A Model-driven Approach to Design Multi-device Forms for Capturing Data in a Medical Study Environment

Master's Thesis

Florian Klein
Koblenzer Str. 1 – 56826 Lutzerath
KleinFlorian@live.de

Bonn, 25th of August 2013

in cooperation with
German Aerospace Center
Simulation and Software Technology

Rheinische Friedrich-Wilhelms-Universität
Institute of Computer Science III
Professor Dr. Armin B. Cremers

universitätbonn

A Model-driven Approach to Design Multi-device Forms for Capturing Data in a Medical Study Environment
Master's Thesis
Author: Florian Klein
Submission Date: August 25, 2013

First Supervisor
Prof. Dr. Armin B. Cremers
Rheinische Friedrich-Wilhelms-Universität Bonn
Institute of Computer Science III

Second Supervisor
Prof. Dr. Rainer Manthey
Rheinische Friedrich-Wilhelms-Universität Bonn
Institute of Computer Science III

In Cooperation with
German Aerospace Center (DLR)
Simulation and Software Technology
Distributed Systems and Component Software

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbständig angefertigt und nicht anderweitig zu Prüfungszwecken vorgelegt wurde. Außerdem wurden keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und wörtliche sowie sinngemäße Zitate als solche gekennzeichnet.

_____

Bonn, den 25.08.2013

Abstract

During medical studies a large amount of data has to be captured for later evaluation. Instead of doing this by pen and paper forms recent approaches use electronic data forms that directly digitize the entered data and store it in a central database. The usage of such electronic forms on mobile devices allows the study staff to carry the forms with them such that they do not have to move to dedicated data capturing stations inside the study facility.

The high diversity of available mobile devices (smartphones and tablets) requires developing electronic data forms in such a way that they adapt to the platform specific features. This is especially challenging due to the differences of the devices' display sizes. Furthermore, the devices offer different possibilities to connect peripheral devices for automating the data capturing process. These differences require a solution for designing multi-device data forms that are usable on the different device types without generating a dedicated form for each device from scratch.

This thesis presents a model-driven method that allows the design of multi-device data forms. The approach works on four model layers that are traversed one after another. Based on the definition of the data to be captured, an abstract version of the form's layout is gathered. This is refined by a concrete definition for each device type until the final user interface of the data form is generated. The transitions between the model layers are done by automatic transformation processes. The developed approach is prototypically implemented in the context of a study management software at the German Aerospace Center (DLR).

The results of a usability study that tested the developed model-driven approach concerning its applicability for the future user group show that the test users were able to complete the given task at an average of about 86%. The test users additionally expressed a high satisfaction when using the implemented prototype. Thus, this model-driven method is assumed to be applicable to design multi-device data forms for use in a medical study environment.

Kurzfassung

Während der Durchführung medizinischer Studien fällt eine große Menge an Daten an, die zur späteren Auswertung erfasst werden. Moderne Methoden der digitalen Datenerfassung ersetzen die händische Erfassung mit Hilfe von Papierformularen durch elektronische Eingabemasken. Dabei werden die Daten in einer zentralen Datenbank gespeichert. Der Einsatz solcher elektronischer Formulare auf mobilen Endgeräten erlaubt es den Studienmitarbeitern die Formulare bei sich zu tragen, was ihnen das Aufsuchen spezieller Datenerfassungsstationen innerhalb der Studienanlage erspart.

Aufgrund der großen Vielfalt mobiler Geräte (Smartphones und Tablets) müssen elektronische Formulare so entwickelt werden, dass sie den gerätespezifischen Eigenarten gerecht werden. Die unterschiedlichen Displaygrößen der Geräte machen dies zu einer besonderen Herausforderung. Darüber hinaus unterscheiden sich die Geräte in ihren Möglichkeiten zur Einbindung externer Geräte, mit deren Hilfe der Vorgang der Datenerfassung teilweise automatisiert werden kann. Um die Entwicklung separater Formulare für jede Geräteart zu vermeiden, wird eine Lösung zur Gestaltung geräteunabhängiger Formulare benötigt.

Diese Arbeit stellt eine modell-getriebene Methode vor, welche die Entwicklung solcher geräteunabhängiger Formulare ermöglicht. Diesem Ansatz liegen vier Modellebenen zugrunde, die nacheinander durchlaufen werden. Basierend auf der Definition der zu erfassenden Daten wird eine abstrakte Version des Formularlayouts abgeleitet. Dieses wird dann spezifisch für jede Geräteart konkretisiert, um letztlich ein gerätespezifisches Formular automatisch zu generieren. Der Übergang von einer Modellebene in die nächste erfolgt mit Hilfe automatischer Modelltransformationen. Der entwickelte modell-basierte Ansatz wurde im Rahmen einer Studienmanagement Software des Deutschen Zentrums für Luft- und Raumfahrt (DLR) prototypisch implementiert.

Die entwickelte Methode wurde mit Hilfe einer Benutzerstudie bezüglich ihrer Anwendbarkeit für die spätere Benutzergruppe evaluiert. Die Ergebnisse der Studie zeigen, dass die Testnutzer die gegebene Aufgabenstellung im Durchschnitt zu 86% lösen konnten und eine große Zufriedenheit bei der Benutzung des Prototyps äußerten. Dies lässt die Annahme zu, dass der erarbeitete modell-getriebene Ansatz zur Gestaltung elektronischer Formulare für den Einsatz in medizinischen Studien geeignet ist.

Danksagung

Zu allererst möchte ich mich bei den beiden Gutachtern meiner Master Thesis Prof. Dr. Armin B. Cremers und Prof. Dr. Rainer Manthey von der Rheinischen Friedrich-Wilhelms-Universität Bonn bedanken, die mir diese Arbeit erst ermöglicht haben.

Außerdem gilt mein Dank meinem Betreuer Dr. Tobias Rho, ebenfalls von der Rheinischen Friedrich-Wilhelms-Universität Bonn, sowie meinen Kollegen beim Deutschen Zentrum für Luft- und Raumfahrt (DLR), insbesondere Jan Flink, Thomas Sauerwald und Doreen Seider, für die Unterstützung und die fachlichen Ratschläge. Ebenfalls danke an Andrea Nitsche für das Korrekturlesen.

Mein besonderer Dank geht an die Mitarbeiterinnen und Mitarbeiter des Instituts für Luft- und Raumfahrtmedizin und der Einrichtung für Simulations- und Softwaretechnik, die die Zeit gefunden haben an der im Rahmen der Arbeit durchgeführten Benutzerstudie teilzunehmen.

Nicht zuletzt möchte ich mich auch bei meiner Familie bedanken, deren Unterstützung während der gesamten Zeit meines Studiums mich stets vorangetrieben hat. Danke auch meinen Freunden, die mich – sicherlich manchmal unwissentlich – motiviert und getrieben haben am Ball zu bleiben

Table of Contents

List of Abbreviations

AFE	**A**bstract **F**orm **E**lement
AFM	**A**bstract **F**orm **M**odel
AID	**A**utomated **I**nput **D**evice
AMSAN	**A**rbeits**m**edizinische **S**imulations**an**lage (Simulation Facility for Occupational Medicine Research)
AUI	**A**bstract **U**ser **I**nterface
CAMELEON	**C**ontext **A**ware **M**odeling for **E**nabling **L**everaging **E**ffective Interact**ion**
CFE	**C**oncrete **F**orm **E**lement
CFM	**C**oncrete **F**orm **M**odel
CRF	**C**AMELEON **R**eference **F**ramework
CUI	**C**oncrete **U**ser **I**nterface
DDM	**D**ata **D**efinition **M**odel
DLR	**D**eutsches Zentrum für **L**uft- und **R**aumfahrt (German Aerospace Center)
DOM	**D**ocument **O**bject **M**odel
DSL	**D**omain **S**pecific **L**anguage
FUI	**F**inal **U**ser **I**nterface
GUI	**G**raphical **U**ser **I**nterface
HTML	**H**yper **T**ext **M**arkup **L**anguage
IDE	**I**ntegrated **D**evelopment **E**nvironment
IDM	**I**nput **D**evice **M**odel
ISS	**I**nternational **S**pace **S**tation
lp	**L**ogical **P**ixel
M2M	**M**odel **to** **M**odel
M2T	**M**odel **to** **T**ext
MBUID	**M**odel **B**ased **U**ser **I**nterface **D**evelopment
mCFM	**M**obile **C**oncrete **F**orm **M**odel
MDSD	**M**odel **D**riven **S**oftware **D**evelopment
MED	**M**odel **E**lement **D**efinition
OASIS	**O**rganization for the **A**dvancement of **S**tructured **I**nformation **S**tandards
OMG	**O**bject **M**anagement **G**roup
OSP	Microsoft **O**pen **S**pecifications **P**romise
T4	**T**ext **T**emplate **T**ransformation **T**oolkit

TCT	**T**ask **C**ompletion **T**ime
UI	**U**ser **I**nterface
UIDL	**U**ser **I**nterface **D**escription **L**anguage
UIML	**U**ser **I**nterface **M**arkup **L**anguage
UML	**U**nified **M**odeling **L**anguage
USIXML	**U**ser **I**nterface **E**xtensible **M**arkup **L**anguage
W3C	**W**orld **W**ide **W**eb **C**onsortium
WPF	**W**indows **P**resentation **F**oundation
XAML	**E**xtensible **A**pplication **M**arkup **L**anguage
XIML	**E**xtensible **I**nterface **M**arkup **L**anguage
XML	**E**xtensible **M**arkup **L**anguage
XSL	**E**xtensible **S**tylesheet **L**anguage
XSLT	**E**xtensible **S**tylesheet **L**anguage **T**ransformation

List of Figures

List of Tables

List of Listings

1 Introduction

This introductory chapter first gives an overview of the working environment and describes the motivation for this thesis. Subsequently, the objectives and limitations are formulated. The chapter finishes with an overview of the thesis' further structure.

1.1 Working Environment

The thesis was written in cooperation with the German Aerospace Center (DLR) at the facility for *Simulation and Software Technology*. DLR is Germany's national aeronautics and space research center. The main research areas are aeronautics, space, energy, transport and security. Currently, DLR has about 7400 employees at 16 national locations and four offices in foreign countries. [DLR13]

The thesis is originated within a software development project which is called :study. The goal of the :study project is to develop a study management software that fully supports the course of a medical study [DLRb]. The expert scientists of the DLR *Institute of Aerospace Medicine* are the main target group of the software. They carry out scientific studies regarding the effects of long term stays in space to the human body (for example on the *International Space Station*, ISS) and explore possible countermeasures [DLRa]. The :study software facilitates and improves the study specific workflows and thereby supports the study staff in their daily work. This is achieved by simplifying the study planning process and the data collection by appropriate applications and devices. One of the core features of the :study software is to facilitate the acquisition of data during a study via electronic data forms. Here it is important that these data forms can be designed by the expert scientists or the study staff themselves. This enhances the flexibility when performing a study and reduces the dependency on computer science experts from other departments. For this purpose, a simple graphical data form editor was developed. For the actual data capturing process, the generated forms are displayed by another application. This software runs on some stationary PCs with touch displays in the study facility. A more detailed introduction into medical studies and the supporting features of the :study software is given in Section 2.1.

1.2 Motivation

Data capturing in the course of medical studies is currently done by pen and paper forms. For the later evaluation, the entered values are manually transferred to decentralized electronic data storages. This process is susceptible for transferring errors and not efficient due to the need of capturing each value twice (once on the paper form and again into an electronic representation). The :study software already solved these problems by offering a system for generating and using electronic data forms. Thereby, the entered values are directly digitized and stored in a central database. Whenever possible, the values are directly read from peripheral devices, such that manual entries become obsolete. The drawback of the current solution is that the electronic

data forms can only be displayed on desktop systems that are installed at dedicated places in the study facility. This does not allow the study staff to capture data where it is actually measured.

Using mobile devices like smartphones and tablets for data capturing in a medical study environment could improve the standard workflow and generate completely new application areas. The study staff can enter data directly where it is collected, such that they do not have to move to any data capturing station. This combines the advantages of the classical pen and paper forms with the ones of the already developed electronic approach.

Due to their features, mobile devices offer even more potential. The decreasing costs make it possible to equip every single study staff member with them. This means that the smartphone or tablet is always used in the context of a specific person, which is an essential difference to the data capturing stations that are used by several people. The personal context information can be used to automatically open an appropriate data form at a specific point in time. Additionally, the devices' internal sensor system can be used to detect the physical environment. Thereby, entered data can automatically be assigned to the right context. These opportunities not just reduce the effort for the user; they also minimize erroneous inputs.

The usage of mobile devices for data capturing requires a possibility to design electronic data forms that are usable on several device types. Thus, the data forms need to be designed in such a way that they transform well if they are displayed on a smartphone, a tablet, or a desktop PC. The enormous difference in the display sizes of these device types makes the design of such multi-device data forms a serious challenge. In addition to the mentioned device types, the diversity of different platforms that are available for each device type requires a platform independent development of the data forms in order to not restrict the users to a specific platform. Available smartphone platforms are for example Android, iOS, or Windows Phone.

In the following course of this thesis, the term "device" stands for the different device types (smartphone, tablet and desktop) whereas the word "platform" denotes the different operating systems and technologies that are available for each device. The notion "target platform" has to be understood as the combination of a specific platform, running on one of the device types.

1.3 Objective

The goal of this thesis is to explore an approach to easily create multi-device data forms for use in a medical study environment. Since the data, which should be captured by a data form, stays the same on the different devices, it would be a significant effort to develop completely independent data forms for each device. This would be a very tedious and non-user-friendly solution. Instead, the common parts of a data form that are consistent for each device and platform should be developed just once. Only the device specific parts should be treated specifically by the designer. Nevertheless, the design of the data forms should be adaptable to the specific requirements of a

device and be consistent with the design guidelines and visual concept of a certain platform.

One of the main differences between the devices is the display size and resolution. The variation of display sizes reaches from full size monitors over reduced tablet screens to small smartphone displays. This is even more disadvantageous because the already small screen of mobile devices is nowadays usually not just used as an output device but also as the (sometimes only) input device in form of a touchscreen. This means that in addition to the data form, also a touch-keyboard has to fit on the screen. Due to these circumstances, it is not possible to simply display the same data forms on a mobile device as on a desktop PC, because this would lead to a very small depiction of the form. As a result the user would often have to zoom in and out which might cause him to lose overview. This would be neither productive nor user friendly. Instead, there should be adapted versions of a data form that meet the specific requirements of a smartphone, a tablet and a desktop PC. Therefore, an important challenge to be solved is to have an appropriate layout of the form's elements on each device. The developed approach should support the designer in generating the device specific layouts.

Also the integration of peripheral devices into multi-device data forms is challenging. An example for this is an electronic laboratory balance whose measured value is automatically entered into the data form, such that the user does not have to transfer the value manually. Due to the different features of mobile devices concerning the possibilities to connect peripheral devices, the approach for designing multi-device data forms also has to deal with these differences. For example, it might be impossible to connect a laboratory balance to a smartphone in the same way as to a desktop PC because the smartphone does not provide the required interface. Instead, it might support Bluetooth communication. So the balance could be integrated using this technology. Some of the missing options to connect peripheral devices to smartphones and tablets can possibly be compensated by using internal sensors or wireless connection technologies, which are often available on mobile devices.

The multi-device data form design approach is exemplarily developed in the context of the :study software. The evaluation of the approach regarding its applicability for the future users and general problems is done by a usability study. Therefore, the important parts of the approach are prototypically implemented. This includes supporting the development of data forms for at least two device types and the integration of peripheral devices to the data forms.

1.4 Limitations

Due to restricting facts regarding the environment and especially the available time for this thesis, there are some limitations to the prototypical implementation as well as to the user study:

For historical and DLR internal reasons, the :study software currently focuses on a homogeneous Windows based environment. Therefore, it is reasonable to firstly integrate Windows based mobile end user devices (Windows Phones and Windows based

tablets). Thus, the prototypical implementation only supports the generation of data forms for these target platforms. Nevertheless, it has been taken care that the acquired approach is also applicable for other platforms like Android or iOS.

Caused by the limited time, the practical realization focuses on the design process of the data forms. This means that the prototypical implementation concentrates on the development of an application for designing multi-device data forms according to the developed approach, which is used for the evaluation by a usability study. The actual application for data capturing on the different target platforms is not of major importance for this thesis and is implemented just rudimentarily to allow the participants of the user study to view their results.

Another limitation concerns the number of possible study participants. Since the target group of the software is rather small, also the number of available test users is limited. Thus, an empirical analysis of the user study is not possible. But, according to Tullis and Albert ([TA08], p. 119), even with a small number of test users it is possible to detect the major usability problems. As the goal of the user study is to evaluate the basic concept, and not to identify particular implementation problems, a small test user group is assumed to be adequate.

1.5 Thesis Outline

Before going into detail, the following paragraphs give a short overview of this thesis and its structure:

Chapter 2 (Foundations) describes foundations that are needed for understanding the content of this thesis. This includes an introduction to medical studies as well as the explanation of basic concepts and technologies that are used throughout the thesis.

In **Chapter 3 (Related Work)**, related work concerning the development of platform-independent user interfaces is presented. Especially model-based methods are described.

Chapter 4 (Conception) explains the overall concept for designing multi-device data forms that was devised during this thesis. This includes the description of similarities and differences to the approaches introduced in the Related Work chapter. Furthermore, the integration of the developed concept into the :study software is described.

Chapter 5 (Implementation) describes the implementation of a prototype that supports the development of multi-device data forms according to the approach introduced in the previous chapter. The overall concept of the prototype is illustrated and the basic parts of the implementation are explained by selected examples.

Chapter 6 (Evaluation) is about the usability study that was conducted for evaluating the developed approach. The study setup is described and the results are discussed.

Finally, **Chapter 7 (Summary and Future Work)** summarizes the results of this thesis and gives a prospect of possible enhancements and future work.

2 Foundations

This chapter starts with a description of basic concepts that are important for understanding the further deliberations in this thesis. This includes an introduction into medical studies as well as an overview of *Model Driven Software Development* (MDSD). Following, the fundamental technologies used for the prototypical implementation are briefly described.

2.1 Medical Studies

The term “medical study” is very extensive. Therefore, the following paragraphs explain what has to be understood by a medical study in the context of this thesis.

The typical goal of a medical study is to explore medical methods regarding their efficiency. The DLR Institute of Aerospace Medicine focuses on the effects of aerospace and road traffic on persons like astronauts, pilots or drivers [DLRa]. To ensure their healthiness, possible physical impacts due to their working environment have to be investigated and countermeasures have to be explored.

Throughout a medical study a group of study participants is located at a confined study location. This ensures that the test subjects live in a controlled environment which is important to achieve proper results. During this time, the participants are subject to medical interventions. The group of study participants is typically divided into two or more subgroups. One subgroup received a specific treatment and the other group is treated differently or gets placebos. This allows to compare the groups and to draw conclusions about the investigated treatment.

The realization of a medical study contains many different aspects that are supported by the :study software. Therefore :study is not one single application but a software system that provides dedicated programs for each aspect. The following sections describe the most important aspects in the context of this thesis and introduce the :study applications that target them. The applications share a common database that holds all data that accumulate during a study. To access this database the applications use a central web service.

2.1.1 Study Planning

Prior to the execution of a medical study a detailed study protocol is developed. This protocol defines all activities that are done by or with the study participants during the study. The notion “activity” in this context denotes all actions and examinations that are concerned with the test subjects. This reaches from simple tasks like getting up in the morning or having lunch, to complex medical examinations like a run on a human centrifuge. The study protocol can be conceived like a schedule at a university. It defines which study participant has which activity at which study day and at which time. Furthermore, the protocol contains information about the room in which an activity takes place (for example a laboratory) and which member of the study crew is responsible for an activity. An excerpt of a study protocol is attached in Appendix A.

The example illustrates the level of detail of such a protocol. The :study software supports the development of a study protocol by the :studycreator application.

Each of the planned activities is an instance of a predefined activity pattern. These activity patterns are used across different studies and contain basic data about the activities of this pattern. This includes a short description of the objective, information about the resources that are needed to perform the activity and a list of parameters that are collected for activities of this pattern during a study. A parameter can be seen as the structure of a captured value. It defines the data type of the value, in which unit the value is captured and stored and possibly some logical thresholds, which must not be exceeded when entering the value. An example for a parameter is the weight of a urine sample, which is stored as a decimal number with the unit gram (g) in the :study database. Figure 1 shows an excerpt of the Entity Relationship Model of the :study database. The depicted part illustrates the important relations between activities, their patterns and parameters. The gray tables contain master data that is reused across multiple studies. The red one belongs to study planning data, and the actual study data is depicted in blue color.

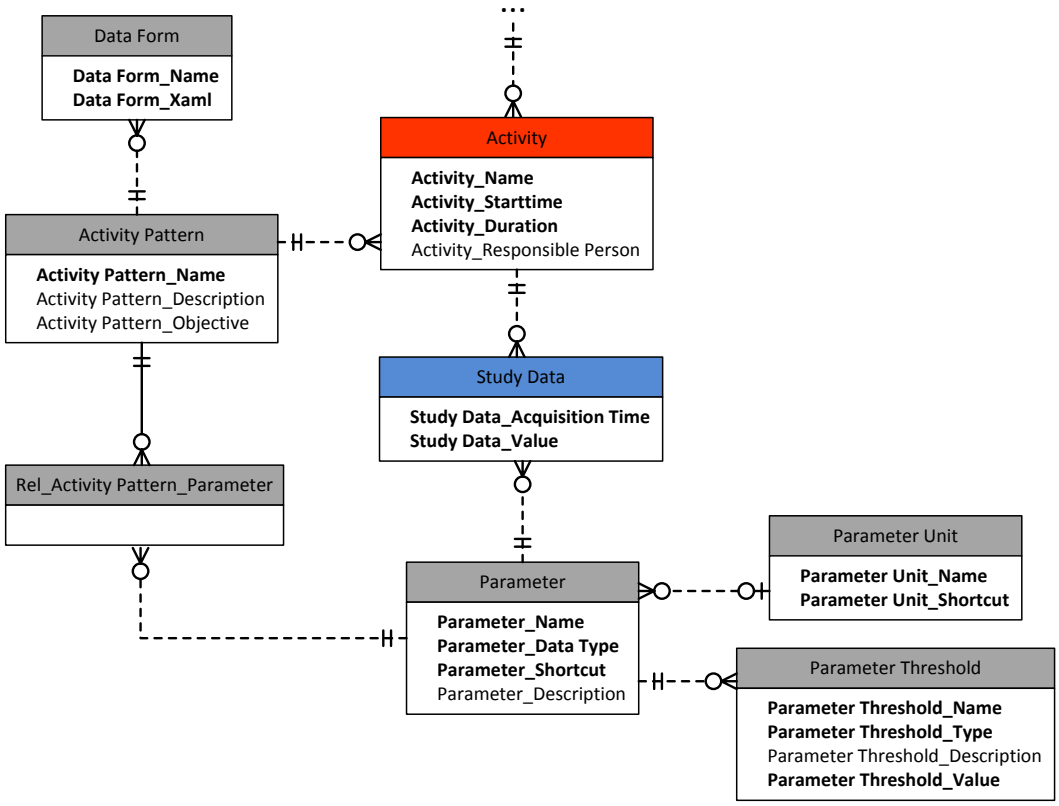


Figure 1: Relevant tables of the :study database

Due to the connection of activities to activity patterns and thereby to parameters, the study protocol already provides meta-information for the values captured during the study. In other words, in the planning phase an activity can be seen as an empty container, for which it is well defined what kind of values have to be added at which point in time during the study execution phase.

For each of the planned activities that require capturing some data, a data form is developed using the :studyforms application. The functionality of :studyforms is limited to the development of the data forms. For displaying the forms to the user and filling in the values the separate :studydata application is available (see Section 2.1.2). In the following course of this thesis, the user of the :studyforms application is also called designer.

The :studyforms application offers an easy-to-use graphical editor for designing the data forms. The user interface is similar to a conventional GUI designer (*Graphical User Interface*) like it is known from common *Integrated Development Environments* (IDE). However, the handling and the available features are much simpler. This is of special importance, because the software is not meant to be used by programmers or GUI designers but by the scientific and supporting members of the study staff. These people do not have a computer science background or experience in using such software. Figure 2 shows the main window of the :studyforms application and illustrates the graphical editor surface.

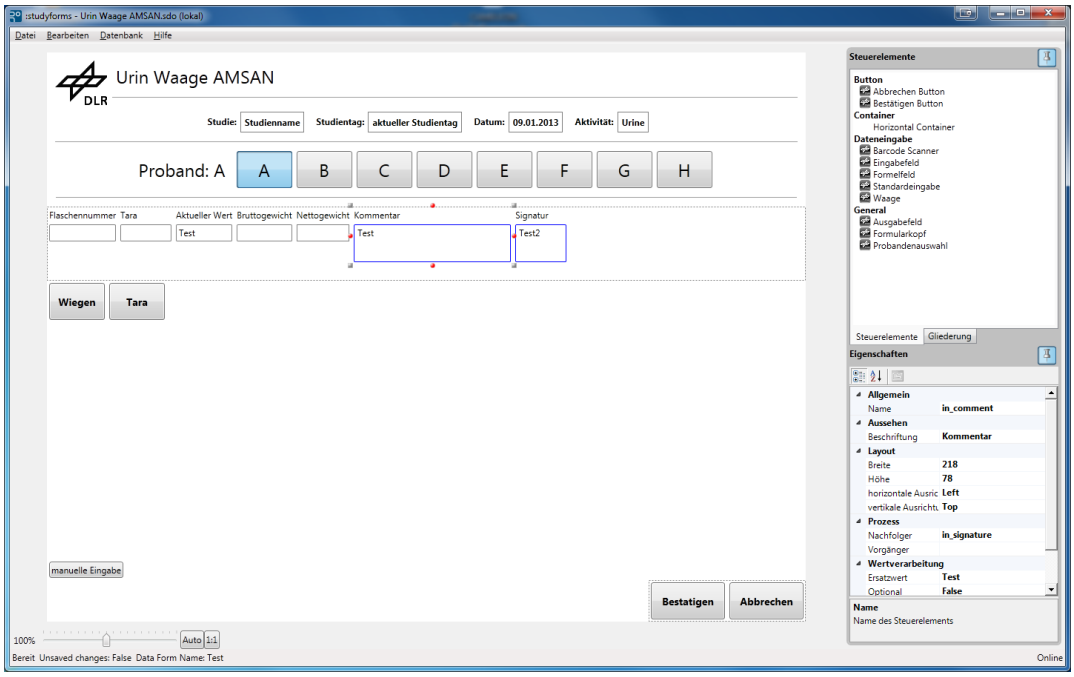


Figure 2: Graphical editor of the :studyforms application

A user starts to design a new data form by first specifying for which activity pattern the form is developed. This limits the available parameters that can be captured by the form because the definition of an activity pattern contains a set of related parameters (compare Figure 1). It is possible that the values of one activity pattern are collected in multiple steps using different data forms. This means that a data form does not have to provide input elements for all parameters of the activity pattern. The other way round, it is not possible that a data form captures values of parameters, which are not assigned to the specified activity pattern.

After setting up the new form, the designer arranges the different elements on the data form using drag and drop. These elements are not simple GUI widgets but more complex interaction elements. The input element, which offers the user the possibility to

enter data manually, not only consists of a text box for entering the actual value, but also includes a caption, describing which value should be entered. Thereby the designer does not have to deal with too many different elements on the form. Additionally, this approach ensures that different forms get a similar look since the design opportunities are restricted. Besides such manual interaction elements, there also exist special elements that make peripheral devices like a barcode scanner or an electronic balance available on the data form. This simplifies the data capturing process by automatically acquiring data using these devices.

The relation between the values entered into the data form and the parameters of the activity pattern is established by assigning the appropriate parameters to the interaction elements by which the values are collected.

2.1.2 Data Capturing

During the study execution phase the previously designed electronic data forms are displayed by the :studydata application. This application thereby builds the connection between the data form and the central :study database. Figure 3 shows the typical data capturing process during a study. The user opens a data form that supports data acquisition for the samples he is currently processing. Since the samples are related to a specific study participant, he first selects the correct one. Then the ascertained data is entered. After that, the user commits these data to the system and proceeds with the next sample until he has finished. If the user accidentally enters the values of a wrong sample, he can reset the whole data form without committing these values.

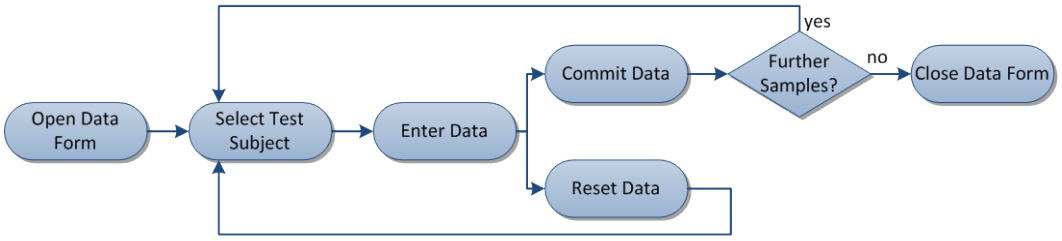


Figure 3: Typical data capturing process in a medical study environment

The submitted values are stored to the :study database and assigned to the activity for which they are acquired. This assignment is done based on the information about the study participant which is selected on the data form and the activity pattern to which the data form is linked. An additional aspect that is taken into account for the association of the data is the acquisition moment. Since the study protocol provides a detailed time schedule of the activities, it is thereby possible to find the right activity in most cases automatically.

2.1.3 Running Example

A very descriptive example of a typical data collection process during a medical study is the acquisition of values for 24h urine samples. This example is used throughout this thesis to explain requirements and parts of an electronic data form

that allow a laboratory assistant to enter the determined values. To get a better impression, the following paragraphs shortly describe the 24h urine examination.

At the beginning of a 24 hours period a study participant has to empty his bladder into the toilet. From this time on, the urine of the participant is collected over a period of 24 hours. The samples are provided in special urine bottles that are closed with a cap. During a study, the participants are usually located in a closed area, which is the *Simulation Facility for Occupational Medicine Research* (AMSAN) at DLR. Here, the samples are collected in a refrigerator until the end of the 24 hours period. In some study settings, already during this sampling process some information like a comment about the sample's visual impression (clear, cloudy, etc.) are recorded.

After the 24 hours collection period, the samples are handed over to the laboratory where they are analyzed by a laboratory assistant. This analysis includes determining a number of specific indicators that are defined in advance of the study. Typical indicators for a urine sample are the pH-value or the hemoglobin value. Furthermore, the samples are weighed. Here it is important, that the net weight of the sample is determined. This means, that the tare weight of the urine bottle, including the cap, has to be known in advance. After determining all relevant indicators, the sample might be aliquoted[1] and frozen for further tests. A simplified version of a data form for capturing the determined values of 24 hours urine samples at the laboratory is shown at the screenshot in Figure 2.

2.2 Model Driven Software Development

Model Driven Software Development (MDSD) describes an approach for generating software by specifying its functionality on an abstract layer using formal models. Stahl et al. define the term MDSD as a generic term for technologies that automatically generate software from formal models ([SVEH07], p. 11).

In this context, a formal model means that a model used in MDSD describes a certain aspect of the software completely. For this purpose, often UML models (*Unified Modeling Language*) are used. Based on one or several such models, the result of a MDSD process is a piece of executable software. In general this can be achieved by generating source code of some kind of programming language out of the defined models or by using an interpreter that reads a model at runtime and executes actions depending on the content of the model. The generation of source code is usually done by automated transformation processes that are integrated into the build process. The general intention of MDSD is to program on a more abstract layer and thereby unify the software architecture and enlarge the interoperability ([SVEH07], p. 13ff). Due to the usage of automated transformations, recurrent patterns in the software are translated to similar source code structures. Once implemented, the transformations can also be used by similar software development projects. This enlarges the reuse of already done work which is usually more difficult with directly implemented source

[1] To aliquot denotes the process of dividing a sample into a set of sub samples that are examined.

code. Here it has to be mentioned, that the implementation of a software using MDSD consist of automatically generated and manually implemented source code parts because models do not describe the whole system ([SVEH07], p. 13).

The *Object Management Group* (OMG)[2] standardized a MDSD approach called *Model Driven Architecture* (MDA). With this standardization, the OMG tries to enhance the ability of developing platform independent software. Therefore, the MDA approach defines four different modeling levels. The highest layer defines the software features in a very abstract fashion that is close to the notion of the domain experts, for which the application is developed. The lower model layers get more and more specific in the sense of leaving the notion of the domain experts and adding implementation details, until the last model layer actually represents the implementation by a concrete programming language. The step from one model layer to the next is done by using *Model to Model* transformations (M2M). The development of a software version for a different target platform can theoretically be achieved by simply changing the transformations between the model layers in a way, that at the end of the transformation process the source code for the new platform results. In practice, this often becomes more complicated because the models are not completely platform independent but contains platform specific parts (compare [SVEH07], p. 15). More information about the MDA approach can be found at the OMG website (see [OMG13b]).

2.2.1 Domain Specific Languages

Closely related to the definition of formal models is the usage of *Domain Specific Languages* (DSL). A DSL is a programming language that targets a special application area. In general, a DSL is specified by a meta-model that defines the available meta-model elements and their interdependencies. This is denoted as the abstract syntax of the DSL that formally defines the structure of the domain. Furthermore, the meta-model includes the static syntax of the DSL, which defines constraints that a model has to match in order to be well-formed. The actual models are implemented using a concrete syntax. This can be a textual or a graphical syntax. Thus, a DSL can contain several concrete syntaxes but always is defined by exactly one abstract syntax. The last part of a DSL is the dynamic semantics, which specify the meaning of the different language elements defined by the meta-model. Meta-models are often defined using UML. Another possible solution is to use XML based models that use *XML Schema Definitions* (see Section 2.3.1) as meta-models. More information about Domain Specific Languages is given in ([SVEH07], p. 97ff).

2.2.2 Transformations

When using multiple model-layers the transition from one model layer to the next is done by automated transformations. This also holds for automatic changes that are applied to a model (refactoring) or for generating the final source code. In general it is

[2] OMG is an international non-profit consortium for standardization in the computer industry. For more information see [OMG13a].

distinguished between *Model to Model* (M2M) transformations that transform one or several source models to a target model and *Model to Text* (M2T) transformations, which generate source code on the basis of a source model. Usually, transformations are implemented using special transformation languages. These languages are specifically optimized for projecting the elements of the source model to the target model. In the course of the Model Driven Architecture approach, the OMG defines the *Query View Transformations* (QVT) language (see [OMG11]). Stahl et al. give a detailed description of Model to Model transformations (see [SVEH07], p. 195ff).

2.3 Used Technologies

The following sections briefly introduce the most important technologies that are used for the prototypical implementation of the model-driven data form design approach. The descriptions are limited to those aspects that are needed to understand the explanations of the implementation in Chapter 5.

2.3.1 XML Schema Definitions

XML Schema Definitions (XSD) is a W3C[3] standard for defining the structure of XML documents [W3C04]. The schema definition itself is also an XML document. This section shortly introduces the XSD concept.

Generally speaking, an XML Schema Definition allows specifying the elements that can be part of an XML document that conforms to the schema definition. Additionally the element hierarchy is defined. The content of each element is specified by its type. XSD distinguishes between simple and complex types. Simple types are predefined basic data types. In addition to these predefined simple types it is possible to define own simple types by restricting another simple type. A complex type defines a new element structure with attributes and sub elements. Optionally, complex types can be based on another complex type that can be extended or restricted.

The usual structure of an XSD document is explained by a simple example. Listing 1 shows a short XML document containing information about the books of a library. Each book has an author and a title as attributes. The XML Schema Definition of the library XML document is given in Listing 2. For the namespace prefix of XSD elements usually the abbreviation *"xs"* is used.

```
 1  <library>
 2     <book>
 3        <author>Bongers, Frank</author>
 4        <title>XSLT 2.0 and XPath 2.0</title>
 5     </book>
 6     <book>
 7        <author>Klein, Florian</author>
 8        <title>A Model-driven Approach to Design …</title>
 9     </book>
10  </library>
```

Listing 1: Example library XML Document

[3] W3C stands for *World Wide Web Consortium*. This is an international community for developing web standards.

The root element of each XSD document is the `xs:schema` element. Inside this root element, the element and type definitions are located. In the library example, the names of the authors are written according to the pattern *"Surname, Forename"*. Therefore, a simple type restricting the `xs:string` type is specified that matches this pattern using a regular expression (Listing 2, l. 3-7). The structure of a book element is defined by a complex type (Listing 2, l. 9-14). The `xs:sequence` element denotes that the author and title sub elements have to be contained in this order in the book element. Finally, the library root element is specified using the `xs:element` XSD element (Listing 2, l. 16-22). This element contains a sequence of book elements. The type of these elements is set to the previously defined book complex type. The `maxOccurs` attribute set to *"unbounded"* denotes that the library element can contain an unlimited number of book elements.

```
 1  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 2
 3     <xs:simpleType name="authorType">
 4        <xs:restriction base="xs:string">
 5           <xs:pattern value="[a-zA-Z0-9]*, [a-zA-Z0-9]*" />
 6        </xs:restriction>
 7     </xs:simpleType>
 8
 9     <xs:complexType name="bookType">
10        <xs:sequence>
11           <xs:element name="author" type="authorType" />
12           <xs:element name="title" type="xs:string" />
13        </xs:sequence>
14     </xs:complexType>
15
16     <xs:element name="library">
17        <xs:complexType>
18           <xs:sequence>
19              <xs:element name="book" type="bookType" maxOccurs="unbounded" />
20           </xs:sequence>
21        </xs:complexType>
22     </xs:element>
23
24  </xs:schema>
```

Listing 2: XML Schema Definition of the example library XML Document

A more detailed explanation of XSD is given in part 0 of the W3C recommendation (see [W3C04]).

2.3.2 XSL Transformations

The term XSL stands for *Extensible Stylesheet Language*. XSL is an XML-based declarative language that is used for describing transformations of XML documents to other XML documents. This process is denoted as *Extensible Stylesheet Language Transformation* (XSLT). An XML document that contains XSLT is named *XSLT-Stylesheet*. This designation has historical reasons and is a bit misleading because the functionality of XSLT is much more extensive than a static description of an appearance in the sense of a style sheet ([Bon08], page 28). XSLT is extensively used for generating presentations for XML documents that describe data just in its structure. The output of an XSL Transformation is not restricted to XML documents. Also HTML files for web pages or plain text files are possible output formats.

In an XSLT transformation at least three documents are involved. These are the XML source document, the XSLT-Stylesheet and the target document. The XML source

document and the XSLT-Stylesheet are passed to an XSLT-Processor. This is a software component that performs the transformation according to the instructions in the style sheet. XSLT-Processors are available for a large amount of application development frameworks and programming languages. The result of the transformation process is the target document. Figure 4 illustrates the transformation process schematically. For accessing resources during the transformation process it is also possible to pass additional XML documents into the transformation process.

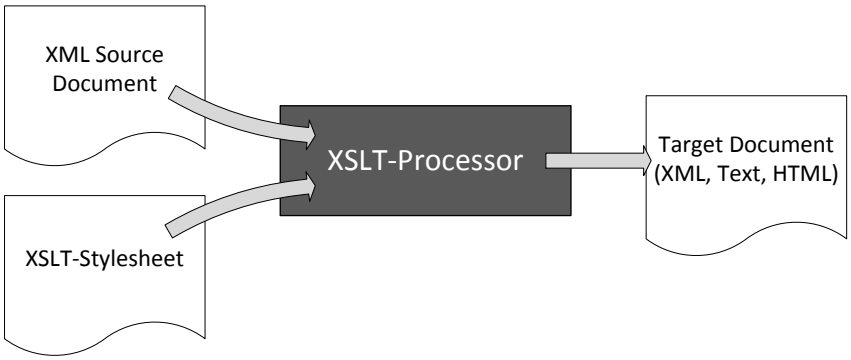


Figure 4: Schematic illustration of the XSL transformation process (according to [Bon08], page 27)

In the following, the general structure of an XSLT-Stylesheet as well as the most important language elements are briefly described. This is done using the book example XML document already introduced in Section 2.3.1 (see Listing 1). This source XML document should be transformed into a target XML document that represents the data in a different structure (see Listing 3).

```
1  <library>
2     <book author="Bongers, Frank">XSLT 2.0 and XPath 2.0</book>
3     <book author="Klein, Florian">A Model-driven Approach to Design …</book>
4  </library>
```

Listing 3: Transformed library XML Document

All elements of the XSL language are part of the XSL namespace and therefore have an *"xsl:"* prefix. The root element of an XSLT-Stylesheet is the `xsl:stylesheet` element. Listing 4 shows the XSLT code implementing the above mentioned transformation.

```
 1  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 2                  version="1.0">
 3
 4     <xsl:template match="/library">
 5        <library>
 6           <xsl:apply-templates />
 7        </library>
 8     </xsl:template>
 9
10     <xsl:template match="book">
11        <book author="{author}">
12           <xsl:value-of select="title" />
13        </book>
14     </xsl:template>
15
16  </xsl:stylesheet>
```

Listing 4: XSLT-Stylesheet for transforming the example library XML document

The specification of the structure of the output document is done using the `xsl:template` element. This is the most important element of an XSLT-Stylesheet. During the transformation process, the XSLT-Processor traverses the XML elements of the source XML document. These source elements are matched to one of the templates defined by the style sheet. Thereby, the output structure of the source elements is determined. For which source elements a template should be used is defined by the match attribute. This is set to an XPath[4] query that determines the elements of the source document for which the template should be applied. The `xsl:applytemplates` element instructs the XSLT-Processor to select the next nodes of the source XML document. In that way, the structure of the target document is built up.

A more extensive description of the XSL Transformation's basics is beyond the scope of this thesis. Additional elements that are used in the following course of the thesis are shortly explained at the respective passages. A detailed explanation of XSLT and all XSL elements is given by Bongers in [Bon08].

2.3.3 XAML

The *Extensible Application Markup Language* (XAML) is an XML based language for defining object trees and the attributes of the objects. The language was developed by Microsoft and is part of the *Microsoft Open Specifications Promise* (OSP), which allows using certain Microsoft technologies in own projects (see [Mic07]). XAML was originally introduced with the .NET Framework 3.0[5] to construct the graphical user interfaces of WPF (*Windows Presentation Foundation*) applications[6]. Thereby, the definition of the application's view becomes independent of the business logic, implemented in any .NET language like C# or Visual Basic. Nowadays, there exist additional specialized XAML versions for the GUI description of Silverlight[7] web applications and Windows 8 *Modern UI*[8] apps. Even for developing apps for Windows Phone, the language is used to declare the user interface. Although these different versions slightly differ, the basic programming concept stays the same throughout these GUI frameworks.

As already mentioned, XAML describes object trees and is therefore closely related to the underlying GUI framework (WPF, Windows Runtime, etc.). The elements of an XAML tree map directly to .NET objects of the GUI framework classes. Attributes

---

[4] XPath is a query language for selecting nodes of an XML document. For more information see [Bon08].

[5] The .NET Framework is a software development framework that is primarily used on Microsoft Windows systems. The currently up to date version is .NET 4.5.

[6] WPF is a programming model for developing Windows- and Browser-based end user applications based on the .NET Framework. A detailed introduction is given by Huber (see [Hub10], p. 39ff).

[7] Silverlight is a software framework for developing rich internet applications. Nowadays it is also used to develop applications for mobile devices.

[8] Modern UI denotes a new graphical user interface introduced with Windows 8 that is optimized for touch operation. The essential features of Modern UI are its graphical simplicity and the fact that applications usually run in full screen mode (no windows). Modern UI applications are based on a new framework called *Windows Runtime*.

are mapped to .NET *Properties*[9] of these classes. Every .NET class providing a default constructor (without any parameters) can be used in XAML. Thus, according to Huber, XAML can be seen as a format for serializing the object structure ([Hub10], p. 141). Hence, there are no features of XAML that could not also be implemented using a .NET language. To get an impression, Listing 5 illustrates the XAML code of a simple WPF Window of a "Hello Word" application. Figure 5 shows the resulting Window.

```
 1  <Window x:Class="WpfApplication1.MainWindow"
 2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4          Title="Hello World Application" Height="100" Width="200">
 5     <StackPanel Margin="12">
 6        <Button Width="150" Content="Greet!" />
 7        <TextBlock HorizontalAlignment="Center"
 8                   Text="Hello World!!!!" />
 9     </StackPanel>
10  </Window>
```

Listing 5: A simple XAML example

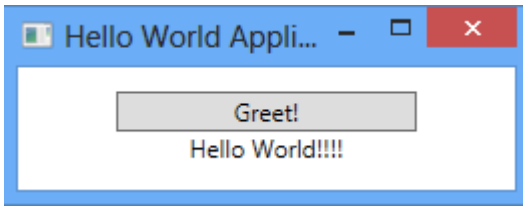


Figure 5: Screenshot of a "Hello World" application resulting from the XAML code of Listing 5

In addition to the strict separation of the GUI definition and the application logic, Huber mentions additional advantages of using XAML in comparison to implementing the user interface using a .NET language ([Hub10], p. 143). Three of them are of special interest for implementing the approach of platform independent data forms in the Windows environment:

- XAML files can be loaded at runtime. This allows exchanging the graphical user interface of an application dynamically. The declared objects in the XAML file are then instantiated and can be used in the application.
- Once loaded, the objects are treated like normal .NET objects. Thus, there are no performance drawbacks using XAML for defining the user interface of an application.
- As the name already suggests, XAML is extensible. This allows using XAML to declare objects of self-specified classes.

The implementation of the UI logic is done using one of the .NET languages in a so called codebehind file. This file contains the event handler, etc. for the GUI elements defined in the XAML file. The compiler integrates these two files and generates an

[9] In the .NET languages private fields are encapsulated using Properties. They can be used like public fields, but allow to add additional code before or after reading or writing the value of the field. A property thereby combines the *Getter* and *Setter* methods, known from other programming languages.

intermediate, binary version of the XAML representation that is deployed with the application.

To better support the separation of GUI design and UI logic the above mentioned GUI frameworks provide additional mechanisms to loosely couple the GUI to the application logic. These include data bindings as well as a command architecture for loosely linking GUI elements to fields of the data classes or methods performing business logic. The consequent usage of these mechanisms yield that the codebehind file does not contain any content. For this implementation strategy, a special design pattern was developed which is briefly introduced in the following Section 2.3.4.

2.3.4 The MVVM Design Pattern

The Model-View-ViewModel design pattern has originally been introduced by Grossman at his blog [Gro05]. The pattern is based on the Model-View-Controller pattern and has been designed to improve the cooperation between GUI designers and programmers. Figure 6 shows the interdependencies between the three components of the MVVM pattern. The three components are:

- **View**: The View represents the Graphical User Interface of the application. This component is implemented using XAML. When using the MVVM pattern, the codebehind files usually do not contain any code which makes it possible to load the View at runtime.
- **ViewModel**: The ViewModel preprocesses the data of the Model to be displayed by the View and provides Properties, to which the View can bind to access this data. Additionally, the ViewModel contains the logic that handles user interaction with the View.
- **Model**: The Model represents the actual data. Usually, the Model consists of classes that just contain data.

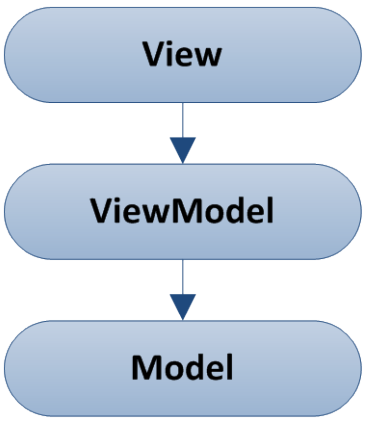


Figure 6: Interdependencies between the components of the MVVM design pattern

The interesting part in the context of this thesis is that the ViewModel, which contains the logic for handling user interaction with the View, does not know the actual implementation of the View and does not contain any GUI elements itself. Thereby, it is possible to exchange the View implemented using XAML easily at runtime. All

user inputs to the View are directly forwarded to the ViewModel to be handled. This is possible via the command architecture and data bindings mentioned at the end of Section 2.3.3. Thus, the View knows about the implementation of its underlying ViewModel. A more detailed explanation of the MVVM design pattern can be found at ([Hub10], p. 507ff).

3 Related Work

The development of multi-device data forms is a special form of developing multi-device user interfaces in general. The diversity of mobile computing devices and available platforms request user interfaces to be suitable for several devices. This results in new requirements to user interfaces that have an impact on their development process. Therefore, a lot of research was done in the field of designing applications and especially user interfaces for targeting multiple devices and platforms. This kind of applications is often called “nomadic applications” (compare for example [MPS03]). In the following sections some of the available concepts and latest research results in this area are introduced. A special focus is laid on model-based approaches because they are closely related to the approach developed by this thesis.

3.1 Model-based User Interface Development

According to Meixner, one method for developing multi-device user interfaces is nowadays commonly known as *Model-based User Interface Development* (MBUID) [Mei11]. The core concept of MBUID is the abstraction of user interface definitions in different layers. Thereby the UI definition gets independent of the concrete UI framework which is later used to provide the UI on a certain target platform. MBUID does not focus on graphical user interfaces but targets any kind of UI. Thus, it for example also includes speech in- and output systems. The approach shares the basic concepts and aims of *Model Driven Software Development* (compare Section 2.1.3). The main difference between these two approaches is that the goal of MBUID is not to build a whole application based on formal models, but just the user interface of the application in order to gather a UI that is suitable on different target platforms.

3.1.1 Core Models

In their summary paper *“Past, Present, and Future of Model-Based User Interface Development”* [MPV11] the authors Meixner, Paternò and Vanderdonckt summarize the MBUID approach with three different abstraction layers that are commonly used for describing a user interface. These are the *Task Model*, the *Dialog Model* and the *Presentation Model*, which are briefly described in the following:

- The *Task Model* defines which tasks a user can actually accomplish by using the UI. The exact definition of a task is done by dividing complex tasks into subtasks until the definition has reached a level of basic input and output operations. The relation of the different tasks is defined by temporal attributes. Thereby it is possible to define which tasks can be done in parallel and which must be sequential.
- The *Dialog Model* connects the task model with the presentation model by defining mappings between tasks and presentation elements and specifies which tasks are available in which state of the application. This information can be derived automatically by evaluating the tasks and their temporal relations from the task model.

- Finally, the *Presentation Model* defines the hierarchical structure of UI elements with which the user actually interacts.

In their paper *"Applying Model-Based Techniques to the Development of UIs for Mobile Computers"* [EVP01] Eisenstein et al. divides the Presentation Model into abstract interaction objects that are platform-neutral and platform specific concrete interaction objects. The Presentation Model assigns several concrete interaction objects to each abstract interaction object which makes the UI portable to any platform. Additionally the authors introduce a further model for describing the features of the different target platforms. This *Platform Model* defines constraints on the platforms which are used at design time to generate a set of user interfaces that are dedicated for a specific target platform. Additionally a run time usage of the Platform Model is considered. A comparable solution is used by the model-driven data form design process introduced by this thesis for integrating peripheral devices into the data capturing process on different target devices.

3.1.2 CAMELEON Reference Framework

The *CAMELEON Reference Framework* (CRF) formalizes a MBUID structure for the development process of multi-target user interfaces. The framework was originally proposed in 2002 by Calvary et al. as a result of the European CAMELEON[10] project [CCT+02]. In 2003, the framework was revised by Calvary et al. in the paper *"A unifying reference framework for multi-target user interfaces"* [CCT+03]. Figure 7 shows a simplified version of the four levels of abstraction that the framework describes. The model layers of the CRF serve as a reference for the model-driven approach for developing multi-device data forms.

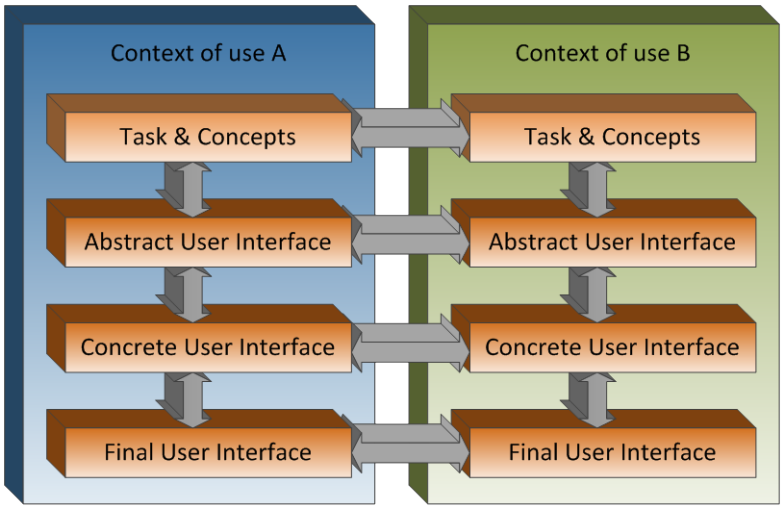


Figure 7: The basic layers of the CAMELEON Reference Framework (according to [LVM+04])

[10] CAMELEON stands for *Context Aware Modeling for Enabling Leveraging Effective Interaction*. More information can be found at the project website [cam04].

At the top level, the *Concepts-and-Tasks Model* describes the tasks for which a user can utilize the UI and their hierarchy. At the *Abstract User Interface* (AUI) level the user interface is defined by abstract user interface elements. These elements are independent of any look-and-feel. This means that an element, which offers the user the possibility to select out of several possibilities, would for example be denoted as "choice". Only at the *Concrete User Interface* (CUI) level the definition whether this choice element is actually implemented as a set of radio buttons or a combo box or in a non-graphical way in case of a speech in- and output system, occurs. Although at the CUI level a concrete look-and-feel of the UI is defined, this level is still independent of any device, platform or UI framework. The *Final User Interface* (FUI) is the fourth level of the CRF. It represents the actual implementation of the user interface using a specific UI framework and is obtained from the CUI by automatically generating source code. The transition between the different abstraction levels is done by semi-automatic transformations between the models or by automatic code generation in the last step, respectively.

3.1.3 The "Graceful Degradation" Approach

In their paper *"Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems"* [FV04] the authors Florins and Vanderdonckt describe an approach for designing multi-target user interfaces that they call "Graceful Degradation". They claim that it must be possible to gain different user interfaces that are suitable for different devices by designing a user interface for the less constraint target platform and applying some predefined transformation rules to that "root interface". These rules are called Graceful Degradation rules. In the paper the authors explain rules for resizing or moving UI elements or changing their orientation. The introduced rules are categorized according to the layers of the CAMELEON Reference Framework.

3.1.4 Constraint-based Layout Management

In *"Multi-device Layout Management for Mobile Computing Devices"* [LCC03] Luyten et al. describe a combination of an abstract definition of a user interface with a constraint-based layout system. The user interface is defined in terms of abstract UI elements. These elements are logically grouped. Elements of the same group at the lowest layer of the grouping hierarchy are always displayed together. The arrangement of the elements is specified by spatial relations (above, left-of, …) between the elements of one group. These constraints allow calculating the UI structure depending on the available screen size. The grouping is used to divide the user interface on small screens. Using the spatial relations, the UI elements are arranged on the available space that is divided into grid cells. This allows generating a device specific user interface at runtime by dynamically building the layout structure and mapping the abstract UI elements to concrete implementations depending on the actual platform.

A similar method of grouping elements and thereby dividing them into different areas is used by the model-driven approach introduced by this thesis to divide a data form into several pages on small screens.

3.2 User Interface Description Languages

In their paper *"A Review of XML-compliant User Interface Description Languages"* [SV03] the authors Souchon and Vanderdonckt give an overview of available XML based *User Interface Description Languages* (UIDL) that can be used for describing one or several of the model layers mentioned in Section 3.1.

This section shortly introduces some prominent UIDL examples. Although none of these languages is directly used by the approach developed in this thesis they serve as a source of inspiration. The reason why the languages are not applicable is that they are considered to be too complex for the future user group not consisting of computer science professionals.

3.2.1 USIXML

Also in the context of the CAMELEON project the *User Interface Extensible Markup Language* (USIXML) was developed [LVM+04]. USIXML is a User Interface Description Language that supports the definition of context-sensitive user interfaces that are device-, platform- and modality independent. Therefore the language provides the possibility to define several models. The AUI and CUI models reflect the same named levels of the CRF. In addition a task-, domain-, context- and UI model are specified. Furthermore USIXML defines a model for the transition between the different models. The modular structure of the language allows starting the development process from any abstraction level and specifying only those models, which are actually needed to satisfy the requirements of the intended user interface.

3.2.2 UIML

The *User Interface Markup Language* (UIML) is an XML compliant UIDL that is standardized by OASIS[11] (see [HSL+08]). UIML allows UI designers to define the structure, style, content and behavior of user interface elements, which are called parts in UIML. Each part is a member of a class. The classes are not predefined. The structure is defined by a virtual tree of parts. The style defines values for different part properties (e.g. the background color, or the font of a text). Additionally it is possible to define the action that occurs when the user interacts with the UI. This is done by defining rule-based behaviors. Each rule defines a condition as well as a sequence of actions that are executed if the condition becomes true.

The relation between this abstract definition and a concrete UI framework is defined in a separate part of a UIML document or even in a different document. This is done by mapping the classes of interface parts and their properties to concrete widgets and their properties of a UI framework. The loose coupling between the abstract definition and the concrete framework ensures UIML do be totally device, platform and framework independent.

[11] OASIS stands for *Organization for the Advancement of Structured Information Standards* and is a non-profit consortium that supports the development of open standards for the information society.

3.2.3 XIML

The *Extensible Interface Markup Language* (XIML) [PE01] allows specifying arbitrary user interfaces. Basically, XIML provides a way to define the elements of a UI, their attributes and relations. The elements are organized in components. In XIML, elements should not be understood as abstract UI elements but as interaction data in a more common sense. The number of components and different elements is not limited by the language and can be extended. Nevertheless, XIML predefines five basic components in its first version, which are task, domain, user, dialog and presentation. Thereby, XIML provides a standard format for exchanging interaction data definitions between different frameworks and applications without specifying how the definitions should be implemented. XIML covers the whole UI development process from design via operation to evaluation.

3.3 Design Environments

In the papers *"Tool Support for Designing Nomadic Applications"* [MPS03] and *"Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions"* [MPS04] Mori et al. present a tool that supports developing multi-device user interfaces using a model-based approach. This tool is called TERESA. It provides a top-down transformation process that follows the CAMELEON Reference Framework. Therefore the tool provides a graph based editor that enables the designer to define a task model. Based on this model different task sets are derived which contains tasks that are available at the same time. The authors call this sets "presentation task sets". The presentation task sets and the task definitions are input for the generation of an Abstract User Interface. From this step on the transformation process follows the CRF and ends with the automatic generation of the final UI. The models on each layer are developed using dedicated XML-based languages.

By this time, the TERESA tool is not further developed. Instead there is a follow-up tool, called MARIAE[12] (MARIA Environment) that uses a similar method for developing interactive applications for several platforms based on web services (see [PSS09]). MARIA is the name of the XML-based language that is used by the tool and supports describing user interface at abstract and concrete levels.

Referring to this thesis, the user interfaces of both tools as well as the underlying XML-based languages serve as a source of inspiration for the development of a model-driven approach to design multi device data forms. Especially many ideas of abstracting UI elements can be transferred to this thesis.

[12] The MARIAE tool is publically available and can be downloaded at [HII].

4 Conception

Based on the findings of the related work research, a model-driven method for designing platform independent data forms for use in a medical study environment is elaborated. This chapter explains this concept in detail.

4.1 Overview

The general approach is based on the abstraction layers, defined by the CAMELEON Reference Framework. Summarized, a data form is developed by running through four different levels of abstraction. The layers are passed one after another. Like it is known from *Model Driven Software Development* (MDSD), the transition from one model layer to the next is performed by automatic transformations from the higher model level to the lower one (compare Section 2.2.2). Thereby, the designer is guided through the model-driven process and just has to add the specific information on each layer until the final transformation to a specific target platform can be done.

In comparison to the CAMELEON Reference Framework, which proposes a model based development process for specifying complete user interfaces for multiplatform applications, the purpose of this thesis is slightly different. Here, the goal is to come up with a solution for designing device independent data forms. This variation results in two important differences. First, the designed data forms are opened and shown by a dedicated application. In terms of the :study software this is the :studydata application. If a data form shall be displayed and filled out on a specific platform, a version of :studydata for this platform is needed. Hence, the data form itself is not supposed to be a standalone application. This method is comparable to using an interpreter for executing a formal model definition in the MDSD approach (compare Section 2.2). The second disparity is the fact that a data form is already a very special kind of user interface. In the context of this thesis, a data form is assumed to have a graphical user interface. This means that all non-graphical solutions, like for example speech systems, that are also included in the general notion of platform independence, are ignored in the further consideration. In addition, when designing a general user interface, the amount of tasks a user can do using this interface can be arbitrarily large and complex. This complexity is defined in the Task Model of the CRF. For a data form, the task is clearly defined and in principle it is the same for each data form. Broadly speaking, a user opening a data form always wants to capture some data. Also the process for doing so always follows the same pattern, which has been explained in Section 2.1.2. To recap, the user opens the form, selects the test subject, enters the data and commits it to the database (compare Figure 3).

Despite these differences, the models defined by the CRF serve as a reference for the developed approach. Figure 8 depicts the basic process for generating multi-device data forms running through four model layers. The first three model layers are part of the actual data form design process. They are generated and edited by a designer. The fourth layer represents the final data form implementation that is specific for an actual target platform. At this level, the designer has no more options for changing.

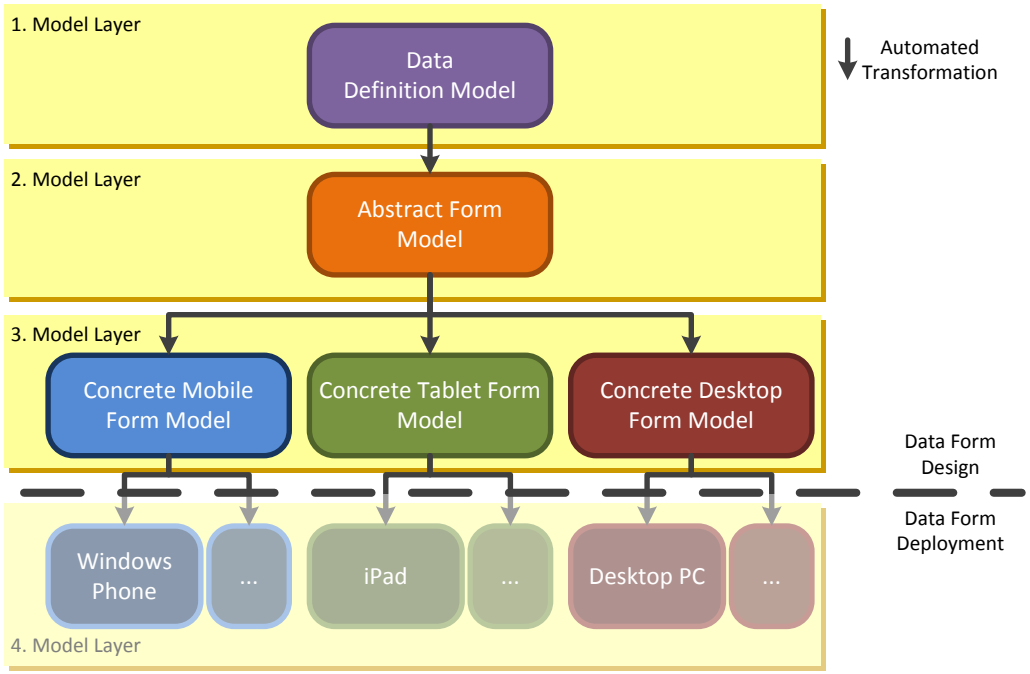


Figure 8: Overview of the general approach for generating multi-device data forms

While a formal definition of a task model is not necessary, there is another important part of the form that has to be specified. The question here is: Which data should actually be captured by the form? This question is very similar to which tasks does a user have to do to achieve his goal. The developed approach therefore starts with specifying a single *Data Definition Model* (DDM). In the DDM, the designer specifies which values are captured by the form and how they are entered. Possible options for entering a value are to fill it in manually or to read it from a peripheral device. The Data Definition Model also defines possible dependencies between some data.

Based on the Data Definition Model, at the second layer an abstract definition of the data form is generated. This *Abstract Form Model* (AFM) is still device independent. It defines the visual appearance of the data form in terms of abstract form elements that are used to capture the specified data. These elements do not contain information about their visual manifestation. They are just characterized by their features and behavior. Thereby, a rough structure of the data form is constructed. Another aspect, which is specified at this layer, is the workflow that guides a user through the data form. This is the order, in which the form elements are activated one after another. Usually this order conforms to the working process of the user entering the data.

At the last design layer, the actual visual appearance of the data form is specified by a *Concrete Form Model* (CFM). At this concrete layer, the definition is split into several models, one for each possible target device. The possible target devices are denoted as *Mobile*, *Tablet* and *Desktop* where *Mobile* mainly stands for smartphones. This classification is considered to be reasonable because it reflects the differences in screen size and other features of the devices. Thereby, the designer is able to define different layouts of the data form for each device that are suitable for the typical display size of a device. The Concrete Form Model makes the visual appearance of the form elements explicit. This means that for each abstract form element of the AFM a

concrete realization is specified. This concrete realization considers the characteristics of the respective target device type. Therefore, an abstract form element can be realized differently on the individual Concrete Form Models. The tendency to larger smartphones and smaller tablets makes it difficult to define a clear border between these two device types. Therefore it is possible that for a very large smartphone device the layout defined in the Concrete Form Model for tablets results in a better depiction and vice versa.

The final step of the design process is the transformation from the Concrete Form Model to the *Final Form* (FF) implementation. This Final Form is the specific realization of the data form for an actual target platform. The particular realization of a Final Form depends on the GUI framework of the respective platform. In case of Windows based platforms, like Windows Phone or Windows Tablets, the Final Form layer is implemented by XAML based user interface descriptions. The XAML files are automatically generated by a transformation from the Concrete Form Model. The generated XAML code can be loaded directly by the versions of the :studydata application implemented for the mentioned Windows based platforms. This means, that the last transformation step is executed explicitly and its result is stored as a XAML document.

If the target platform does not provide such a feature for loading user interfaces at runtime, there is also the possibility to do the last transformation step implicitly. In this case, the respective version of the :studydata application interprets the relevant Concrete Form Model at runtime and directly instantiates the user interface elements of the data form. In addition, also a mixed solution is conceivable: The Concrete Form Model is transformed into some intermediate model, which is then interpreted by the :studydata application. More particularly, this solution is more or less the same as the generation of XAML files. The only difference is that the intermediate model is proprietary. Therefore, a dedicated interpreter has to be implemented for the :studydata application to read this proprietary intermediate model. This is a difference to the XAML solution, because the GUI frameworks of the above mentioned Windows platforms support loading XAML files directly.

The described approach ensures that a designer does not have to implement things twice. All information, that is valid across the different target devices, is stated in the first two model layers. This is mainly information about the captured data and the capturing methods. Only at the Concrete Form Model layer, the development is split into the three device types. The information that is given on this layer, for example the data form's layout or the size of the form elements is specific for the respective device type.

To guide the designer through the design process traversing the three model layers, a support tool for the explained method is needed. In terms of the :study software, this tool is an exhaustively extended version of the :studyforms application. The most important challenge that has to be solved by this application is to enable a designer without computer science background to easily work with the described model-driven method.

In addition to the already mentioned models, which are traversed one after another during the design process, a further model for defining automated input devices is defined. This *Input Device Model* (IDM) provides information about available peripheral devices that can be used for gathering values in an automated way. This concept is explained in detail in Section 4.3. Prior to this, the following section describes the three introduced model layers of the design process in more detail.

4.2 Model Layers

The following sections explain the goals and the structure of the upper three model layers, which are part of the actual design process, in detail. For each of the layers a *Domain Specific Language* (DSL) is defined, which allows to specify a clearly defined part of a data form. The abstract syntax (compare Section 2.2.1) of the languages is specified in a corresponding meta-model for each model layer. These meta-models are described by UML class diagrams that specify the available elements and their interdependencies. Common attributes of several elements within the same meta-model are composed in abstract base classes of the concerned elements. These abstract elements are themselves no parts of the DSL and cannot be used in the models. For the concrete textual syntax for the DSLs, the XML format has been chosen. The reasons for this choice are the platform independence of the XML format as well as the tree-like structure of an XML document that projects the hierarchical structure of the models. This textual syntax is also used to store the models during and after the design process. To validate the models against their meta-models, the defined class diagrams will later be transformed to XML Schema Definitions. This eases the validation in the :studyforms application. For the purpose of editing the models in :studyforms a graphical syntax is introduced for the models (see Chapter 5).

4.2.1 Data Definition Model

One of the disadvantages of the former, purely graphical data form design process (compare Section 2.1.1) is the mixing of data and their visual appearance on the data form. The design process is only based on arranging graphical form elements that are linked to a parameter of the database. This causes that even attributes like an initial value of some data, which is closely related to the data itself, is set on the input element that graphically represents that data. To overcome this drawback, the separate *Data Definition Model* (DDM) is introduced. On one hand, the DDM is the first step of the model-driven design process, from which the graphical appearance of the data form is gathered. On the other hand it represents an independent data layer that defines which data is collected by the data form and exists beside the lower model layers. It thereby forms the connection between the central :study database and the data form. Figure 9 shows the meta-model of the Data Definition Model.

The DDM is specified in a hierarchical structure. The root of this structure is the `DataModel` element that holds a reference to the activity pattern the form is related to. The actual pattern definition is not part of the model but stored in the :study database. Additionally, the `DataModel` element holds the name of the form. For build-

ing the hierarchic data structure of the DDM, the meta-model defines three elements. These are `DataGroup`, `DataDefinition` and `VolatileDataDefinition`.

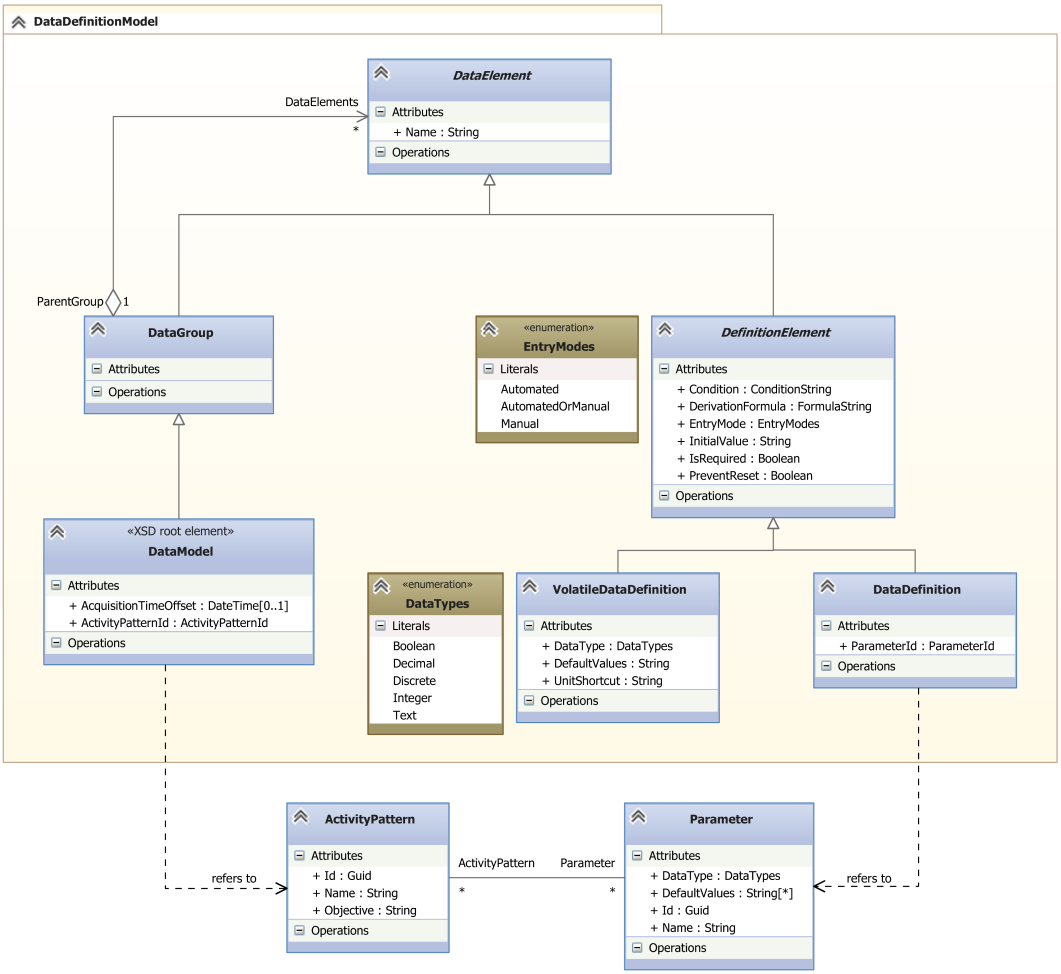


Figure 9: Meta-model of the Data Definition Model

`DataGroup` is a simple grouping element that has no further attributes, just child elements. It is used to group data logically within the form. This grouping eases the transformation of the form to several devices in the upcoming transformation process. For the 24h urine example a possible group would be the weight, which is composed of the gross, tare, and net weights. For a flexible structuring it is possible to nest data groups. Each of the DDM elements is identified by a name attribute that has to be unique within the same group.

With the two supported data definition elements, `DataDefinition` and `VolatileDataDefinition`, the designer specifies which values are captured by the form. The two elements have several attributes in common:

- For each of the elements it is possible to define an initial value (attribute `InitialValue`). To this value, the data is set after the form is initially opened or reset.
- Using the `IsRequired` attribute, the designer can specify whether the data has to be set to a valid value when the values of the form are submitted, or whether entering a value is optional. The validity of a value is checked by the :studydata

application according to the defined data type as well as possible thresholds defined for a parameter in the :study database.

- The `DerivationFormula` attribute allows to specify that a value should be calculated based on the entered values of other data definitions. This is for example the case for the net weight of 24h urine samples. This value is calculated as the difference of the determined gross weight and the tare of the urine bottle. The reference to another value is stated by the name of its data definition element and its parent elements. These names build a unique path with the pattern *"ModelName/GroupName/.../ReferencedElementName"* starting at the root of the DDM. For the gross weight of the before mentioned example this path could for example be *"24hUrineLab/Weight/GrossWeight"*.
- The need to enter a value can also be dependent on other values. For example, some data only has to be gathered if for others specific values are entered. This can be achieved by setting the `Condition` attribute to a logical expression that defines this dependency. The references to other elements are again expressed via their paths. The possibility to define such a condition leads to a context sensitive behavior of the data form. This is one of the crucial advantages an electronic data form has over its paper representation. Thereby it is possible to hide all currently unnecessary input elements of the form, which allows the user to focus on his current task. This is especially important on mobile devices since they are usually used casually during gaps in the actual task of the user (compare [HB11], p. xxix). Additionally, it avoids unreasonable data sets since the entering of specific data is not possible under certain circumstances.
- The `EntryMode` attribute is one of the central attributes that is evaluated by the transformation from the Data Definition Model to the Abstract Form Model (see Section 4.6.1). It allows defining whether the value of a data definition will be entered manually by the user of the data form or captured in an automated way by means of a peripheral device. It is also possible to offer both options for example to handle exceptional cases like hardware defects or connection issues.

The essential difference between the two types of data definitions is that a `VolatileDataDefinition` represents data that will not be stored in the database. Such data is just used within the context of the data form as a base for calculating other values or for supervisory purposes. For example, the calculated net weight of a 24h urine sample should be displayed in the form, such that the user can verify the value. However, since storing redundant information should be avoided, the value of the net weight will not be stored in the database.

As opposed to this, the value of a `DataDefinition` element is stored in the database when the form is submitted. Therefore, each `DataDefinition` refers to a parameter of the :study database. This enables the system to assign the entered values to the right parameter. The reference is established by storing the unique surrogate key of the parameter as an attribute of the `DataDefinition` element in the DDM. The parameter definitions in the database contain information that is needed by the data form. This is the data type, the default values and the unit in which the parameter is captured. To avoid holding redundant information for the `DataDefinition`

elements these details are not part of the Data Definition Model. Instead, this information is loaded from the database during the design process and when the form is opened and displayed to the user. This ensures that the form always provides up to date default values and units. Since the `VolatileDataDefinition` elements are not linked to a parameter of the database, for these elements the needed information is part of the DDM. Therefore, the meta-model allows specifying the following attributes on a `VolatileDataDefinition` element:

- The `DataType` specifies of which type the entered values have to be. It can be set to one of the available `DataTypes`. The `DataType` set to *"Discrete"* means that the entered value has to be one of the specified default values.
- The `DefaultValues` attribute define a list of standard values which are commonly entered for this data definition. These elements can be offered for selection by the user, which eases to fill in the data form. The values are specified in form of a comma separated list.
- For values that are entered in a specific unit, the `UnitShortcut` attribute defines this unit by its abbreviation. This abbreviation can be shown in the data form near to the corresponding value.

To get an impression of a Data Definition Model, Listing 6 shows a shortened version of a Data Definition Model for the 24h urine laboratory data form. The example is shown using the concrete XML syntax.

```
 1  <DataModel Name="24h-Urine Laboratory" ActivityPatternId="083c0f2f-…">
 2     <DataDefinition Name="BottleNumber" ParameterId="334d7112-…"
 3                     IsRequired="true" EntryMode="AutomatedOrManual" />
 4     <DataGroup Name="Weight">
 5        <DataDefinition Name="GrossWeight" ParameterId="b0279a2a-…"
 6                        IsRequired="true" EntryMode="Automated" />
 7        <DataDefinition Name="TareWeight" ParameterId="4c94c6e7-…"
 8                        IsRequired="true" EntryMode="AutomatedOrManual" />
 9        <VolatileDataDefinition Name="NetWeight" IsRequired="true"
10                                EntryMode="Manual" DataType="Decimal"
11                                DerivationFormula="…/…/GrossW - …/…/TareW" />
12     </DataGroup>
13     <DataDefinition Name="pHValue" ParameterId="47aac2a6-…"
14                     IsRequired="true" EntryMode="Manual" />
15     <DataDefinition Name="Comment" ParameterId="dd080c92-…"
16                     IsRequired="true" EntryMode="Manual" />
17  </DataModel>
```

Listing 6: Example Data Definition Model for the 24h urine laboratory data form

4.2.2 Abstract Form Model

The *Abstract Form Model* (AFM) is comparable to the Abstract User Interface of the CAMELEON Reference Framework. In this step of the data form design process, the focus shifts from the pure data view to the actual interaction elements that are displayed in the form and used to enter the values. The AFM therefore defines the visual elements of the form in the sense of abstract user interface elements. The reason why they are denoted as abstract is, that at this design layer their concrete realization is not yet defined. This means, that the elements are characterized only by their basic behavior or the task they support. However, the visual appearance of the elements on the data form is still undefined. An example for such an abstract UI element is a field that shows and allows editing numerical values. This field does not define how the

editing is done but just that it is possible. A concrete version of this field could allow entering the value directly via the keyboard or by pressing an increment or decrement button. These abstract UI elements will be denoted as *Abstract Form Elements* (AFE) in the following course of this thesis.

The AFM defines the visual structure of the form by grouping the Abstract Form Elements and thereby building a hierarchical structure. This structure reflects the structure of the Data Definition Model and is the base for the layout of the elements in different device specific versions of the form on the next design layer. Additionally to the visual structure, the Abstract Form Model also specifies the data form's internal interaction workflow. This has to be understood as the order in which the elements on the form are activated and filled in by the user. Usually this workflow starts with selecting a test subject to which the entered data is captured. Then the data is entered and submitted to the database. The interaction workflow does not specify which of the entered data is stored to the database. This is distinguished at the Data Definition Model by the two alternative data definition elements `DataDefinition` and `VolatileDataDefinition`.

The following sections describe the available AFEs and their attributes in detail. Figure 10 gives an overview of the Abstract Form Model's meta-model.

The root element of the AFM is the `AbstractForm` element. The attributes `FirstElement` and `LastElement` allow specifying the start and end of the interaction workflow. The `FirstElement` attribute is set to the element that will be activated when the form is opened or reset. The `LastElement` attribute specifies the element after which the workflow is finished and restarts for processing the next sample of another test subject.

The `AbstractCompound` element corresponds to a `DataGroup` in the Data Definition Model. It allows structuring the Abstract Form Elements in the sense of grouping them to different graphical areas on the data form. Like in the Data Definition Model, each AFE is identified by a name attribute that has to be unique within the same `AbstractCompound` element. This allows referencing AFEs by their path, starting at the root element of the Abstract Form Model.

In general, the AFEs can be divided into two groups. Some elements only make information visible on the form. They are denoted as *Output Only Elements* in the following paragraphs. With the other elements, the user can interact somehow. Therefore they are called *Interaction Elements*.

Output Only Elements

On the Abstract Form Model, there are two pure output elements available. These are `AbstractDisplayElement` and `Description`.

AbstractDisplayElement

This element displays the value of a referenced data definition on the form. The `Caption` attribute is used to set up the heading text that is displayed nearby the val-

ue on the data form. An output element is for example useful to show an automatically calculated value or a value that was read from a peripheral device.

Description

The `Description` element represents additional textual information on the data form. It is not linked to an element of the DDM. Instead, the displayed text is specified at design time by setting the `Text` attribute of the `Description` element.

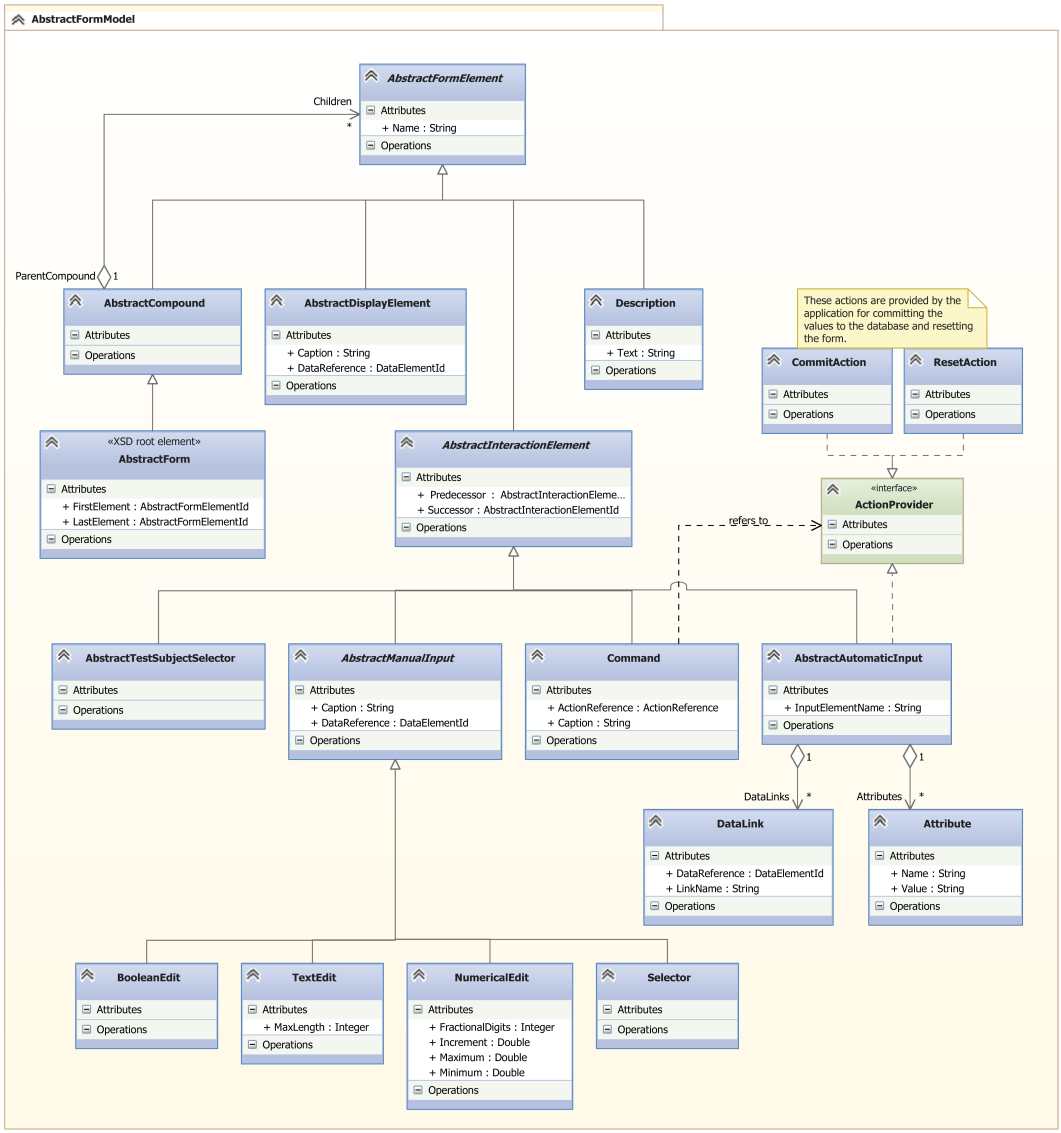


Figure 10: Meta-model of the Abstract Form Model

Interaction Elements

All other Abstract Form Elements offer some sort of interaction with the user. Therefore, all of them can be part of the data form's interaction workflow. To define that activation order, the elements have a `Successor` and `Predecessor` attribute. These attributes hold a reference to the next and the previous interaction element in the workflow. The reason for the ability of setting the predecessor element separately is that in some situations it is reasonable to activate another element than the previous element in the workflow when navigating back. An example for such a situation is a

faulty result after a barcode scan. In this case, it is more sensible to correct this error manually than to scan the barcode again. Therefore, if the user navigates in backward direction, starting at the element following the barcode scanner, the manual input element that shows the faulty result should be activated instead of the barcode scanner (compare Figure 11). This behavior cannot be stated in general. The actual decision how to proceed in such a situation has to be made during the form's design process.

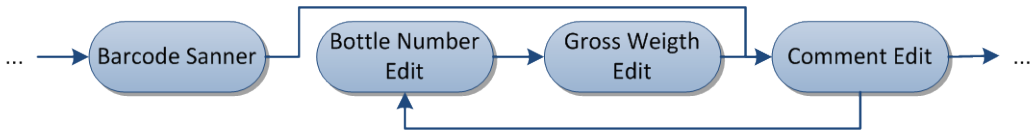


Figure 11: Example for an internal data form workflow with backward navigation

The purpose and the specific attributes of the interactional Abstract Form Elements are now explained in detail. Depending on the data type of the referenced value the designer can choose out of four different manual input fields, each of them is intended to handle values of a specific data type. These manual input fields are the first four of the described AFEs. All of them reference to the data definition of the DDM they represent on the form. This reference is held by the `DataReference` attribute. Additionally they have an option to set a caption string (attribute `Caption`).

BooleanEdit

This field supports data definitions with *"Boolean"* data type. It just toggles the value between *true* and *false*.

TextEdit

The `TextEdit` element focuses on the *"Text"* data type. It allows entering alphanumerical strings and special characters. In some cases it might also be feasible to use this type of input field for *"Integer"* or *"Decimal"* values. This is for example the case, if a decimal number with a large amount of fractional digits has to be entered. In that case, the direct input via the keyboard is much more comfortable. The maximal number of characters entered to the text edit field can be restricted by setting the `MaxLength` attribute to an integer greater zero.

NumericalEdit

For entering numerical values of the data types *"Integer"* or *"Decimal"* the `NumericalEdit` element should be used. Depending on its actual realization on the Concrete Form Model layer it provides different types of features that simplifies entering a numerical value. This could for instance be two spin buttons for incrementing or decrementing the value. The minimal and maximal allowed value as well as the number of displayed fractional digits can be set by respective attributes. In case of an Integer value, the `FractionalDigits` attribute is set to zero.

Selector

The `Selector` element stands for an input field that allows the user to choose one value out of several suggestions. It is used for values of the type *"Discrete"*. The

available values are either specified in the Data Definition Model or loaded dynamically from the :study database when the form is opened (compare end of Section 4.2.1). Hence, the `Selector` element has no further attributes. The provision of a multi select option is considered as being not reasonable. This is caused by the general concept of data capturing provided by the :study software, which assumes that for each parameter a unique value is captured at a specific point in time. This assumption also holds for the value of a `VolatileDataDefinition` because it is seen as a volatile equivalent of a parameter.

AbstractTestSubjectSelector

The `AbstractTestSubjectSelector` element allows determining to which study participant the entered data is linked when it is stored to the database. Since the concept of data acquisition in :study assumes that a data form is always filled out in the context of a specified study participant, there is no explicit data definition for the current test subject in the DDM. Therefore, the test subject selector does not need a reference to an element of the DDM. Instead, the data model that holds the currently selected test subject as well as all available test subjects is provided by the :studydata application that shows the data form and handles the access to the database. The reference to this data is established by the actual implementation of the `AbstractTestSubjectSelector` for the respective target platform. During the design process, the relation can be assumed to be defined implicitly.

AbstractAutomaticInput

The `AbstractAutomaticInput` element is the most complex element of the AFM. It represents any kind of peripheral device that is included into the data form to automate the entry of selected values into the form. Examples for such devices are a barcode scanner or a laboratory balance.

The available automated input devices as well as their attributes and the actions they provide are defined in the Input Device Model (see Section 4.4). To avoid the need for adapting the meta-model of the AFM every time a new automated input device is added to the system, the devices are not directly used in the Abstract Form Model. This means that there is not a separate Abstract Form Element for each input device defined in the IDM. Instead, all devices used by a form are represented via the `AbstractAutomaticInput` element. Which kind of peripheral device is concerned is specified by setting the `InputElementName` attribute of such an element to the name of one of the input devices defined in the IDM. To assign values to the different attributes of the devices defined in the IDM and add references to the Data Definition Model, the element allows defining sub elements of the types `Attribute` and `DataLink`. These sub elements refer to one of the data link or property definitions defined for the concerned device in the Input Device Model. The references are established by setting the `Name` or respectively `LinkName` attribute to the name of the respective data link or property definition. The intended value is set to the `Value` or respectively `DataReference` attribute. An example for this is given in Listing 7, l. 7.

Command

The Command element represents an interaction element for triggering actions on the data form. Actions are provided by automatic input devices or by the data form itself which provides the *"submit"* and *"reset"* actions. These special actions are used for triggering the underlying :studydata application to store the entered values in the database or to reset all values to their initial values. The implementation of this behavior is part of :studydata application. The link to an action is established by setting the `ActionReference` attribute to the path of the data form or automatic input element, followed by the name of the triggered action. Available actions for an automated input device are defined in the Input Device Model. The pattern for this kind of reference is specified as follows: *"ModelName/Group1Name/.../ReferencedElementName::ActionName"*.

Listing 7 shows a simplified version of an Abstract Form Model, which is consistent to the example Data Definition Model of Listing 6.

```
 1  <AbstractForm Name="24h-Urine Laboratory">
 2     <AbstractTestSubjectSelector Name="TestSubjectSelector" />
 3     <TextEdit Name="TextEdit_BottleNumber" Caption="Urine Bottle Number"
 4               DataReference="…/BottleNumber MaxLength="6" />
 5     <AbstractCompound Name="Weight">
 6        <AbstractAutomaticInput Name="Balance" InputElementName="Balance">
 7           <DataLink LinkName="WeightValue" DataReference="…/GrossWeight" />
 8        </AbstractAutomaticInput>
 9        <Command Name="Command_Weight" Caption="Weight"
10                 ActionReference="…/…/Balance::Weight" />
11        <AbstractDisplayElement Name="DisplayElement GrossWeight"
12                    Caption="Gross Weight" DataReference="…/GrossWeight" />
13        …
14     </AbstractCompound>
15     <NumericalEdit Name="NumericalEdit_pHValue" Caption="pH-Value"
16                    FractionalDigits="2" DataReference="…/pHValue" />
17     <TextEdit Name="TextEdit Comment" Caption="Comment"
18               DataReference="24h-Urine Laboratory/Comment" />
19     <AbstractCompound Name="Finishing">
20        <Command Name="Command_Submit" Caption="Submit"
21                 ActionReference="24h-Urine Laboratory::Commit" />
22     </AbstractCompound>
23  </AbstractForm>
```

Listing 7: Example Abstract Form Model for the 24h urine laboratory data form

4.2.3 Concrete Form Model

The *Concrete Form Model* (CFM) concretizes the Abstract Form Elements defined in the AFM for a specific target device type. At this model layer the designer specifies the actual layout of the data form by setting the height and width of the now *Concrete Form Elements* (CFE). Since these details vary significantly depending on which device the form is displayed on, the design process is split at the Concrete Form Model layer. This means, that a CFM is defined for each of the three possible target devices types (*Mobile*, *Tablet* and *Desktop*) independently. Thus, a CFM is specific for one device type, but still independent of the target platform. The Concrete Form Model is therefore comparable to the Concrete UI of the CAMELEON Reference Framework (compare Section 3.1.2).

Although the heterogeneity of display sizes is much lower within one device type than across the different devices, it is still not possible to define the data form's layout by fixed positions and sizes of the form elements. Instead, the CFM also orders the

elements in a hierarchical structure. But, there is an essential difference in the meaning of this structure in comparison to the two higher model layers explained above. The group elements are not just simple containers that group their children logically or define abstract areas in the data form. Instead, they are concrete layouts, which define how their child elements are visually arranged on the data form and therefore also have several layout attributes themselves. These layout containers adapt the size and positions of the elements depending on the actual screen size at runtime and thereby assure a proper layout on each platform. More details about the handling of different screen sizes and resolutions are explained in Section 4.5.

The available Concrete Form Elements are now explained in detail, starting with the possible layout options. An overview of the CFM's meta-model is depicted in Figure 12.

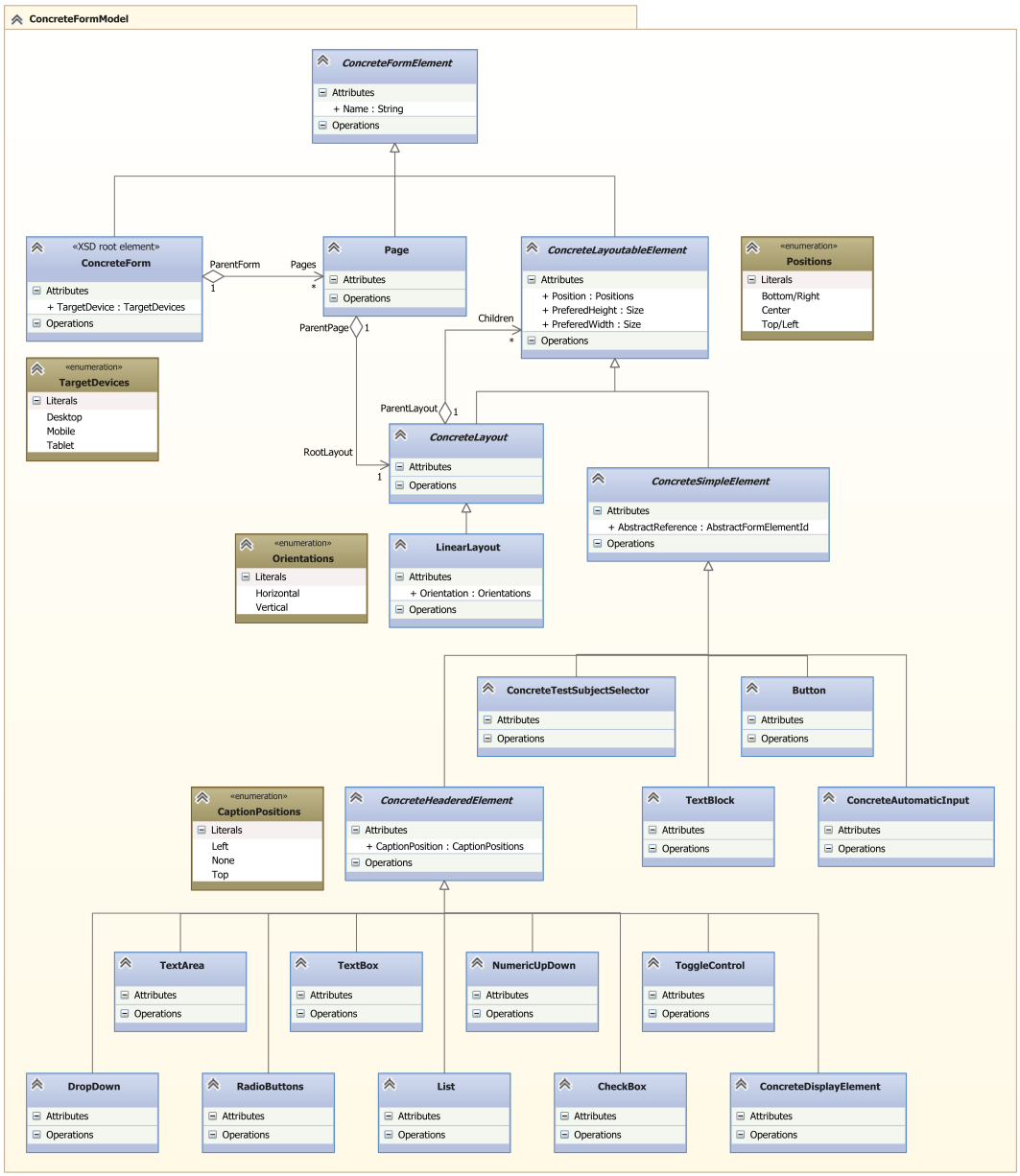


Figure 12: Meta-model of the Concrete Form Model

The root element of the CFM is the `ConcreteForm` element. Its only attribute defines the target device of the model and can be set to either *"Mobile"*, *"Tablet"* or *"Desktop"*. The root element contains one or several `Page` elements as children. A page represents a section of the data form. The elements within one page are visible at the same time. The page concept has mainly been introduced for smartphone devices whose display sizes are most limited. On such devices it is reasonable to divide the data form into several pages that are displayed one after another. On desktop and tablet devices it can be assumed that the data form is usually designed as a single page. For sake of flexibility, it is nevertheless also possible to define several pages on a CFM targeting these devices. Each `Page` element has exactly one child element, which is the root layout container of the page. For the time being, the only layout container that is defined by the CFM is the `LinearLayout`. This layout arranges its child elements in a single column or a single row depending on the value of the `Orientation` attribute (inspired by [Gooa]). The `Orientation` attribute can be set to *"Vertical"* for arranging the child elements in a column one below the other, or to *"Horizontal"*, which causes the child elements to be stacked side by side in one row. Figure 13 illustrates this simple concept. The `LinarLayout` is chosen as first layout container because it allows easily arranging elements on the data form. In spite of this simplicity, also complex layout structures can be achieved by arbitrarily nesting several layout containers. However, the meta-model of the CFM is defined flexible enough to be extended by further layouts in a future version.

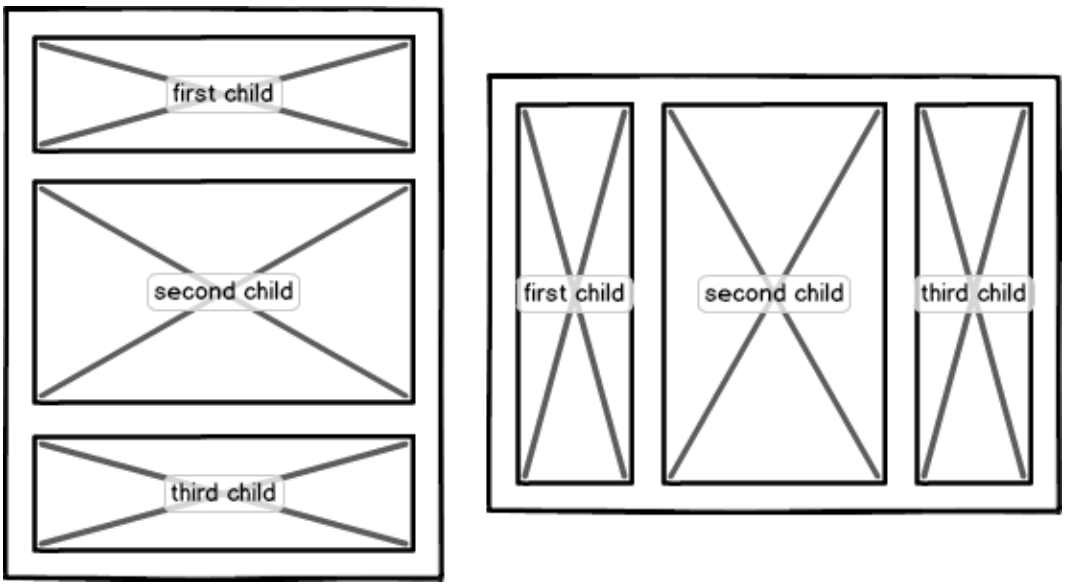


Figure 13: Linear layout with vertical orientation (left) and horizontal orientation (right)

Additionally to the obligatory name attribute for identifying and referencing the model elements, each of the CFEs has attributes for specifying the element's size and position. Due to the different possible display sizes and resolutions of the target platforms the sizing and positioning is not done by fixed pixel values. Instead, the values are defined in a platform independent fashion. The solution for this is inspired by the *Grid* panel of the Windows Presentation Foundation (compare [Hub10], p. 348ff). The size is determined by the `PreferedHeight` and `PreferedWidth` attributes.

The reason why the attributes are denoted with the "preferred" prefix is that the actual size of the element on the target device might differ from the specified values. This is the case if the target display is too small to satisfy all defined size values. The two attributes can be set in three different modes:

- Setting the value to the string *"auto"* yields the according size to be calculated automatically. This means that the element takes exactly the space that is needed to display its content without cutting something off. If the available size is smaller than the size needed, it is still possible that parts of the content are cut.
- The second option is to set the size relative to the available size of the parent layout container. This relative size value is specified by a decimal value, followed by the *"*"* (Star) symbol. All child elements that use a relative size share the available space of their parent layout container. Here it is important, that the distribution of available space only works in the orientation direction of the parent linear layout. Setting the size value of the other direction to any relative size (using the Star symbol) results in the container size in that direction, regardless of the preceding weight value. The available space of a layout container calculates as the size of the container minus the sizes of automatically or fixed sized elements within that layout. The decimal value of the relative size serves as a weight value that denotes which amount of available space the element receives. This makes it possible to divide the available space equally (all weights are equal) or weighted.
- In some cases, it is also reasonable to set the size of an element to a fixed value. This is done by a plain decimal value, specifying the size in pixels. But, due to the diversity of resolutions, it is not possible to specify the size in real device pixels. Instead, the value is specified in terms of *Logical Pixels* (lp). A detailed explanation of this is given in Section 4.5.

The `Position` attribute determines the alignment of an element within the area of its parent container. Since for now, the only layout container is the `LinearLayout`, which stacks its child elements either in vertical or in horizontal direction, there is just a single Position attribute, and not one for each axis. Valid position values are *"Top/Left"*, *"Center"*, and *"Bottom/Right"*. The value always refers to the axis complementary to the linear layout's orientation. Figure 14 illustrates this by an example: If an element is a child of a linear layout arranging its children in a column (`Orientation` attribute set to *"Vertical"*) the position attribute defines the element's alignment on the horizontal axis. If the size of the element is set to "*" (Star), and thus not smaller than the parent container, the `Position` attribute does not cause a visible difference.

In addition to the layout attributes mentioned until now, those elements for which it is possible to define a caption in the Abstract Form Model, have an attribute `CaptionPosition`. This attribute defines where the specified header should be positioned relative to the actual field showing the value. Valid values for this attribute are *"None"*, *"Top"*, *"Left"* or *"Auto"*. The first option results in hiding the caption on the data form. The next two options display the caption above or to the left of the value field, respectively. Since the :study software is expected to be used in a cultural

environment with a left to right reading direction, arranging the caption to the right or to the bottom of the value field is assumed to be unreasonable. To prevent design errors caused by unintentionally using such inappropriate position values these options are not provided. With the *"Auto"* option, the designer can specify that the caption should be displayed either above or to the left of the input field depending on the current aspect ratio of the elements' parent layout container. This option is especially useful on the Concrete Form Models targeting smartphone or tablet devices since the user might use them in landscape or portrait mode. Changing the screen orientation also leads to a change of the aspect ratio of a layout container with relative sizes. Thus, the feature allows reacting dynamically to a change of the device orientation. The actual implementation of this feature is part of the :studydata application on the respective platform.

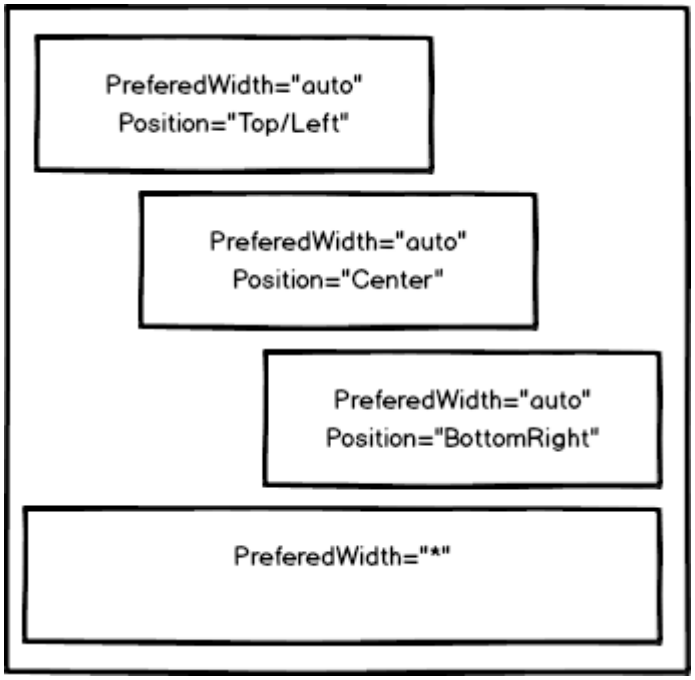


Figure 14: Positioning of elements in a linear layout with vertical orientation

The remaining Concrete Form Elements (in Figure 12 denoted as `ConreteSimpleElements`) are the concrete versions of one or several Abstract Form Elements. While for the manual input fields of the Abstract Form Model there are different options to implement them on the concrete layer, for the other AFEs there exists exactly one concrete element that represents it. Except the layout attributes, the Concrete Form Elements have no further specific attributes. The reason for this is that all properties which can be specified are defined on the abstract layer because they are valid for each of the possible target devices. Only the visual appearance of an Abstract Form Element might differ on different target devices. To make the properties, defined on the Abstract Form Model layer, available for the Concrete Form Elements, each CFE provides the attribute `AbstractReference`. This attribute holds the reference to the abstract base element of the CFE on the Abstract Form Model. Thereby, the attributes set to the referenced AFE are available to its concrete implementation. In addition, this referencing mechanism avoids copying data from the Ab-

stract Form Model to the Concrete Form Model. This eases the propagation of changes between the model layers (see Section 4.3).

Following, the available `ConcreteSimpleElements` are described in detail. Furthermore it is stated which Abstract Form Elements they represent and a wireframe image illustrates the visual appearance of the elements. The illustration has to be understood as an example. The actual visual appearance differs depending on the device type and the platform. Moreover it is possible that not all of these elements are available for each device type.

ConcreteDisplayElement

Represented AFE: `AbstractDisplayElement`

The display element shows the value of the referenced data definition on the data form. Editing the value is not possible. The fact that this is a read only field is expressed by the blue colored border of the value field. This optically distinguishes this element from the `TextBlock` and `TextBox` elements (see below). The decision for the blue color has no special reason.

Figure 15: Concrete Form Element `DisplayElement`

TextBlock

Represented AFE: `Description`

This is a simple output element for predefined static textual content. It just prints the text to the screen and inserts line breaks if necessary. To clearly distinguish between additional informative text and displayed data, this element cannot be used to represent data defined in the Data Definition Model.

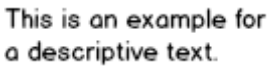


Figure 16: Concrete Form Element `TextBlock`

TextBox

Represented AFEs: `TextEdit` or `NumericalEdit`

The `TextBox` element is a single line field for entering alphanumerical characters. The element does not allow line breaks. It is intended to be used for strings that have a fixed maximal length or which are expected to be rather short (up to 30 characters). A second use case for this element is a numeric value that is supposed to lie in a wide range. An example for this is a decimal number with a large amount of fractional digits. In this case entering the value using the `NumericUpDown` (see below) element would be tedious.

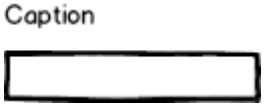


Figure 17: Concrete Form Element `TextBox`

TextArea

Represented AFE: `TextEdit`

A `TextArea` allows entering a free text. This includes the possibility to enter line breaks. The text length is not limited.

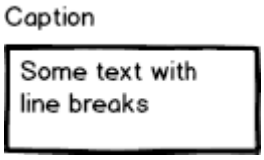


Figure 18: Concrete Form Element `TextArea`

NumericUpDown

Represented AFE: `NumericalEdit`

This is an input element for numeric values. It provides an input field for entering the value via keyboard as well as buttons for incrementing or decrementing the current value by the step size specified at the Abstract Form Model.

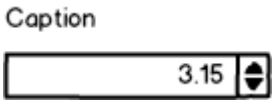


Figure 19: Concrete Form Element `NumericUpDown`

CheckBox

Represented AFE: `BooleanEdit`

This element enables the user to set a Boolean value by checking or unchecking a checkbox.

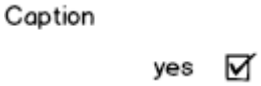


Figure 20: Concrete Form Element `CheckBox`

ToggleControl

Represented AFE: `BooleanEdit`

The `ToggleControl` is an alternative for the `CheckBox` which is especially used on mobile devices. It provides a stylized switch that signifies the current state of the Boolean value. By clicking or touching the switch, the state can be changed.

Figure 21: Concrete Form Element `ToggleControl`

DropDown

Represented AFE: `Selector`, `TextEdit`

If the available screen size is very limited, the `DropDown` element is a good realization for the abstract `Selector` element. It shows the currently selected value similar to a `TextBox` and shows the available values on a popup or full page menu if the

user clicks or touches it. The `DropDown` element can alternatively be used for textual values. Then the user can either enter text manually or select one of the default values suggested in the drop down menu.

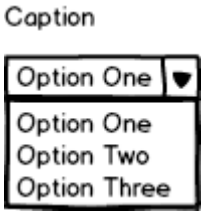


Figure 22: Concrete Form Element `DropDown`

List

Represented AFE: `Selector`

The `List` element displays a list of possible values on the screen and lets the user select one of these values by clicking or touching it. Since this element always shows all available options it needs a lot of screen space. If the element size is smaller than the space needed to show all options, a scroll bar is shown.

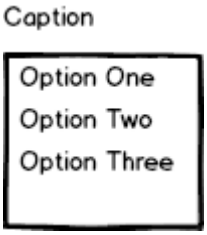


Figure 23: Concrete Form Element `List`

RadioButtons

Represented AFE: `Selector`

A further alternative for implementing the abstract `Selector` element is the `RadioButtons` element. It provides a dynamically arranged set of radio buttons; one for each possible value. The user can select the value by marking one of the radio buttons. Like the `List` element, this element always displays all available options on the screen. Since the options are presented by one radio button each, no scrolling is possible to limit the needed screen size.

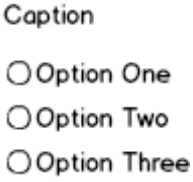


Figure 24: Concrete Form Element `RadioButtons`

ConcreteTestSubjectSelector

Represented AFE: `AbstractTestSubjectSelector`

The `ConcreteTestSubjectSelector` element allows selecting the current test subject out of a list of all study participants. The actual realization of this element

differs depending on the target device and platform. On a desktop system it can for example be a set of toggle buttons that carry the codes[13] of the available test subjects (see Figure 25 left). However, on a mobile device it might be implemented as a simple drop down box due to the limited available space (see Figure 25 right).

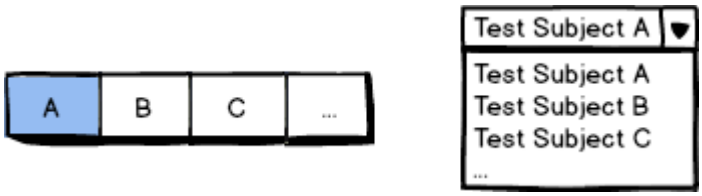


Figure 25: Examples for the Concrete Form Element `TestSubjectSelector`

Button

Represented AFE: `Command`

With the `Button` element the user of a data form triggers actions of automated input elements or the *"commit"* and *"reset"* actions of the data form itself.

Figure 26: Concrete Form Element `Button`

ConcreteAutomaticInput

Represented AFE: `AbstractAutomaticInput`

This element is the representation of an automated input device on the concrete layer. Since the actual implementation of this type of element depends on the target platform (see Section 4.4) this element just acts as a dummy for which the position and size is determined.

4.3 Model Changes

An essential problem that arises when using a model-driven approach that includes several model layers is the consistency between the models on the different layers. The following paragraphs explain this problem in more detail. Following it is described how the problem is, at least partially, solved by the described approach for designing multi-device data forms.

The general problem arises after a transformation from one model layer to the next initially took place. From this state on, the designer can make changes to the target model in order to edit it for his purposes. However, at any time in the design process the designer is allowed to make changes at the upper model layers, which are the source models of the initial transformation. This possibility is needed since otherwise there would be no option to correct errors at that layers or add missing elements. To ensure that the models of the source and target layer of the transformation do not be-

[13] For reasons of anonymization, an alphanumerical code is assigned to each study participant. This code is used for storing data concerning this participant in the :study database. Matching these codes to real names is only possible offline, which means outside the :study software.

come inconsistent, changes made to one of the affected models need to be propagated to the respective other model. Thus, the problem can be divided into two parts: On one hand, changes made to the higher model layer need to be propagated to the lower model layer. The other way round, depending on the structure of the models, it is also possible that changes made to the lower model layer need to be propagated back to the source model. For the model-driven approach described in this thesis there might also arise the question whether changes on a Concrete Form Model should be automatically propagated inside the CFM layer to models targeting other devices. Since the intention of splitting the models of this layer is to allow designing independent versions for the different target devices, this is considered to be not useful.

The propagation of changes made on lower model layers back to the respective source model is solved by the way the models are designed and work together. Each of the described model layers (Data Definition Model, Abstract Form Model and Concrete Form Model) just contains information that is specific for this layer. This means that for example information that is contained in the Abstract Form Model layer is not copied to the Concrete Form Model layer when transforming from AFM to CFM. Instead, a reference to the source elements is added to the elements of the CFM. Figure 27 illustrates the difference between copying information to the next model layer (left) and referencing the source element (right) by the example of the `TextEdit` Abstract Form Element and `TextBox` Concrete Form Element.

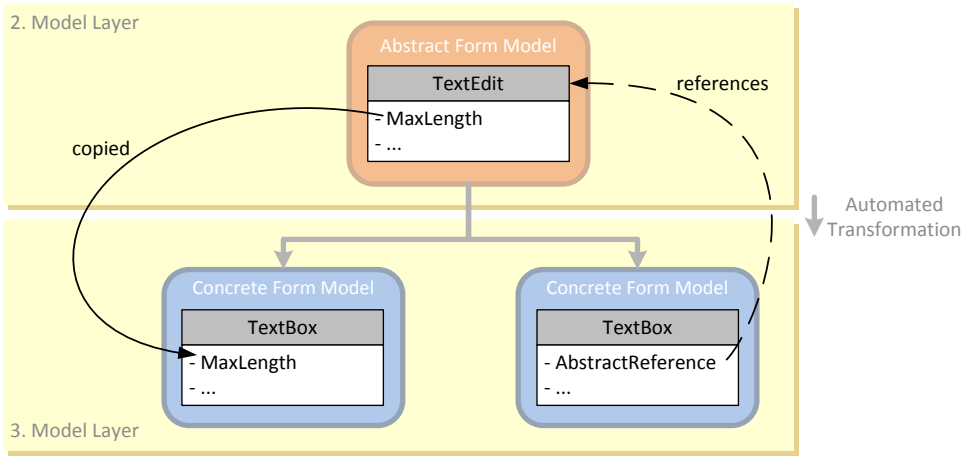


Figure 27: Copying of information versus referencing the source element by the example of the `TextEdit` Abstract Form Element and `TextBox` Concrete Form Element

Since the Concrete Form Model thereby does not contain all information directly, a transformation from the CFM to a Final Form additionally needs the AFM as a second source model to access the information set at the abstract layer. This information is needed for generating a Final Form implementation that considers all settings made by the designer. Since the described model-driven approach is intended to be used in a limited area using dedicated applications that support the designer in applying the approach, namely the :studyforms application, this is assumed to be applicable. Because the lower model layers do not contain details that may be changed by the designer and that are also applicable to the higher model layer, there is no need for propagating changes from lower model layers to higher ones.

The problem of propagating changes made to the source model down to the target model(s) after the initial transformation is serious. The problem is that most of the available transformation approaches are stateless (compare [Tra08]). This means that after the initial transformation was performed, the only possibility to apply changes on the source model to the target model is to rerun the transformation again. This produces a new version of the target model from scratch regardless of the already existing version. The question is now what to do with the two versions of the same model. The easiest solution is to overwrite the old version with the new one. In the course of this thesis, this solution is used in the prototypical implementation. The essential drawback of this method is that the changes to the target model made by the designer get lost. Since this behavior is not acceptable for a later productive implementation of the model-driven approach, possible solutions for the problem are explained in the following paragraphs. To implement one of the described approaches remains open for future work.

A simple idea that arose during this thesis is to merge the two versions of the models by some merging algorithm. Since this approach does not consider the relations between source and target elements it may not properly work if the elements are renamed and reordered.

Tratt and Clark propose a solution that is based on logging the changes to the source model [TC03]. These logs are denoted as *change deltas*. The change deltas are the input for *delta transformations* that generate deltas to the target model that are implied by the change deltas. In a second step, these deltas are applied to the target model. Instead of dividing this process in two separate steps it would also be possible that the delta transformations directly perform the changes to the target model. A drawback of this solution is that the number of different possible changes can be very high. This results in a large number of needed delta transformations which have to be implemented manually. Furthermore, the application of delta transformations might overwrite intentionally made changes to the target model. This could be avoided by marking regions of the target model that contain changes made by the designer as protected. This *Protected Regions* approach is for example used by Xtext[14] (compare [Die]). Therefore, special keywords are introduced that mark the start and the end of such regions. The code that is enclosed by a Protected Region is not changed by the transformation process.

A transformation language that directly supports change propagation joins the above mentioned solutions in one transformation framework. This means, that the transformations propagate changes on the source model to the target model without destroying manually made changes on the target model. Therefore, the transformation framework includes a method for transforming change deltas and additionally considers removed elements. To achieve this, such a transformation framework needs to distinguish elements on the target model to specific source elements. An example for a change propagating model transformation is given by Tratt [Tra08].

[14] Xtext is a framework for developing and generating programming and domain specific languages.

For the implementation of the model-driven approach introduced by this thesis, a mixed solution of tracking changes to the source model and protected regions in the target model that contains manually made changes is conceivable. This means, that the tool which supports a designer in using the model-driven approach tracks changes on the higher model layers after an initial transformation to a lower model layer is done. This tracking includes added and removed elements as well as attribute changes. Since the elements on the lower model layers reference their base element of the higher layer, each tracked change can be propagated to the lower layer by altering the referenced element at the lower model appropriately. If a data definition for example is deleted at the DDM layer also the Abstract and Concrete Form Elements that reference this data definition are deleted. If a new data definition is added or an attribute of an existing one is changed, it is conceivable to only transform this single data definition again and insert the transformation result into the Abstract Form Model. Thereby the transformation logic remains encapsulated at the transformation process. Inserting a new element can be done based on the element's siblings in the hierarchical structure. A drawback of this solution is that the transformation of a single element does not consider the element in context of its whole model. To prevent overwriting manually made changes at the lower model layer special XML comments can be inserted around those parts of the model's XML representation that mark these regions as not changeable. Due to these Protected Regions it is possible that the propagation of changes causes conflicts that have to be solved manually by the designer or by a complete retransformation.

4.4 Automated Data Acquisition

As already mentioned in the introduction (Chapter 1) the data capturing workflow can be optimized by reading values directly from peripheral devices, instead of entering them manually to the data form. Since it is not predictable which kind of samples have to be processed during future studies, it is not possible to define a closed set of peripheral devices that can be integrated into the data forms for gathering values in an automated way. Therefore, a flexible system is needed, that allows defining such *Automated Input Devices* (AID). The idea to solve this problem is to formally define the available devices in a separate model, the *Input Device Model* (IDM).

In the preceding chapters, already two possible devices that are used to optimize the data capturing process are mentioned: A barcode scanner and an electronic balance. These devices serve as examples for the definition of Automated Input Devices. Therefore their application is concretized by the example of the data capturing process of 24h urine samples (compare Section 2.1.3). The usage of the scale device is intuitive: The bottles, containing the urine samples, are weighed using the balance and the result is entered automatically into the respective field on the data form. The barcode scanner is used in a more complex fashion: The urine bottles that are collected during the 24 hours period are equipped with a barcode. This barcode contains a unique bottle number, the tare weight of the bottle and the code of the study participant, to which the sample belongs. This means, that for each study participant, a set of urine bottles is available. Thus, a participant only uses bottles that carry his test

subject code. The cap of the bottle carries a separate barcode which only contains the tare weight of the cap. This makes the caps exchangeable between the bottles which eases the handling for the study staff. Hence, when processing the samples in the laboratory, both barcodes need to be scanned. Thereby, the right test subject is selected on the data form. Furthermore, the bottle number, which is needed to match the sample to one of the initially captured ones, and the tare weight of the urine bottle are entered automatically to the data form. The tare weight is needed to calculate the net weight of the urine sample out of the determined gross weight. The development of this barcode system was not part of this thesis, but was done in an earlier stage of the :study project.

During the design process, a peripheral device is represented by an Automated Input Device element on the Abstract or Concrete Form Model. The `AbstractAutomaticInput` element references to one of the defined AIDs in the Input Device Model. The definitions in the IDM include information about the attributes a designer can set for an Automated Input Device. Additionally possible actions provided by an AID are defined. These actions can be made available on the data form by the designer, such that they can be triggered by the later user of the data form (see `Command` element in Section 4.2.2).

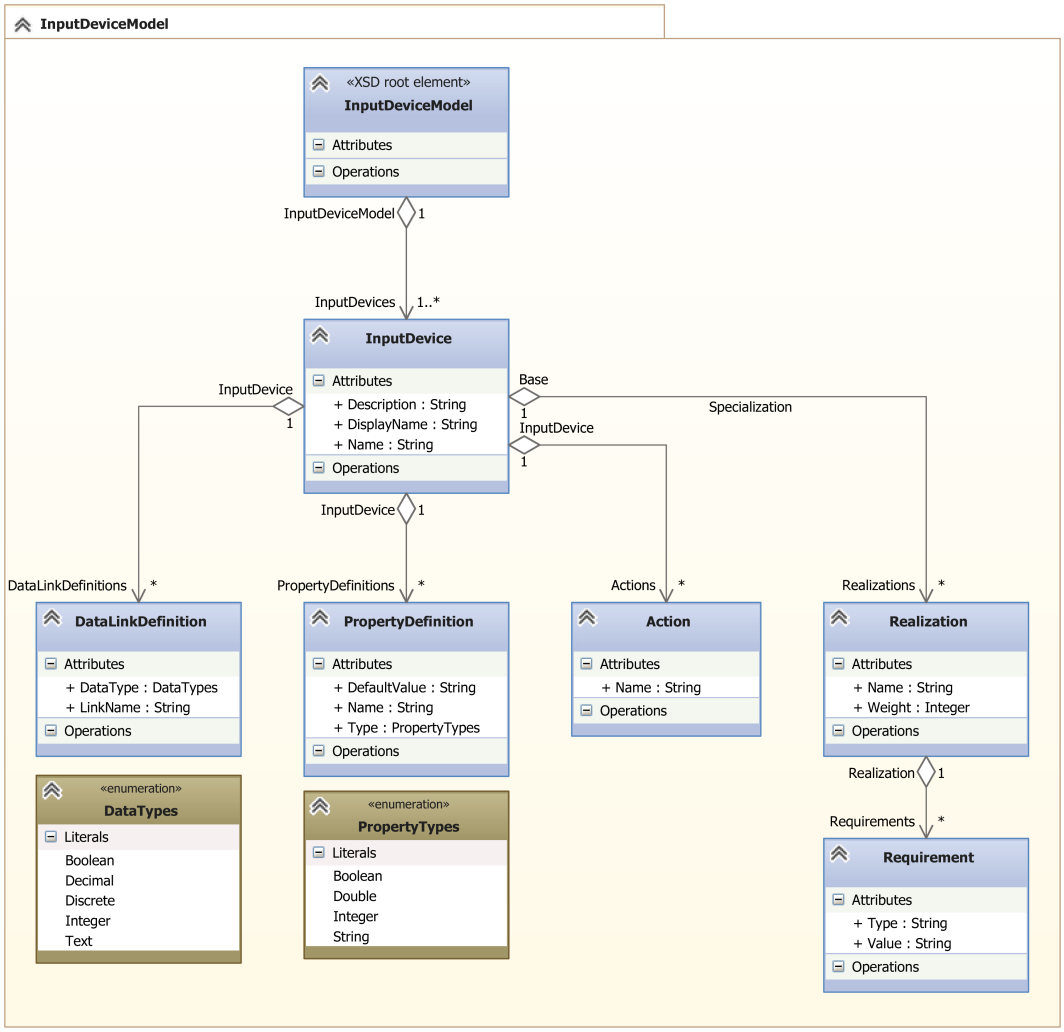


Figure 28: Meta-model of the Input Device Model

As already mentioned above, the IDM is not a direct part of the data form design process. The model does not need to be newly generated or adapted each time a new form is designed but is static. During the design process it provides information about the available Automated Input Devices that can be added to a data form. Only if a new AID is added to the system, the model has to be adapted. The definition and deployment of new Automated Input Devices is done by the developers of the :study project. It is not intended to be done by a data form designer himself.

Figure 28 depicts the meta-model of the Input Device Model. Like the other models, the IDM is stored using the XML format. The definitions of Automated Input Devices are not ordered in a hierarchical structure within the model. Nevertheless, the structure of an XML document requires a single root element. Therefore, the artificial element `InputDeviceModel` which has no attributes builds the model's root element and contains the list of defined devices.

The definition of an input device is done using the `InputDevice` element. Each of these elements is identified by a unique name (attribute `Name`). This name is used to reference a specific AID out of the Abstract Form Model (compare to the `AbstractAutomaticInput` element in Section 4.2.2). Additionally, the element allows to specify a `DisplayName` and a `Description` that are used in the :studyforms application to present the Automated Input Device and describe its functionality to the data form designer.

In contrast to the input fields that are filled out manually by the user, an Automated Input Device can provide more than just one of the values captured by a data form. As a reminder: The urine barcode scanner provides the bottle number and the tare weight of the bottle. This requires the possibility to bind an AID to several data definitions of the Data Definition Model. Which values are provided by an AID is defined by specifying a set of `DataLinkDefinition` elements as sub elements of an `InputDevice` element. A `DataLinkDefinition` characterizes a provided value by a name and the data type of the value. The list of possible data types matches the one defined for the data type of data definitions in the DDM.

Similar to the definition of the provided values, for each Automated Input Device further properties can be specified that change its behavior or its appearance on the data form. Such properties are defined using the `PropertyDefinition` element. Besides the attributes for defining the name and the data type of the property, a property definition contains the attribute `DefaultValue`. This can be used by the developer of the Automated Input Device to specify a default value that is used for the attribute if the designer of the data form does not set a value for that property. The available data types for a property definition are limited to the basic types *"Boolean"*, *"String"*, *"Integer"* and *"Double"*. An example for such an attribute definition is a Boolean flag that can be set for the urine barcode scanner AID and specifies whether the tare weight of the barcode scanner is used by the data form. This information is important because the cap of a urine bottle carries a separate barcode including the tare weight of the cap. Thus, if the tare weight is required, both barcodes have to be scanned before the data capturing workflow can continue. If a urine sample is just

collected without weighing it, scanning the barcode of the cap is needless since it only contains the tare weight of the cap which is of no interest in this situation.

The last aspect that is important for the abstract definition of an Automated Input Device is the information about the actions provided by the device element. An action has to be understood as a functionality of the AID that is triggered by the user of a data form. The available actions of an input device are defined using the `Action` element. The `Name` attribute specifies the identifier of the action. It is used to reference the action from a `Command` element on the Abstract Form Model. The electronic balance AID provides two actions. The *"Weight"* action causes the current weight value of the device to be inserted into the related field on the data form. The *"SetTare"* action triggers the balance device to tare.

Beside the already mentioned attributes, the `DataLinkDefinition`, `PropertyDefinition` and `Action` elements additionally contain a `Description` attribute. It is used to explain the details of the respective aspect of an Automated Input Device. For a `DataLinkDefinition`, this description for example gives details about the provided value or for a `PropertyDefinition` the effect this property has to the AID is explained. The descriptions are displayed to the designer in :studyforms and help to understand how to use the AID.

Due to the different features of the target devices that are used for data capturing, the actual implementation of an Automated Input Device depends on the target device on that it is used. This means that the :studydata application, which is be implemented for each target device and platform, has to include implementations for the AIDs defined in the Input Device Model. Additionally, the functionality and behavior of an AID can vary on different devices and platforms. Taken the barcode scanner example from above, it might be the case that it is realized using a real barcode scanner device on a desktop system, whereas on a mobile device it uses the integrated camera to read the barcodes. As a consequence, the requirements an AID imposes are different depending on which device type and on which platform it is used. For instance, the barcode scanner requires a port for attaching the barcode scanner device on a desktop system whereas the mobile version of the input device depends on an integrated camera. Since the actual implementation of the logic of an AID has to be done specifically for each target platform, it is also conceivable that there are platforms on which the :studydata application does not include an implementation for some AIDs at all.

For these reasons the Input Device Model contains information about the available implementations of each Automated Input Device on the target platforms. In the last transformation step, which generates a Final Form for a specific platform out of a Concrete Form Model, this information is used to select the concrete realization of an Automated Input Device that is available for the respective platform and fits the features of the target device. Therefore, for each AID the available implementations are defined in the IDM using the `Realization` element. The `Name` attribute of this element is set to the name of the class that implements the AID functionality on a target platform. For each realization a set of requirements explains on which platforms this specific realization can be used and which device features the AID need to

operate properly. A `Requirement` element therefore has a `Type` and a `Value` attribute. With the `Type` attribute, it is determined to which aspect a requirement refers. An example for this is the platform, on which the AID is available. To keep it open for future development on the device market and new device features, the IDM meta-model does not define a fixed list of allowed requirement types. Thus, also the possible values of each requirement type are not predefined by the IDM's meta-model. A list of conceivable requirement types and values combinations as well as a short explanation is given in Table 1.

Type	Values	Description
Platform	WP, Android, iOS	Restricts the availability to a specific target platform
PlatformVersion	WP7, WP8, …	Restricts the availability further to a specific version of a platform
Camera	Required, Front, Back	Requires a camera; possibly explicitly at the front or back of the device
Port	USB, Serial, …	Requires a specific port for connecting a peripheral device
Bluetooth	Required, 4.0, …	Requires Bluetooth wireless connection

Table 1: Possible requirement types and values of automated input device definitions

The transformation that generates a Final Form implementation out of a Concrete Form Model matches these requirements to the features of the intended target device. This can be hard coded to the transformation's implementation if the features of the devices of a specific platform are fixed. In this case, the developer of the transformation needs to consider the used requirement types and their possible values. If the diversity of the end user devices of a specific platform, and thereby the variety of the devices' features is too high, it is also conceivable to provide the information about the actual target device dynamically to the transformation. This could for example be done by an additional *Target Device Model* which defines the capabilities of the target device. This model can be conceived as the counter part of the requirement definitions of the Input Device Model. The transformation uses this model to match the requirements with the capabilities of the target device. Thereby it is possible to determine the realization of an Automated Input Device that is applicable to the intended target device's platform and features. Since the usage and the characteristics of such a Target Device Model depend on the implementation of the transformation from the Concrete Form Model to the Final Form, this model is not defined further in the context of this thesis.

During the transformation from the Concrete Form Model to the Final form it is possible that the matching of requirements of an Automated Input Device and capabilities of the target device yields to more than just one realization of an AID that is applicable for the target device. For handling this case, the `Realization` element contains the `Weight` attribute. It is set to an arbitrary integer value. If several realizations are relevant the one with the highest weight value is chosen. When defining an

AID in the Input Device Model it has to be taken care that the weight values are unique throughout the realizations of an AID.

Listing 8 shows the definitions of the above mentioned electronic balance and urine barcode scanner Automated Input Devices. The description attributes are omitted in the given listing. The reason why the barcode scanner AID does not contain a `DataLinkDefinition` for the test subject contained in the barcode is that the Data Definition Model does not provide a data definition for the current test subject. It is set directly by the implementation of the barcode scanner AID.

```
 1 <InputDevice Name="UrineBarcodeScanner" DisplayName="Urine Barcode Scanner">
 2    <DataLinkDefinition LinkName="BottleNumber" DataType="Text" />
 3    <DataLinkDefinition LinkName="TareWeight" DataType="Decimal" />
 4    <PropertyDefinition Name="IsScanCap" Type="Boolean"/>
 5    <Realization Name="UrineBarcodeScanner">
 6       <Requirement Type="Port" Value="USB" />
 7    </Realization>
 8    <Realization Name="UrineBarcodeScannerWithCamera">
 9       <Requirement Type="Camera" Value="Back" />
10    </Realization>
11 </InputDevice>
12
13 <InputDevice Name="Balance" DisplayName="Laboratory Balance">
14    <DataLinkDefinition LinkName="WeightValue" DataType="Decimal" />
15    <Action Name="Weight" />
16    <Action Name="SetTare" />
17       <Realization Name="LaboratoryBalance" Weight="10">
18          <Requirement Type="Port" Value="Serial" />
19       </Realization>
20       <Realization Name="BluetoothBalance" Weight="5">
21          <Requirement Type="Bluetooth" Value="Required" />
22       </Realization>
23 </InputDevice>
```

Listing 8: Input Device Model definitions of the barcode scanner and electronic balance Automated Input Devices

4.5 Element Sizing

Until now, the described concept considers the crucial differences in the screen sizes of smartphone, tablet and desktop devices by defining an individual layout of a data form for each device type. This makes it feasible to assume that the screens of different machines that use the same Concrete Form Model have almost the same size. The critical part of this assumption is the word “almost” because there are still small variations. Typical displays of today’s smartphones for example have sizes between 3.5” and 5”. Of cause these values are much closer together in comparison to a tablet with about 7” to 10” but there are still differences. In addition to that, there are smartphones on the market whose display sizes are close to the lower end of the tablet screen sizes.

In addition to the physical size of the displays, their resolution as well as their aspect ratio is another crucial point. For some platforms the supported resolutions are very limited. For example Windows Phone 8, which is the current up to date version of the Windows Phone operating system, only supports three different resolutions with two aspect rations (compare [Mic13b]). For others, like Android, the diversity is much higher because the vendor does not limit the amount of different display sizes that strict. An example for this is the Android system which is used on smartphones and

tablets. Figure 29 illustrates the differences in resolutions supported by Windows Phone 8. The graphic shows that even if the supported resolutions are limited, the differences cannot be ignored.

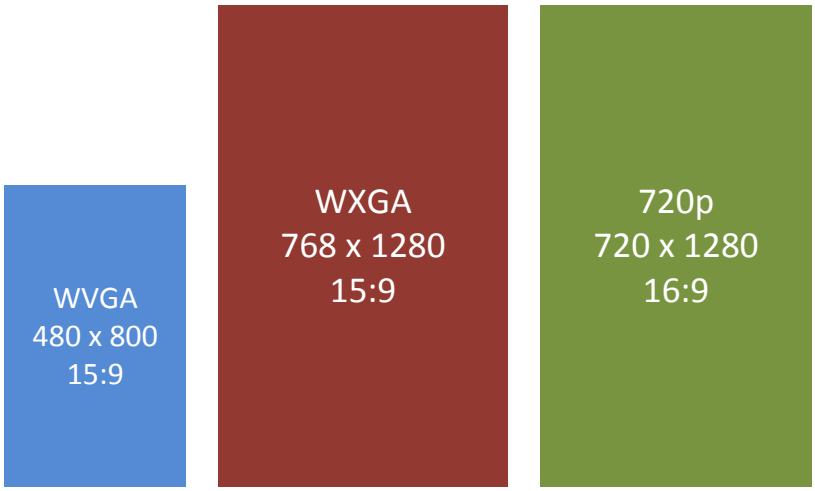


Figure 29: Supported resolutions of the Windows Phone platform (according to [Kuh12])

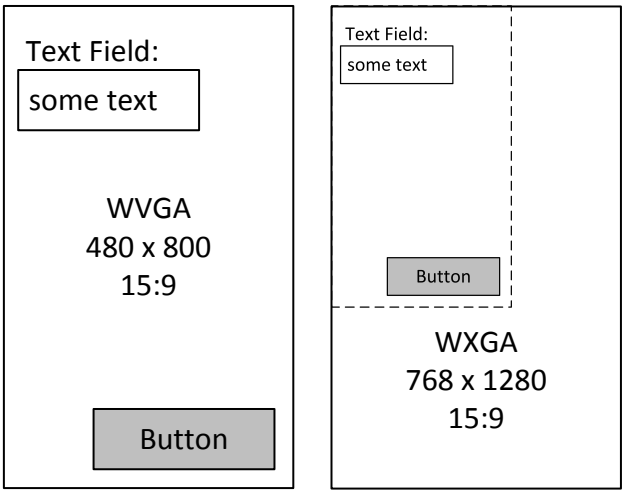


Figure 30: Comparison of elements with fixed sizes and positions on two different resolutions

Another aspect is that even if the supported resolutions are limited, it is still possible that devices with the same physical display size use different resolutions. This results in variable pixel densities or in other words, in different sizes of a single pixel.

The described differences can be neglected as long as the designer of a data form only uses automatic or relative sizes. Unfortunately, this would not result in a nice design that considers the design concept and guidelines of the respective platform. For example the height of a button or a text field is usually predefined in the design guidelines, whereas the width should be determined automatically. Therefore, a good design also contains elements with fixed sizes. If these sizes would be specified in terms of real device pixels this would yield in a very different presentation on devices with different pixel densities. Figure 30 depicts a text box and a button with a fixed size of 100x300 pixels on a WVGA resolution (left) and on a WXGA resolution (right),

which is exactly 1.6 times larger. The positions of the elements are also assumed to be given by fixed pixels relative to the upper left corner. Although the actual display size stays the same, the text box on the right is larger than on the left because the resolution changed and therefore each single pixel has a different size.

To overcome the described problem the vendors of the different operating systems for mobile devices mainly use two different solutions:

The first one, which is used for example by Windows Phone, is to define a minimal baseline resolution on which base the design of the GUI is done (compare [Kuh12]). If the GUI is displayed on a screen with a resolution higher than the baseline, it is scaled up, independent of the pixel density. This means that on displays with the same physical size but with different resolutions, the GUI elements keep the same size, while on larger screens the elements become larger. Since the variation of display sizes of current Windows Phone devices is rather small, this approach is reasonable.

The second option, for example used by Android, specifies the size of a GUI element in terms of *Density Independent Pixels* (see [Goob]). Also with this approach, a GUI element whose size is defined using this kind of pixel values keeps the same size on two equally sized screens with different resolutions. The difference to the first approach is that this also holds if the display becomes larger and has a higher resolution. Instead of becoming larger, the GUI elements keep their size and potentially more elements would fit on the screen.

Due to the fact that the described approach for designing platform independent data forms already splits the platforms in different device types, the variation of screen sizes is considered to be rather low for devices that use the same version of the data form. This assumption is applicable because the :study software is used in a closed environment where it can be influenced which devices are used. Therefore, it is reasonable to compensate the described screen differences by scaling up the form such that it fills the whole screen. This behavior ensures that the user gets a data form with the same layout although he uses different devices of the same type and the same platform. Thus, the specification of fixed sizes is done in terms of *Logical Pixels* (lp) which are normed to a baseline resolution that is different for each device type. The baseline resolutions of the considered device types are defined in Table 2. The values are based on low-end resolutions of currently available devices of each type.

Device Type	Baseline Resolution
Mobile	480 x 800
Tablet	1366 x 768
Desktop	1440 x 900

Table 2: Defined baseline resolutions for the different target devices

For desktop devices this might be a too strict assumption because the variation of screen sizes is much higher for this device type. Here it might be reasonable to display the data form in a non-fullscreen mode. Since this thesis mainly focuses on mobile devices, this is not further explored and remains open for future work.

4.6 Transformations

As already mentioned, the transitions from one model layer to the next one are supported by automatic transformations (compare Section 2.2.2). The following sections explain the transformations applied during the model-driven data form design process.

4.6.1 Data Definition Model to Abstract Form Model

The first automatic transformation takes the Data Definition Model as the source and builds a basic version of the Abstract Form Model, from which the designer can start to build up the data form's layout. Thus, the main objective of the transformation is to generate a first, abstract version of the data form's graphical user interface. Therefore, the transformation has to map the elements of the DDM to user interface elements that are suitable to show or manipulate the defined data. To determine appropriate Abstract Form Elements, the transformation uses the information that is directly defined for the data definition elements, as well as information that is implicitly available due to some data definitions referencing parameters of the :study database. The result of this transformation step is an abstract version of the data form that is complete in respect to the Data Definition Model. This means that the data form contains all elements that are necessary to use the form for capturing the specified data. Only if the designer likes to add further features to the data form (for example Automated Input Device elements for capturing some of the values in an automated way) or to do some other changes, the resulting Abstract Form Model has to be adapted. But, the generated AFM can be used for the further design process even without such changes.

Due to the similar hierarchical structure of the Data Definition Model and the Abstract Form Model, the transformation starts by mapping the DDM's root element `DataModel` to an instance of the `AbstractForm` element, which builds the root of the AFM. The name of the new element is taken from the source element without any adaptions. This helps the designer to better understand the relations between the DDM and the AFM and avoids evolving lengthy names during the transformation steps that would be the result if some layer specific pre- or postfix would be added. The same holds for the transformation of `DataGroup` elements. They are mapped to `AbstractCompound` elements, containing exactly these Abstract Form Elements that relates to the child data definitions of the source data group.

This results in an abstract layout structure of the resulting Abstract Form Model that reflects the grouping of data definitions on the DDM layer. The structure ensures that the UI elements representing data of the same group are spatially close together on the data form's user interface. On one hand this is an intuitive assumption; on the other hand this is proven to be a good way for grouping related elements because it conforms to the proximity rule of the Gestalt Laws (see [HB11], p. 515). The described behavior makes clear that it is essential for the transformation process that the designer builds plausible groups of data definitions already at the DDM layer. Regardless of that, grouping elements is still possible on the Abstract Form Model.

For mapping the two data definition elements `DataDefinition` and `VolatileDataDefinition` of the DDM to appropriate Abstract Form Elements the transformation takes three attributes of the data definitions into account. The decision tree in Figure 31 illustrates graphically how the data definition elements are mapped to Abstract Form Elements.

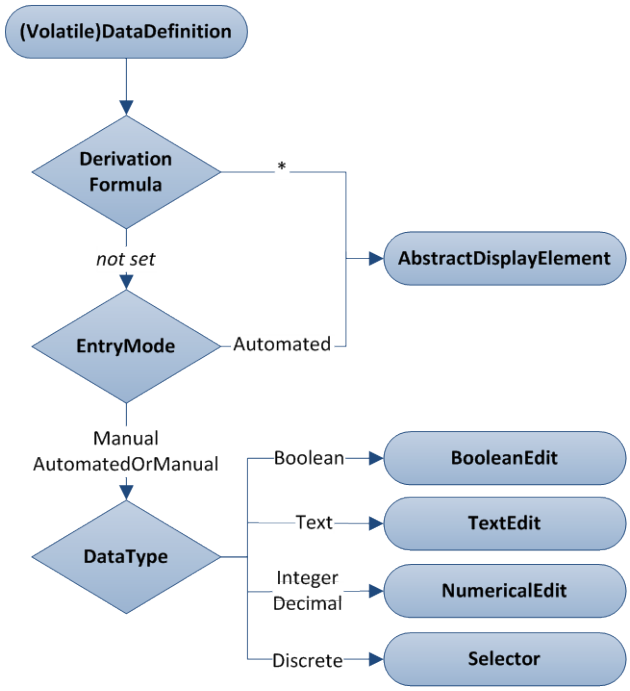


Figure 31: Mapping data definitions to Abstract Form Elements

The first two of the considered attributes are `EntryMode` and `DerivationFormula`. They allow deciding whether the Abstract Form Element that represents the data definition should allow entering and changing the value or whether the value is gathered in an automated way and should just be displayed to the user. If the `EntryMode` is set to *"Automated"* or the designer has specified a derivation formula, the data definition is mapped to the `AbstractDisplayElement`. The name of this element is set to the name of the data definition prefixed by *"DisplayElement_"*. This simplifies to recognize the control's type during the design process in the :studyforms application. For the `Caption` attribute, the name of the data definition is taken purely.

If the `EntryMode` attribute of a data definition is set to *"Manual"* or *"AutomatedOrManual"* and no derivation formula is specified, the transformation inserts one of the available manual edit fields to the generated Abstract Form Model. Which type of edit field is inserted is determined by taking the data type of the data definition element into account. In case of a `VolatileDataDefinition` the data type is defined on the DDM. For a "normal" data definition the data type is read from the referenced parameter in the :study database. Thus, a data definition with *"Boolean"* data type is transformed to a `BooleanEdit` element and a data definition of type *"Text"* results in a `TextEdit` element. A `NumericalEdit` element results from *"Integer"* or *"Decimal"* data types. For decimal data the `FractionalDigits` at-

tribute of the generated `NumericalEdit` element is set to *"2"*, which is assumed to be an appropriate default value for the amount of fractional digits of a decimal number. For `NumericalEdit` elements that refer to an integer data definition, the number of fractional digits remains at its default value, which is defined to be zero. The *"Discrete"* data type specifies that the value has to be set to one of a set of predefined values. Therefore for such a data definition a `Selector` element is added to the AFM.

Like for the `AbstractDisplayElement`, the Name and the Caption attributes of the manual edit elements are set to the name of the referenced data definition prefixed by the element name of the manual input field (for example *"TextEdit_"*). For all AFEs that are generated from a data definition of the DDM the `DataReference` attribute is set to the path which points to that data definition.

Another aspect of the Abstract Form Model that can be preset by the transformation is the activation order of the form elements that require some user interaction (compare Section 4.2.2 – Interaction Elements). Therefore, the `Successor` attributes of these elements are set to the next such element in the tree structure. As the tree structure represents the graphical layout of the data form this basic activation order fits to the users' intuitive reading direction which is (at least in the western world) from top-left to bottom right (compare [Wes98], p. 108).

All the transformation's details mentioned until now ensure that each data definition of the DDM is represented by an appropriate form element on the data forms GUI. In addition, the attributes of the generated Abstract Form Elements are set to suitable or at least plausible default values. But for generating a complete abstract data form that could be used without any changes by the designer, two important details are still missing: The form does not provide any elements for selecting the current test subject and for submitting or resetting the entered values. Since these functionalities are not represented in the Data Definition Model, the transformation inserts appropriate elements additionally:

As the first sub element of the Abstract Form Model's root element an `AbstractTestSubjectSelector` element is inserted. This is caused by the typical data form fill in process that starts with first selecting the study participant, to whom the entered data is assigned to (compare Section 2.1.2, Figure 3). Although this selection could be done automatically by some automated input device, there should usually also be the possibility to do a manual selection, which exactly is the scope of the `AbstractTestSubjectSelector` element. To make sure that this element is also the first one that is activated after opening or resetting the data form, a reference to that element is set to the `FirstElement` attribute of the `AbstractForm` element.

In the same way, two `Command` elements are added to a separate compound named *"Finishing"* at the end of the Abstract Form Model's tree structure. The `Caption` attributes of these elements are preset to *"Submit"* and *"Reset"* and the `ActionReference` is set to the respective actions of the data form. To complete the data

form internal workflow, the `LastElement` attribute of the `AbstractForm` element is set to the path of the inserted submit command.

Appendix B illustrates the Data Definition Model to Abstract Form Model transformation by an excerpt of the 24h urine laboratory data form.

4.6.2 Abstract Form Model to Concrete Form Model

The second automatic transformation generates a Concrete Form Model out of the Abstract Form Model. This means, a concrete layout of the data form is generated from the structure of the abstract elements on the AFM. Furthermore, the Abstract Form Elements are translated into their concrete versions. These concrete versions as well as the data form's layout, gathered from the abstract structure of the AFM, depend on the actual target device of the Concrete Form Model. Therefore, specific transformations have to be available, which produce the Concrete Form Model for one of the introduced target devices types (*Mobile*, *Tablet*, and *Desktop*). A `BooleanEdit` element on the Abstract Form Model is for example transformed to a `CheckBox` element on the Desktop Concrete Form Model, whereas on the Mobile Concrete Form Model it is transformed to a `ToggleControl` because this control is optimized for touch interfaces. Since this thesis focuses on data forms for mobile devices and due to the limited time, the transformation to the *Mobile Concrete Form Model* (mCFM) is emphasized. This is explained in the following paragraphs. For comparison reasons, at the end of this section also some basic thoughts about the transformation to the Tablet Concrete Form Model are introduced. The transformation to the Desktop Concrete Form Model is not treated in the context of this thesis.

Due to the very limited available space on a smartphone screen, it is not possible for the mobile version of a data form to display all elements simultaneously. Instead, the elements on the Mobile Concrete Form Model need to be divided in a sensible way. Thus, the transformation to the mCFM does not preserve the element's structure of the Abstract Form Model. For defining groups of elements that are displayed at the same time, the Concrete Form Model allows specifying `Page` elements. Based on the element compositions on the AFM layer, the transformation divides the concrete elements into several pages. In general one can say that each `Composition` element defined on the Abstract Form Model, results in at least one page on the Mobile Concrete Form Model. This method requires that the designer does not build compositions with a large amount of child elements that do not fit on one page. To handle such cases automatically by the transformation remains open for future work.

The transformation process starts with transforming the `AbstractForm` element to a `ConcreteForm` element, whose `TargetDevice` attribute is set to *"Mobile"*. The sub elements are transformed into their concrete equivalents. Since for some of the abstract elements there are several options to represent them on the concrete layer, the mapping between Abstract Form Elements and Concrete Form Elements has to be defined. This mapping can be different for the transformations targeting different devices types because the used concrete elements have to fit to the features of the

respective device. Therefore, properties like the display size of the target device or the type of input device (keyboard, touch, etc.) is taken into account. Additionally, the values of the Abstract Form Elements' attributes have to be considered.

Table 3 shows the mapping between Abstract and Concrete Form Elements, applied by the Abstract to Mobile Concrete Form Model transformation.

Abstract Form Element	Concrete Form Element
AbstractDisplayElement	ConcreteDisplayElement
Description	Label
BooleanEdit	ToggleControl
TextEdit *(limited number of characters)*	TextBox
TextEdit *(unlimited number of characters)*	TextArea
NumericalEdit	NumericUpDown
Selector	DropDown
AbstractTestSubjectSelector	ConcreteTestSubjectSelector
AbstractAutomaticInput	ConcreteAutomaticInput
Command	Button

Table 3: Mapping between Abstract and Concrete Form Elements for the mobile platform

The separation of the Concrete Form Elements to several pages is done according to a predefined algorithm that is formulated in pseudo code in Listing 9. The algorithm is defined recursively. It runs through the structure of the Abstract Form Model, starting with the direct child elements of the `AbstractForm` root element. All elements prior to the first compound element are put into one page, named *"Page 1"*. Then the algorithm is called recursively for the child elements of this first compound element. Thus, the child elements of the compound element are put to a separate page, which is named according to the compound's name. Thereafter, the algorithm proceeds with handling the remaining child elements of the root element that come after the first compound in the same way. This results in generating a separate page for each group of consecutive elements that are not part of a `Compound` element. Additionally, for each compound, at least one page is generated. The reason, why not always just one page is generated per compound is that compounds can be nested. If this is the case, also the child elements of a compound are transformed into several pages according to the described method.

The functioning of the algorithm is illustrated by Figure 32. The transformation process inserts a linear layout with vertical orientation to each generated `Page` element. The transformed child elements are actually added to this root layout container on each page. Stacking the elements vertically is a typical design used for forms on mobile devices (compare [Nei12], p. 39ff).

```
1  call generatePage(rootElement.ChildElements, "Page")
2
3  method generatePages(elements, pageName)
4     set pageIndex to 1
5     foreach element in elements do
6        if element is Compound then
7           call generatePages(element.ChildElements, element.Name)
8           pageIndex++
9        else
10          transform element
11          if page with name <pageName + pageIndex> not exists then
12             generate page with name <pageName + pageIndex>
13          end
14          add transformed element to page with name <pageName + pageIndex>
15       end
16    end
17 end
```

Listing 9: Pseudo code for generating pages out of the Abstract Form Model structure

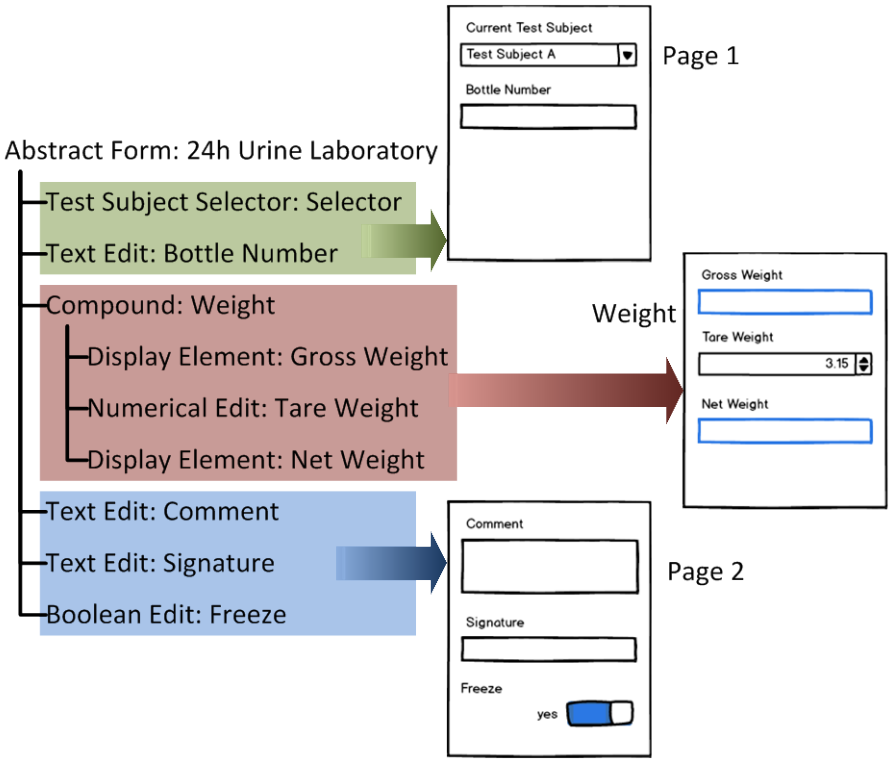


Figure 32: Dividing Concrete Form Elements to several pages

The larger space offered by the displays of tablet devices in comparison to most smartphones allows to put more elements into one page or even to place all elements of a data form on a single page. Thus, the layout structure within the pages becomes more complex on the Tablet Concrete Form Model. For building groups on that layout structure, the compounds of the Abstract Form Model can be utilized in a similar way as for building pages on the mCFM. For the evaluation of the introduced model-driven approach, a simple version of the Abstract to Tablet CFM transformation is used. It builds vertical linear layout containers like in the Mobile CFM and arranges them in a parent horizontal linear layout side by side in one page. Assuming that the tablet device is used in landscape mode, this is a trivial but applicable approach. A limitation of this solution is that a large number of groups results in small columns. To solve this problem, a further elaboration of the transformation to the Tablet Concrete Form Model has to be done which remains open for future work.

4.7 Final Form Implementation

The generation of a Final Form implementation out of one of the Concrete Form Models highly depends on the actual target platform. In the context of the :study project, and therefore also by this thesis, Windows based end user devices are focused. Therefore, this section illustrates how a Final Form implementation for the Windows Phone platform is gathered from the Mobile Concrete Form Model. A detailed description of the Final Form implementation requires background knowledge about the Windows Phone GUI framework, which is beyond the scope of this thesis. Therefore, the general method is described:

Basically, the implementation follows the MVVM design pattern (see Section 2.3.4). The UI- and business logic is implemented by the Windows Phone :studydata application. The actual UI definition is done using XAML. The XAML code is generated automatically by a transformation from the Mobile Concrete Form Model. This transformation builds up the Concrete Form Elements by standard GUI widgets that are available in the Windows Phone GUI framework. These widgets bind to the data model as well as features provided by the :studydata application. The data model provides the data defined in the Data Definition Model. The resulting XAML code for the different pages is integrated into a single XML document. This document as well as the data definition model are stored in the central :study database. Therefore, the database structure as well as the web service has to be extended in order to be able to handle several versions of the same data form, targeting different platforms.

To ensure that each page of the data form displays conjoint information as well as common form features in the same way and on the same place, the content of the single pages is inserted into a uniform page frame. This frame is implemented by the :studydata application and therefore not part of the data form's XAML code. Figure 33 illustrates this frame. It conforms to the Windows Phone design guidelines [Mic13a], which ensures that the data forms feel natural for users who are familiar with this platform.

At the top of the frame, the page header displays static information like the data form name and the name of the related activity pattern. Below this information, the code of the currently selected test subject is stated in big letters, such that the user always is aware to which study participant the currently entered data is linked. Additionally to the test subject code, the name of the current study day for that test subject and the name of the study are shown. At the bottom of the page frame, the standard Windows Phone *Application Bar* provides three common functions available for every data form. With the left one, the user closes the data form and navigates back to the :studydata application's main page. The latter two buttons allow the user to navigate in backward and forward direction through the data form's internal workflow. The area between the page header and the Application Bar contains the content of the different pages.

For loading a data form, the :studydata application invokes an appropriate method on the web service and passes the device type and platform of the device the data form is loaded from. For Windows Phone this is device type *"Mobile"* on platform *"Win-*

dows Phone". The web service returns the Data Definition Model of the queried data form as well as the Final Form implementation corresponding to the given parameters. The :studydata application uses the Data Definition Model to build up the data model, to which the view of the data form binds. Using the XAML code for the different data form pages, the application instantiates the view by inserting the pages' content into the common page frame. The page content area of the frame therefore uses a special UI element that manages several content pages and always displays just one of them. For moving from one page to the next, the user has to perform a swipe gesture. Figure 34 illustrates the first three pages of the 24h urine data form on Windows Phone.

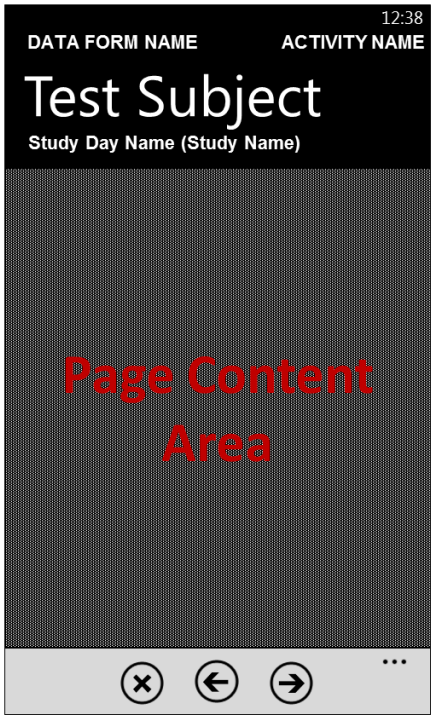


Figure 33: Common page frame of the Windows Phone :studydata application

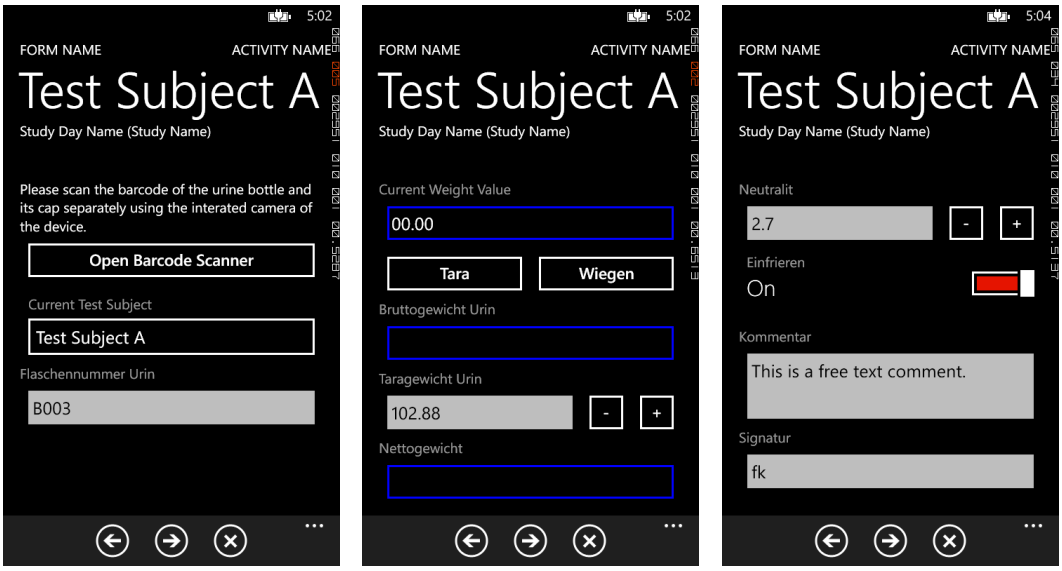


Figure 34: Example 24h urine laboratory data form on Windows Phone

5 Implementation

This chapter explains the prototypical implementation of the model-driven approach described in the previous chapter. In the context of the :study software, the resulting application prototype can be seen as an advanced version of the :studyforms application. To clearly distinguish the two versions of the :studyforms application, the new application prototype is denoted as model-driven :studyforms application.

5.1 Requirements

Before starting with the description of the implementation, this section explains the most important requirements for the prototype. This includes functional and non-functional requirements that arise from the project context and from the fact that the model-driven method is exploratory and is thus subject to constant changes.

- The software should allow developing multi-device data forms according to the method described in Chapter 4. The different abstraction layers should be clearly identifiable.
- Like the former version of :studyforms, that provided a simple and intuitive graphical editor, the model-driven version is intended to be used by expert scientists and supporting study staff. These are users who do not have an intuitive understanding how a model-driven approach works and what it is good for. At the first view, this results in a conflict between the developed method and a simple and intuitively usable application. But, the very limited target group mitigates the problem, because future users will be trained in using the tool. Nevertheless, the model-driven approach has to be encapsulated in such a way, that it is easily usable by the target user group.
- The development of the models during the design process should be supported by some kind of graphical editor. Additionally, for the Concrete Form Model layer a direct graphical editor, comparable to the former :studyforms application, should be available. This editor should give an impression how the data form will look like on the different target platforms.
- Since the described model-driven approach for designing multi-device data forms is still under exploration, changes to the defined meta-models should be easily adaptable to the prototype. Ideally, this can be done without recompiling the tool. This does not include major structural changes, but for example adding or removing attributes to the meta-models. More sophisticated changes should be easily added into a flexible application structure.
- For the same reason as the previous requirement, the transformations should be easily changeable. This allows to rapidly integrate improvements based on users' experiences and needs.
- The prototype focuses on Windows based target devices. Therefore the last transformation process, which results in the final user interface, should generate appropriate XAML code that can be loaded directly by the :studydata applications on Windows based target devices.

5.2 Implementation Concept

At the beginning of the development there were some thoughts about using some DSL tool support for defining the described meta-models and thereof automatically build a graphical editor that could be extended. In the .NET environment, such a tool support is provided by the *Visual Studio Visualization and Modeling SDK* (VSVMSDK)[15]. Unfortunately, this SDK only allows developing tool support as a Visual Studio plugin. This would require installing a Visual Studio distribution on every system that is used by the study team to design the needed data forms. Beside the fact that this would be an essential effort and cause licensing problems, the Visual Studio environment is assumed to be too complex for being used by the target user group. For these reasons, it has been decided to implement a standalone application, based on the *Windows Presentation Foundation* (WPF) framework. Whenever possible, state of the art technologies are used to implement the different aspects of the application.

As already mentioned in Chapter 4.1, the models on the different layers of the design process are stored using XML as a concrete textual syntax. The general idea of the :studyforms' implementation is to work directly on the XML documents of the different model layers. This does not mean that the user has to develop the models by writing XML in a text editor, or that changes to the models are done directly on the XML documents' files. The models' XML documents are loaded, but the :studyforms application does not contain a dedicated class structure that contains a class for every element of the meta-models, to which the XML documents could be parsed. Instead, the application internally works on the .NET framework XML classes, which reflects the *Document Object Model* (DOM).

To ensure that the loaded XML documents are consistent to the defined meta-models and that changes, which are done by the designer, do not break this consistency, the XML documents have to be validated towards the meta-models. For this purpose, the meta-models are implemented as *XML Schema Definitions* (XSD) (see Section 5.4). Since the capability of validating an XML document against an XML Schema Definition is part of the .NET Framework, this is an easy way to ensure that the models' XML documents are compliant to their meta-models. In addition, the XML Schema Definitions can be loaded easily at runtime. This allows reacting to changes on the meta-models by just replacing the XSD documents. Changes to the source code of the prototype are not needed.

In addition to the validation purpose, the XSD files are used for gathering information about the attributes of the model elements. This information is used to provide the available properties of each model element to the designer. Depending on the attributes' type that is specified in the XML Schema Definition it is possible to offer dedicated editors or dialogs for setting the values of the attributes. In addition to the XSD files, the :studyforms application obtains information from XML based *Model Ele-*

[15] The VSVMSDK allows building graphical or form-based Domain Specific Language designers for Visual Studio. For more information about this SDK see [Mic12a].

ment Definition (MED) files. These files are maintained additionally to the XML Schema Definitions and contain information about the elements and their attributes that is not extractable from the XSD meta-model but which is needed by the :studyforms application. Such a file exists for each of the three model layers of the design process. The MED files define which model elements the designer is allowed to add to the models manually and to which values the attributes of newly added elements are initialized. Furthermore, for each of the available elements and their attributes a short descriptive text is given. These descriptions are displayed on the User Interface of the :studyforms application to support the designer in developing a data form.

Gathering the needed information from the XSD and MED files makes the implementation of the :studyforms application flexible to changes on the meta-models of the design layers. New attributes can simply be added to the definition files and are then available in the application. This generic approach is limited to changes that do not force to implement additional editors for setting the values of attributes of a special type. Also major structural changes to the meta-models are not covered by this approach.

For the implementation of the transformations from one model layer to the next, it has been decided to use XSL Transformations (see Section 5.5). The advantage of this solution is again the possibility to load XSLT-Stylesheets easily at runtime before performing the transformation. This allows to make changes on the transformations' implementations without the need to rebuild the :studyforms application. Furthermore, XSL Transformations are a state of the art approach for transforming a source XML document into a target XML document.

5.3 Graphical User Interface

One of the essential challenges when developing the prototype was to find an appropriate *Graphical User Interface* (GUI) that allows the designer to build up the models on the different layers and, at the same time is easy and intuitive to use for people that are not familiar with model driven approaches. A starting point for the development was the already existing :studyforms application, that allows the designer to directly work with the data form elements (compare Figure 2). Due to the model-driven approach the realization of such a *"what you see is what you get"* (WYSIWYG) editor is not possible because the elements of the Data Definition Model and the Abstract Form Model are not directly graphically represented on the resulting data form. Nevertheless, also for these two model layers, the application should provide some kind of graphical editor.

Since on all of the described model layers, a model with a tree like structure has to be defined by the designer, it has been decided to represent the models by a tree view. On the Concrete Form Model layer, the designer can switch between this tree view editor and an editor showing a graphical visualization of the data form. This is possible because the elements defined on this layer have a fixed graphical appearance and the arrangement of the elements on the data form is defined by the layout attributes.

However, the graphical editor can only give an impression of the data form's graphical user interface on each of the target device types. Since the design of the Concrete Form Elements differs for the various target platforms, the real appearance on the device might be slightly different. Dissimilarities can additionally be caused by different pixel densities on the target devices or by platform guidelines. For expert users, there is also the option to work directly on the XML document. Therefore, on each model layer a text editor showing the model's XML code is available. In the prototype, this text editor only allows to view the XML of the models. Changes in this editor have no effect.

Figure 35 shows the main window of the model-driven :studyforms application. The red numbers denote the most important parts of the GUI. They are briefly described in the following paragraphs:

1. Menu Bar: The menu bar offers standard features for opening or saving data forms or copying and pasting single elements on the models. Furthermore, the *Transform* menu enables the designer to trigger the transformation processes between the model layers. The *Master Data* menu provides access to master data like activity patterns and parameters stored in the :study database.

2. Tool Bar: The tool bar provides often used features without navigating through the menu structure.

3. Editor Area: The largest area of the :studyform's main window shows the editor for developing the models on the three design layers. The current model can be selected on the tab bar at the top of the editor area (a). Via the tab bar at the bottom (b) the designer can switch between the available editors on each model layer. On the screenshot, the selected model is the Data Definition Model. Thus, the graphical editor is not accessible because it is only available for a Concrete Form Model.

4. Toolbox: The toolbox shows the available model elements that can be added to the models using drag and drop. The content of the toolbox depends on the currently selected model. The available elements are grouped in categories. When dragging the mouse over one of the elements, a tooltip with a short description of the element appears.

5. Properties: The properties area shows the attributes of the model element currently selected in the editor. The designer can change the values of the attributes using input fields or separate dialogs that are appropriate for the attribute's type. At the bottom of the property grid, a brief description of the available attributes is given (c). This makes the usage of the property grid more intuitive for the designer.

6. Validation Errors: The validation errors area shows errors indicating mismatches between the models and their meta-models. For the prototype, this area is of special importance because it gives hints to implementation errors. At the final version of the :studyforms application, this area might be dispensable, because invalid changes to the models should be prevented by the application.

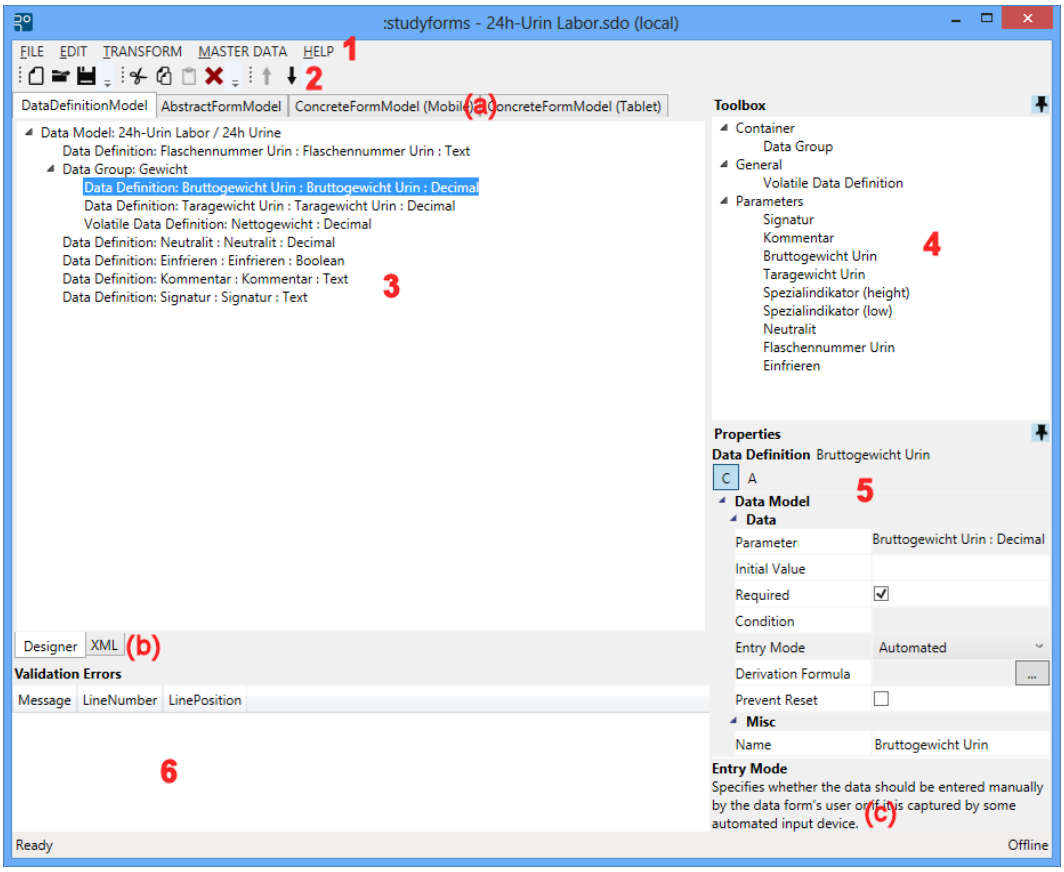


Figure 35: Main window of the model-driven :studyforms prototype

5.4 Model Validation

To make sure, that the generated models conform to the abstract syntax defined by their meta-models, the XML model documents have to be validated when they are loaded by the :studyforms application or when the designer performs changes to the models.

This validation process could have been hard coded into the prototype, but this would be hard to maintain in case of changes on the meta-models. Since this is exactly one of the requirements mentioned in Section 5.1 another solution has been found: The meta-models are transformed into *XML Schema Definitions* (XSD) (see Section 2.3.1). The defined schema definitions are then used to validate the model's XML documents and are loaded dynamically by the prototype. This means that no changes to the validation source code of the :studyforms application are needed in order to react to changes on the meta-models.

To generate the XML schema definitions automatically out of the meta-models, the meta-models have first been defined as UML class diagrams. For this purpose, the Visual Studio 2012 Ultimate[16] edition provides a separate project type, the *Modeling*

[16] Visual Studio 2012 is the currently up to date IDE from Microsoft for developing applications for the Windows platforms.

projects. This project type allows developing different kind of UML models, one of which is UML class diagrams. The resulting class diagrams of the meta-models of the three abstraction layers have already been shown in the figures in Chapter 4.2. The advantage of using the Visual Studio integrated modeling capabilities is that the generated class models can be accessed by own code using the *UML Modeling Extensibility API* (see [Mica]). Since all model elements are stored in a common model store, the meta-models are designed in different packages. This eases the generation of dedicated XSD files for each meta-model. In the following, the transformation from the meta-models to corresponding XSD documents is explained. Thereafter, the implementation of the automatic generation of XSD files is shortly described. The given examples are taken from the Concrete Form Model.

Every class of the meta-model is transformed into an XSD complex type. If the class is denoted as "abstract", the generated complex type is also set to be abstract. As a consequence, the complex type denoted as abstract cannot be used as an element in an XML document that conforms to that schema definition. The generalization relations are realized by defining a complex type that extends the complex type generated from the base class. Here, it is important to mention that this extension mechanism does not have to be understood like an inheritance hierarchy. In addition to its own attribute and element definitions, an extended complex type contains all attribute and element definitions of its base type. But there is no possibility to cast the extended type to the base type. Both complex types are completely independent. Listing 10 shows the definition of the `ConcreteLayoutableElement` element that is set to be abstract and that extends the `ConcreteFormElement` complex type.

```
1  <xs:complexType name="ConcreteLayoutableElement" abstract="true">
2     <xs:complexContent>
3        <xs:extension base="ConcreteFormElement">
4           <xs:attribute name="PreferedHeight" type="Size" />
5           <xs:attribute name="PreferedWidth" type="Size" />
6           <xs:attribute name="Position" type="Positions" />
7        </xs:extension>
8     </xs:complexContent>
9  </xs:complexType>
```

Listing 10: XSD complex type `ConcreteLayoutableElement`

For each attribute of a class, an attribute specification is added to the complex type representing that class. The name and the data type of the attribute match the specifications of the attribute on the meta-model. The data type is mapped to the corresponding XSD simple type. Others than standard data types are defined manually in a separate XSD document that is included into the schema definitions. This is for example the case for the `Size` data type of the `PreferedWidth` and `PreferedHeight` attributes of the `ConcreteLayoutableElement`. As described in Section 4.2.3, possible values for these attributes are plain double values, double values followed by a *"*"* (Star) symbol or the string *"auto"*. Therefore the `Size` data type is defined as a simple type that is based on the `xs:string` simple type. The definition of that `Size` simple type is shown in Listing 11. The base type is restricted to only allow the above mentioned values using two regular expressions. The first one defines the double values, optionally with the star sign at the end (Listing 11, l. 3). The second one adds the possibility to set the value to *"auto"* (Listing 11, l. 4).

```
1  <xs:simpleType name="Size">
2     <xs:restriction base="xs:string">
3        <xs:pattern value="[0-9]*[.[0-9]*]?\*?" />
4        <xs:pattern value="auto" />
5     </xs:restriction>
6  </xs:simpleType>
```

Listing 11: XSD simple type definition of the `Size` data type

The enumerations defined in the meta-models are also transformed to XSD simple types. For these cases, the simple type definitions are based on the `xs:string` simple type, but the restriction is done by specifying the list of allowed values. These are equivalent to the enumeration's list of literals. Listing 12 shows the simple type definition of the `Positions` enumeration. The list of acceptable values is set using the `xs:enumeration` element. The definition of the allowed values is case sensitive, which means that the values of the `Position` attribute can only be set in the way they are specified in the simple type definition.

```
1  <xs:simpleType name="Positions">
2     <xs:restriction base="xs:string">
3        <xs:enumeration value="Top/Left" />
4        <xs:enumeration value="Center" />
5        <xs:enumeration value="Bottom/Right" />
6     </xs:restriction>
7  </xs:simpleType>
```

Listing 12: XSD simple type definition of the `Positions` enumeration

Another important part of the meta-models that have to be represented in the XML documents are the aggregation relations between the model elements. An example for this is the aggregation between a layout container and its child elements on the Concrete Form Model. These relations are represented in the XML document in form of sub elements of another element. To express such sub elements in an XSD, an element definition is added to the parent complex type definition. For generating the XML Schema Definition of the defined meta-models, it has been decided that the parent element of an association is the element from which the association is navigable. This expects that each association in the meta-models is navigable only in one direction. The direction is depicted by an arrow at the end of the association in the class diagrams. In case of the above mentioned example, its child elements are navigable from the layout container. In the XML document this means that the children are sub elements of the layout container element. As already described above, XSD does not provide a real inheritance mechanism for complex types. Thus, it is not sufficient to define a single sub element with the type set to the abstract base type of the allowed sub elements because the concrete child elements do not match this type. When validating the XML according to its schema definition, this would cause a validation error. Instead, for each possible sub element an element definition is added as a sub element to the complex type definition. To express that the child elements can occur in an arbitrary order, the element definitions are part of an `xs:choice` definition. The multiplicity of the aggregation's child end is taken to set the minimal and maximal occurrence of the sub elements. Listing 13 shows a shortened version of the complex type definition of the `ConcreteLayout` element.

```
 1  <xs:complexType name="ConcreteLayout" abstract="true">
 2     <xs:complexContent>
 3        <xs:extension base="ConcreteLayoutableElement">
 4           <xs:choice minOccurs="0" maxOccurs="unbounded">
 5              <xs:element name="LinearLayout" type="LinearLayout" />
 6              <xs:element name="TextBox" type="TextBox" />
 7              …
 8           </xs:choice>
 9        </xs:extension>
10     </xs:complexContent>
11  </xs:complexType>
```

Listing 13: XSD complex type definition of the `ConcreteLayout` element

The last aspect that needs to be defined in a schema definition is the root element of the XML tree structure. To indicate the root element of the meta-model in the class diagram, a UML profile is introduced, that defines the stereotype[17] *<<XSD root element>>*. This stereotype is applied to the class that builds the root element of the model. In the Concrete Form Model, this is the `ConcreteForm` element (compare Figure 12). To express the root element in the schema definition, an element definition is added at the document level of the XSD. The type of this definition is set to the name of the corresponding complex type.

The actual generation of XSD files from the defined class diagrams is done using T4 text templates (see [Micb]). T4 stands for *Text Template Transformation Toolkit*, which is a template engine integrated into Visual Studio. The result of a T4 transformation is an arbitrary text file (for example an XML or a source code file). The template contains static text parts that are directly inserted into the target file and control blocks, which are implemented in C# or Visual Basic and are used for implementing logic for inserting dynamic content into the generated file. Such a text template allows implementing a *Model to Text* (M2T) transformation (compare Section 2.2.2) that generates XML Schema Definition files according to the defined class diagrams of each model layer. The application of the transformations can be included into the build process, which ensures that the :studyforms application always works with the latest meta-model definitions.

For each of the four defined meta-models, one T4 text template is generated. In these template files, first the package containing the meta-model, for which the template generates the schema definition, is loaded. Listing 14 illustrates this for the package `ConcreteFormModel` that contains the corresponding meta-model. Since the depicted code should not be just copied to the output file of the transformation, but instead be evaluated by the template engine, it is marked as a control block by the surrounding <# … #> symbols.

[17] A stereotype is an extension to existing UML model elements that is used to further specify the purpose of an element.

```
 1  <#
 2     // Get modeling project.
 3     string projectPath =
 4        this.Host.ResolvePath(@"..\..\Models\Models.modelproj");
 5     IModelingProjectReader project =
 6        ModelingProject.LoadReadOnly(projectPath);
 7     // Get package from model store.
 8     IPackage package = project.Store.Root.NestedPackages.
 9        SingleOrDefault(p => p.Name == "ConcreteFormModel");
10  #>
```

Listing 14: Loading a meta-model package from the modeling project

A part of the actual template definition is shown in Listing 15. The `xs:schema` and the `xs:include` tags are copied directly into the output file of the transformation. The `xs:include` tag includes a further XSD file that contains the manually maintained simple types, like the type `Size` (see above). Between the opening and closing `xs:schema` tags a further control block is defined. Inside this block it is iterated over the elements, defined in the package and an appropriate template is applied to each of these elements (see Listing 15, l. 5-8). Thereby, the XML Schema Definition file is generated iteratively when the text template is executed.

```
 1  <?xml version="1.0" encoding="utf-8"?>
 2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 3     <xs:include schemaLocation="DataTypes.xsd"/>
 4  <#+
 5        foreach (IType t in package.OwnedTypes)
 6        {
 7           …
 8        }
 9  #>
10  </xs:schema>
```

Listing 15: T4 text template part for generating the XML schema definition

5.5 Transformations

The transformation processes that guide the designer through the model-driven design process and automatically generate a starting model on the always next model layer are implemented using the XSLT (*Extensible Stylesheet Language Transformation*) technology (see Chapter 2.3.2). The .NET framework supports performing XSL Transformations natively. The framework therefore includes the class `XslCompiledTransform` that provides methods for loading an XSL-Stylesheet and for performing the transformation. This class can be seen as a .NET integrated XSLT processor.

As described in Chapter 2.3.2 an XSLT document defines templates that are applied to specific elements of the source XML document. Which template is applied for which element is specified by the pattern attribute of each template. The basic concepts of the transformation processes are described in Chapter 4.6. Following, the implementation of these transformations using XSLT is illustrated at the example of the transformation from the Data Definition Model to the Abstract Form Model. Therefore, selected template examples are shown and described. Additionally, some challenges that occurred during the implementation are explained. A detailed explanation of the implemented XSLT-Stylesheets is beyond the scope of this thesis, but the shown examples allow imagining how the transformations work.

As the transformation from the Data Definition Model to the Abstract Form Model preserves the structure of the source model, the templates applied to the root element `DataModel` and to the `DataGroup` element have to transform those elements themselves, and insert the Abstract Form Elements resulting from the transformation of their children as sub elements. The XSL template that matches the `DataModel` element is shown in Listing 16. The template inserts the `AbstractForm` element to the target model. The name of the `AbstractForm` element is set to the name of the `DataModel` element of the source model. As described in Section 4.6.1, the first child element, added to the `AbstractForm` definition, is an `AbstractTest-SubjectSelector` element. Further child elements are added by applying templates that match the child elements of the `DataModel` element in the source model. At the end of the template, the *"Finishing"* compound is added. This `Compound` element contains the `Command` elements for submitting and resetting the data form.

```
 1 <xsl:template match="DataModel">
 2    <AbstractForm Name="{@Name}">
 3       <AbstractTestSubjectSelector Name="TestSubjectSelector" />
 4       <xsl:apply-templates />
 5       <AbstractCompound Name="Finishing">
 6          <Command Name="Command_Reset" Caption="Reset"
 7                   ActionReference="…::Reset" />
 8          <Command Name="Command_Submit" Caption="Submit"
 9                   ActionReference="…::Commit" />
10       </AbstractCompound>
11    </AbstractForm>
12 </xsl:template>
```

Listing 16: XSL template matching the `DataModel` element

Beside the two grouping elements `DataModel` and `DataGroup`, the Data Definition Model contains the actual data definition elements `DataDefinition` and `VolatileDataDefinition`. The XSL templates that transform these data definition elements to appropriate Abstract Form Elements do not only consider the name of the source element in their match statement, but also the attributes `DataType`, `EntryMode`, and `DerivationFormula`. According to the decision tree in Section 4.6.1 (see Figure 31) there are five different Abstract Form Elements, to which the transformation process potentially transforms the data definition elements. Since the `DataDefinition` and `VolatileDataDefinition` elements are treated separately, this sums up to ten templates. Listing 17 shows the template that matches `DataDefinition` elements, whose data type is *"Integer"*, whose `EntryMode` attribute is not set to *"Automatic"* and whose `DerivationFormula` attribute is empty. For such data definitions, the template inserts a `NumericalEdit` Abstract Form Element. The reference to the source element in the Data Definition Model is set by adding the `DataReference` attribute using the `xsl:attribute` element. The path to the source data definition element is inserted by directly calling an additional template that builds up the path and inserts it into the target model.

```
 1 <xsl:template match="DataDefinition[param:GetDataType(@ParameterId)=
 2                     'Integer' and @EntryMode!='Automated' and
 3                     @DerivationFormula='']">
 4    <NumericalEdit Name="NumericalEdit_{@ParameterId}"
 5                   Caption="{param:GetDataName(@ParameterId)}">
 6       <xsl:attribute name="DataReference">
 7          <xsl:call-template name="getPath" />
 8       </xsl:attribute>
 9    </NumericalEdit>
10 </xsl:template>
```

Listing 17: XSL template for transforming a `DataDefinition` element to a `NumericalEdit` element

In contrast, Listing 18 illustrates the XSL template that transforms a `DataDefinition` element to an `AbstractDisplayElement`. This template does not consider the data type of the data definition and the template matches data definitions whose `EntryMode` attribute is set to *"Automated"* or for which a derivation formula is defined.

```
 1 <xsl:template match="DataDefinition[@EntryMode='Automated' or
 2                     @DerivationFormula!='']">
 3    <AbstractDisplayElement Name="DisplayElement_{@ParameterId}"
 4                      Caption="{param:GetDataName(@ParameterId)}">
 5       <xsl:attribute name="DataReference">
 6          <xsl:call-template name="getPath" />
 7       </xsl:attribute>
 8    </AbstractDisplayElement>
 9 </xsl:template>
```

Listing 18: XSL template for transforming a `DataDefinition` element to an `AbstractDisplayElement`

One of the crucial parts of the transformation from the Data Definition Model to the Abstract Form Model is the fact that the `DataDefinition` elements do not directly contain information about the data type of the data definition. Instead, these elements hold a reference to a parameter in the :study database that provides the needed information. Unfortunately, XSL does neither support querying a database directly, nor working with a web service. This means that it is not possible to gather the information about the data type of a `DataDefinition` element directly in the XSL Transformation. In order to nevertheless access the needed information, the extensibility property of XSLT is used. XSL allows including additional instructions that are provided by the XSLT processor. The .NET integrated XSLT processor therefore provides the possibility to pass *XSLT Extension Objects* (see [Mic12b]) to the XSLT-Stylesheet. Such an XSLT Extension Object is an arbitrary .NET class that contains methods, which are available from the XSLT-Stylesheet. To access the data type and the name of a parameter from the :study database, the class `ParameterConverter` is implemented. This class provides public methods that accept the unique identifier of a parameter, and returns the requested information after querying the :study database. Here, it is important to mention that this kind of extensions to the core features of XSLT is no .NET specific opportunity. Extensions are part of the W3C recommendation of XSLT (see [W3C99], Chapter 14).

Listing 19 shows a simplified source code excerpt of the :studyforms application that initiates the transformation from the Data Definition Model to the Abstract Form Model. After generating a new `XslCompiledTransform` object and loading the

appropriate XSLT-Stylesheet (Listing 19, l. 1-2), an instance of the `ParameterConverter` class is passed to the transformation process. This is done by adding the instance to the `XsltArgumentList` class. This argument list is passed to the transformation (Listing 19, l. 4-6). Thereafter, the transformation is performed.

```
1  XslCompiledTransform xslt = new XslCompiledTransform(true);
2  xslt.Load("DataDefinitionToAbstractTransformation.xslt");
3
4  XsltArgumentList xslArg = new XsltArgumentList();
5  ParameterConverter paramConverter = new ParameterConverter();
6  xslArg.AddExtensionObject("urn:parameter-conv", paramConverter);
7
8  xslt.Transform(DataDefinitionModelXmlDocument, resultDocument);
```

Listing 19: Source code excerpt calling the DDM to AFM transformation

The transformations from the Abstract Form Model to the Concrete Form Models targeting different device types work in a similar way as the described transformation. The XSLT-Stylesheets contain a template for each of the Concrete Form Elements that are potentially generated by the transformation process. The division of the elements to several pages is done by an XSL implementation of the algorithm described in Section 4.6.2 (see Listing 9).

Another difficulty that had to be solved arose from the implementation of the transformation of the Mobile Concrete Form Model to the Windows Phone Final Form. The problem is that this transformation needs to consider the Mobile Concrete Form Model as well as the Abstract Form Model as source model, since the mCFM does not contain all needed information. The Concrete Form Elements instead reference Abstract Form Elements contained in the AFM. To enable the transformation to access the information from the Abstract Form Model, the AFM is given as a parameter to the transformation. XSL therefore allows declaring global parameters that can be set when the transformation is called. Setting the value of a global parameter is done similar to passing an Extension Object to the transformation (see above). Listing 20 shows a shortened version of the call of the Mobile Concrete Form Model to Windows Phone Final Form transformation. Passing the Abstract Form Model as a parameter to the transformation takes place in line 7.

```
1  XslCompiledTransform xslt = new XslCompiledTransform(true);
2  xslt.Load("XslTransformations/MobileToWPTransformation.xslt");
3
4  XsltArgumentList xslArg = new XsltArgumentList();
5  DynamicExtension dynExtension = new DynamicExtension();
6  xslArg.AddExtensionObject("urn:dyn-ext", dynExtension);
7  xslArg.AddParam("abstractModel", AbstractFormModelXmlDocument);
8
9  xslt.Transform(MobileConcreteFormModelXmlDocument, resultDocument);
```

Listing 20: Source code excerpt calling the mCFM to Windows Phone transformation

For accessing attribute values of the referenced Abstract Form Elements, an XPath expression is built up from the path given by the `AbstractReference` attribute of a Concrete Form Element. This is done dynamically during the execution of the transformation. Since the XSLT version supported by the .NET framework does not allow evaluating such dynamically generated XPath queries, this is done using an extension. Therefore, the class `DynamicExtension` is implemented that contains just one method `Evaluate`. This method takes the Abstract Form Model's XML and the

XPath query that should be evaluated on the given XML. The result of the query is returned by the method. An instance of that class is given as an Extension Object to the XSLT processor before executing the transformation (see Listing 20, l. 4-6). The `Evaluate` method is then used by the XSLT-Stylesheet to access attribute values on the Abstract Form Model by the dynamically generated XPath queries.

6 Evaluation

For evaluating the introduced model-driven data form design approach as well as the developed :studyforms prototype, a usability study has been conducted. Here it has to be mentioned that due to the limited time, some parts of the prototype are implemented as a mockup without any functionality. This has been done to be able to include important parts of the model-driven concept into the task the study participants have to fulfill. Thus, in the context of this evaluation, the term "usability" does not focus on the Graphical User Interface of the prototype but on the general applicability of the model-driven approach for the future target group and on the way the :studyforms prototype implements this approach. Therefore, the study concentrates on the users' satisfaction and on finding general drawbacks or advantages.

The study is conducted as a formative study. This type of usability study is also called explorative ([RC08], p. 29) or inductive ([SB11], p. 163) study. According to Sarodnick and Brau, a formative study is suitable for the given conditions because it focuses on the analysis of prototypes and tries to find drawbacks and possible improvements ([SB11], p. 163). Thus, it does not compare different alternative systems but tests just one prototype.

The following sections first explain the considered usability attributes, the setup of the user study as well as the task, the participants have to solve. Thereafter, the results of the user study are presented and discussed.

6.1 Considered Usability Attributes

For the usability evaluation of the model-driven data form design process using the :studyforms prototype four usability aspects are examined by the study. These aspects are mainly selected from the five usability attributes defined by Nielsen ([Nie93], p. 26ff). Following, the investigated usability attributes are introduced. Furthermore, it is expressed which metrics are used to make a point about the different usability attributes. *"A metric is a way of measuring or evaluating a particular phenomenon or thing."* ([TA08], p. 7) The user study is designed in such a way that a statement to each of the four aspects can be made.

Satisfaction

In the context of this user study, satisfaction is treated from two perspectives: On the one hand, the users' satisfaction with the model-driven data form design process using the :studyforms prototype is considered. On the other hand, the satisfaction with the resulting data forms is of interest. Therefore, the users are shown their development results on a Windows Phone smartphone. The two types of satisfaction are measured by three questions for each of the two types on a questionnaire.

Learnability

The learnability attribute gives information about how easy the usage of a system can be learned by its users. Since the :studyforms application is not intended to be used

intuitively and the number of future users is limited, this usability attribute is considered to be less important. It is measured by three questions on a questionnaire filled out by the study participants.

Efficiency

Efficiency is *"the amount of effort, required to complete the task"* ([TA08], p. 8). In the context of this user study, the efficiency is measured by the *Task Completion Time* (TCT) which is the time needed by the test users to complete the task. The TCT starts with generating a new data form and ends with generating the Final Form for the target platform given in the task description. Thus, reading the task is not included in the TCT.

Effectiveness

The effectiveness concerns the ability of a user to complete the given task ([TA08], p. 8). Although effectiveness is not one of the five usability attributes mentioned by Nielsen it is considered to be an important aspect for the evaluation of the results of the user study. In the context of this user study, the participants get one global task that is formulated in an abstract and superficial way and targets several aspects of the tested approach. Therefore it is possible that the study participants do not complete the task with all its aspects but partially. In contrast to this, Nielsen defines test tasks more granularly and dedicated to specific aspects (compare [Nie93], p. 185f) such that they are solvable completely or not at all. The effectiveness allows comparing the results of the test users even if they do not fulfill the task completely. Therefore, a list of 20 steps that are considered to be the most important for completing the task was elaborated prior to the study. According to these steps, a percentage is calculated that indicates how much of the task a user was able to do. This value is used as the metric for measuring the effectiveness in this study.

6.2 Setup of Usability Study

In this section the general setup of the user study is explained. This includes the selection of study participants as well as a description of the study procedure and the used questionnaires.

Since the user study is done in the course of this master's thesis, the study is conducted by just one supervisor, who is familiar with the prototype. This also prevents distractions by other people and the participants do not feel observed by too many eyes. The supervisor and the author of the thesis are the same person.

All data capturing in the context of the study was done pseudonymized and with permission of the test users. Since all participants were German native speakers, all material handed out to the test users was written in German language to avoid confusion or misunderstandings caused by the language.

6.2.1 Selection of Study Participants

For the study participants, future users of the :study software were obtained from the DLR *Institute of Aerospace Medicine*. These users are the target group of the :studyforms application and can be expected to have sufficient background knowledge about typical medical studies performed by this institute. At the same time, these users are not expected to be familiar with model-driven approaches like they often are used in computer science. To compare the performance of these users, a second group of participants is obtained from the facility for *Simulation and Software Technology*. These users are expected to have no experience in executing medical studies, but feel familiar with abstracting things using model-based approaches. Following, the first group of participants is denoted as domain experts, the latter one as computer science professionals or reference group. The comparison of the two user groups allows distinguishing between general drawbacks of the model-driven approach and problems that are caused by misunderstandings or inexperience of the domain experts. This simplifies to classify the severity of observed problems.

Altogether, eight users participated in the study, four on each group. The low number of test users is caused by the small amount of available domain experts and the limited time for this thesis. The study focuses on finding general problems of the model-driven approach and its implementation for the future users. Tullis and Albert state, that *"five participants per significantly different class of users is usually enough to uncover the most important usability issues"* ([TA08], p. 119). The authors further claim, that *"In most of the usability tests we're [sic] [the authors] conducted over the years, regardless of the total number of test participants, we're [sic] seen most of the significant issues after the first four or five participants."* Based on these statements, the number of eight test users in total is assumed to be sufficient for this study.

6.2.2 Study Procedure

According to Sarodnick and Brau an appropriate test atmosphere is very important to reduce stress for the participants ([SB11], p. 240). Therefore, at the beginning of each test session, the test procedure is explained to the participant. Furthermore, the goal of the study is specified and it is clearly stated that it is the system which is under test, not the user. Since a formative user study is done using a prototypical implementation, it is also important to state that the later application can look differently, and that some common features like an undo redo function are not yet implemented.

During the introduction, the participant is asked to fill in a questionnaire about his personal background and foreknowledge that might influence the performance of the user with the software. The details of the questionnaire are given in Section 6.2.4.

Thereafter, the study participant is requested to read an information sheet about the :studyforms application and the underlying model-driven data form development process (see Appendix C). The information describes the general model-driven approach and which parts of the data form are specified at which model layer. Since the participants are not necessarily familiar with building models in general on one hand or with

the domain of medical studies on the other hand, questions of the participants are answered during reading the introduction.

The :studyforms application is not intended to be used by totally novice users. Instead, the future users will be trained before using the software. To also take this training phase into account, after the theoretical introduction by reading the text, the participant also get a practical insight into the :studyforms prototype. This is done by means of showing how to develop a simple data form for initially capturing 24h urine samples (compare Section 2.1.3). The demo includes the major parts of the model-driven design process and already shows how some of the parts of the user task (given in Section 6.2.5) can be solved. But, not all aspects needed to fully solve the task are shown. Again, during this demo the participant is allowed to ask questions.

After the demo, the task description is given to the participant and he is asked to start working. To get a better impression of what the user thinks during performing the task the "think aloud" method is used. Therefore, before the user starts working, he is requested to express his thoughts, confusion, frustration and even delights loudly (compare [RC08], p. 204). The supervisor observes the progress of the user and takes notes.

When the participant finishes the task, he is shown the resulting data form of his developing on a Windows Phone device. Since the prototypical implementations of the :studyforms and the Windows Phone :studydata applications do not yet have a database integration, the transfer of the data form to the smartphone device is done by the supervisor. The user is then asked to explore the resulting data form. The :studydata application prototype allows to view the data form's graphical user interface. Missing business logic in the prototype is explained by the supervisor.

The study participant is finally asked to fill in a questionnaire regarding his satisfaction with the model-driven data form design approach and the resulting data form. At the end, in most of the study sessions a short discussion took place. The supervisor asked open questions and the participant had the opportunity to make additional comments.

6.2.3 Technical Infrastructure

The demo during the introduction phase of a test session and the performance of the user task was done on a Windows 8 based PC with standard office hardware features. At this system, the :studyforms prototype is installed. At the same machine, the used development environment is installed. This is needed for transferring the developed data form to the mobile device and running it for demo purposes at the end of each test session.

To ease the evaluation of the "think aloud" study and to be able to retrospectively hear again the thoughts of the users, the voice of the user and the screen content of the machine were captured during the task performance. Furthermore, a logging functionality is implemented to the :studyforms prototype. Thus, during performing the task by the user, important events like starting to develop a new data form or performing a transformation from one model layer to the next, are logged. The log also in-

cludes timestamps for each event which allows calculating the Task Completion Time.

6.2.4 Questionnaires

During a study session, the test users filled in two questionnaires. The first one targets the personal background (age, gender, profession) of the participants as well as possible foreknowledge that is important for the study. This concerns the general experience with computer systems and with software for generating electronic data forms as well as knowledge in the area of *Model Driven Software Development* (MDSD) and in the execution of medical studies at DLR. The user expresses his experience in these areas on a 7 step Likert scale. The questionnaire about the test users' background is attached in Appendix F.

The second questionnaire is divided into four areas. It is given in Appendix G. A translation of the questions is given when presenting and discussing the results in Section 6.3. Two areas cover questions about the user's satisfaction. One of them focuses on the :studyforms prototype and the model-driven approach in general, the second one targets the result of the design process. A third area contains questions regarding the learnability of the :studyforms application and the model-driven approach. The asked questions are inspired by standard usability questionnaires and adapted to the given context (see [Lew95], [WHG97]). All questions are formulated as a statement. The participant answers by expressing his affirmation to the statement on a Likert scale from 1 (strongly agree) to 7 (strongly disagree). In the fourth area, the participant has the chance to give additional free text comments, not restricted to the other areas of the questionnaire.

6.2.5 User Task

The participants had to accomplish one single task. Since the goal of the user study is to find out whether the future users are able to use the model-driven data form design process for their needs, the granularity of the task description is rather crudely. This means, it does not specify the single steps, the user has to perform in order to fulfill the task. Instead, it describes what the user wants to achieve and just gives hints to the model elements or attributes he has to use. The task is selected in such a way that it describes a realistic data form which is not too complex for the context of the user study. The participants were free in naming the form elements and grouping them together. All the needed master data had already been added to the :study database prior to the study. The task description as it was handed out to the study participants is attached in Appendix D. Following, it is summarized in English language:

The user should develop a data form for capturing data of 24h urine samples in the laboratory after they were initially captured at the study facility (compare Section 2.1.3). The data form should later run on smartphones and tablets. Table 4 defines the parameters that should be captured by the data form and stored to the central :study database.

Parameter	Data Type
Bottle Number	String (max. 6 characters)
Gross Weight (of the sample)	Decimal Number
Tare Weight (of the sample)	Decimal Number
pH-Value	Decimal Number
Freeze (Indicates, that the sample should be frozen for later processing)	Boolean Value
Comment	String (unlimited)
Signature	String (max. 3 characters)

Table 4: Parameters of the data form that is developed by the participants of the user study

The bottle number and tare weight should be captured in an automated way by a barcode scanner. Via this scanner, the current test subject is selected automatically. Since the barcode could be damaged, also a manual entry of these data should be possible.

The sample's gross weight should be captured by an electronic balance. Here, no option for entering the value manually should be available. Two buttons should enable the user of the data form to take the current weight value and to tare the balance.

Additionally, the data form should contain a button for submitting the values to the database and to reset all input fields on the data form.

6.3 Presentation and Discussion of Results

In this section, the results of the user study are presented and discussed. In total eight test users participated in the study. The evaluation of the questionnaire asking about background information yields, that in average the participants were 36 years old, the youngest was 27 years old and the oldest person had an age of 49 years. Two of the test users were female, the other six participants were male. The answers of the participants concerning their profession and the DLR institute or department they work on, allows identifying four computer science professionals and four domain experts. This grouping is confirmed by the answers about the experience in medical studies at DLR. All computer science professionals have no experience at all in this area, whereas the domain experts have medium to high experience. The experience in MDSD is contrary. Only one computer science professional stated like all domain experts to have no experience in MDSD. The general experience about computer systems is high (all participants answered with 5 or better). This seems to be not comparable between the two user groups because the participants evaluated themselves based on their understanding of that field. Nevertheless, it shows that no novices in working with computers participated. The answers concerning the experience with other software for generating electronic data forms are wide spread and do not allow to give any tendency for the two user groups.

6.3.1 Usability Attributes

Following, the study results are presented and discussed in the context of the considered usability attributes (see Section 6.1). Here it has to be mentioned that the person who evaluated the results is the same as the supervisor during the study. Although this might bias the results, it is assumed to be sufficient in that early phase of development. A comparison of the target and reference user group is just done if the results show identifiable differences between these two groups. The answers from the second questionnaire are presented on bar charts using a Likert scale from one to seven, like it is used on the questionnaire. The bar charts do not distinguish between the computer science professionals and the domain experts. Additionally, the mean and the standard deviation were calculated.

Satisfaction

The first three questions of the questionnaire are about the user's satisfaction concerning the convenience with the developed solution for designing multi-device data forms. Two questions are regarding the handling of the developed :studyforms prototype with the underlying model-based approach in general. The third one targets dedicatedly the benefits of the model-driven approach for designing data forms for multiple devices.

The two questions regarding the :studyforms prototype and the underlying approach in general are:

Q1: *The software allows generating data forms easily.*
Q2: *The software forces the user to perform unnecessary working steps.*

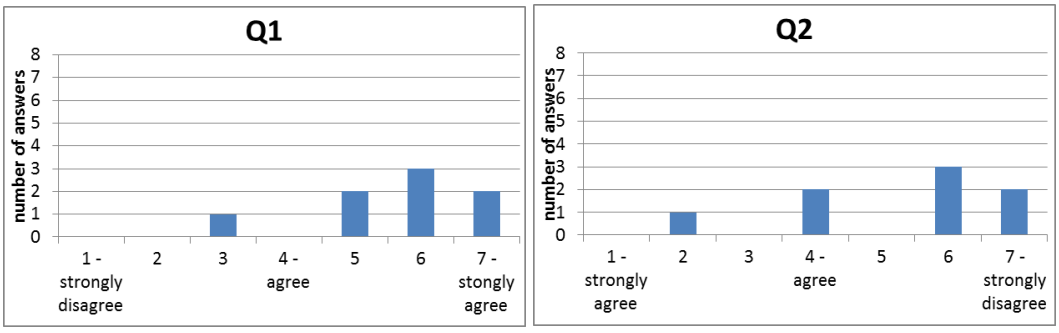


Figure 36: Answer values regarding the satisfaction with the :studyforms prototype in general

In average, the users answered to Q1 with M1=5.63 with a standard deviation of σ1=1.22. This allows reasoning that the users had no severe problems in using the prototype. However, the mean value for Q2 is M2=5.25 with σ2=1.64. Here, it has to be mentioned that the scale is inverted for question Q2, since the question is asked in such a way, that the stronger the users agree, the worse is the result. Although this mean value is still above the middle value, this is the worst result of all questions on the questionnaire. This shows that some users seem to feel uncomfortable with designing the data form in three different model layers. A reason for this might be that the prototype does not yet propagate all changes in higher model layers to the lower ones, if the transformation has already initially been done (compare Section 4.3).

Some of the test users (including the two who answered question Q2 worst) had to do manual changes on the lower model layers again because they did the transformations more than just once and thereby overwrote their modifications.

The question specifically targeting the model-driven approach is:

Q3: *The model-based approach facilitates the development of data forms for multiple target platforms (tablets and smartphones).*

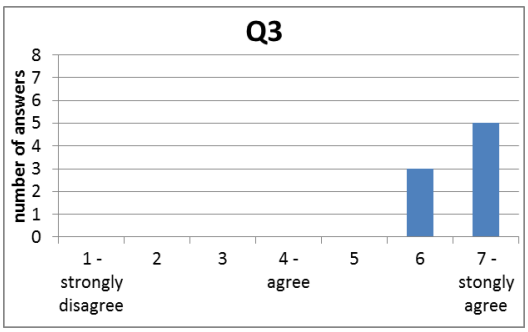


Figure 37: Answer values regarding the satisfaction with the underlying model-driven approach

This question has been answered with a mean value of M3=6.63. The standard deviation σ2=0.48 is rather low. These results allow concluding that the users see the benefit of the model-driven approach for designing multi-device data forms. This is especially interesting since although some of the study participants felt to be forced to do unnecessary working steps (compare Q2), the users at the same time felt that the approach eases the design process.

The user's satisfaction concerning the results of the design process, that is the data form itself, is determined by the three last questions of the questionnaire. The first two of them focus on the structure and the design of the generated data form. The very last question is about the general usage of such data forms for data capturing in a medical study environment.

The questions regarding the visual appearance of the generated data form are the following:

Q7: *I am satisfied with the appearance of the generated data form which I have seen on the smartphone device.*

Q8: *The generated data form meets my expectations of a data form for the given task description on a smartphone device (or tablet, except element arrangement).*

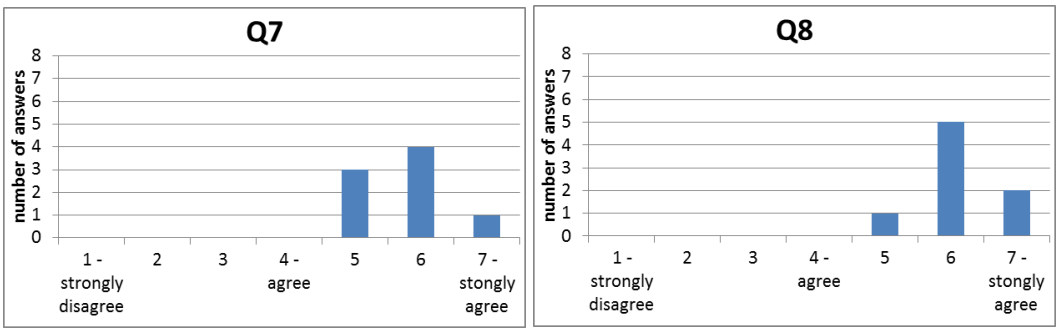


Figure 38: Answer values regarding the satisfaction with the structure and the design of the generated data form

With a mean value of M7=5.75 for Q7 and M8=6.13 for Q8 the study participants evaluate the data form, resulting from their development, uniformly very positive. The standard deviations calculate as σ7=0.66 and σ8=0.60. These values allow concluding that the users were throughout satisfied by their results they have seen on the smartphone. Thus, the arrangement of the elements on the data form as well as the separation into several pages and the navigation between these pages seems to be appropriate. The fact that the generated data form also matches the user's imagination of a data form considering the given task description (this is Q8) allows reasoning further, that the type of generated data forms can be used intuitively. This is affirmed by the low standard deviation of the answers, because the users seem to have a similar imagination of the structure and design of such a data form, which is achievable by the features of the developed model-driven approach using the :studyforms prototype. Here, the answers of the domain experts, which all answered with a value of 6, is especially important since these are the people that are familiar with the current data capturing processes during medical studies (using paper forms). The more the electronic data forms are comparable to their current paper versions, the easier the introduction of the new approach will be.

The last question regarding the user's satisfaction with the resulting data form targets the applicability of such kind of data forms in the intended target environment. The question is formulated as follows:

Q9: *In general, I can imagine using such kind of data forms for data capturing on smartphones or tablets in a study situation.*

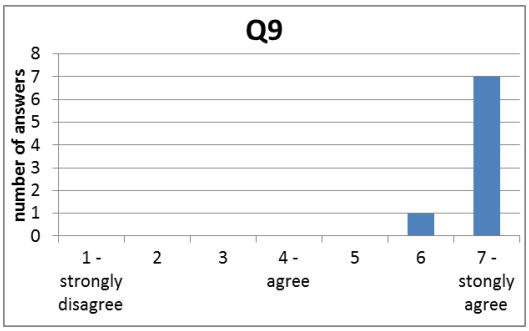


Figure 39: Answer values regarding the general usage of mobile data forms in a medical study environment

Figure 39 shows that question Q9 was answered almost optimally with a mean value of M9=6.88 and a standard deviation of σ9=0.33. Although the answers are uniformly positive, it has to be distinguished between the domain experts and the reference group. Since the domain experts are much more experienced with medical studies, their answers can be given a higher weight. The responses of the future users show that they are open for this new approach and willing to replace the current paper forms with an electronic version. Due to their profession, the test users of the reference group are familiar with slipping into special domains for finding possibilities to support domain experts by appropriate software. Based on the explanations at the beginning of each study session they were able to simulate a study situation. Therefore the answers of these test users are also taken into account.

Learnability

For gathering information about the learnability of the developed system, three questions are asked on the questionnaire. The first of these questions is about the time needed to get into the software in general:

Q4: *It took a long time to learn how to use the software.*

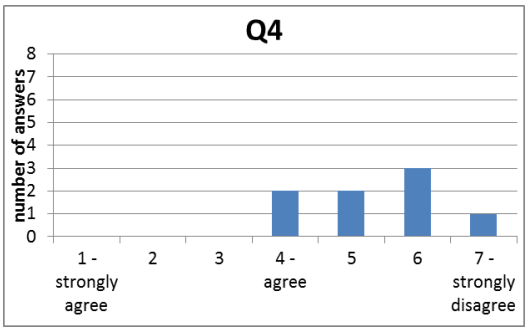


Figure 40: Answer values regarding the time needed to learn how to use the software

First of all it has to be stated that for this question the scale has been inversed, such that higher values represent a lower hurdle to get familiar with the usage of the software. The study participants answered with a mean of M4=5.38 with a standard deviation of σ4=0.99 to this question. This shows that getting into the software was in general not a serious problem for the test users. One of the study participants stated in the comments section of the questionnaire that the user interface of the prototype is unusual and that one has to try a bit until one gets the general idea. But, the user expressed further that he feels able to use the software with some practice. Since the software is intended to be used by persons that are trained in using it, the observed training curve for the novice test users is seen non-critical.

The second question in the area of learnability regards the transparency of the automatic transformations between the model layers:

Q5: *The automatic transformations between the model layers are comprehensible for me.*

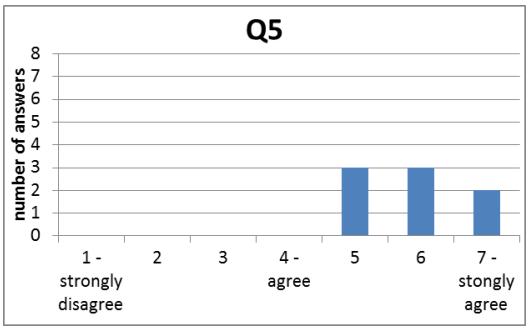


Figure 41: Answer values regarding the transparency of the automatic transformations

The mean value of the answers to this question is M5=5.88 and the standard deviation is σ5=0.78. For this question it is also interesting to look at the mean values of the two user groups. The domain experts answered in average with 5.50 and the computer science professionals with 6.25. Thus, there is a tendency that the transformations are more comprehensible for the computer science professionals than for the domain experts. This seems to confirm the preliminary thoughts that the reference group feels more familiar with the model-driven approach than the target group. However, the answers are positive and allow concluding that the model-driven approach is applicable for the future users of the system.

The last question targets the applicability of the software in the future:

Q6: *I can imagine using the software in the future to generate electronic data forms for capturing data using PCs, tablets and smartphones.*

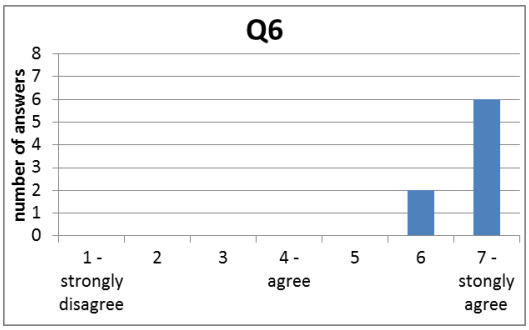


Figure 42: Answer values regarding the applicability of the software in the future

Figure 42 shows that the answers to question Q6 are uniformly positive. The mean value calculates to M6=6.75 and the standard deviation is σ6=0.43. This shows that the users of both groups can imagine using the software in a productive environment and are open to the model-driven approach, used by the software to design multi-device data forms. One test user concluded in the comments section of the questionnaire that there is a need to be able to abstract in order to use the software because of the unusual user interface of the prototype. Due to the answers of this user to the other questions this is taken as a neutral statement. In a later version of the prototype the user interface might become more familiar by introducing more symbols on the GUI for the different model elements in the tree views and the toolbar. Furthermore, the naming of the transformation processes and the model elements could be improved to

be easier understandable for the future user group. These two suggestions for improving the prototype were made by two other study participants.

Efficiency

The efficiency of the test users in fulfilling the given task is measured by the time the users need to design the data form. This is the so called *Task Completion Time* (TCT). Figure 43 shows the overall average Task Completion Time in minutes (left) and the average per user group (mid and right). The TCT starts with generating a new data form and ends with the final transformation to the Windows Phone 8 platform. Unfortunately, the given task description (see Appendix D) does not define a clear end of the task. Therefore, the supervisor of the study gave a short hint to do this last transformation, if the test users said to be finished without having it done.

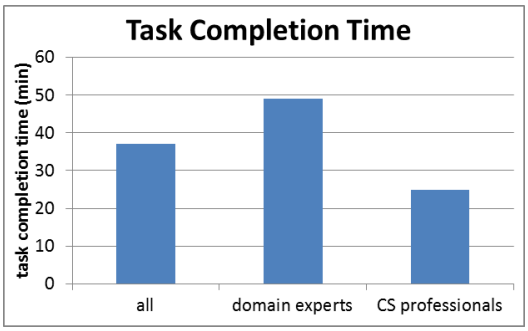


Figure 43: Average task completion time needed by the test users to fulfill the task

The average task completion time over all test users is about 37 minutes. One of the domain experts stated during the user study that currently it takes between 10 and 40 minutes to generate a paper data form. The broad timespan is caused by the dependence on the complexity of the data form whereby the data form designed during the user study is assumed to be not trivial. This allows concluding that the time needed for generating the electronic data form is comparable to the time currently needed to generate a paper form. Hence, using the model-driven :studyforms application for designing electronic data forms results in a comparable efficiency to generating a paper form. Thus, the advantages of the electronic multi-device data forms, like the simplification of the data capturing process and minimization of entry errors, are not impaired by a more time consuming data form design process.

An interesting detail is the fact that the domain experts need in average double as much time (about 49 minutes) as the computer science professionals (about 26 minutes). Thus, the domain experts seem to need more time to get into the details of the prototype and the approach in general. But, it also has to be taken into account that the domain experts spend more time to group the elements in a meaningful way and to order them according to the real data capturing workflow. Since the computer science professionals have no experience with medical studies, their grouping and ordering was less reasonable and took less time.

Effectiveness

The effectiveness of the test users is measured by means of how much of the given task they fulfilled. Therefore, in the preparation phase of the study, a list with 20 issues was composed, which contains the most important steps the users had to do in order to generate a data form that is compliant with the given task. Following, these steps are called subtasks. The list of subtasks is attached in Appendix E. During the study, the supervisor marked which subtasks were done by the test users. These results have again been reviewed afterwards. The number of successfully performed subtasks allows calculating a percentage that reflects how much of the task a user has done. If the users got stuck performing some of the subtasks, the supervisor gave a short hint. These hints were counted and documented. Minor mistakes like setting the `EntryMode` attribute to *"AutomatedOrManual"* instead of *"Automated"* were not taken into account. Here, it is more important that the user recognized that he must change it and where to change it.

In average, the users were able to complete the task to 86% successfully. All users solved the task to at least 80%. These results are considered positively especially because of the missing experience of the test users with the application. Most of the participants needed one hint from the supervisor, the maximal number of given hints is 2. Taking into account that the users saw the software for the first time, this can be neglected. The fact that the percentage value of task completeness is above 80% for all test users also validates the comparison of efficiency among the participants (see above).

6.3.2 Observed Potential for Improvement

As already mentioned in Section 6.2.2 during performing the task, the test users were requested to express their thoughts aloud. By this method, the supervisor was able to take notes. The evaluation of the notes gives information about concrete usability problems that are caused by the model-driven approach or its kind of implementation in the :studyforms prototype. Following, these observations are explained and possible solutions are given. The order of the mentioned problems does not reflect their severity.

- The missing propagation of major changes on the upper model layers (compare Section 4.3) to the lower ones, after the transformation already had been done initially caused problems. Some of the users did the transformations again and therefore had to rework their changes to the lower model layers. For other users, the models just became inconsistent. During the debriefing, one of the users proposed to use a Wizard like user interface that guides the user from one model layer to the next to solve the problem. But, not allowing the user to move back in the design process is assumed to be too strict. Thus, in a future version the propagation of changes throughout the model layers needs to be enhanced. Introducing a Wizard is nevertheless an idea that should be further investigated.
- Several users had problems in understanding how the Automated Input Devices work. This caused problems in using them because the users could not imagine how the AID elements, which are represented just by a placeholder in the Con-

crete Form Model, will work and look like on the final data form. To overcome this problem, the functionality of the available Automated Input Devices should be described in more depth directly on the user interface.

- Related to the previous problem, some users also had problems in connecting the command elements to the balance device element. Here, the problem was that the users tried to specify the command's action before they added the balance device to the data form. Thus, they did not know how to specify the action of the command. Concerning these problems, most of the hints had to be given by the supervisor. After the short explanation, the users were able to set up the action references. Therefore this problem might not be crucial for trained users and if the functionality of the AID elements is explained on the user interface.
- Most of the users did not set the data type of the volatile data definition for the net weight. Since this value is calculated automatically, some users expected the system to set the data type based on the data types of the values involved in the calculation. This should be done in a future version.
- The `EntryMode` attribute of the net weight data definition caused some confusion. This value should be set to *"Automated"* by the system, if a derivation formula is specified and no longer be editable as long as a derivation formula is set.
- Several users searched the attribute for setting the maximal length of a string value on the Data Definition Model because in their mental model it belongs to the data definition, not to the input field. Thus, this property should be moved to the higher model layer.

6.3.3 Conclusions

The results of the usability study show that in average, the users were able to fulfill about 86% of the given task in about 40 minutes. This time is comparable to the time currently needed to create paper data forms with similar complexity. Thus, the model-driven approach can be used with similar efficiency as the current data form design process.

The comparison of the domain expert user group with the reference group of computer science professionals yields that the members of the reference group felt more familiar with the model-driven approach and just need about half the time to perform the given task than the domain experts. The results of the "think aloud" method shows that the domain experts spend much more time thinking about a reasonable grouping of the elements on the data form. Since the members of the reference group do not know the details about the data capturing processes in a study environment, they did the grouping in a more intuitive way and therefore needed less time. This partially mitigates the difference in the Task Completion Time.

Based on the results of the questionnaire it can be concluded that the test users were very satisfied with the model-driven approach and the prototype as well as with their development results. In average the users answered with a 6 on a 7 step Likert scale. The evaluation of the questions about the users' satisfaction yields no mentionable differences for the two user groups. Also the learnability was rated with a 6 in average, which shows that the test users in general easily became familiar with the proto-

type. Nevertheless, there is a training curve that maybe can be further limited by increasing the :studyform application's usability and providing more guidance through the model-driven process for the designer. The comparison of the answers about the learnability of the two user groups shows a slightly better result for the computer science professionals which was expected due to their experience with model-driven approaches in general.

In spite of the good results of the questionnaire, the evaluation of the think aloud method identified several problems of the model-driven approach or its implementation respectively. The most severe problem is the missing propagation of changes on the upper model layers to the target models after the initial transformation is performed. This drawback needs to be overcome before the model-driven approach for designing data forms can be used in a productive environment because it forces the designer to do unnecessary work which reduces the users' satisfaction and might cause errors in the sense of inconsistent models. The other detected problems are of less importance because they can easily be solved by minor changes to the approach or its implementation.

7 Summary and Future Work

This chapter summarizes the developed approach to design multi-device data forms for capturing data in a medical study environment and the results of its evaluation. Thereafter, an outline of issues that remains open for future work is given.

7.1 Summary

During the thesis, a model-driven approach for designing multi-device data forms was developed. This approach is used to develop data forms in the context of the :study software that optimizes the data capturing processes of medical studies conducted at the DLR Institute of Aerospace Medicine. The approach works on four model layers that are traversed one after another by the designer. The transitions from one model layer to the next are done by automatic model transformations.

First, the data that should be captured by the data form is defined at the *Data Definition Model* (DDM). This data model also serves as the link between the data form and predefined parameters in the central :study database. From these data definitions, an abstract version of the data form's graphical user interface is generated. This *Abstract Form Model* (AFM) is independent of the used target device. On this abstract layer, the designer specifies which interaction elements are used for gathering the defined data. Furthermore, peripheral devices are integrated into the data form at this layer. These devices can be used to partially automate the data entry process. On the next layer, a *Concrete Form Model* (CFM) is designed for each type of target device the data form will be used on. The considered device types are smartphones, tablets and desktop devices. This separation is based on the different display sizes of these device types which has to be considered by the design of the data form. Thus, on the concrete layer, the design process is split into specific models for each device type. These models are still independent from the actual platform and application framework of the target device. The splitting allows defining dedicated layouts of the data form that fit to the features of the respective device type. Furthermore, the Concrete Form Model concretizes the Abstract Form Elements from the layer above in such a way that the used interaction elements are appropriate concerning the available display size and device features. Depending on the target platform, a last transformation generates the *Final Form* (FF) implementation for this platform. The Final Form is not a standalone application, but is interpreted by a dedicated application for data capturing that is available for each platform a data form should be used on.

Based on the developed model-driven approach, a prototypical implementation of an application that supports designing multi-device data forms was done. An important requirement to this application is that it needs to be usable by people without computer science background. Thus, building up the models is supported by a tree-like graphical editor to which new model elements can be easily added using drag and drop. On the Concrete Form Model an additional graphical editor is available that already illustrates the later visual appearance of the data form. Since the Concrete Form Model is still independent from the actual target platform, the exact design of

the data form is only visible at the target device. The prototype focuses on generating data forms for mobile devices, especially for smartphones, and provides a transformation to the Final Form implementation for Windows Phone 8.

The developed model-driven approach and the implemented prototype were evaluated by a usability study. The study was set up as a proof of concept study whose goal was to explore whether the model-driven approach is applicable for the future target group and to find severe problems of the approach. Therefore, eight test users were asked to design a data form according to a given task description. Four of the test users were experts in the domain of medical studies at DLR and thus future users of the application. The other four test users were computer science professionals who act as a reference group. The users' satisfaction was determined by a questionnaire. During the usability study, the "think aloud" method was used to get information about general usability problems of the prototype. The answers of the questionnaire yield that the users are highly satisfied with the implemented prototype. The test users had no severe problems in using the prototype for designing a data form according to the given task description. The overall result of the study is that the model-driven approach is applicable for the future users. However, the study revealed some general problems that remain open for future work.

7.2 Future Work

The following paragraphs give an overview about issues that still need to be done and possible improvements to the model-driven approach.

Due to the limited time for the thesis, the prototypical implementation focuses on mobile devices and especially on smartphones. Therefore, the finalization of the transformation to the Tablet Concrete Form Model and the development of a transformation to the Desktop Concrete Form Model remain subject for future work. The same holds for the transformations to Final Form implementations for these two device types.

Especially for the Desktop Concrete Form Model it would be interesting to investigate the usage of a constraint based layout system like it is described by Luyten et al. (see Section 3.1.4) because the variation of screen sizes on this device type is not as limited as for the other two introduced types. Additionally, the transformation to the Mobile Concrete Form Model could be extended in such a way that large groups are divided automatically into several pages. Therefore the exploration of a sensible division algorithm could be subject for future work. Moreover, a further evaluation of the log files gathered during the usability study could reveal additional potential for improving the model-driven approach.

Furthermore, the prototype does not propagate changes on higher model layers to the lower ones after the initial transformation is performed (compare Section 4.3). Since this was one of the main drawbacks discovered during the usability study, a future version should handle such changes and perform suitable transformations to the lower model layers without destroying changes that were manually made by the designer. Possible solutions for this problem have been introduced in Section 4.3.

In order to prevent the designer from developing unreasonable data forms, several constraints should be introduced. For example it should not be possible to link a `NumericalEdit` element at the Abstract Form Model to a data definition with data type *"Text"*. There are several other design opportunities similar to this example that can result in an inconsistent state that should be prevented by the application in advance.

One aspect that is not yet covered by the model-driven approach in general is the possibility to display already captured data on a data form. For the 24h urine example it would for instance be useful to display the urine bottles initially captured at the study facility at the data form used in the laboratory. Thereby the user can easily see which samples he still has to process further. To achieve this, some kind of data query definitions could be added to the Data Definition Model. The values provided by these elements could then be displayed by some output elements on the data form.

References

[Bon08] Frank Bongers. *XSLT 2.0 & XPath 2.0*. Galileo Press, Bonn, second edition, 2008.

[cam04] Cameleon Project - plasticity of user interfaces. [online] http://giove.isti.cnr.it/projects/cameleon.html, July 2004. [Accessed: August 22, 2013].

[CCT+02] G. Calvary, J. Coutaz, D. Thevenin, L. Bouillon, M. Florins, Q. Limbourg, N. Souchon, J. Vanderdonckt, L. Marucci, and C. Santoro F. Paternò. The CAMELEON Reference Framework. Technical report, September 2002.

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15:289–308, 2003.

[Die] Daniel Dietrich. Protected Regions – Core. [online] https://github.com/danieldietrich/xtext-protected-regions/blob/master/plugins/net.danieldietrich.protectedregions.core. [Accessed: August 22, 2013].

[DLRa] DLR. Institute of Aerospace Medicine. [online] http://www.dlr.de/me/en. [Accessed: August 22, 2013].

[DLRb] DLR. :study. [online] http://software.dlr.de/p/study/home/. [Accessed: August 22, 2013].

[DLR13] DLR. DLR at a glance. [online] http://www.dlr.de/dlr/en/desktopdefault.aspx/tabid-10443/637_read-251, 2013. [Accessed: August 22, 2013].

[EVP01] Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. Applying Model-Based Techniques to the Development of UIs for Mobile Computers. In *Proceedings of the 6th International Conference on Intelligent User Interfaces*, IUI ’01, pages 69–76, New York, NY, USA, 2001. ACM.

[FV04] Murielle Florins and Jean Vanderdonckt. Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, IUI ’04, pages 140–147, New York, NY, USA, 2004. ACM.

[Gooa] Google. Linear Layout. [online] http://developer.android.com/guide/topics/ui/layout/linear.html. [Accessed: August 22, 2013].

[Goob] Google. Supporting Multiple Screens. [online] http://developer.android.com/guide/practices/screens_support.html. [Accessed: August 22, 2013].

[Gro05] John Grossmann. Introduction to Model/View/ViewModel pattern for building WPF apps. [online] http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx, October 2005. [Accessed: August 22, 2013].

[HB11] Steven Hoober and Eric Berkman. *Designing Mobile Interfaces*. O'Reilly Media Inc., 2011.

[HII] HIIS Laboratory. MARIAE. [online] http://giove.isti.cnr.it/tools/Mariae/. [Accessed: August 22, 2013].

[HSL+08] James Helms, Robbie Schaefer, Kris Luyten, Jean Vanderdonckt, Jo Vermeulen, and Marc Abrams. User Interface Markup Language (UIML) Version 4.0, January 2008.

[Hub10] Thomas Claudius Huber. *Windows Presentation Foundation - Das umfassende Handbuch*. Galileo Press, Bonn, second edition, 2010.

[Kuh12] Peter Kuhn. Windows Phone 8: Multiple Screen Resolutions. [online] http://www.silverlightshow.net/items/Windows-Phone-8-Multiple-Screen-Resolutions.aspx, December 2012. [Accessed: August 22, 2013].

[LCC03] Kris Luyten, Bert Creemers, and Karin Coninx. Multi-device Layout Management for Mobile Computing Devices. Technical report, Expertise Centre for Digital Media Limburgs Universitair Centrum, Wetenschapspark 2 B-3590 Diepenbeek, 2003.

[Lew95] James R. Lewis. IBM computer usability satisfaction questionnaires: psychometric evaluation and instructions for use. *International Journal of Human-Computer Interaction*, 7(1):57–78, January 1995.

[LVM+04] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. USIXML: A User Interface Description Language for Context-Sensitive User Interfaces. In *Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces with XML"*, pages 55–62, 2004.

[Mei11] Gerrit Meixner. Modellbasierte Entwicklung von Benutzungsschnittstellen. *Informatik-Spektrum*, 34:400–404, 2011.

[Mica] Microsoft. API Reference for UML Modeling Extensibility. [online] http://msdn.microsoft.com/en-us/library/vstudio/ee517354.aspx. [Accessed: August 22, 2013].

[Micb] Microsoft. Codegenerierung und T4-Textvorlagen. [online] http://msdn.microsoft.com/en-us/library/vstudio/bb126445.aspx. [Accessed: August 22, 2013].

[Mic07] Microsoft. Open Specification Promise. [online] http://www.microsoft.com/openspecifications/en/us/programs/osp/default.aspx, February 2007. [Accessed: August 22, 2013].

[Mic12a] Microsoft. Visual Studio Visualization and Modeling SDK. [online] http://archive.msdn.microsoft.com/vsvmsdk, September 2012. [Accessed: August 22, 2013].

[Mic12b] Microsoft. XSLT Extension Objects. [online] http://msdn.microsoft.com/en-us/library/tf741884.aspx, August 2012. [Accessed: August 22, 2013].

[Mic13a] Microsoft. Design library for Windows Phone. [online] http://msdn.microsoft.com/library/windowsphone/develop/fa00461b-abe1-41d1-be87-0b0fe3d3389d%28v=vs.105%29.aspx, August 2013. [Accessed: August 22, 2013].

[Mic13b] Microsoft. Multi-resolution apps for Windows Phone 8. [online] http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj206974%28v=vs.105%29.aspx, July 2013. [Accessed: August 22, 2013].

[MPS03] Giulio Mori, Fabio Paternò, and Carmen Santoro. Tool support for designing nomadic applications. In *Proceedings of the 8th international conference on Intelligent user interfaces*, IUI '03, pages 141–148, New York, NY, USA, 2003. ACM.

[MPS04] Giulio Mori, Fabio Paternò, and Carmen Santoro. Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions. *IEEE Trans. Softw. Eng.*, 30(8):507–520, August 2004.

[MPV11] Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. Past, Present, and Future of Model-Based User Interface Development. *i-com*, 10(3):2–11, 2011.

[Nei12] Theresa Neil. *Mobile Design Pattern Gallery*. O'Reilly Media Inc., second edition, 2012.

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993.

[OMG11] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. [available online: http://www.omg.org/spec/QVT/1.1], January 2011. Version 1.1.

[OMG13a] OMG. About OMG. [online] http://www.omg.org/gettingstarted/gettingstartedindex.htm, August 2013. [Accessed: August 22, 2013].

[OMG13b] OMG. MDA - The Architecture of Choice for a Changing World. [online] http://www.omg.org/mda/, February 2013. [Accessed: August 22, 2013].

[PE01] Angel Puerta and Jacob Eisenstein. XIML: A Universal Language for User Interfaces, 2001.

[PSS09] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. Support for authoring service front-ends. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, EICS ’09, pages 85–90, New York, NY, USA, 2009. ACM.

[RC08] Jeffrey Rubin and Dana Chisnell. *Handbook of Usability Testing - How to Plan, Design, and Conduct Effective Tests*. Wiley Publishing Inc., Indianapolis, second edition, 2008.

[SB11] Florian Sarodnick and Henning Brau. *Methoden der Usability Evaluation - Wissenschaftliche Grundlagen und praktische Anwendung*. Verlag Hans Huber, Bern, second. edition, 2011.

[SV03] Nathalie Souchon and Jean Vanderdonckt. A review of XML-compliant User Interface Description Languages. pages 377–391. Springer-Verlag, 2003.

[SVEH07] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt Verlag, second edition, May 2007.

[TA08] Tom Tullis and Bill Albert. *Measuring the User Experience - Collecting, Analyzing and Presenting Usability Metrics*. Morgan Kaufmann, 2008.

[TC03] Laurence Tratt and Tony Clark. Issues surrounding model consistency and QVT. Technical Report TR-03-08, Department of Computer Science, King’s College London, December 2003.

[Tra08] Laurence Tratt. A change propagating model transformation language. *Journal of Object Technoloy*, 7(3):107–126, March-April 2008.

[W3C99] W3C. XSL Transformations (XSLT) - Version 1.0. [online] http://www.w3.org/TR/xslt, November 1999. [Accessed: August 22, 2013].

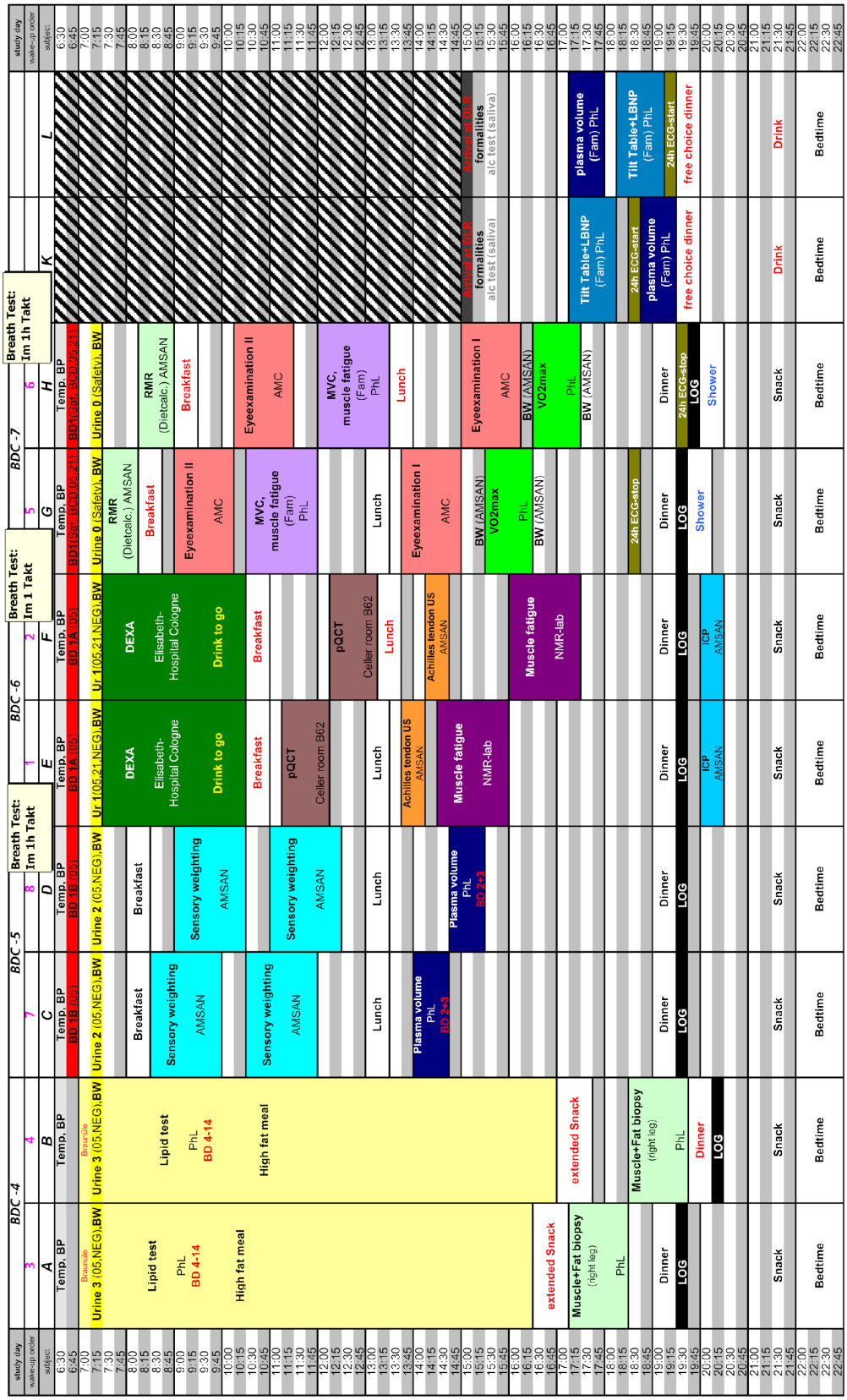
[W3C04] W3C. XML Schema Part 0: Primer Second Edition. [online] http://www.w3.org/TR/xmlschema-0/, October 2004. [Accessed: August 22, 2013].

[Wes98] Ivo Wessel. *GUI-Design: Richtlinien zur Gestaltung ergonomischer Windows-Applikationen*. Carl Hanser Verlag, München, Wien, 1998.

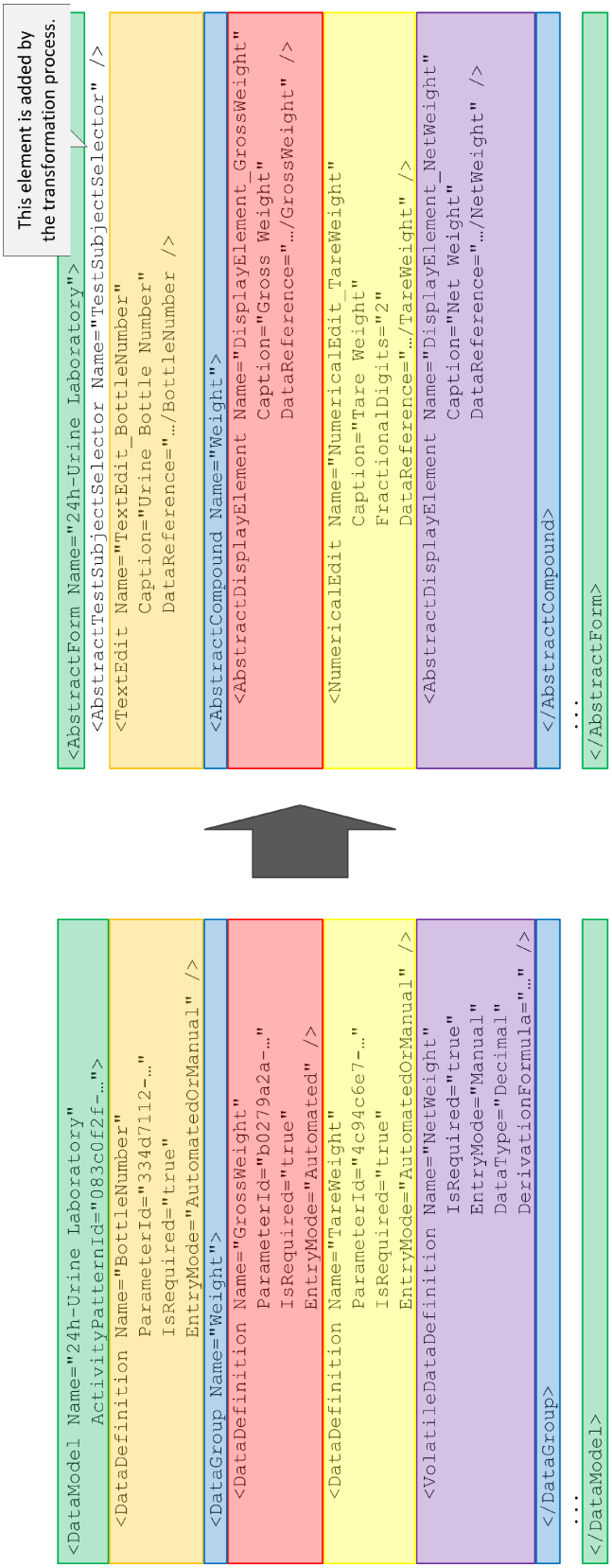
[WHG97] H. Willumeit, K.C. Hamborg, and G. Gediga. IsoMetricsS, June 1997.

Appendix

A Example Study Protocol



B Transformation Example



C Introduction Sheet

Einführung

Bei der zu testenden Software :studyforms handelt es sich um eine Anwendung, die es ermöglicht elektronische Formulare zur Datenerfassung im :study Projekt zu erstellen. Da Smartphones und Tablets immer mehr den Weg in unser alltägliches Leben finden und gerade bei der Datenerfassung im Umfeld medizinischer Studien enorme Vorteile bieten können, sollen die erstellten Formulare nicht nur auf einem Desktop PC mit großem Monitor bedienbar sein, sondern auch auf oben genannten Geräten. Diese bieten erheblich kleinere Displays zur Anzeige von Informationen und sind meist nur mit Hilfe eines Touchscreens zu bedienen, was es unmöglich macht, dasselbe Formular auf den drei unterschiedlichen genannten Gerätearten sinnvoll einzusetzen. Nichtsdestotrotz sollen mit einem Formular auf allen Geräten die gleichen Parameter erfasst werden. Um zu verhindern, dass unterschiedliche Ausführungen des gleichen Formulars für die unterschiedlichen Gerätearten von Grund auf entwickelt werden müssen, erfolgt die Formularerstellung mit Hilfe eines modell-getriebenen Ansatzes in drei Schritten. Hierbei steht jeder Schritt für das Erstellen einer Modellebene. Der Übergang von einer Modellebene in die nächste erfolgt jeweils durch eine automatische Transformation. Abbildung 1 verdeutlicht den Prozess der Formularentwicklung grafisch. Die einzelnen Ebenen werden im Folgenden näher erläutert:

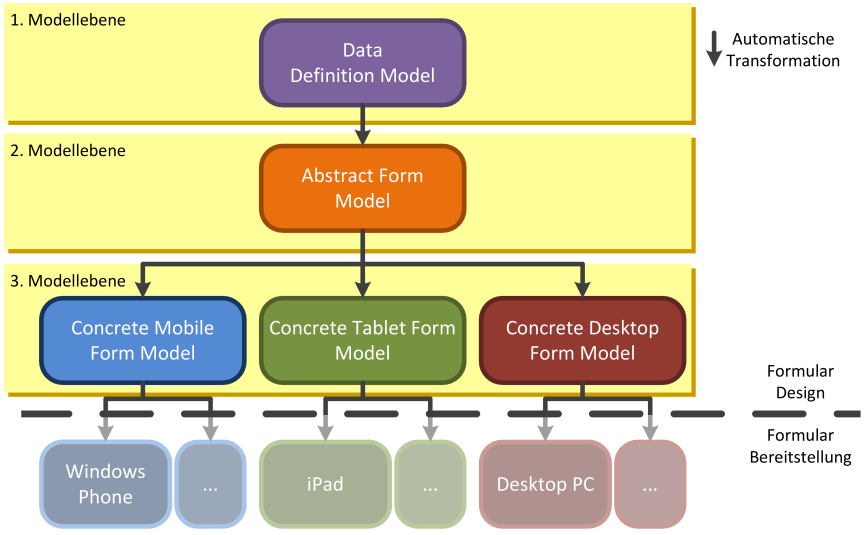


Abbildung 1: Übersicht modellgetriebener Ansatz zum Erstellen von Formularen

Im ersten Schritt wird im Datendefinitionsmodell (Data Definition Model) definiert, welche Daten mit dem Formular erfasst werden sollen. Unter Daten sind in diesem Zusammenhang in der :study Datenbank vordefinierte Parameter zu verstehen, die während einer Studie zu einer bestimmten Aktivität erfasst werden. Die Definition der zu erfassenden Daten erfolgt mittels Datendefinitionselementen, denen ein Parameter aus der :study Datenbank zugeordnet ist. Das Datendefinitionsmodell stellt damit sozusagen die Verbindung zwischen dem Formular und der :study Datenbank dar. Darüber hinaus ist es möglich, dem Datendefinitionsmodell sogenannte flüchtige Datendefinitionen (Volatile Data Definition) hinzuzufügen, deren Werte nicht in die Datenbank übertragen werden. Dies ist zum Beispiel sinnvoll, wenn es sich um einen, aus anderen Werten

berechneten Wert handelt, der nur zu Kontrollzwecken auf dem Formular angezeigt wird. Um redundante Datenhaltung zu vermeiden, werden solche Werte in der Regel nicht in der Datenbank gespeichert. Zu jeder dieser Datendefinitionen (allgemeine und flüchtige) können ergänzende Angaben gemacht werden. Hierbei kann es sich zum Beispiel um einen Initialwert handeln, der für eine Datendefinition nach dem Öffnen bzw. Zurücksetzen des Formulars angenommen wird, oder um eine Kennzeichnung, die angibt ob für eine Datendefinition überhaupt ein Wert angegeben werden muss, oder ob die Eingabe optional ist. Ein weiterer wichtiger Aspekt des Datendefinitionsmodells ist die Möglichkeit Datendefinitionen logisch zu gruppieren. Auf Basis dieser Gruppierung ist das System in der Lage die Fülle an Eingabeelemente auf den sehr begrenzten Anzeigebereich eines Smartphones zu verteilen.

In der zweiten Modellebene (Abstract Form Model) findet eine Zuordnung der definierten Parameter zu abstrakten Eingabeelementen (die später auf dem Formular sichtbaren Felder) statt. Die Angaben, die in diesem Schritt gemacht werden, sind noch vollkommen unabhängig von den Gerätearten, auf welchen das Formular betrieben werden soll. Da die Eingabeelemente sich jedoch von Gerät zu Gerät stark unterscheiden, werden sie in diesem Schritt noch als „abstrakt“ bezeichnet. Bei einem solchen abstrakten Eingabeelement kann es sich neben den bekannten Feldern, beispielsweise zur manuellen Eingabe von Text oder Ziffern, auch um automatisierte Eingabeelemente handeln. Darunter kann man sich zum Beispiel ein Eingabeelement vorstellen, das den Wert einer mit dem System verbundenen Laborwaage ausliest, und automatisch in das dafür vorgesehene Feld einträgt. Um den späteren Anwender beim Ausfüllen des Formulars so gut wie möglich zu unterstützen, kann weiterhin die Reihenfolge, in der die einzelnen Eingabeelemente des Formulars ausgefüllt werden sollten, festgelegt werden. Diese Reihenfolge sollte sich an den einzelnen Arbeitsschritten orientieren, die beispielsweise ein Labormitarbeiter beim Verarbeiten einer Probe durchführt.

Die dritte Modellebene (Concrete Form Model) liegt spezifisch für die drei möglichen Gerätearten (Desktop PC, Tablet, Smartphone) vor. Hier wird im Wesentlichen das Layout des Formulars (Größe und Position der einzelnen Elemente) festgelegt.

Die Abbildung auf eine spezielle Ausführung der jeweiligen Geräteart (zum Beispiel auf ein Windows Phone) erfolgt schlussendlich durch eine weitere Transformation. So kann ein für Mobiltelefone gestaltetes Formular automatisch auf verschiedene Plattformen (z.B. Windows Phone, iPhone, Android) übertragen werden.

D User Task Sheet

Aufgabenstellung

Es soll ein Formular erstellt werden, mit dem die Datenerfassung während der Verarbeitung von 24h-Urinproben im Labor erfolgt, nachdem die Proben zuvor im AMSAN (Arbeitsmedizinische Simulationsanlage) erfasst wurden. Das Formlar soll am Ende auf Smartphones und Tablets angezeigt werden können. Bei der Verarbeitung sollen die folgenden Parameter erfasst und verknüpft zu einem Probanden in der zentralen Datenbank abgelegt werden:

Parameter	Datentyp
Flaschennummer	Zeichenfolge (maximal 6 Zeichen)
Bruttogewicht der Probe	Dezimalzahl
Tara der Probe	Dezimalzahl
pH-Wert (Neutralit)	Dezimalzahl
Einfrieren (Gibt an, ob die Probe zur späteren Weiterverarbeitung eingefroren werden soll.)	Boolescher Wert
Kommentar	Unbegrenzte Zeichenfolge
Signatur (der Person, welche die Probe bearbeitet hat)	Zeichenfolge (maximal 3 Zeichen)

Die benötigten Parameter wurden bereits zur :study Datenbank hinzugefügt. Neben den erforderlichen Parametern, enthält die Datenbank allerdings auch noch weitere Parameter.

Beim Erstellen des Datendefinitionsmodells ist darauf zu achten, dass die Datendefinitionen sinnvoll gruppiert werden.

Zusätzlich soll das Nettogewicht (Bruttogewicht - Tara) automatisch berechnet und zu Kontrollzwecken auf dem Formular angezeigt werden. Da es sich hierbei um einen berechneten Wert handelt, und das Speichern redundanter Informationen in der Datenbank zu vermeiden ist, soll das Nettogewicht nicht in die Datenbank geschrieben werden (Volatile Data Definition).

Das Erfassen der Flaschennummer und des Taras der Probe soll automatisch unter Zuhilfenahme eines Barcodescanners (Urine Barcode Scanner) erfolgen. Die Auswahl des Probanden mit Hilfe des Barcodescanners erfolgt an dieser Stelle implizit. Jede Urinflasche ist mit einem Barcode versehen, der die entsprechenden Informationen beinhaltet. Da dieser Barcode in Ausnahmefällen beschädigt sein kann soll es zusätzlich möglich sein die genannten Daten manuell einzugeben (EntryMode = AutomatedOrManual).

Das Bruttogewicht soll automatisch von einer mit dem System verbundenen Waage (Laboratory Weight) übernommen werden. Um Übertragungsfehler auszuschließen soll hier keine Möglichkeit zur manuellen Eingabe bestehen (EntryMode = Automated). Die Übernahme soll durch einen Klick auf einen entsprechenden Button ausgelöst werden. Ein weiterer Button (Command) soll das Tarieren der Waage ermöglichen.

Darüber hinaus soll das Formular jeweils einen Button zum Speichern der eingegebenen Daten in der Datenbank und zum Zurücksetzen aller Eingabefelder des Formulars enthalten.

E List of Subtasks

The following list contains important subtasks that need to be done by the test user in order to fulfill the user task. Based on this list a percentage is calculated, that indicates how much of the task a test user was able to do. The order of the list does not necessarily reflect the execution order.

1. Generate a new data form and assign it to the right activity pattern
2. Add `DataDefinition` elements for each of the required parameters (overall 7 `DataDefinition` elements)
3. Add a `VolatileDataDefinition` for the net weight
4. Enter the right formula to calculate the net weight (gross weight – tare weight)
5. Set the DataType attribute of the net weight data definition to *"Decimal"*
6. Set the `EntryMode` attribute of the bottle number data definition to *"AutomatedOrManual"*
7. Set the `EntryMode` attribute of the tare weight data definition to *"AutomatedOrManual"*
8. Set the `EntryMode` attribute of the gross weight data definition to *"Automated"*
9. Group the data definitions in a meaningful way
10. Limit the maximal possible number of characters for the bottle number to 6
11. Limit the maximal possible number of character for the signature to 3
12. Add the barcode scanner AID element
13. Assign the barcode scanner AID element to the correct data definitions
14. Add the electronic balance AID element
15. Assign the electronic balance AID element to the correct data definition
16. Add the "Weight" command and assign it to the right action
17. Add the "Tare" command and assign it to the right action
18. Transform to the Mobile Concrete Form Model
19. Transform to the Tablet Concrete Form Model
20. Transform to the Windows Phone 8 Final Form implementation

F Questionnaire about the User's Background

Fragebogen zum Hintergrund der Testperson

Bitte beantworten Sie die folgenden Fragen wahrheitsgemäß (Fragen 1-5) bzw. nach eigener Einschätzung (Fragen 6-8).

Persönlicher Hintergrund:

1. Wie alt sind Sie? _____
2. Geschlecht: ☐ weiblich ☐ männlich
3. Was ist Ihr Beruf? _____
4. Falls abweichend: Was ist Ihre aktuelle Tätigkeit? _____
5. In welchem Institut / in welcher Abteilung arbeiten Sie? _____

Technischer Hintergrund:

6. Als wie erfahren würden Sie sich selbst im Umgang mit EDV-Systemen einschätzen?

   sehr erfahren ☐–☐–☐–☐–☐–☐–☐–☐ unerfahren

7. Als wie erfahren sehen Sie sich selbst im Umgang mit Software zum Erstellen von elektronischen Formularen?

   sehr erfahren ☐–☐–☐–☐–☐–☐–☐–☐ unerfahren

8. Wie gut kennen Sie sich auf dem Gebiet der Modellgetriebenen Softwareentwicklung (MDSD) aus?

   sehr gut ☐–☐–☐–☐–☐–☐–☐–☐ gar nicht

9. Wie vertraut sind Sie mit der Durchführung medizinischer Studien des Instituts für Luft- und Raumfahrtmedizin des DLR?

   sehr vertraut ☐–☐–☐–☐–☐–☐–☐–☐ gar nicht

Vielen Dank für die Teilnahme an dieser Studie!

G Questionnaire about the User's Satisfaction

Fragebogen zur Nutzerzufriedenheit

Die folgenden Fragen sollen Aufschluss darüber geben, wie zufrieden Sie mit der Bedienung der Software bzw. mit dem Vorgehen zum Erstellen eines Formulars sind. Bitte geben Sie Ihre Zustimmung zu jeder Aussage auf der jeweiligen Skala an.

Zweckmäßigkeit:

1. Die Software ermöglicht ein einfaches Erstellen von Formularen.

   stimmt voll ☐–☐–☐–☐–☐–☐–☐ stimmt nicht

2. Die Software zwingt mich, überflüssige Arbeitsschritte durchzuführen.

   stimmt voll ☐–☐–☐–☐–☐–☐–☐ stimmt nicht

3. Der modell-basierte Ansatz erleichtert mir die Erstellung eines Formulars für mehrere Zielplattformen (Tablets und Smartphones).

   stimmt voll ☐–☐–☐–☐–☐–☐–☐ stimmt nicht

Erlernbarkeit:

4. Es hat lange gedauert, bis ich die Bedienung der Software erlernt habe.

   stimmt voll ☐–☐–☐–☐–☐–☐–☐ stimmt nicht

5. Die automatischen Transformationen zwischen den Modellebenen sind für mich nachvollziehbar.

   stimmt voll ☐–☐–☐–☐–☐–☐–☐ stimmt nicht

6. Ich kann mir vorstellen mit der Software zukünftig Formulare zur elektronischen Datenerfassung auf PCs, Tablets und Smartphones zu erstellen.

   stimmt voll ☐–☐–☐–☐–☐–☐–☐ stimmt nicht

Ergebnis:

7. Ich bin mit dem Aussehen des erzeugten Formulars, welches ich auf dem Smartphone gesehen habe, zufrieden.

   stimmt voll ☐–☐–☐–☐–☐–☐–☐ stimmt nicht

8. Das generierte Formular entspricht meinen Vorstellungen eines Formulars für die gegebene Aufgabenstellung auf einem Smartphone (bzw. Tablet, abgesehen von der Ausrichtung der Elemente).

   stimmt voll ☐–☐–☐–☐–☐–☐–☐ stimmt nicht

9. Ich kann mir generell vorstellen mit Formularen dieser Art Daten mit Hilfe eines Tablets oder Smartphones in einer Studiensituation zu erfassen.

   stimmt voll ☐–☐–☐–☐–☐–☐–☐ stimmt nicht

Anmerkungen:

_____

_____

_____

_____

Vielen Dank für die Teilnahme an dieser Studie!

H Contents of the Attached DVD

The following table gives an overview of the content of the DVD attached to this thesis:

Folder Name	Description of the Content
Thesis	This Thesis as a PDF document
Online References	Referenced websites as offline versions
Source Code	The Visual Studio Solution containing the source code of the prototype
Transformations	The implemented XSLT transformations
XML Schema Definitions	The automatically generated XSD files of each model layer
User Study Material	The questionnaires and texts handed out to the test users
User Study Results	The log files and data forms developed by the test users