this reason the approach mentioned as option b) was implemented.

It is worth mentioning that except for the layout model there is one another model not mentioned yet. The other model represents data filled in a form. As generated form templates may contain references to complex data (e.g. the *workingGroup* item from Listing 4) the library should also be able to provide existing records along with the form template so as a user can choose an existing record instead of creating a new one.

4.3.1 Templates

A form template represents a *form* with its layout. A form in our concept consists of *fields* and *sub-forms*. A *field* is a type of a form *item* used for simple, unstructured input values such as text, number, date etc. The graphical representation of a form *field* is typically one input box. Another type of form items are *sub-forms*. These items represent complex values which require their own *form* with several more items to be filled. *Sub-forms* are of the same structure as top-level *forms*.

This abstraction leads to the design at UML class diagram in Figure 5. The `FormItem` interface defines common methods for objects that can be added to a *form*, i.e. *fields* and *sub-forms*. They are implemented by the `FormField` and `Form` classes respectively. The `AbstractFormItem` class implements their common behaviour in order to minimize code duplicity.

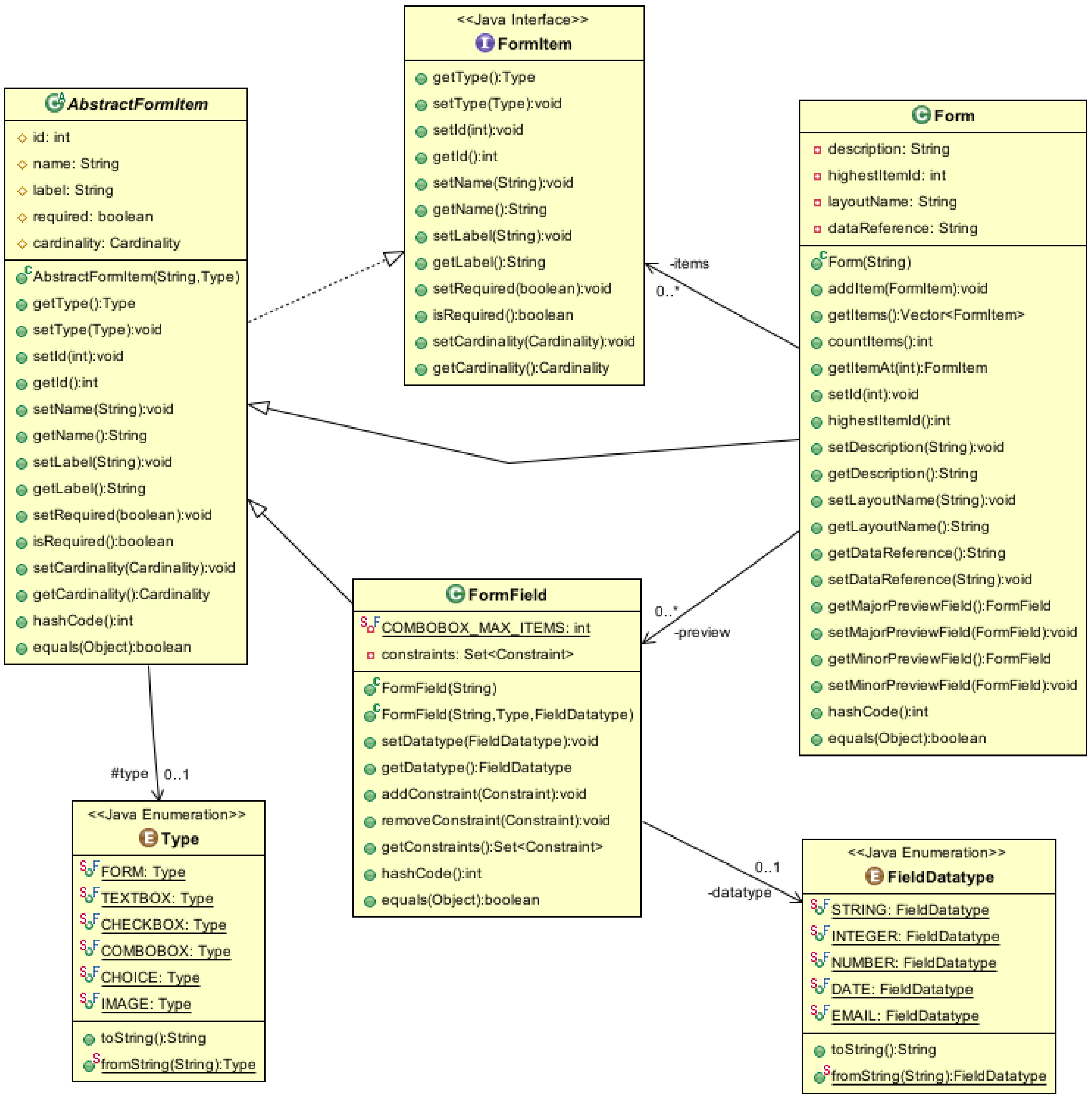


Figure 5: UML class diagram of the internal form model.

4.3.2 Data

The data model has two fields of usage:

a) It represents form data.

b) It represents database records offered to a user in a form.

However, both cases represent the same kind of data because forms are generated from a database structure. Moreover, those records being offered in a form are defined by a sub-form, so it does not matter whether the data has been just entered by a user or whether they were loaded from a database.

The data model structure copies the architecture of the form model described in section 4.3.1 Templates. The difference is that data model objects encapsulate input values instead of describing form layout. Their purpose is to represent data in a hierarchical structure corresponding to the structure of a related form. The data model is depicted in the UML class diagram in Figure 6.

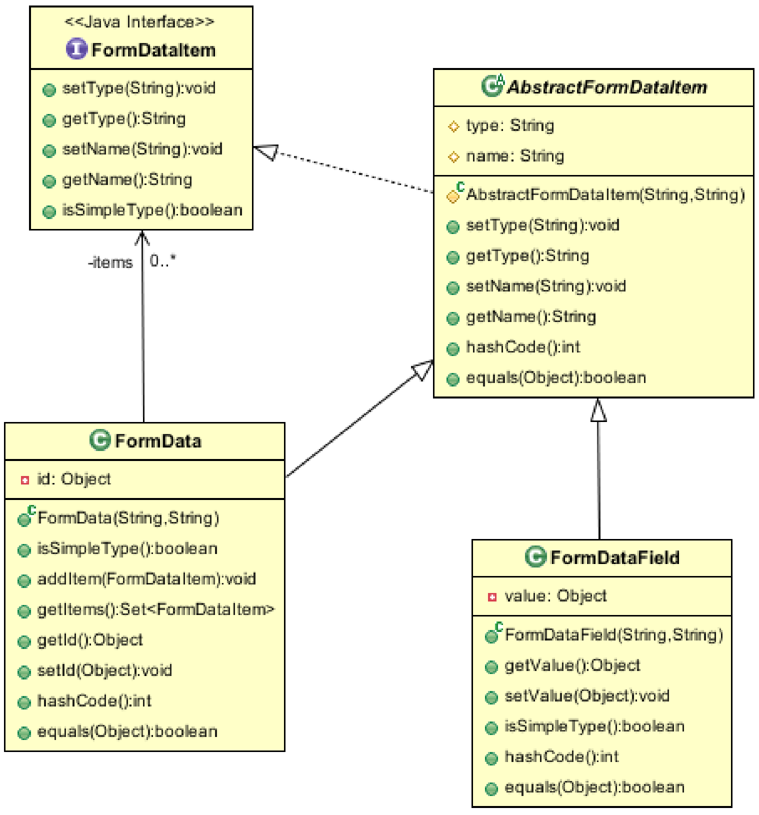


Figure 6: UML class diagram of the internal data model.

The `FormDataItem` interface defines a data item which can be either simple (obtained from a form *field*) or complex (obtained from a *sub-form*). The simple one is implemented by `FormDataField`. This object encapsulates an input value typed to `java.lang.Object`, but it should be either a primitive datatype or one of defined "simple" types like `java.lang.String` or `java.util.Date`. Complex data items are implemented by the `FormData` class. Their values are composed of other data items the same way as *sub-forms* are composed of form *items*. In order to avoid code duplicity, an abstract class, `AbstractFormDataItem`, is used again.

4.4 Parsers Implementation

This chapter describes creation of the internal model from an object-oriented data model. The data model must be annotated as described in section 4.2 Data Model Annotations. The crucial concept in this phase is *reflection*.

According to [Fo05], reflection is *"the ability of a running program to examine itself and its software environment, and to change what it does depending on what it finds"*. [Fo05] gives a thorough description of all features provided by the reflection API in Java. Reflection is used in many popular applications and frameworks such as Apache Tomcat, Spring, Hibernate, JUnit and many others.

Reflection is a very powerful tool but it has its drawbacks. The most important ones are performance and security issues. Reflective method invocation has considerably poorer performance than statical method calls since types are resolved at runtime. Security issues include restrictions if running under a security manager and possible impacts of accessing code which is not supposed to be accessed directly (private members). Last but not least, a reflective solution is much worse readable and maintainable than a corresponding statically typed code.

4.4.1 Class Parser

The most important use-case of the template generator library consists in generating form templates from an object-oriented data model. The data model is made up of POJOs or, more exactly, JavaBeans. These classes must be annotated as described in chapter 4.2. Form templates are generated according to the structure of these classes. For this reason the generator does not need any instances in this phase, the form model is generated completely from class objects. The library is generic, it parses any data model. It is obvious that such classes cannot be known at compile-time, which is a clear indication for reflection.

Let's suppose we have a class object to be parsed. Figure 7 depicts the basic flow chart of this process.

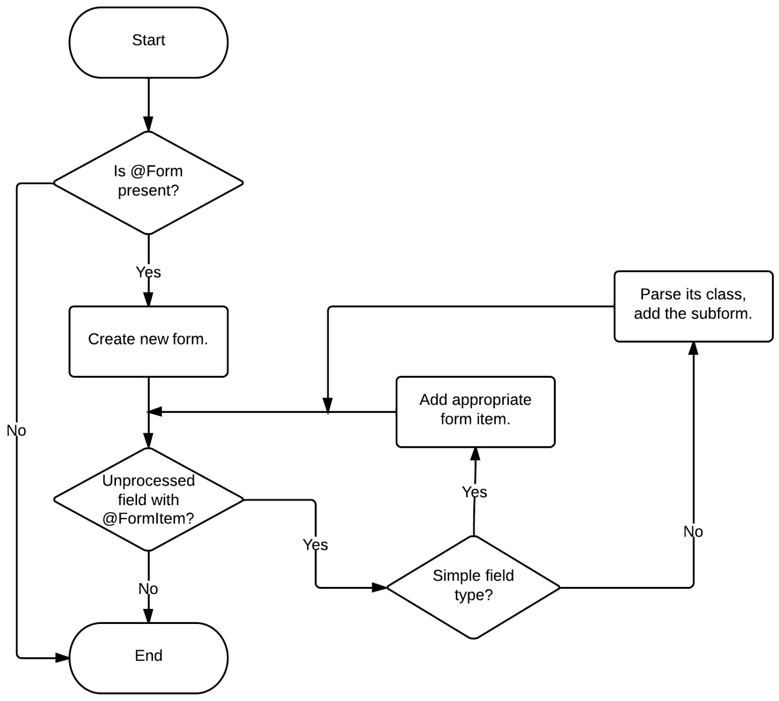

Start
Is @Form present?
Yes
No
Create new form.
Parse its class, add the subform.
Add appropriate form item.
Unprocessed field with @FormItem?
Yes
No
Yes
No
Simple field type?
End

Figure 7: Flow chart of the class parsing process.

The parser first checks if a `@Form` annotation is present for the class being parsed. If not, the process ends. If `@Form` was found, a new form model is created. Next the parser iterates over all fields marked with a `@FormItem` annotation. If the type of the field is *simple*, an appropriate form field is added to the form. Otherwise, the field's class is parsed the same way and the resulting form is added as a sub-form. Types are considered *simple* if a user can input their values in one input box. It includes primitive Java types, their object wrappers, strings and dates. An overview of *simple* field types and their mappings gives Table 5.

All objects involved in the transformation process are implemented in the `cz.zcu.kiv.formgen.core` package. The class parser is implemented by `ClassParser`. This object uses `TypeMapper` which implements mapping from Table 5. In addition, `TypeMapper` provides convenience methods for detecting wrapper classes of primitive types and converting them in both directions.

Field's type in Java	Type of form item

`byte, java.lang.Byte`
`short, java.lang.Short`
`int, java.lang.Integer`
`long, java.lang.Long`
	textbox (integer)

`float, java.lang.Float`
`double, java.lang.Double`
	textbox (number)

`boolean, java.lang.Boolean`	checkbox

`char, java.lang.Character`
`java.lang.String`
	textbox (string)

`java.util.Date`	textbox (date)

Table 5: Simple types and their mapping.

`ClassParser` provides a public method with the following signature

```
Form parse(Class<?> cls);
```

which is responsible for running the parsing process. It returns a `Form` object generated from the `cls` class. This method calls a private recursive method `_parse()` from Listing 5.

The `_parse()` method is the core of the whole process. First it creates a new `Form` object for the class being currently parsed. Next it iterates over all fields annotated with `@FormItem`. The `ReflectionUtils.annotatedFields()` method is a convenience method which returns a collection of fields annotated with a specified annotation. An instance of `TypeMapper` is used to determine whether the current field is simple type. If so, an appropriate `FormField` object is created and added to the form.

Special handling is required for collections since they are mapped to form items with cardinality set to -1, which indicates multiple values. These items are created by the `createFormSet()` method. The generic collection type must be parameterized so as the `createFormSet()` method can determine type of contained objects and create a proper type of form item. Finally, if the field is neither a simple type nor a collection, a sub-form is created by recursively calling the `_parse()` method.

```
private Form _parse(Class<?> cls, String formName, int id) {
    Form form = createForm(formName, cls);  // create the form model object
    form.setId(id++);  // id is unique for every item in a form template

    // iterate over all fields annotated with @FormItem
    for (Field f : ReflectionUtils.annotatedFields(cls,
                        cz.zcu.kiv.formgen.annotation.FormItem.class)) {

        if (mapper.isSimpleType(f.getType())) {
            FormField item = createFormField(f, id++);  // create form field
            form.addItem((FormItem) item);
            if (isPreviewField(f))  // check whether the field is preview
                setPreviewField(form, item, f);

        } else if (Collection.class.isAssignableFrom(f.getType())) {
            FormItem set = createFormSet(f, id++);  // create a set
            form.addItem(set);
            id = set.getId() + 1;

        } else {
            // parse the referenced object recursively
            Form subform = _parse(f.getType(), f.getName(), id++);
            form.addItem(subform);
            id = subform.highestItemId() + 1;

            // process the @FormItem annotation
            cz.zcu.kiv.formgen.annotation.FormItem annotation =
              f.getAnnotation(cz.zcu.kiv.formgen.annotation.FormItem.class);
            if (!annotation.label().isEmpty())
                subform.setLabel(annotation.label());
            subform.setRequired(annotation.required());
        }  // end if
    } // end for

    return form;
}
```

Listing 5: The core method of the parsing process.

4.4.2 Data Parser

The template generator library is able to process the data itself, not only their structure. The model was described in section 4.3.2 Data. It is closely related to a form template since it represents data entered in this form.

The process of converting data to the internal model is implemented in `DataParser`. It uses an algorithm similar to converting classes to form templates as described in the previous section. The main difference consists in parsing instances of data POJOs instead of their classes. However, it is necessary to inspect the class structure with reflection just as in the previous case. Having an object reference, say `obj`, its class object can be obtained very easily:

```
Class<?> cls = obj.getClass();
```

Now we have the class object and can iterate over its fields annotated with `@FormItem`. The algorithm follows that one from the `_parse()` method, except it is simpler because it does not deal with form layout issues such as ids, labels, field restrictions and so on. It only needs the field's name and its value. Having an object reference `obj` and its field object `field`, the value is obtained using the `value()` method from Listing 6. This method is implemented in the `ReflectionUtils` class.

```
public static Object value(Field field, Object obj) {
    if (field == null || obj == null)
        return null;

    // first try to get the value using its getter method
    try {
        // getterName() returns getter's name using standard conventions
        Method getter = obj.getClass().getMethod(getterName(field));
        return getter.invoke(obj);  // invoke the getter method

    } catch (Exception e) {

        // check the field's accessibility
        if (!Modifier.isPublic(field.getModifiers()))
            field.setAccessible(true);  // set the field accessible

        // try to get the value directly
        try {
            return field.get(obj);  // read the field's value
        } catch (Exception e2) { /* log the error here */ }
    }

    return null;
}
```

Listing 6: The `value()` *method for retrieving value of a field reflectively.*

4.5 Objects Builder

In addition to converting POJOs to form templates and their data, the library offers a transformation in the reversed direction. It means that the tool is able to instantiate the original POJOs from an internal data model. This process involves reflective construction and direct setting values of fields.

The process is implemented in `cz.zcu.kiv.formgen.core.SimpleObjectBuilder`. The most important thing the builder needs to know is the class of an object to be built. This information can be

a) contained in the form-data model, or

b) passed to the builder along with the model.

The main disadvantage of the option a) is that it brings an application-specific information to the model. For that reason the option b) was implemented. A disadvantage of this approach is the fact that the caller must know the object's class. Listing 7 shows the implementation of this approach.

```
public <T> T buildTyped(FormData formData, Class<T> type)
                                     throws ObjectBuilderException {
    try {
        // cast the object to required type
        return type.cast(createInstance(type, formData));
    } catch (ClassCastException e) {
        throw new ObjectBuilderException("Build error.", e);
    }
}

// creates a new instance of type and fills it with the specified data
protected Object createInstance(Class<?> type, FormData data)
                                     throws ObjectBuilderException {
    try {
        Object instance = type.newInstance();  // reflective instantiation
        fill(instance, data);                  // fill the object with data
        return instance;
    } catch (Exception e) {
        /* log the error here... */
        throw new ObjectBuilderException("Build error.", e);
    }
}
```

Listing 7: Building original POJOs.

The `buildTyped()` method is a generic method, its return type is determined by the second argument. `SimpleObjectBuilder` also offers a `build()` method which creates the instance the same way, but the result is typed to `java.lang.Object`. The line

```
obj = type.newInstance();
```

creates a new object of the required type using a reflective construction. The class type must have a non-parametric constructor so as the object can be instantiated this way. This condition is always satisfied for JavaBeans. The newly created object is passed to the `fill()` method afterwards. This method sets its fields using a direct reflective access.

Listing 8 gives an example of `SimpleObjectBuilder` usage.

```
FormData data;
...  // initialize the form-data model

ObjectBuilder builder = new SimpleObjectBuilder();

// let the builder instantiate Person
// the form-data model must represent this entity
Person person = builder.buildTyped(data, Person.class);
```

Listing 8: A sample usage of `SimpleObjectBuilder`*.*

4.6 odML Serialization

The template generator library is designed to be able to support various transport formats in the future. Currently the implemented format is odML. Templates in this format were described in Chapter 3 Automatically Generated Templates.

4.6.1 Odml-java-lib Adaptation

Odml-java-lib is a small open-source library for handling odML. It is written in Java and its source code is available in a GitHub[18] repository. It provides primarily implementation of objects used in odML - `Section`, `Property` and `Value`. They can be constructed in various ways and linked together to create the odML tree. In

18 Odml-java-lib on GitHub: https://github.com/G-Node/odml-java-lib

addition the library offers `Writer` and `Reader` able to (de)serialize the odML tree to/from a XML file. All mentioned objects are located in the `odml.core` package.

The problem with the library was that `Writer` and `Reader` objects were designed to work only with files. Such an approach is not very flexible, stream operations are needed instead. Because of a bad design of `Reader` and `Writer` it was not possible to add the required functionality using inheritance. For that reason the source code of odml-java-lib was modified and required stream operations were added directly in this library. One of the goals of this modification was to respect the existing interface and behaviour of affected objects so as we can raise a pull request to the original project.

Listing 9 shows basic usage of the `odml.core.Writer` class after stream operations were added. Exception handling was omitted for simplicity. This approach is much more general since streams provide higher level of abstraction than files. A stream can be easily directed to a file if needed, as the listing shows. The `odml.core.Reader` class was modified in the same fashion.

```
Section root;
... // initialize the odML tree here

OutputStream stream;  // any output stream can be used
stream = new FileOutputStream("example.odml");

Writer writer = new Writer(root);
writer.write(stream);  // write the odML tree to the stream
```

Listing 9: Basic usage of the `Writer` *after modification.*

Another feature added to odml-java-lib are redefinitions of `equals()` and `hashCode()` methods for `Section`, `Property` and `Value` classes. These methods are defined by the `java.lang.Object` class, but they need to be overriden in custom classes in order to work properly. They are used when comparing objects for equality, which is very useful in unit testing (see chapter 6.1 Unit Tests).

4.6.2 Writer and Reader Implementation

The public API of the template generator offers `Writer` and `Reader` interfaces located in the `cz.zcu.kiv.formgen` package. `Writer` provides methods for writing the internal model to an output stream using a transport format. `Reader` is able to read it back from an input stream. Their odML implementations are located in the `cz.zcu.kiv.formgen.odml` package. `OdmlWriter` is able to write the model to an output stream in the odML format and `OdmlReader` loads the odML representation from an input stream. Both of these objects need to convert between the internal model and the appropriate odML tree. The odML tree can be writen to a stream or read from a stream using the odml-java-lib API.

The conversion is implemented by `cz.zcu.kiv.formgen.odml.Converter`. This object offers following methods:

- `Section layoutToOdml(Form)`
- `Section dataToOdml(FormData)`
- `Form odmlToLayoutModel(Section)`
- `Set<FormData> odmlToDataModel(Section)`

The first two methods convert the internal model to odML tree. They both return the root section of the tree. Their implementation details are not very interesting, it involves iterating over all items of a form and creating appropriate section structure with defined properties. The other two methods create the internal model from the odML tree likewise.

`Converter` is used by both `OdmlWriter` and `OdmlReader`. Methods provided by these objects are all implemented using the same pattern. As an example the `OdmlWriter.writeLayout()` method is shown in Listing 10. First it performs some checks for null values. Next it creates the root section of the odML document. Then the `Converter.layoutToOdml()` method is called and its result is added to the root section. Finally, the odML document is written to the stream using `odml.core.Writer` provided by the odml-java-lib API.

```
public void writeLayout(Form form, OutputStream outputStream)
                                                    throws OdmlException {
    if (form == null)  // nothing to be written
        return;
    if (outputStream == null)  // no stream reference
        throw new NullPointerException("Output stream is null!");

    try {
        // create the root section of the odML document
        Section root = new Section();

        // convert the model and add it to the root section
        root.add(new Converter().layoutToOdml(form));

        // write the odml document to the stream
        odml.core.Writer writer = new odml.core.Writer(root);
        writer.write(outputStream);

    } catch (OdmlConvertException e) {
        throw new OdmlException("Could not convert to odML.", e);
    }
}
```

Listing 10: The `OdmlWriter.writeLayout()` *method.*

4.7 Public API

The public API of the template generator library is depicted in the UML class diagram in Figure 8. It offers several interfaces and their implementations providing a convenient way to control transformation processes. They provide a facade for parsers and builders described in chapters 4.4 and 4.5 respectively.

The `LayoutGenerator` interface offers methods used to generate the internal model of form templates from class objects of an object-oriented model. It is implemented by `SimpleLayoutGenerator`. There are several overloaded `load` methods:

- `load()` takes as its arguments class objects to be parsed
- `loadClass()` takes fully qualified class names
- `loadPackage()` takes a fully qualified package name (all classes in the specified package will be parsed)

All these methods return the internal model generated from their arguments. They can also be called more times and the generator accumulates the internal model. The whole model can be obtained using `getLoadedModel()` afterwards. It contains all templates since the generator has been constructed or since the `clearModel()` method has been called.

The `DataGenerator` interface, implemented by `SimpleDataGenerator`, is used to generate the internal model of form data. Its methods follow the same conventions as the `LayoutGenerator` interface. There are two `load()` methods, taking objects as their arguments, which return the generated form-data model. They can be called more times again, `DataGenerator` provides the `getLoadedModel()` and `clearModel()` methods like `LayoutGenerator`.

The `Writer` interface is responsible for writing the internal model to an output stream. It is implemented by `OdmlWriter` which supports the odML format. There are two overloaded methods, taking the internal model and the target output stream as their arguments:

- `writeLayout()` for writing form templates
- `writeData()` for writing form data

The `Reader` interface provides the ability to read the model from an input stream. The odML format is supported by `OdmlReader`. `Reader` provides two methods:

- `readLayout()` for reading form templates
- `readData()` for reading form data

The `ObjectBuilder` interface defines methods used to instantiate original POJOs from the internal data model. It provides two methods. The `buildTyped()` method returns the object typed to its instantiating class, while `build()` returns `java.lang.Object`. The latter must be used if the type is not known at compile-time. `ObjectBuilder` is implemented by `SimpleObjectBuilder` described in chapter 4.5. `PersistentObjectBuilder` extends this class to provide the ability of working with persistent objects. It means that data can contain references to existing records. A user of `PersistentObjectBuilder` has to implement `PersistentObjectProvider` which is responsible for retrieving persistent objects by their primary key.

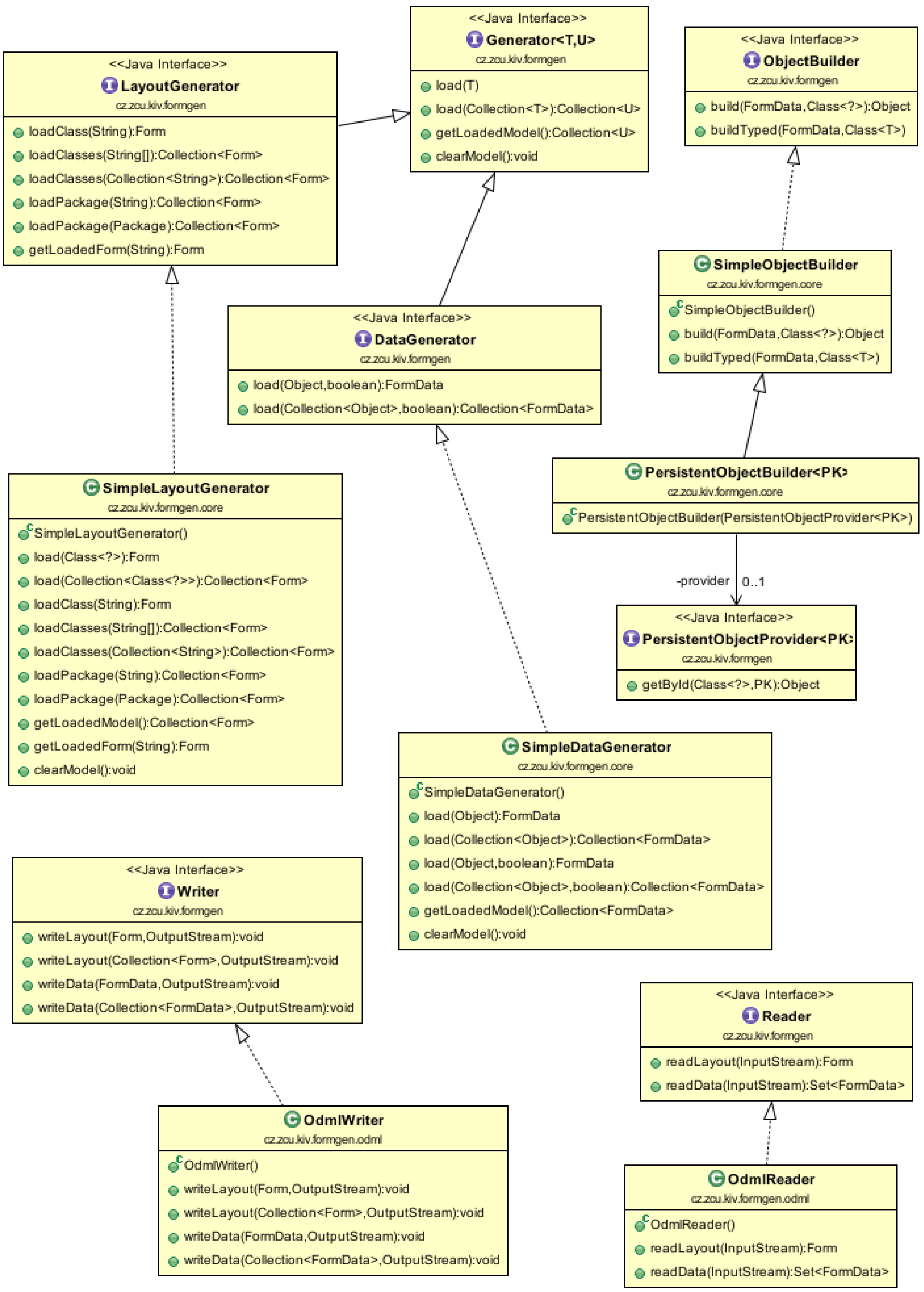

<<Java Interface>>
LayoutGenerator
cz.zcu.kiv.formgen
loadClass(String):Form
loadClasses(String[]):Collection<Form>
loadClasses(Collection<String>):Collection<Form>
loadPackage(String):Collection<Form>
loadPackage(Package):Collection<Form>
getLoadedForm(String):Form
<<Java Interface>>
Generator<T,U>
cz.zcu.kiv.formgen
load(T)
load(Collection<T>):Collection<U>
getLoadedModel():Collection<U>
clearModel():void
<<Java Interface>>
ObjectBuilder
cz.zcu.kiv.formgen
build(FormData,Class<?>):Object
buildTyped(FormData,Class<T>)
SimpleObjectBuilder
cz.zcu.kiv.formgen.core
SimpleObjectBuilder()
build(FormData,Class<?>):Object
buildTyped(FormData,Class<T>)
<<Java Interface>>
DataGenerator
cz.zcu.kiv.formgen
load(Object,boolean):FormData
load(Collection<Object>,boolean):Collection<FormData>
SimpleLayoutGenerator
cz.zcu.kiv.formgen.core
SimpleLayoutGenerator()
load(Class<?>):Form
load(Collection<Class<?>>):Collection<Form>
loadClass(String):Form
loadClasses(String[]):Collection<Form>
loadClasses(Collection<String>):Collection<Form>
loadPackage(String):Collection<Form>
loadPackage(Package):Collection<Form>
getLoadedModel():Collection<Form>
getLoadedForm(String):Form
clearModel():void
PersistentObjectBuilder<PK>
cz.zcu.kiv.formgen.core
PersistentObjectBuilder(PersistentObjectProvider<PK>)
-provider
0..1
<<Java Interface>>
PersistentObjectProvider<PK>
cz.zcu.kiv.formgen
getById(Class<?>,PK):Object
SimpleDataGenerator
cz.zcu.kiv.formgen.core
SimpleDataGenerator()
load(Object):FormData
load(Collection<Object>):Collection<FormData>
load(Object,boolean):FormData
load(Collection<Object>,boolean):Collection<FormData>
getLoadedModel():Collection<FormData>
clearModel():void
<<Java Interface>>
Writer
cz.zcu.kiv.formgen
writeLayout(Form,OutputStream):void
writeLayout(Collection<Form>,OutputStream):void
writeData(FormData,OutputStream):void
writeData(Collection<FormData>,OutputStream):void
<<Java Interface>>
Reader
cz.zcu.kiv.formgen
readLayout(InputStream):Form
readData(InputStream):Set<FormData>
OdmlWriter
cz.zcu.kiv.formgen.odml
OdmlWriter()
writeLayout(Form,OutputStream):void
writeLayout(Collection<Form>,OutputStream):void
writeData(FormData,OutputStream):void
writeData(Collection<FormData>,OutputStream):void
OdmlReader
cz.zcu.kiv.formgen.odml
OdmlReader()
readLayout(InputStream):Form
readData(InputStream):Set<FormData>

Figure 8: The public API of the template generator library.

Chapter 5
EEGBase: Implementing Web Services

The template generator is intended to be used by a server such as EEGBase. The server works with an underlying database and thus it knows the data model. Generated templates are used primarily in applications in various mobile devices. For that reason it is necessary to propose and implement a proper way of transfer between the server-side application and mobile devices. In the terms of the Service-Oriented Architecture (SOA) the server application will provide a service for client ones.

Such a service must meet following requirements:

- universal and platform independent
- available on the Internet
- using open standards
- support for security
- easy to use by mobile applications

The solution that meets all these requirements is called web services.

5.1 Web Services

Web services are considered a new generation of distributed computing. They are designed to eliminate disadvantages of RPC-like technologies. They communicate with textual messages (XML or equivalent), are platform-independent and rely on well-established and non-proprietary protocols. The main difference between web services and other distributed applications is that web services are typically delivered over the HTTP protocol.

Several features that distinguish web services from other distributed software systems are mentioned in [Ka13]:

- **Open infrastructure**
  Web services piggyback on existing, standardized and vendor-independent protocols and languages like HTTP, XML and JSON.
- **Platform and language transparency**
  Web services and their clients can be used in different programming languages, hardware platforms and operating systems.
- **Modular design**
  Existing web services can be re-used and composed to new ones and so on.

Web services are very useful in the field of mobile technologies. Various mobile platforms can take advantage of their openness and platform-independence and use them equally without taking care of implementation details. Clients of web services are rarely web browsers but rather specialized applications on various networked devices.

5.1.1 SOAP and REST

Web services are delivered in two flavours – **SOAP-style** and **RESTful** ones. Both approaches are described in [Ka13]. Let's examine which one suits better needs of this project.

SOAP originally stood for Simple Object Access Protocol, but this acronym was later officially dropped. SOAP is considered a successor of XML-RPC. It is designed to be transport-neutral, which means it can be delivered by any transport protocol, HTTP being the most commonly used one (the reason is that with HTTP there is no problem passing through firewalls). A description of a SOAP-based web service is provided in a WSDL (Web Service Description Language) file, which is a XML document describing the service as a collection of operations, their parameters and return types. They can be exposed in public UDDI (Universal Description, Discovery and Integration) registries.

REST stands for Representational State Transfer. Its base principles were proposed in Roy Fielding's PhD dissertation [Fi00]. The central abstraction in RESTful architecture is a *resource*. It can be any entity identified by a Uniform Resource Identifier (URI). URIs are standardized names for resources and must be structured according to their specification [RFC05]. In RESTful architecture, the HTTP protocol, apart from being the transport mechanism, can be seen as an API for manipulating resources using so called CRUD operations, which stands for Create, Read, Update and Delete.

REST is not standardized as opposed to SOAP. The advantage of RESTful architecture is its simplicity in comparison to the complexity of SOAP-based web services. REST completely relies on the HTTP protocol which is widely supported. With REST there is no need to use WSDL files, client stubs and other related things, the only needed thing is to know the public API of the service. This feature makes REST the primary choice for many simple applications because, generally speaking, it is easier to implement. Another advantage of RESTful approach is the size of messages being transferred – RESTful ones are usually smaller than with SOAP due to its complex standards. Finally REST, unlike procedural SOAP, is stateless and orientated towards data, which complies with the intent of providing form templates. After this comparison the RESTful approach was chosen.

5.2 Analysis of Required Functionality

The primary purpose of the implemented web service is the transfer of generated form templates to mobile devices. But form templates, described in Chapter 3, do not include existing records used when filling a form. It follows that the web service should be able to provide this data as well. Moreover, a user of a mobile device can design his own template and it will be useful if he can upload the template to the server and thus make it available for other devices. Finally, all the form templates are intended for users to enter new data. The mobile device should be able to upload this data to the server.

This consideration shows following use-cases:

- managing form templates
- providing existing records
- uploading new data

The first item, managing form templates, includes a complete CRUD API for templates. A user will be able to download available templates and upload, update and delete his own ones. The next use-case, providing existing records, involves only downloading data from the server. Finally the last item, uploading new data, enables users to upload new data to the server after submitting a form.

5.3 Storing Templates

Form templates implemented in this work are XML files, one file per template. The server can store them

a) in a filesystem, or

b) in a database.

Both these approaches have their pros and cons. The most important issues concern performance and simplicity of use. Filesystems were designed to manage files. For that reason it could be considered the best solution. But it brings following disadvantages:

- It is necessary to choose a proper directory and keep this information (e.g. in a configuration file). This must be taken into consideration by application deployment etc.
- Files are stored apart from the rest of data (i.e. the database) which brings non-uniformity and complicates for example backup processes.

These issues are eliminated with database. On the other hand the database can grow very quickly, especially with large files.

Another issue is the performance. According to Microsoft Research [Se06] the best performance for small files (under 250 kB) can be achieved with a database. On the other hand files greater than 1 MB are typically better handled by a filesystem. One of reasons is that filesystems mostly have better fragmentation handling.

After this research database was chosen as a more suitable solution for storing the templates. They are not very large, no more than hundreds of kilobytes. Storing such files in a database brings more benefits.

5.3.1 Database Extension

The first step was a creation of a new table in the underlying database. EEGBase currently uses PostgreSQL. Barring ID, the table must contain following items:

- **form name** which identifies a form
- **layout name** which identifies a concrete layout
- reference to **owner** (foreign key)
- **content** of the template itself

A form template is identified by a pair of names – the *form name* and the *layout name*. There can be several templates for one form, i.e. it contains the same items but their layout is different. The reference to *owner* serves for access rights.

Listing 11 contains the SQL script that was used to create this table.

```
CREATE TABLE form_layout (
    form_layout_id INTEGER PRIMARY KEY NOT NULL,
    form_name VARCHAR(50) NOT NULL,
    layout_name VARCHAR(50) NOT NULL,
    content BYTEA NOT NULL,
    person_id INTEGER
);

ALTER TABLE form_layout
    ADD CONSTRAINT form_layout_unique_idx
    UNIQUE (form_name, layout_name);

ALTER TABLE form_layout
    ADD CONSTRAINT form_layout_person_fk
    FOREIGN KEY (person_id) REFERENCES person (person_id)
    ON DELETE SET NULL;
```

Listing 11: The SQL script for PostgreSQL.

The only interesting thing is the way of storing the template file itself. First, it could be stored either as a textual content or as a binary one. In the terms of Large Objects (LOBs) it means either the Character LOB (CLOB) or the Binary LOB (BLOB). CLOBs can be useful for text searching, but they can cause issues with encoding. Since there is no need to search for text in stored form templates, it is appropriate to store them in the binary form which avoids mentioned encoding issues.

PostgreSQL provides two options how the store a binary content:

a) **OID** which is a reference to a BLOB stored in a special table

b) **BYTEA** which is a byte array stored directly with the table

If using OID the content is stored in a special system table. The content is split to parts of the same length that are stored as BYTEA. It is advantageous for very large content because it can be handled in parts. On the other hand the OID type is not handled transparently as a standard table column unlike the BYTEA type. Since form templates are generally small in size, BYTEA is used.

5.3.2 Persistent Objects Implementation

EEGBase takes advantage of the Object-Relational Mapping (ORM) provided by Hibernate[19]. ORM enables an object-oriented code to work with persistent objects. Describing this concept and Hibernate is out of scope of this work. A very thorough description can be found in [Pe06].

In EEGBase, data-layer objects are located in the `cz.zcu.kiv.eegdatabase.data` package. It contains several subpackages, POJOs being located in the `pojo` subpackage and DAOs in the `dao` subpackage.

Listing 12 shows interesting parts of the implemented `FormLayout` entity which represents stored templates. It contains fields corresponding to the `form_layout` database table described in the previous section. Hibernate mapping is controlled using an annotation-based approach.

```
@Entity
@Table(name = "FORM_LAYOUT")
public class FormLayout implements Serializable {

    private int formLayoutId;
    private String formName;
    private String layoutName;
    private byte[] content;
    private Person person;

    // constructors (including a non-parametric one)

    @Column(name = "FORM_NAME", nullable = false, length = 50)
    public String getFormName() {
        return formName;
    }

    // other getters and setters

}
```

Listing 12: The `FormLayout` *entity.*

19 Hibernate home page: http://hibernate.org

The next step is to implement the DAO object. Its interface is defined by `FormLayoutDao` and implementation is provided by the `SimpleFormLayoutDao` class. The interface defines methods for accessing and manipulating `FormLayout` persistent objects. `SimpleFormLayoutDao` implements those methods using Hibernate. There are two possibilities of querying for data records with Hibernate – Hibernate Query Language (HQL) or Criteria. HQL is a query language similar to SQL except it works with objects. Criteria provides an object-oriented API which is better especially for dynamic queries where HQL would require uncomfortable string concatenation.

For that reason Criteria were used. An example is given in Listing 13. The `getLayoutsCount` method returns number of layouts for a given form owned by a specified person. If either the `owner` or the `formName` or both arguments are null, the corresponding restriction is not applied, i.e. the method can be used to count all layouts regardless of their owner and the form they describe.

```
public int getLayoutsCount(Person owner, String formName) {

    // create the criteria object
    DetachedCriteria criteria = DetachedCriteria.forClass(type);

    // count records
    criteria.setProjection(Projections.rowCount());

    // add the owner restriction
    if (owner != null)
        criteria.add(Restrictions.eq("person.personId",
                                     owner.getPersonId()));

    // add the form restriction
    if (formName != null)
        criteria.add(Restrictions.eq("formName", formName));

    // query and return as single integer
    return DataAccessUtils.intResult(
                getHibernateTemplate().findByCriteria(criteria));

}
```

Listing 13: The `getLayoutsCount` *method demonstrating usage of Criteria.*

5.4 RESTful Web Services with Spring

EEGBase takes advantage of the Spring Framework[20]. Although being lightweight, Spring offers a very rich functionality. Describing this framework in more detail is out of scope of this work, a reader is referred to [Wa08] instead. The following sections expect some knowledge of this domain.

Some RESTful services were already implemented in EEGBase last year by Petr Miko. They are described in [Mi13]. Their implementation is located in the `cz.zcu.kiv.eegdatabase.webservices.rest` package. The new service is placed in its subpackage named `forms` and conventions established by Petr Miko's work are followed.

5.4.1 Message Format

If a client calls a web service the result is often an object, i.e. a complex type. This object must be written in the HTTP message by the sender and read by the receiver. Terms *marshalling* and *unmarshalling* are commonly used for this processes respectively. The representation of an object in the message must have a standardized format. In REST, XML and JSON are typically used. JSON, although originating from the Javascript language, is an open, language-independent and human-readable data format, specified in RFC document [RFC14]. Recently this format has became popular thanks to its simplicity and lower verbosity compared to the older XML. Our web service supports both of them, if possible. However, odML currently defines only XML serialization, for that reason JSON is not supported for the transfer of the odML templates themselves.

In EEGBase the Java Architecture for XML Binding (JAXB) is used for XML marshalling. The only required thing for this purpose are data containers provided with JAXB annotations. The data containers are conventionally placed in the `wrapper` subpackage of the concrete RESTful service's package. Following containers were implemented:

- **`AvailableFormsDataList`** – list of available form names
- **`AvailableLayoutsData`** – form name and layout name of a form template
- **`AvailableLayoutsDataList`** – list of `AvailableLayoutsData`

20 Spring Framework: http://projects.spring.io/spring-framework

The conversion to JSON is provided by the Jackson[21] library. This library requires no special annotations for the data containers, the JAXB-annotated ones mentioned above can be used without changes.

5.4.2 Service Object Implementation

A *service* in Spring is a special bean which provides defined functionality to other components. According to the Domain-Driven Design it is *"an operation offered as an interface that stands alone in the model, with no encapsulated state"*. It is usually used as an intermediate layer between the data layer and controllers (see next section). The service object should contain the logic of a web service itself while the controller contains the logic related to HTTP requests handling.

The `FormService` interface defines methods for manipulation form templates and related data records. If appropriate, they return marshallable data containers described in the previous section. The interface is implemented by the `FormServiceImpl` class. Listing 14 shows a snippet of this class. The code was slightly adjusted in order to point out important things.

The `@Transactional` annotation is used to wrap execution of an annotated method in a database session. If used for the whole class it is applied to every method within this class. The `readOnly` parameter indicates that no writes will be performed during the session. Methods that need write access to persistent objects must be annotated with the following line

```
@Transactional(readOnly=false, propagation=Propagation.REQUIRES_NEW)
```

which overrides the `@Transactional` annotation which is used for the whole class.

The service bean implements two extra interfaces – `InitializingBean` and `ApplicationContextAware`. The first one defines the `afterPropertiesSet()` method which is called during the creation of the bean. This is done by Spring during the start of the application. It is utilized to re-generate form templates. The body of `afterPropertiesSet()` contains code that loads data model classes, generates corresponding form templates and updates them in the persistent storage. Since the data model cannot change during runtime it is sufficient to perform this once at the application's startup.

The `ApplicationContextAware` interface defines the `setApplicationContext()` method. It is used to pass an instance of the `ApplicationContext` during the

21 Jackson JSON Processor: http://wiki.fasterxml.com/JacksonHome

creation of the bean again. Having access to the application context, the bean can interact with the Spring container. Since the web service provides access to any data records used in any of defined forms, it would be uncomfortable to write specialised methods for every data entity. Moreover, if a new form was defined in the data model, it could cause a need to modify the web service implementation so as it would be able to provide data records required by the new form. For that reason a generic solution was introduced, which is based on retrieving a DAO for the required data entity dynamically from the container. This approach is possible thanks to the fact that all DAOs in EEGBase implement the `GenericDao` interface which provides sufficient functionality for the web service.

```
@Service
@Transactional(readOnly = true)
public class FormServiceImpl implements FormService,
                          InitializingBean, ApplicationContextAware {

    @Autowired private FormLayoutDao formLayoutDao;
    @Autowired private PersonDao personDao;
    private ApplicationContext context;
    // other fields omitted

    public void afterPropertiesSet()
    { /* method body */ }

    public void setApplicationContext(ApplicationContext ctx)
    { /* method body */ }

    // get the number of available forms
    public RecordCountData availableFormsCount() {
        RecordCountData count = new RecordCountData();
        count.setPublicRecords(formLayoutDao.getAllFormsCount());
        count.setMyRecords(formLayoutDao
                    .getFormsCount(personDao.getLoggedPerson()));
        return count;
    }

    // other methods

}
```

Listing 14: The `FormServiceImpl` *class.*

5.4.3 Controller Implementation

The purpose of a *controller* in Spring is to handle HTTP requests. In our case it defines endpoints of the web service. The `FormServiceController` class was implemented for this purpose. Listing 15 shows a snippet of this class. The controller is responsible only for handling HTTP communication. The logic of the web service is placed in the `FormService` object as described in the previous section.

```
@Controller
@RequestMapping("/form-layouts")       // URL mapping
@Secured("IS_AUTHENTICATED_FULLY")     // users must be authenticated
public class FormServiceController {

    /** The service object providing data. */
    @Autowired private FormService service;

    @RequestMapping(value = "/count", method = RequestMethod.GET)
    @ResponseBody
    public RecordCountData availableLayoutsCount(
              @RequestParam(value = "form", required = false) String form) {

        if (form == null)
            return service.availableLayoutsCount();
        else
            return service.availableLayoutsCount(form);
    }

    // other content omitted

}
```

Listing 15: The `FormServiceController` *class.*

The `availableLayoutsCount()` method is used to handle GET requests for the **/form-layouts/count** path. The value of the `@RequestMapping` annotation is simply appended to the controller's URL. This methods is used to get number of available layouts stored in the server's database. It returns the `RecordCountData` object, which is a marshallable data container, because there are two numerical values to be returned – number of public layouts and number of user's own layouts. The `@ResponseBody` annotation tells Spring that the return value should be marshalled in the body of the response.

The method has one argument named `formName`. It is annotated with `@RequestParam`. This annotation indicates that the value will be obtained from the URL as the value of a parameter named “form”. It may or may not be present, null is passed in `formName` in the latter case. The URL may be e.g.

/form-layouts/count?form=someFormName

In this case the `availableLayoutsCount()` method is called with the `formName` argument's value set to “someFormName”. The `service` object is used afterwards to obtain the return value.

5.5 Provided RESTful API

5.5.1 Querying Available Templates

Following URLs provide means to discover which templates are available on the server. They support only the GET method. Both XML and JSON responses are supported, XML being the default one. JSON can be requested using the Accept header in the HTTP request:

Accept: application/json

Description of individual supported URLs follows.

- /rest/form-layouts/count[?form=<formName>]

  This URL returns number of available templates. There are two values – number of templates owned by the current user and number of all templates. The optional *form* parameter restricts result to a given form (there can be more templates for one form).

- /rest/form-layouts/available[?form=<formName>&mineOnly=true]

  This URL returns a list of available templates. Every entry of the list contains the name of the form and name of the layout. The optional *form* parameter restricts results to a given form. Another optional parameter is *mineOnly*. If set to true, only templates owned by the current user are returned.

- /rest/form-layouts/form/count

  Functionality of this URL is similar to the first one, but this one returns number of distinct forms for which at least one template is available.

- **/rest/form-layouts/form/available[?mineOnly=true]**

  This URL is similar to the second one, except this one returns list of names of distinct forms for which at least one template is available. The *mineOnly* parameter can be used again.

5.5.2 Transferring Templates

Only the XML format is supported. The URL is following:

/rest/form-layouts?form=<formName>&layout=<layoutName>

A client can use all CRUD operations:

- GET downloads a template from the server.
- POST uploads a new template to the server.
- PUT updates a template owned by the current user.
- DELETE deletes a template owned by the current user.

For POST and PUT requests the content-type must be application/xml.

5.5.3 Transferring Data

The URL is following:

/rest/form-layouts/data?entity=<entityName>[&id=<recordId>]

It supports only the XML format again. Unlike with templates, only two operations are supported for this URL:

- GET downloads an odML document with all records of the given entity, or with just one record specified by the optional id parameter.
- POST uploads an odML document with a new record to the server.

In addition the service provides two more URLs so as a client can query for existing data records similarly to querying for available templates. They support both the XML and JSON responses.

- **/rest/form-layouts/data/count?entity=<entityName>**

  This URL returns the number of records for the given entity.

- **/rest/form-layouts/data/ids?entity=<entityName>**

  Returns a list of all records' IDs (for the given entity).

Chapter 6
Testing

Testing is an integral part of the software development process. It is also the most often neglected part. The most primitive way of testing, manual testing or debugging, consumes too much time and human resources and is much less reliable than automated testing. The only exception is the manual testing end-user interface, which often needs a human interaction.

This chapter describes how the implemented solution was tested. Unit testing was used primarily. It is described in the following section. In addition, a standalone software tool was used for the purpose of manual testing the implemented RESTful web service's endpoint. It is also described later in this chapter.

6.1 Unit Tests

Unit testing is a method of testing individual code units separately. In the object-oriented world the unit is usually an object. These objects are tested in isolation, without worrying about its role in the surrounding system [Ra06]. The goal is to verify that each object behaves correctly, which increases the probability of the correct behaviour of the whole system considerably.

According to [Ra06] unit tests must be:

- **Automated**, i.e. they can be run by a computer.
- **Repeatable**, i.e. that executing the same test under the same conditions must give the same result.
- **Self-verifying**, i.e. the test itself must know the expected result and decide whether it passed or failed.

There are many frameworks for unit testing in Java. JUnit, being one of the most popular ones, was used in this work. JUnit is very thoroughly described in [Ra06].

6.1.1 Testing Template Generator

The template generator library respects the standard Maven directory structure, i.e. tests are located under the **src/test/java** directory. Test cases for individual objects are placed in classes named after the tested class with the “Test” suffix. For example test cases for `ClassParser` are located in the `ClassParserTest` class. Test classes are placed in the same package hierarchy as the tested ones.

Unit tests are usually based on the evaluation whether a returned value equals an expected one. In Java objects are compared to equality using the `equals()` method. The first thing necessary to start writing tests is to implement the `equals()` and `hashCode()` methods in *value objects*. Value objects are those returned by methods as their results. In the template generator library, value objects constitute the internal model, i.e. they are located in the `cz.zcu.kiv.formgen.model` package.

After implementing `equals()` and `hashCode()` it is important to test these methods before using them in other tests. According to [Ra06] writing tests for `equals()` is quite complex, but a solution is to use `EqualsTester` provided by an open-source project GSBase[22]. Listing 16 shows how to test `equals()` with `EqualsTester`.

Next it is advisable to test as much functionality as possible, the more the better. The only exception are methods that are too simple to break. They are typically getters and setters. [Ra06] suggests not to test individual methods, but rather a behaviour or a use-case. This can be equal to testing a method, but not always.

```
@Test
public void testEquals() {
    Form a = new Form("name");         // control object
    Form b = new Form("name");         // object equal to the control one
    Form c = new Form("anotherName");  // not equal object
    Form d = new Form("name") { /* trivial subclass */ };
    new EqualsTester(a, b, c, d);      // test the equals() method
}
```

Listing 16: Testing `Form.equals()` *with* `EqualsTester`*.*

Testing a concrete behaviour includes at least testing a regular successful usage and a proper extreme case. The latter may be expected to throw an exception. Listing 17 shows such a test case for `ClassParser`. The parser is asked to parse `null` and

22 GSBase: http://gsbase.sourceforge.net

add it as a subform to `form`. It should throw an exception, which is first caught and `form` is checked whether it was not modified. The exception is then thrown. If the exception's type is `NullPointerException`, the test passes.

```
@Test(expected = NullPointerException.class)
public void testParse_addToFormNull() throws Exception {
    Form form = new Form("superForm");
    try {
        new ClassParser().parse(null, form);
    } catch (Exception e) {
        assertEquals(new Form("superForm"), form);
        throw e;
    }
}
```

Listing 17: Expecting a test case to throw an exception.

Table 6 gives an overview of implemented test cases for individual objects.

Package	Tested object	Number of test cases
cz.zcu.kiv.formgen.core	ClassParser	6
	DataParser	2
	PersistentObjectBuilder	1
	SimpleDataGenerator	4
	SimpleLayoutGenerator	11
	SimpleObjectBuilder	7
cz.zcu.kiv.formgen.model	Form	2
	FormData	2
	FormDataField	1
	FormField	2
cz.zcu.kiv.formgen.odml	Converter	4
	OdmlReader	2
	OdmlWriter	3

Table 6: Overview of tested objects and implemented test cases.

All implemented tests can be run using the following command line

```
mvn test
```

which runs all tests and prints their results. It should be run after every modification in the source code in order to verify that it has not caused a defect. Currently all tests are successful.

6.1.2 Testing EEGBase Extension

EEGBase, like the template generator library, is a standard Maven project, i.e. tests are located under the **src/test/java** directory. It follows the same naming conventions, appending the “Test” suffix after the name of a tested object.

Running tests for EEGBase requires Spring's context, because tested objects are designed to live inside Spring's container.. For this purpose a special test context is used. Beans and other required components are created within this context as usual, but it is possible to define some special configuration for the testing environment. For example the H2 in-memory database[23] is used instead of the Postgres database used for deployment.

During this work five types of objects were created in EEGBase – controller, service object, DAO object, data entity and marshallable data containers. The last two, data entity and containers, do not require any testing since they contain no logic. The controller provides an end-user interface of the web service and will be tested manually (see section 6.2 Manual Testing). It follows that the only objects for unit testing are the service and DAO objects.

The DAO object, `SimpleFormLayoutDao`, extends `SimpleGenericDao` which implements basic CRUD operations. For that reason they do not need to be tested within `SimpleFormLayoutDao`. The service object, `FormServiceImpl`, contains logic of the implemented web service. It uses the `SimpleFormLayoutDao` object to access stored templates. For that reason the DAO object was tested first. Test cases for service object assume DAO working correctly. An alternative approach would be usage of a mock DAO object.

Table 7 gives an overview of implemented test cases together with a short description. All implemented test cases pass successfully. They can be run with the maven command line `mvn test`.

23 H2 Database Engine: http://www.h2database.com/html/main.html

Tested object	Test-case method	Tested behaviour
`SimpleFormLayoutDao`	`testFormAccess()`	Querying for names and counts of stored forms.
	`testLayoutCounts()`	Querying for counts of stored templates by specified criteria.
	`testGetLayout()`	Retrieving a specified template.
`FormServiceImpl`	`testAvailableForms()`	Querying for available forms.
	`testAvailableLayouts()`	Querying for available templates.
	`testGetLayout()`	Retrieving a specified template.
	`testCRUDLayout()`	CRUD operations for templates.
	`testGetOdmlData()`	Retrieving data records.

Table 7: Overview of test cases implemented in EEGBase.

6.2 Manual Testing

The implemented web service's endpoint was tested manually in addition to unit tests described in the previous section. A standalone application called RESTClient[24] capable of testing web services was used especially in the initial stage of development. Furthermore, the service was tested using a mobile client developed by Jaroslav Hošek within his master thesis.

Following scenarios were used for the purpose of manual testing:

Scenario 1: Templates manipulation

1. Query available templates.
2. Download a chosen one.
3. Upload the template under a different name.
4. Query available templates to verify it was saved.
5. Update the template from step 3 by uploading a different content.
6. Download the updated template.
7. Delete the template.
8. Query available templates.

24 WizTools.org RESTClient: http://code.google.com/p/rest-client

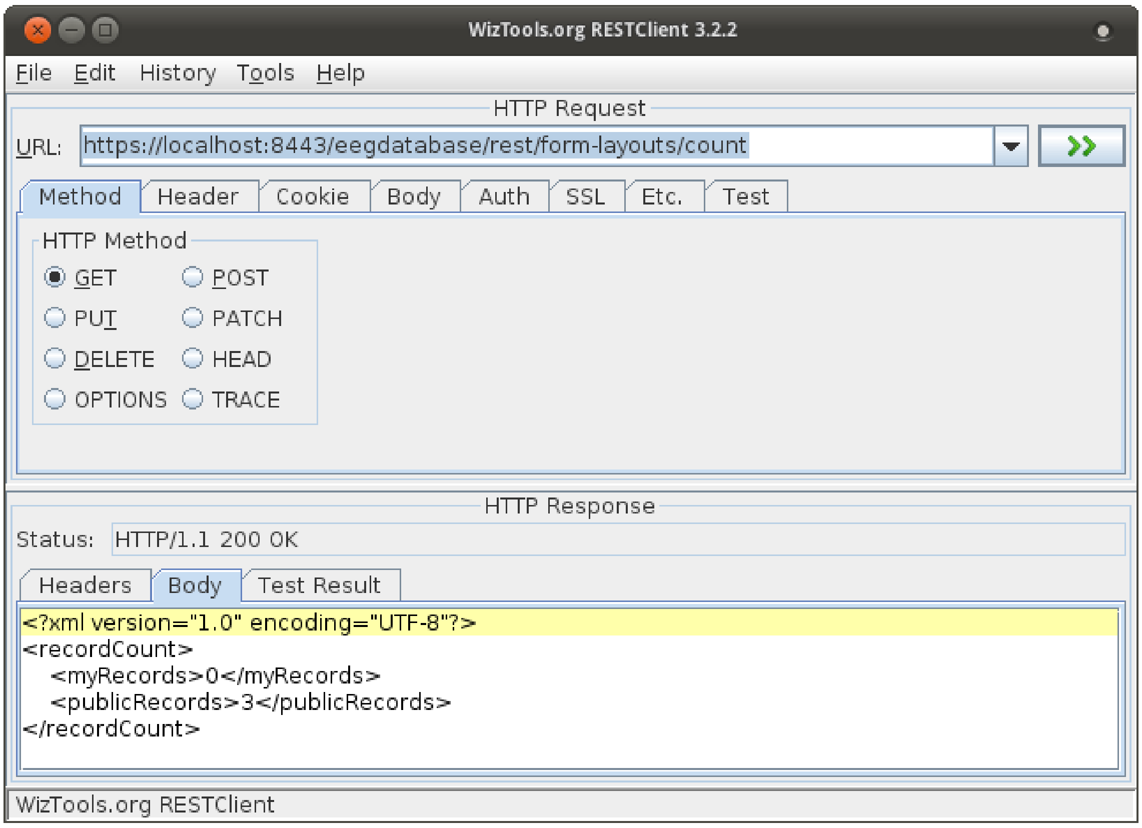
Scenario 2: Data manipulation

1. Download all records of any type referenced by a generated template.
2. Upload a new record.
3. Download the record from step 2 by its ID.

6.2.1 RESTClient

RESTClient is a Java application and a framework to test RESTful web services. The application offers both command line and graphical interface. It is capable to test variety of HTTP communications. Its features include support for all common HTTP methods, setting headers, cookies and body of a request, authentication, encryption with SSL and many other things.

Figure 9 depicts the graphical user interface after a successful query for the count of available layouts. The URL is entered in the upper part of the application window. In the central part the GET method is chosen from the combo-box. The lower part of the window shows the response from the server.


WizTools.org RESTClient 3.2.2
File Edit History Tools Help
HTTP Request
Method Header Cookie Body Auth SSL Etc. Test
HTTP Method
HTTP Response
Headers Body Test Result
WizTools.org RESTClient

URL: https://localhost:8443/eegdatabase/rest/form-layouts/count

HTTP Method: ● GET ○ POST ○ PUT ○ PATCH ○ DELETE ○ HEAD ○ OPTIONS ○ TRACE

Status: HTTP/1.1 200 OK

```
<?xml version="1.0" encoding="UTF-8"?>
<recordCount>
   <myRecords>0</myRecords>
   <publicRecords>3</publicRecords>
</recordCount>
```

Figure 9: RESTClient - graphical user interface.

The graphical application is easy to use and provides all required functionality to test the implemented web service. Although some requests can be tested in a web browser as well, it is not sufficient to test the whole public API. The only supported HTTP method after entering a URL in a web browser is GET, but the RESTful API supports also POST, PUT and DELETE. RESTClient allows a user to choose the required method very easily. It is also very simple to add required request headers, such as `Accept: application/json`, which is impossible or at least difficult to achieve with a browser.

6.2.2 Mobile Application for Android

Jaroslav Hošek has been developing a mobile application for Android in parallel with this project. The application will be described in his master thesis. It allows users, to design their own forms and use them to enter data from their experiments. This functionality is similar to a mobile application described in [Mi13]. In addition to that, the new application is also a client of the RESTful web service created within this work. During the development we have closely cooperated, mainly with defining the format of templates.

The developed mobile application was used to test the implemented RESTful API as its dedicated client. Revealed bugs, shortcomings and additional requirements were continuously fixed.

Chapter 7
Conclusion

The goal of this thesis was to propose and implement a framework for automatic generation of platform-independent graphical templates. They enable users to work with different data management systems without the need to implement a specialised application for each of them. The problem was presented in the context of collecting metadata from neuroscience experiments.

The very first challenge in this work was the selection of a format for the mentioned templates. After an exploration of the current state of the art we have ascertained there is not a satisfactory solution. For that reason we have decided to propose our own specification of required graphical templates. We have selected odML with its XML serialization as the underlying format. This selection has brought the advantage of using the same format for the transfer of both graphical templates and related data records (which is a primary purpose of odML).

The next step was a proposal and implementation of a tool for automated templates generation from an arbitrary data model. It was implemented using the Java programming language, which brought the advantage of working with an object-oriented data model. An analysis showed that the data model needs to be annotated so as the generation process can be controlled. Java annotations were used for this purpose. The tool parses the data model using reflection and provides odML templates output.

A use case verifying the implemented solution was also presented in this work. The tool has been deployed in a data management system for electrophysiological experiments called EEGBase. A communication API for client applications using RESTful web services was created. Besides template manipulation the web service also offers means to transport related data. A client application for Android has been developed in parallel within a separate thesis. The mobile client has successfully verified the implemented solution.

The goals of this thesis were achieved. Beyond the assignment it dealt with data transport which forms an integral part of all graphical templates use-cases. Both the template generation tool and the EEGBase extension were thoroughly tested and revealed bugs were fixed. The source code is clearly arranged and provided with documentation comments, and thus ready for future modifications and extensions by other programmers.

Possible future enhancements of the proposed solution includes an extension of the graphical templates specification. The presented one covers basic needs of our project, but some extra features might be useful, e.g. defining more types of form items. An alternative approach to enhancing the proposed format is to add support for another one such as HTML. The template generation tool was designed with regard to this possibility.

List of Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
BLOB	Binary Large Object
CLOB	Character Large Object
CRUD	Create, Read, Update, Delete
DAO	Data Access Object
EEG	Electroencephalography
ERP	Event-Related Potentials
GUI	Graphical User Interface
HQL	Hibernate Query Language
INCF	International Neuroinformatics Coordinating Facility
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
LOB	Large Object
MIME	Multipurpose Internet Mail Extension
ODML	Open MetaData Markup Language
ORM	Object-Relational Mapping
POJO	Plain Old Java Object
RDF	Resource Description Framework
REST	Representational State Transfer
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
WSDL	Web Service Description Language
WADL	Web Application Description Language
XML	Extensible Markup Language

Used software

Software	Description
Eclipse 4.3	Java IDE
GNU/Linux 3.2	Operating system
LibreOffice 4.2	Office suite
RESTClient 3.2	Tool for manual testing RESTful web services

References

[Au11] AUSTIN, Jim, Tom JACKSON, Martyn FLETCHER, Mark JESSOP, Bojian LIANG, Mike WEEKS, Leslie SMITH, Colin INGRAM and Paul WATSON. CARMEN: Code analysis, Repository and Modeling for e-Neuroscience. *Procedia Computer Science.* [online]. 1-3 June 2011, vol. 4, p. 768-777. ISSN 1877-0509. [accessed 25 Apr 2014]. DOI 10.1016/j.procs.2011.04.081. Available: http://www.sciencedirect.com/science/article/pii/S1877050911001396.

[Bj07] BJAALIE, Jan G. and Sten GRILLNER. Global Neuroinformatics: The International Neuroinformatics Coordinating Facility. *The Journal of Neuroscience.* [online]. 4 Apr 2007, 27(14), 3613-3615. [accessed 24 Jan 2014]. DOI 10.1523/JNEUROSCI.0558-07.2007. Available: http://www.jneurosci.org/content/27/14/3613.full.

[Fa05] FALLON, L.. Electroencephalography. *Gale Encyclopedia of Neurological Disorders.* 2005. Encyclopedia.com. [accessed 13 May 2014]. Available: http://www.encyclopedia.com/doc/1G2-3435200133.html.

[Fi00] FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures.* Irvine, 2000. Doctoral dissertation. University of California.

[Fo05] FORMAN, Ira R. and Nate FORMAN. *Java Reflection in Action.* Greenwich: Manning, 2005, 273 p. ISBN 19-323-9418-4.

[Ga08] GARDNER, D. et al. The Neuroscience Information Framework: A Data and Knowledge Environment for Neuroscience. *Neuroinformatics.* September 2008, vol. 6, iss. 3, pp. 149-160. DOI 10.1007/s12021-008-9024-z. Available: http://link.springer.com/article/10.1007/s12021-008-9024-z.

[Gi09] GIBSON, F., P. OVERTON, T. SMULDERS, S. SCHULTZ, S. EGLEN, C. INGRAM, S. PANZERI, P. BREAM, M. WHITTINGTON, E. SERNAGOR, M. CUNNINGHAM, C. ADAMS, C. ECHTERMEYER, J. SIMONOTTO, M. KAISER, D. SWAN, M. FLETCHER and P. LORD. Minimum Information about a Neuroscience Investigation (MINI): Electrophysiology. *Nature Precedings.* [online]. 9 Apr 2009. [accessed 15 Apr 2014]. Available: http://hdl.handle.net/10101/npre.2009.1720.2.

[Gr11] GREWE, J., T. WACHTLER and J. BENDA. A Bottom-up Approach to Data Annotation in Neurophysiology. *Frontiers in Neuroinformatics.* [online]. 30 Aug 2011, vol. 5, 18 p. ISSN 1662-5196. [accessed 3 Feb 2014]. DOI 10.3389/fninf.2011.00016. Available: http://www.frontiersin.org/Journal/10.3389/fninf.2011.00016/full.

[He08] HERZ, Andreas V. M., R. MEIER, M. P. NAWROT, W. SCHIEGEL and T. ZITO. G-Node: An integrated tool-sharing platform to support cellular and systems neurophysiology in the age of global neuroinformatics. *Neural Networks.* [online]. 2008, vol. 21, ISSN 1070-1075. [accessed 24 Feb 2014]. DOI 10.1016/j.neunet.2008.05.011. Available: http://www.g-node.org/publications/NN2436.pdf.

[Je08] JEŽEK, Petr. *Design of application for EEG processing (Návrh aplikace pro zpracování EEG signálů).* Pilsen, 2008. Master thesis. University of West Bohemia, Faculty of Applied Sciences.

[Je12] JEŽEK, Petr. *Ontology Development in EEG/ERP Domain.* Pilsen, 2012. Doctoral dissertation. University of West Bohemia, Faculty of Applied Sciences.

[Je13] JEŽEK, Petr, R. MOUČEK, Y. Le FRANC, T. WACHTLER and J. GREWE. Framework for automatic generation of graphical layout compatible with multiple platforms. *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on.* 15-19 Sept 2013, pp. 193-194. DOI 10.1109/VLHCC.2013.6645264. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6645264&isnumber=6645226.

[Ka13] KALIN, Martin. *Java Web Services: Up and Running.* 2nd ed. Sebastopol (California): O'Reilly, 2013, 338 p. ISBN 978-144-9365-110.

[Mi13] MIKO, Petr. *Mobile system for management of EEG/ERP experiments.* Pilsen, 2013. Master thesis. University of West Bohemia, Faculty of Applied Sciences.

[Mo14] MOUČEK, R., P. JEŽEK, L. VAŘEKA, T. ŘONDÍK, P. BRŮHA, V. PAPEŽ, P. MAUTNER, J. NOVOTNÝ, T. PROKOP and J. ŠTĚBETÁK. Software and hardware infrastructure for research in electrophysiology. *Frontiers in Neuroinformatics.* [online]. 7 Mar 2014, vol. 8. [accessed 5 Apr 2014]. DOI 10.3389/fninf.2014.00020. Available: http://www.frontiersin.org/Neuroinformatics/10.3389/fninf.2014.00020/abstract.

[Pe05] PECINOVSKÝ, Rudolf. *Java 5.0: Novinky jazyka a upgrade aplikací.* Brno: Computer Press, 2005, 152 p. ISBN 80-251-0615-2.

[Pe06] PEAK, Patrick and Nick HEUDECKER. *Hibernate Quickly.* Greenwich: Manning, 2006, 425 p. ISBN 978-193-2394-412.

[Ra06] RAINSBERGER, J. B. and Scott STIRLING. *JUnit Recipes: Practical Methods for Programmer Testing.* Greenwich: Manning, 2005, 721 p. ISBN 19-323-9423-0.

[RFC05] RFC 3986 (STD 66). *Uniform Resource Identifier (URI): Generic Syntax.* 2005. [online]. [accessed 2 Apr 2014]. Available: http://tools.ietf.org/html/rfc3986.

[RFC14] RFC 7159. *The JavaScript Object Notation (JSON) Data Interchange Format.* 2014. [online]. [accessed 4 Apr 2014]. Available: http://tools.ietf.org/html/rfc7159.

[Se06] SEARS, Russell, Catharine VAN INGEN and Jim GRAY. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem. *Microsoft Research.* 2006, 10 p. [online]. [accessed 8 Apr 2014]. Available: http://research.microsoft.com/pubs/64525/tr-2006-45.pdf.

[Wa08] WALLS, Craig and Ryan BREIDENBACH. *Spring in Action.* 2nd ed. Greenwich: Manning, 2008, 730 p. ISBN 19-339-8813-4.

Appendix A: Listings

The following listing contains a XML serialization of the sample form template from chapter 3.3.4 Sample Template (Figure 3).

```
<section>
    <type>form</type>
    <name>Form1</name>
    <reference>example.Person</reference>
    <property> <name>label</name> <value>Person<type>string</type></value> </property>
    <property> <name>previewMajor</name> <value>fullName<type>string</type></value> </property>
    <property> <name>previewMinor</name> <value>gender<type>string</type></value> </property>
    <section>
        <type>textbox</type>
        <name>fullName</name>
        <property> <name>label</name> <value>Full name<type>string</type></value> </property>
        <property> <name>datatype</name> <value>string<type>string</type></value> </property>
        <property> <name>required</name> <value>true<type>boolean</type></value> </property>
        <property> <name>cardinality</name> <value>1<type>int</type></value> </property>
        <property> <name>maxLength</name> <value>50<type>int</type></value> </property>
        <property> <name>id</name> <value>1<type>int</type></value> </property>
    </section>
    <section>
        <type>combobox</type>
        <name>gender</name>
        <property> <name>label</name> <value>Gender<type>string</type></value> </property>
        <property> <name>required</name> <value>true<type>boolean</type></value> </property>
        <property>
            <name>values</name>
            <value>M<type>string</type></value>
            <value>F<type>string</type></value>
        </property>
        <property> <name>id</name> <value>2<type>int</type></value> </property>
        <property> <name>idTop</name> <value>1<type>int</type></value> </property>
    </section>
    <section>
        <type>form</type>
        <name>address</name>
        <reference>example.Address</reference>
        <property> <name>label</name> <value>Address<type>string</type></value> </property>
        <property> <name>previewMajor</name> <value>town<type>string</type></value> </property>
        <property> <name>required</name> <value>false<type>boolean</type></value> </property>
        <property> <name>cardinality</name> <value>-1<type>int</type></value> </property>
        <property> <name>id</name> <value>3<type>int</type></value> </property>
        <property> <name>idTop</name> <value>2<type>int</type></value> </property>
        <section>
            <type>textbox</type>
            <name>town</name>
            <property> <name>label</name> <value>Town<type>string</type></value> </property>
            <property> <name>datatype</name> <value>string<type>string</type></value> </property>
            <property> <name>required</name> <value>true<type>boolean</type></value> </property>
            <property> <name>cardinality</name> <value>1<type>int</type></value> </property>
            <property> <name>maxLength</name> <value>50<type>int</type></value> </property>
            <property> <name>id</name> <value>4<type>int</type></value> </property>
            <property> <name>idTop</name> <value>6<type>int</type></value> </property>
        </section>
        <section>
            <type>textbox</type>
            <name>street</name>
            <property> <name>label</name> <value>Street<type>string</type></value> </property>
            <property> <name>datatype</name> <value>string<type>string</type></value> </property>
            <property> <name>required</name> <value>true<type>boolean</type></value> </property>
            <property> <name>cardinality</name> <value>1<type>int</type></value> </property>
            <property> <name>maxLength</name> <value>50<type>int</type></value> </property>
            <property> <name>id</name> <value>5<type>int</type></value> </property>
        </section>
        <section>
            <type>textbox</type>
            <name>number</name>
            <property> <name>label</name> <value>Number<type>string</type></value> </property>
            <property> <name>datatype</name> <value>integer<type>string</type></value> </property>
            <property> <name>required</name> <value>true<type>boolean</type></value> </property>
            <property> <name>cardinality</name> <value>1<type>int</type></value> </property>
            <property> <name>minValue</name> <value>1<type>int</type></value> </property>
            <property> <name>id</name> <value>6<type>int</type></value> </property>
            <property> <name>idTop</name> <value>5<type>int</type></value> </property>
        </section>
    </section>
</section>
```

Listing A.1: XML serialization of the sample form template.

Appendix B: CD

The enclosed CD contains source codes of the implemented solution and other related electronic resources. Its content is following (directories are enclosed in square brackets):

- **[EEGBase]** – snapshot of the EEGBase project (14 May 2014)
    - **[Source code]** – complete source code
    - **eegdatabase-2014-05-14-SNAPSHOT.war** – compiled WAR archive
- **[odML]** – odML related resources
    - **odml.xsd** – XML schema for odML
- **[odml-java-lib adaptation]** – adaptation of the odml-java-lib library
    - **[Release]** – release of the adapted library
    - **[Source code]** – complete source code
- **[Sample templates]** – sample odML templates from EEGBase
    - **experiment.odml** – generated form template for saving experiments
    - **data.odml** – sample document with data records
- **[Template generator]** – final version of the implemented tool
    - **[Example]** – example of use
    - **[Javadoc]** – javadoc documentation
    - **[Release]** – release of the tool including required dependencies
    - **[Source code]** – complete source code
- **[Thesis]** – this thesis
    - **[Source]** – source text of this thesis including used images
    - **MasterThesis_JakubKrauz.pdf** – text of this thesis in PDF
- **[Tools]** – various software tools used during this work
    - **[Apache Ant]** – build tool used in odml-java-lib
    - **[Apache Maven]** – project management tool
    - **[RESTClient]** – tool for testing RESTful web services