ADAPTIVE FORMS

A XML-Based Framework for Dynamic Data Entry

Master's Thesis by

Henrik Gammelmark

geemark@cs.au.dk

20031899

Supervised by

Niels Olof Bouvin

bouvin@cs.au.dk

April 30, 2009

DEPARTMENT OF COMPUTER SCIENCE

FACULTY OF SCIENCE

AARHUS UNIVERSITY

Abstract

A framework for defining adaptive input forms in a UI- and device-independent manner is presented. Forms defined in the defined XML language can be pre-adapted based on the use context and adapt themselves at runtime by changing the value, visibility and validation status of each input field dynamically. Adaptation semantics and complex validation rules are expressed using XPath and embedded into the form definition. Turing-complete adaptations and validation is made possible through hook methods.

The host application can transparently inspect and manipulate the form via a type-safe API or XML DOM at runtime, either on-demand or event-initiated, while the end-user is interacting with the form. The decoupled architecture means that the host application need not know anything about how or where the end-user interacts with the form, and thus the form can be transmitted (possibly in a partially completed state) to another application using a different UI layer.

Forms will always be in a valid state when submitted, and data is transparently converted to and from simple Java types or even JavaBeans. No additional layer of validation is needed, as this is performed locally independent of the UI or client in use.

The full feature set has been implemented as a reference prototype in Java, released as open source. The prototype UI is a XHTML- and Ajax-based web GUI.

Keywords
input forms, form adaptation, AdapForms, validation, XML, single authoring, XPath, state tree

This thesis, the prototype implementation of the framework designed herein, associated resources and live demonstrations are available in digital form at the website accompanying the thesis:

http://adapforms.gammelmark.eu

Acknowledgements

First and foremost I would like to thank my supervisor **Niels Olof Bouvin**, who has provided constructive criticism throughout the duration of the thesis period and ensured that I was on the right track with my work. Niels also contributed ideas and hints when I got stuck, for which I am grateful.

The **eGov+** project deserves recognition for providing the motivation behind the thesis from abstract ideas within the research area of *Adaptive Documents*. Thanks to **Morten Bohøj** for acting as my primary technical link to eGov+, and for his continuous stream of feedback and improvement suggestions for the prototype implementation, originating from his work on applying the framework to an eGov+ prototype.

My appreciation also goes to **Claus von Meyeren Christiansen**, volunteering for experimenting with the nearly-finished prototype implementation as a developer, and trying out the framework in practice through a range of use scenarios.

For testing the usability of the prototype implementation from the average end-user point of view, I enlisted the help of **Bodil Gammelmark** and **Kaj Gammelmark** as well as web designer **Filip Skov Lundby**. They contributed with valuable usability observations and ways of thinking that I had not considered myself.

I would also like to thank **Thomas Loftager Nielsen** for proof-reading the final draft of the thesis, pointing out typos, linguistic errors and paragraphs in need of clarification or restructuring.

Finally, I wish to thank everyone else not explicitly mentioned, who has contributed to this thesis directly or indirectly, be it fellow students, family or friends.

The work carried out in this thesis has been influenced one way or the other by all the works listed as references and related work, which have provided a scientific foundation. Concurrently, appreciation goes to the developers of all the various free and open source tools and software libraries I have used for the prototype implementation as well as for authoring the thesis itself. This thesis has been typeset using LaTeX, the free document preparation system.

Contents

List of Figures

PART I

INTRODUCTION

Defining the problem domain and concepts used throughout the thesis

CHAPTER 1

Problem statement

1.1 Vision

The user of an IT system is presented with a digital form, but is only confronted with the relevant options and questions, based on the context of use and prior knowledge about the user. The form is capable of adapting further on the fly, as the values entered limit or expand the options in relevance. The form may be saved in a partial state and returned to at a later time. The context of the form may contain a *user role*, allowing specific parts of the same form to be filled out only by the relevant persons or authorities at different points in time.

From the developer perspective the underlying framework facilitates that a single form, along with a set of semantic rules describing the adaptive properties, can be deployed on multiple platforms without the need to write UI-specific code for the task.

The user experience may be enhanced because of the more personalized forms that is specifically tailored, thus increasing the overview and minimizing the probability of erroneous input.

1.2 Challenges

The following are the primary thesis questions of interest, each with elaborating subquestions:

- Which is the best way to represent the forms and their adaptive semantics?
  - Is it possible to reuse a sub-form across multiple forms? Is it for instance beneficial to support reuse of an “address” form as opposed to rewriting it every time?
  - How comprehensive should the representation/language be? How complex can it become, before it becomes too cumbersome to be of practical use for a developer?
  - Can arbitrarily long lists of sub-forms be supported, and will it be usable in practice? What about recursive forms?
- How can user input and context metadata be represented, to allow for easy transfer between host application, framework and a client (for instance a web browser)?
  - How can this be integrated into traditional web applications, which are typically based on key-value pairs and/or relational databases?

- Which is the best way to ensure the validity of data, both in individual fields, but also that the form in its entirety has been filled in correctly?
  - How should the user be informed about erroneous input or missing information in an easily comprehensible way, even when the rules are complex?
  - How will the data entry order influence the system? How should the system respond if, for instance, the user returns to a previous field and makes a change that affects parts of the form, which may already have been filled out.
  - If different parts of the form is filled out by different persons or legal entities, how can dependencies between their respective parts of the form be handled?

1.3 Abstract architecture

On the abstract level, the framework will consist of the following components, which will be concretized as the project work progresses, either by using existing technologies and tools or by building new ones. Note that this does not neccesarily reflect the final architecture, but is to be thought of on the abstract level only:

- A language to specify forms in, as well as model the semantic interrelations between the elements.
- A preprocessor that is given a form and a set of input data (known information about the user, usage context etc.) and outputs a personalized form that may have preprinted values and is tailored to the specific needs of the user. The input data may include previously entered data in case of a partially completed form.
- A runtime component that acts on user input via UI events and checks data validity as well as resolves dependencies between fields. This could be displayed to the user by using color codes, icons or similar, and choices made by the user may cause the form to change (add/remove/modify form elements).
- A UI-specific renderer that translates between the generic formats and the concrete UI technology in use.

Figure 1.1 on the following page depicts the abstract architecture, with major actions sequentially numbered in execution order. Actions 3 through 6 may be repeated a number of times, while the user interacts with the runtime component indirectly via the UI layer.

The core parts and the UI specific parts will be seperated to the extend possible, in such a way that the core framework can be reused to display the same unmodified form on a range of different devices or platforms; for instance a classical website in a browser, a mobile device, vocal representation for the hearing impaired etc.

1.3.1 Implementation

A concrete proof-of-concept implementation of the framework will be done in Java, and will support a web-based interface through the use of Java Servlets/JSP.

The purpose of the implementation is to experiment with the raised questions and hopefully lead to a refined framework that will be usable in practice.

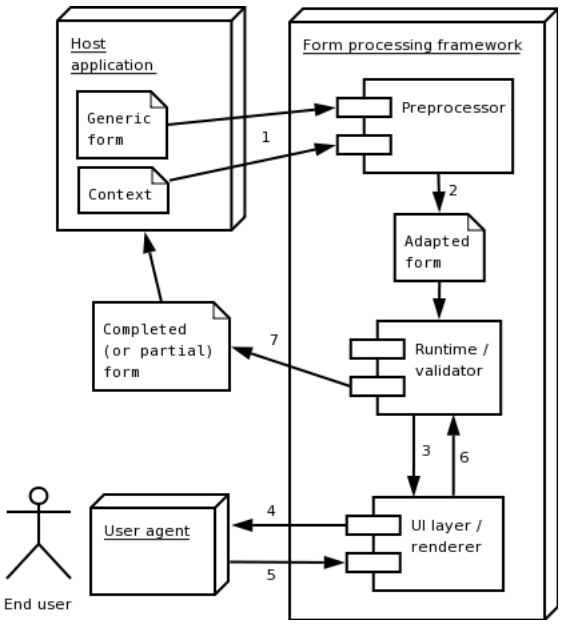


Figure 1.1: Abstract framework architecture

1.4 eGov+ as a case

eGov+ is a Danish research project with the aim of enhancing both the user experience and administration efficiency of electronic government-citizen interaction, primarily in the form of web applications.

The basic idea of having more agile adaptive forms originates from this project. While this thesis is not part of the project, the proposed framework is developed with eGov+ prototype use contexts in mind.

Application and adminstration of *parental leave* is part of the eGov+ project as an area that applies complex forms that could benefit from simplification. This will be adopted as a concrete case in the thesis to the extent possible, taking data sensitivity considerations into account.

More information on eGov+ is available at [15] (in Danish).

1.5 Scope limitation

Although there are many interesting angles of the problem domain, it is necessary to restrain the focus area to a limited subset.

The thesis may sketch solutions for applying the proposed framework in practical host applications, but emphasis is on representing and handling the individual input form, and thus building actual form- or case-management systems is out of scope.

Another aspect that will only receive little attention, is study of user interaction patterns and documentation of user experiences. In other words, emphasis is first and foremost on creating a functional prototype framework.

CHAPTER 2

Concepts and definitions

This chapter introduces some central concepts and notions that provide a basis for understanding and discussing the proposed framework.

2.1 Form types

The following table compares a few aspects of plain old paper forms, static HTML forms as they are used in general today and finally adaptive forms as proposed by this thesis.

	Paper form	**Static HTML forms**	**Adaptive forms**
Boundaries	Very agile. Post-its and manual marking makes it possible to break the boundaries of the predetermined form.	The very strict formalities make it impossible to not adhere to the rules.	SImilar to HTML forms. However, it is possible[1] to store a partially completed form, and return to it at a later time and complete it.
Feedback	None.	Validation errors on submit.	Realtime validation errors and adaptation of elements.
Multi-party input	Different sections may be filled out by different people or authorities, but there is no way to ensure this.	Traditionally there will be completely disjoint input forms, one for each user-type and no clear conceptual correspondance between them.	The same form may be displayed to different parties in turn, and *user roles* ensure that the data is visible and editable to the relevant parties only.
Tangibility	Very tangible. One physical copy that is passed around.	Traditionally a HTML form is not conceptually related to a form entity. Often, the user will never see the input form again after it has been filled out.	Like HTML forms, but with a more document feel. The host application may choose to promote this feeling by providing a virtual folder of the user documents or similar.

	Paper form	Static HTML forms	Adaptive forms
Transparency	It is not always clear which fields are required in the given context and in which format or order data is expected.	The same considerations as for paper forms apply, with the exception that the form can usually not be submitted while it contains errors.	Only the fields relevant to the user in the given situation are visible at any given time and it is clearly indicated which are optional.

2.2 Concepts

The following concepts are used throughout the thesis:

Concept	Definition
Host application	The domain-specific application making use of the framework. The forms are considered belonging to the domain of the application.
Form	A form is the central concept of this thesis. It consists of a number of *form elements* with various properties, and semantics specification defining the *form adaptation*.
Form element	The visible elements of a form, like labels and input fields. A description of all supported elements can be found in section 2.2.1 on the next page.
Form structure	The static hierarchy of form elements a form consists of.
User role	A user can take on one of a number of possible roles when interacting with a form. The role dictates which form elements can be seen or edited by the current user. The roles and their privileges are mandated by the host application.
Form context	The context in which the form is displayed. This may include, but is not limited to, stored information about the user and the *role* of the active user. The context is defined by the host application. Technically, previously entered data (in case of a partially completed form) is treated as part of the context, although conceptually it is not.
Form adaptation	The act of changing either the form data, the relevant elements, validation status or similar of the form, collectively known as the *form state*.
Form semantics	The collective adaption- and validation-rules of the form.
UI layer	The user interface layer of the framework. This stands in contrast with the core framework which contains the functionality shared among all UI techniques.

[1] If the host application has a save/load mechanism for it

When referring to a specific class of individuals, the thesis distinguises between the following stereotypes, which are not necessarily disjoint:

Stereotype	Definition
User or *End-user*	Unless otherwise specified, the *user* is the end-user that interacts with the software frontend via the UI layer. He does not know anything about the underlying framework, and should not need to either; from his perspective, it is simply a form that is being filled out.
Form author	A person who creates adaptive forms within the host application, with only very limited programming knowledge. Although it can be argued that the form author is also a user of the framework, he is not considered a *user* per se. In many situations, the form author may be a *developer*.
Developer	Person (or development team) who uses the framework as part of a host application as a black-box component. Knowledge is limited to interfacing with the framework, authoring forms, and responding to events raised by the framework.
Framework developer	Person (or development team) who knows the internal workings of the framework, and may change or improve it by adding or modifying code. An obvious task of the framework developer could be to add support for a new user interface technology.

2.2.1 Form elements

Below is listed the different types of form elements (input field types etc.) supported in the framework. This is primarily for clarity and avoiding ambiguity, since most of the terms are widely standardized in one form or the other.

Element type	Explanation	Example usage
Label	Read-only text output	Pre-printed or computed value
Text	Unrestricted text, single- or multi-line	Name, street, description
Secret	Text field with asterisks (or similar) to mask the entered text	Password, credit card number
Integer	Ranged or unranged integer (Java *long*)	Age, monthly income
Decimal	Ranged or unranged decimal number (Java *double*)	Price
Toggle	*true* or *false* indication (e.g. check box)	*"Currently unemployed"*
Choice	Single selection among fixed mutually exclusive choices (e.g radio buttons, list or dropdown)	Marital status, country
Multi-choice	Multiple selections among fixed choices (e.g list or multiple check boxes)	Categories, interests

Element type	Explanation	Example usage
Date	Single date selection	Birthday, employment date
Group	Logical part of a form that act as a grouping of related[2] elements. Groups will help divide the form into parts easier to overlook, or merely to structure the underlying data model.	*"Contact details"*
Repeat (repeated subform)	A collection of form elements, that can be repeated in a list-like manner. Repeats may be restricted to occur a specific number of times, or the user may be allowed to add any number of entries desired.	Customer list, each with name and address
Bean	Essentially a *group* that is automatically populated with form elements inferred from a JavaBean. This is elaborated in section 5.4 on page 35.	Complex domain structure, classes to be automatically populated
HelpText	Useful help text to guide the user	*"In the section below, please take into consideration. . . "*

2.3 Benchmark form

In order to have a common reference point and an example to refer to, I have created a form that is used as a benchmark for comparing technologies and ideas throughout the thesis.

The form is not intended to be of any practical use or even make sense, but it aims at utilizing most, if not all, of the features I wish to support, from the simple text field to the more complex adaptations and validations. Note that in most experiments and examples, only few parts of the form, required to draw conclusions or prove a point, will be used.

The form, along with a natural-language description of its semantics, is described in appendix A.

[2]The actual relation may be conceptual (i.e. grouping address elements together), user-centric (different sections to be filled out by various users), or any arbitrary domain-specific relation

CHAPTER 3

Related work

The following includes a discussion of various articles and software systems, which in one way or the other relates to the vision of this thesis. All works covered have influenced the work in this thesis either directly or indirectly by their use of architectural styles, language definitions, technologies or merely abstract ideas and notions.

Each is related to the design goals and vision discussed in the previous chapters, and a summary is provided at the end of the chapter. The goal is not to cover the works in their entirety, but focus on the elements that relate to my own work. However, basic introductions are needed to establish an overview.

The main points of interest when evaluating the works are the following:

- Definition of UI-independent forms
- Definition of adaptation semantics
- Import and export of entered data, and interoperability with other technologies
- Personalization/individualization of the forms

3.1 Dissertation: Web User Interaction

Although specifically aimed at web user interaction and supporting the ever-growing need for a more application-centric web platform, the Ph.D. dissertation [10] holds some valuable discussions and ideas for this thesis. The primary concern of the dissertation is the development of a (primarily conceptual) framework for web applications that allow tailoring of technology-independent user interfaces (supporting browsers, mobile phones, PDAs etc.).

As such, the purpose is different and the scope is much wider than that of this thesis. Nevertheless, some of the considerations apply equally well to both contexts.

3.1.1 Choice of language class

Differences between procedural and declaractive languages for authoring independent interfaces are discussed, and a considerable set of languages and technologies are investigated to find an answer for which approach is best suited. The languages are compared according to a long list of criteria in [10, table 3.2], of which only very few are relevant within the scope of this thesis. The difference is summed up as follows:

" Declarative UI languages have usually a higher semantic level while traditional, procedural programming languages have more expressive power. It is essential that a balance between semantic level and expressive power is found. " [10, chap. 3]

It is concluded that the declarative approach is superior in terms of defining semantics - especially in situations where the UI author does not have developer skills, which very well could also be the case when designing adaptive forms.

More concretely the dissertation speaks strongly of XForms[1] which, as it turns out, has a lot in common with this thesis. It will be covered in more detail in section 3.2.3 on page 13.

3.1.2 UI adaptation

The dissertation deals with four different types of adaptation: Spatial, temporal, interaction and media - all of which primarily deals with the process of adapting the application to a specific use context or device, cf. [10, section 2.5]. As such, there is not much focus on runtime adaptation besides what XForms provides.

The following general abstraction techniques are identified, which is related here to my own problem domain:

- **Multiple authoring**
  The process of writing a separate form for each context, device or UI technology to be used. This is in direct conflict with the vision of this thesis.
- **Unique portable code**
  Examples are code created for specific UI toolkits (e.g. Java Swing) or a generic client (e.g. web browsers understanding scripts). This usually involves using a procedural approach, which is not advisable cf. the conclusion above.
- **Transcoding**
  Conversion from one UI-specific implementation/declaration of a form to another. This requires that only interaction techniques and data structures compatible with both the original and the target systems are used, which is hard to ensure in practice. Furthermore, it forces the developer to know the low-level details of at least one of the used languages and a complex transcoder for each pair of languages should be implemented.
- **Multireification**
  A single high-level description of a form is created, from which a unique UI-specific implementation is reifed (or derived). The high-level description abstracts away all the technical details, thus limiting the required knowledge to construct a form. The element types, structures and semantics of the language being used to describe the form, defines a common abstract set of what is possible (lowest common denominator), regardless of the UI layer being used.

[1] Note that the dissertation author has co-specified the XForms language

- **Abstraction-reification**
  Essentially a combination of transcoding and multireification, where a low-level implementation is first abstracted or simplified into a high-level representation, which in turn is reified. Most of the disadvantages of transcoding remains, but for each user interface type, now only a single converter to and from the common high-level language is needed.

Based on this comparison, multireification is the one that most neatly fits the vision, and thus this will be the starting point when designing a high-level language for adaptive forms.

3.2 Form processing

3.2.1 FormGen: CFG-based approach

FormGen [6] is a Java GUI generation tool that allows input of structured data into a generated adaptive form, graphically displayed as an application window. The article identifies three main tasks related to creating an adaptive forms system:

- Data model
- Dynamic behaviour (adaptation)
- Graphical form layout

FormGen is based on a context-free grammar (CFG) that defines the data structure, and the relation between them. An interesting feature is the support for recursive and ambiguous grammars, allowing the data structure to be infinitively expanded one level at a time on user request. The dynamic behaviour is implicitly defined through the set of possible productions in the grammar.

The backbone of the runtime system is a tree structure representing the current data structure. As the user expands the data structure (creating CFG productions), new nodes are added to the tree, and coupled to the relevant new GUI elements, possibly displayed in a new window. In this way, the hierachical data model and the GUI is directly linked at all times.

Before use, the CFG is used to generate Java classes for both the data model (*classgen* tool) and the GUI components (*formgen tool*). Since the basic builing blocks are limited to abstract notions such as lists, buttons and tuples, the actual GUI generation tool (formgen) may be replaced by another compatible implementation, while maintaining the same data model. This can therefore be considered an example of multireification, since the same abstract definition may literally be compiled into different concrete implementations. The downside is, that this will have to take place at compile-time, as the host application needs to be implemented directly against the compiled classes; at least the data model.

I am inspired by the way the structures of the data model and UI are synchronized, yet transparently to the host application, in shape of a tree. A variation of this will be used in the thesis, with the exception that the host application should not be bound to know the structure at compile-time.

The idea of having both the structure of the form and the semantics tied together in an abstract representation is also neat, but the reliance upon grammars is not particularly convenient, as disjoint parts of the form are tied together by design, making it hard to support reuse of subforms.

3.2.2 Dynamic Forms

Dynamic Forms [9] provides adaptive data entry in a X Window GUI[2]. The form and rules are based on a declarative Form Description Language (FDL) as opposed to a CFG.

Values entered in a field may be used as a basis for calculation of other field values, as known from spreadsheets. The fields are defined in an object-oriented manner, and thus can have properties assigned.

As an example, consider the following snippet of FDL which defines a single entry field:

```
{EntryField other_service_phone,
  Prompt "Other service phone",
  InputMask "999 999-9999",
  Width 12,
  Flags {NONEWLINE, GROUPALIGN},
  DependsOn customer_info,
  Fprocs {FPComputeFCC},
  Formula other_service_phone_2,
  Help {644}
};
```

This definition holds information about the field, both as an abstract named entity, but also as to how it is displayed, how it relates to other fields and what the valid value range is.

The GUI components are rearranged automatically, leaving the developer with only limited influence on the graphical layout. An implicit checklist makes sure that the user can remain informed of which parts of the form have been filled out correctly and which have not. The authors claim the checklist to be implicit, since in fact it is the section headlines that are color-coded to display their collective validation status (see figure 3.1).

Figure 3.1: Dynamic Forms: Checklist. Source: [9]

This is an example of single authoring, also potentially allowing multireification, as all fields are defined abstractly, with only a limited number of GUI-related attributes. Although this framework resembles the

[2]Cross-platform windowing system

thesis vision, it does lack a few mechanisms. For instance, there is no pre-customization based on the current user, and already known user master data. Furthermore, the visibility adaptations are limited to dependency on whether values have been entered in other fields (*DependsOn* attribute). Any logic beyond this (including the dynamic field calculations) must be expressed using procedures written in the C programming language.

Various user feedback and overview mechanisms (such as the color-coded validation) are carried over directly to my problem domain along with certain layout ideas. The notion of having each form element defined as an object that can have properties attached to it, and be freely moved around the form to re-arrange the elements logically, is also one that will be reused.

3.2.3 XForms

XForms offers simple validation and adaptation in a web application. It is currently under development by W3C (World Wide Web Consortium) and is scheduled for inclusion in the final version of XHTML 2.0. It is targeted at replacing the well-known HTML forms and geared towards todays more dynamic web sites.

As is the case with all recent W3C recommendations, it makes intensive use of XML:

- The **XForm model** is an ad-hoc application-specific XML structure used to describe the data model to be filled out. In the simplest case, this maps 1-1 between XML nodes and input fields, but may be more complex. Support for repeated and nested elements exists, but the recursion level is limited; this means that arbitrary tree structures are not possible, cf. [10, chapter 3].
- The **XForm user interface** is basically the XForms equivalent of the traditional XHTML form elements (<*input*>, <textarea> etc.).
- The data submitted from the form is the XForm model listed above, populated with entered or calculated values.

Note that, as is the case with traditional HTML forms, XForms require a host language in order to make sense and operate. XForms can be incorporated into any XML language with XForms support, which for the most part is likely to be XHTML in practice though.

As for validation, datatypes can be specified on each element, using the XML Schema types (with a few exceptions) and marking of *required* or *readonly* fields can be done using XML attributes.

Values can be calculations of other field values or functions. Logic functions include basic boolean, string and date operations. Examples include taking the average of the values of a nodeset (*avg(node-set)*), determining the current date and time (*now()*) and a conditional value assignment (*if(boolean-test, true-string, false-string)*) similar to the Java *choice operator* (*?*).
In fact, all functions in the XPath core library can be used. XPath is covered in more detail in section 3.3.4 on page 18.

Most interestingly though, elements can have *relevant* attributes, which defines whether they should be displayed or not depending on the evaluation of an expression. The relevance of a node or set of nodes can be changed at runtime.

One of the goals of XForms is to move as much of the processing and validation as possible to the client, to reduce the application load of the server and increase the responsiveness of the web application. For more involved interaction, asynchronous requests to the server is supported.

XForms 1.1 is formally defined in [5]. For more casual reading and examples, refer to [16].

Benchmark form

Listing 3.1 shows how part of the benchmark form could look when implemented using XForms. The example makes use of the *required* property for some data fields, as well as demonstrating a simple case usage of the *relevant* property, by showing the *state* field iff *country* has the value *"US"*. Also notice the type designations, using XSD datatypes.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xf="http://www.w3.org/2002/xforms">
  <head>

    <!-- XForm model -->

    <xf:model>
      <xf:instance>
        <person>
          <birthday xsi:type="xsd:date" />
          <name>
            <first xsi:type="xsd:string"/>
            <middle xsi:type="xsd:string"/>
            <last xsi:type="xsd:string"/>
          </name>
          <country xsi:type="xsd:string"/>
          <state xsi:type="xsd:string"/>
        </person>
      </xf:instance>

      <xf:submission action="http://example.com/submit" method="post" id="submit-all"/>
      <xf:bind nodeset="/person/state" relevant="/person/country='US'" />
      <xf:bind nodeset="/person/name/first" required="true()" />
      <xf:bind nodeset="/person/name/last" required="true()" />
      <xf:bind nodeset="/person/country" required="true()" />
    </xf:model>

  </head>
  <body>

    <!-- XForm user interface -->

    <h1>Person details</h1>

    <xf:input ref="/person/name/first">
      <xf:label>First Name:</xf:label>
    </xf:input>
    <br /><br />

    <xf:input ref="/person/name/middle">
      <xf:label>Middle Name(s):</xf:label>
    </xf:input>
    <br /><br />

    <xf:input ref="/person/name/last">
      <xf:label>Last Name:</xf:label>
    </xf:input>
    <br /><br />

    <xf:input ref="/person/birthday">
      <xf:label>Birthday:</xf:label>
    </xf:input>
    <br /><br />

    <xf:select1 ref="/person/country">
      <xf:label>Country:</xf:label>
      <xf:item>
              <xf:label>Denmark</xf:label>
              <xf:value>DK</xf:value>
```

```
60          </xf:item>
61                  <xf:item>
62                   <xf:label>USA</xf:label>
63                   <xf:value>US</xf:value>
64          </xf:item>
65        </xf:select1>
66        <br /><br />
67
68        <xf:input ref="/person/state">
69          <xf:label>State:</xf:label>
70        </xf:input>
71        <br /><br />
72
73        <xf:submit submission="submit-all">
74          <xf:label>Submit</xf:label>
75        </xf:submit>
76
77      </body>
78  </html>
```

Listing 3.1: XForms example

Notice the need to specify both the abstract data format (*XForm model*), and the actual appearance (*XForm user interface*, using XHTML in this example).

Support issues

A significant downside to using XForms is that it requires browser support, either natively or via plugins, and although it is currently a W3C *recommendation*[3], current browser support is extremely limited and it will presumably take a long time before it becomes mainstream.

However, implementations exist that simulates XForms by performing translations to XHTML and JavaScript. Examples include Orbeon Forms [27] (Java server-side implementation), Chiba [28] (Java server-side implementation) and FormFaces [19] (platform-independent JavaScript implementation).

Discussion

XForms is to some extend multi-modal, but it is still very web-centric and it may seem artificial to apply it to e.g. a desktop GUI application, since it depends on a host language such as XHTML.

There is no straightforward classification of the UI adaptation scheme, given the classes defined earlier. In part, this is single authoring, because the XForm model and the defined semantic rules are indeed reusable without modifications (allowing for multireification) but it is still dependent on a specific UI language for defining the interface. In theory, one could simply write only the XForms specific tags and leave out all the XHTML tags, but that would result in a pile of input elements with no formatting, and the page may not display properly, since it would not be a wellformed XHTML document. So as far as the UI goes (which is a required part), multiple authoring in some shape or form must be applied.

That being said, XForms does provide a lot of the adaptivity, validation and user feedback mechanisms I have envisioned, using a relatively easy-to-understand declarative XML language. For these reasons, I will use it heavily as an inspirational source when developing my framework, as will become clear in part II.

[3] In fact, XForms 1.0 has *recommendation* status, while XForms 1.1 is still a *candidate recommendation* as of April 2009

3.2.4 FormEncode

FormEncode [17] is a package for use within Python-based web applications. It consists of a number of related tools that can be used independently of each other. The goal of the package is to provide semi-automated form validation to diverse web applications across multiple technologies, such as XML-PRC, web forms, databases etc.

The core of FormEncode is a validation schema defined in Python syntax, that maps from a domain data structure to a collection of validators to use for each data node or subtree (compound data structure). As a side-effect of the validation, the input data is converted to the expected data format if needed (and possible).

FormEncode provides a utility named htmlfill, capable of taking an arbitrary HTML form with values entered as input, process it according to a validation schema, and annotate the HTML form with validation error messages where relevant. The unprocessed HTML form may have special placeholder elements to let htmlfill know where to insert error messages and in which format.

In addition, FormEncode allows the developer to take the inputs entered and extract them in the data formats expected in a more “Python-friendly” manner, as opposed to reading and converting the raw HTML query strings.

FormEncode supports compound tree-like data structures and lists of similar items, which are two features I would like to make use of also. For this to work with htmlfill, a mapping is performed from the tree-paths of the actual hierachical structure to the flat data structure imposed by HTML. Tree branches are seperated by dots, and repeats are handled by appending an integer to the element name. Example: “book-1”, “book-2” etc. A very similar scheme, albeit with a slightly different syntax, is used in the protoype implementation discussed later in this thesis, where the exact same challenge is encountered.

The advantage of the FormEncode approach is that the developer can design a form in any way and shape he desires, and then create a validation schema that handles the data conversion and validation semi-automatically. There are a few downsides, however. Firstly it requires double work, and a constant synchonization between the two, in case of changes. Secondly, even if htmlfill is designed to augment an arbitrary HTML form, it still requires that the host application pipes the entire HTML form though the framework, and that the form itself has some annotations, such as the aforementioned placeholders.

This can be seen as a hybrid between multireification and multiple authoring; the actual logic and validation can potentially be applied to any kind of form, provided that a language-specific interpreter exists. However, the developer is still required to define the static form and perform the aforementioned mapping between a hierarchy and the flat form by hand, both time consuming and error prone.

While the basic design differs from the thesis vision, there is a relevant element of data exchange present of particular interest. The very idea of having automatic type conversion between the actually desired types (rich Python types in this case) and the extremely simple type system supported by HTML (basically just key, value mappings), is a great one. As a consequence, this has also found its way into my proposed framework, as elaborated in part II.

3.3 Representing and querying data and meta-data

3.3.1 OIOXML

OIOXML [32], or *Offentlig Information Online XML*, is a coordinated effort to standardize the data exchange processes within the public sector of Denmark.

It is a suite of languages / data types (primarily XML) that coexist on different levels (refer to [7, chapter 4] for details):

- **Core class**
  Central concepts in general use across all government bodies, including social security numbers, adresses, names etc.
- **Domain class**
  Domain specific concepts in use by only a few government bodies.
- **NDR class**
  Data formats that adhere to the OIO-NDR conventions[4], but does not qualify to be mandatory standards due to limited use. However, their use is recommended where applicable.
- **Adoption class**
  Internationally adopted standards, not subject to the OIOXML guidelines.

To register an official OIOXML standard, a proposal must be sent to the relevant authorities and undergo inspections and hearings, possibly followed by inclusion into one of the categories mentioned above. The elaborate process is described in detail in [7, chapter 6].

All data formats are publicly available at the OIO InfoStructureBase repository [33].

OIOXML is relevant to this thesis due to the eGov+ relations already mentioned in section 1.4 on page 4. If parts of the work are to be used in the public sector (at least at the prototype level), OIOXML support will be beneficial, if not mandatory, and it should be at least kept in mind when designing the framework.

3.3.2 PalCom Hierarchical Maps

Hierarchical Maps [13], also called h-maps, is a data structure implemented for use within the PalCom EU project [35]. The data structure is intented for multi-purpose storage of data and state of varying complexity using an ad-hoc tree-like structure, that changes over time.

Each piece of information is stored in its own leaf node, and is identified by a unique path, where each part of the path (seperated by *"/"*) represents a level in the tree, starting from the root. Each internal node of the tree holds a map of its children, that allows it to identify them by name (local path part). Thus, it is essentially a hierarchy of maps (hence the name), where all data is stored in the leafs. Querying or updating the structure requires navigating to the relevant node(s), which may occur explicitly by traversing the tree, or implicitly by supplying fully qualified paths.

Each leaf is annotated with an automatically inferred data type, which guarantees a mild degree of type safety on data retrieval, even though the type can change over time if the value is changed.

[4] Offentlig Information Online: Navngivnings- og Design-Regler (naming and design conventions)

Due to the tree structure, serialization to and from XML is made easy, allowing the structure to be either persisted to storage or transmitted (in part or entirety) over a network for introspection. By explicitly maintaining the order of similarly named elements (name uniqueness is not enforced), the transformation can occur in both directions without loss of data.

The original purpose of the h-map is to allow for both local and remote process state introspection when used in the PalCom service architecture, as well as a means of communication. Therefore, nodes can be annotated with permission bits, denoting who can read and write to them (local vs remote).

In addition to holding values, a leaf may have registered one or more message handlers, which are invoked in order, when a message (local or remote) is destined for the corresponding tree path. This may be regarded as the PalCom service analogy to UDP ports (paths) and sockets (message handlers).

The idea of using a hierachical data structure to store data as well as meta-data and external references is one that I will utilize in my architecture, as it fits nicely with the hierachical data models I am looking to support. The straightforward mapping to and from XML is also very convenient, and allows for easy data exchange between XML-based applications. A modified version of the message handlers will serve as the basis for an event triggering mechanism.

3.3.3 Ruby on Rails scaffold

Ruby on Rails (RoR) [26] is a web-development framework built in the Ruby programming language. The goal is to speed up and simplify the development process of classic web-applications with a database backend which are often very similar in nature. To prevent reinventing the wheel for each application, RoR provides a large number of commonly used features and mechanisms for both client- and server-side code, at the expense of dictating the architecture of the project.

At first glance it has little to do with this thesis, but a particular mechanism is of particular interest: The *scaffold* feature. Given a data model object (usually mapped to an SQL database table of the same name in plural form), RoR can generate a collection of webpages (a scaffold) capable of listing, adding, editing and removing instances of this model type directly from a web browser.

If the model is changed, the scaffold changes also. The specific type and display of each HTML user input element is dynamically inferred from the data type of the corresponding member in the model class.

This kind of direct mapping between the user-entered data and the domain models, is a very convenient one, and I will work on incorporating a similar feature into the framework designed in this thesis. Bridging the gap between the host application and the end user becomes a bit easier for the developer when the list of form elements are automatically inferred from the data models, and actually goes beyond the concept of single authoring.

3.3.4 XPath

XPath [3] expressions provide a convenient way of identifying and querying nodes and values within a XML document. In the simplest case, an XPath expression is a pointer to a named node, or collection of nodes within a XML document.

At first glance, this is an obvious way of referring to form elements, as it is indeed also done in XForms. Nodes may be referenced by both relative and absolute paths, and their parameters and any *Character data*[5] content may also be accessed.

[5] *Special <![CDATA[* nodes holding verbatim data, without the need for character escapes

Apart from selecting nodes, an XPath expression may compute a value of a range of types such as string, integer or boolean. For this purpose, a large collection of functions are provided that operate on nodes and the aforementioned simple values. More than 100 standard functions are described in the official specification of the XPath 2.0 function library [2]. Depending on the implementation, it may be possible to extend the function library with domain-specific functions. This is supported by the generic Java XPath API.

This is not meant to be a tutorial on XPath, but for the sake of illustration, a few expressions and their results are listed here:

```
/foo/bar              // Selects all <bar> nodes nested within the root node <foo>
foo[2]/bar            // Selects all <bar> node nested within the second <foo> node
bar/@baz              // Select the parameter "baz" of all <bar> nodes in the current context
bar[@baz='foobar']    // Select all <bar> nodes having a parameter "baz" with value "foobar"
avg((1,2,4)) > 2      // Calculate if the average of a list of values is greater than 2
```

At the time of writing, XPath 2.0 is the latest version that has W3C Recommendation status. XPath 2.0 is a subset of XQuery 1.0, a more expressive query language.

3.4 Discussion

Although all the works covered here supports a subset of the thesis vision, none of them fits the problem statement entirely. One of the common shortcomings is that they do not conceptualize the form as an entity that is filled out and passed around, but they rather work merely as a way of gathering structured data from a single user and processing the input directly. In other words, the focus is generally only on the user interface and not on how the data should be stored and re-retrieved.

Few attempts are made to automate and integrate the process of pre-adaptation, so that each user (or class of users) will experience the exact same form differently. The very elaborate framework envisioned in the Ph.D. thesis [10] does, as previously stated, perform pre-adaptation, although this is not at the user-level, but at the media/device level. Granted, that this comparison is not entirely fair, since [10] deals with complete web applications as opposed to just forms.

The tool resembling the vision the most is XForms, but it does not support single authoring (the UI needs to be specified) and is very web-centric. However, it will serve as one of the primary sources of inspiration, along with Dynamic Forms, XPath and h-maps.

As for descriptive languages, data structures and processing techniques, I have found numerous examples to follow, which provide a good basis for designing a framework that will support single authoring with multireification. It has become clear that a new declarative language is probably the best approach, and a variation of the h-map data structure seems to support this elegantly. As will become evident, many ideas and techniques will be used in combination.

Many of the works covered has influenced the thesis indirectly with abstract notions, ideas or details beyond what is made explicit in this chapter. Throughout the remainder of the thesis, references will be made to this chapter where appropriate, to demonstrate the origin of certain concrete ideas and concepts discovered during the research.

PART II

FRAMEWORK DESIGN

Discussion of the design of the framework and the prototype implementation

CHAPTER 4

Architecture

This chapter discusses the overall architecture of the proposed framework. Most subjects are covered at a glance and references are given to more detailed discussions in later chapters. The framework in its entirety is denoted AdapForms.

Generally, the framework is built using the traditional MVC (Model-View-Controller) pattern, where the core framework essentially comprises the *Model* and *Controller* parts, by providing the data structures as well as the logic making the model dynamic. The *View* part is UI-specific and thus highly coupled to the display- and interaction-capabilities of the technology chosen. *How* to specifically display data to, and interact with the user, is not the primary concern of the framework. Instead the focus is on determining *what* to display and what to do with the input gathered. An important goal of the architecture is to have the host application manipulate the form and interact with the user exclusively via local proxy interfaces, regardless of how the UI exposes the form and where it is physically being executed. Therefore, this chapter will not go into details about presentation.

4.1 Form lifecycle

In order to describe the software architecture, it is beneficial to define the lifecycle of an adaptive form, so that the various phases can be related to concrete software elements. The major phases stem directly from the abstract definition in the problem statement, but are presented here in greater detail. The *Instantiation* and *Initialization* phases can be thought of as a single phase in the abstract sense, but the technical distinction will be made more clear in the following. In general, the reasoning behind the phase outline will be discussed in the corresponding chapters.

Phase	Purpose and activities
Definition	The form syntax and semantics are defined in a file external to the framework. The XML language for this purpose is one of the primary concerns of chapter 5.
Parsing and loading	The form is parsed and its dependent definitions and objects are parsed, ending up in a single *Form* entity representing the static form, with all its form elements (explicitly specified, or inferred from templates, beans etc.). The form structure may be viewed as an abstract structured tree, where each node is a form element. See figure 4.1 on page 24 for a visual illustration of the process.

Phase	Purpose and activities
Instantiation	A concrete instance of the form is created. Using the static form as a basis, the instance holds current values, validation status etc. of all nodes in the static form structure. The *FormInstance* is essentially holding the collective state of the instantiated form. Multiple users can share the same *Form*, but they will each have a unique *FormInstance*. This is also illustrated in figure 4.1.
Initialization	The form instance is initialized with default values and, if available, stored form data from a previous session. The role of the current user (if any) is also selected at this point. All relevant semantic rules are triggered, causing the form to be pre-adapted as much as possible before the user encounters the form. This phase is elaborated in section 4.5 on page 27.
User interaction	Input from the user (or the host application) is passed to the instantiated form, and the instance may respond with a list of validation errors and form adaptations, triggered by the loaded semantics. This is called an *adaptation cycle*, and will be discussed in section 4.4 on page 27 and chapter 6. Each user interaction triggers a new adaptation cycle. Adaptations may be scripted with XPath expresions and queries. To facilitate complex domain-specific validation and reasoning (possibly requiring lookups in domain databases or similar), the host application may register hooks with the instance, which are triggered whenever a given form element changes its value or state (*observer* pattern). Reacting on user input and performing adaptations is the subject of chapter 6.
Submission	At form submission, the instance is checked for any consistency problems and validation errors, and the event is then parsed (along with the instance) to a callback handler, specified by the host application at instantiation.

4.2 Form paths

A recurring scheme is the use of *form paths*. A path uniquely identifies a form element, and is used throughout the framework for multiple purposes. A few examples follows:

- Look up form elements (form structure tree).
- Access or update the internal state of the form instance (state tree).
- Register form hooks from the host application.
- Define adaptation and validation targets.
- Define semantic rules.
- Exchange unique element identifiers between the core framework and the UI layer.

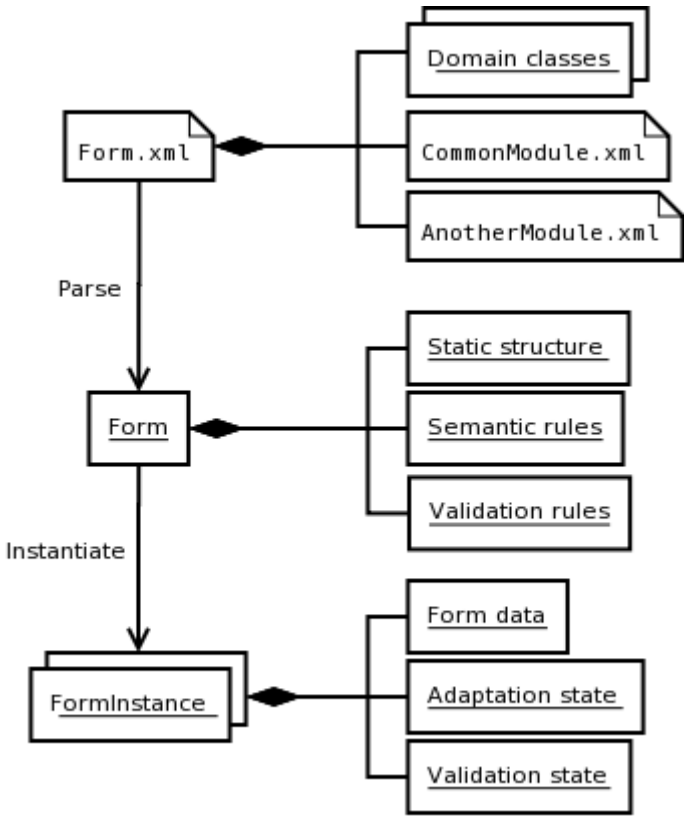


Figure 4.1: Parsing and instantiation of a form

A path resembles the filename structure of UNIX-based file systems, and is heavily inspired by XPath and in part by h-maps, both presented in section 3.3 on page 17. An example form path looks like this: */applicant/spouse/name/first*. Paths may be fully qualified (indicated by a leading slash) or relative to the context they are used in.

For the most part it is possible to create a 1-to-1 mapping between a form element in the static form with the representation of that field in the instantiated form, simplifying things. However, this is not possible in the case of repeated subforms, where the user may specify any number of occurences of a given data type. In this case, the path to the definition of an element in the subform (e.g. */applicant/jobs/company*) will have to map to multiple “instances” of the subform; one for each job entered.

This is solved by adding an extra annotation level to the instance paths, so the paths become */applicant/jobs[1]/company* and */applicant/jobs[2]/company* for the first and second repeat (job), respectively. This syntax is also borrowed from XPath. When making references to the static form definition, the numbering scheme is omitted. When defining semantic rules, both versions may be used, depending on whether it is a general rule or a rule specific to a single repeated entry.

4.3 Element state tree

The form instance holds state for each form element present in the instance[1]. The actual state is held in a hierachical data structure closely resembling the hierachical maps discussed in section 3.3.2 on page 17, using form paths as unique keys.

This is also inspired by how FormGen (cf. section 3.2.1 on page 11) stores state, but with the major difference that the host application need not be implemented against specifically generated tree classes. This decision allows any host application to load any adaptive form and process it at runtime, although it of course may not always be able to do anything sensible with the entered data besides saving or forwarding it. The tradeoff for this approach is that the API exposed to the host application must be more general and therefore may be less natural to use, than if the actual tree was generated specifically for the domain.

At any given time, the following state parameters are available for all elements:

- **Form path**
  The full path to the element in question. This can not be changed at runtime.
- **"Relevant" state parameter**
  Flag indicating whether the element is currently relevant or not. Elements there are not deemed relevant are hidden from the user.

Any form element capable of holding a value, also holds the following state:

- **Element value**
  The current value entered into the element.
- **"ReadOnly" state parameter**
  Flag indicating whether the user may change the value of this element. It is not possible to remove the *ReadOnly* flag of an element, if the current user role is not allowed to write to the element.
- **"Required" state parameter**
  Flag indicating whether the user is required to enter a value into this element.
- **List of associated hooks**
  A list of form hooks currently registered at this path (analogue to h-map message handlers). The hook mechanism will be covered in section 6.1 on page 37. The list of hooks is not exposed directly to the host application, however.
- **Validation problems**
  Current validation problems with the element. Validation is the subject of chapter 7.

In figure 4.1 on the previous page, the state tree is represented by the *Form data*, *Adaptation state* and *Validation state* collectively.

All of the mentioned state entries can be read and manipulated by the host application, with the exceptions explicitly stated above. Changing the state will in most cases trigger an adaptation of the form instance.

Access to reading and manipulating the state of a given element is exposed by the *ElementState* interface. The following simple example shows how the host application could obtain and manipulate a certain form instance:

[1] Including any elements hidden from the user, due to the current user role selection or relevance setting

```
ElementState state = instance.elementState("/path/to/element");
state.setRelevant(true);
state.setValue(42);
```

As opposed to the original h-map, the set of possible nodes is restricted to match the paths dictated by the static form structure, with the exception of *Repeat*s, as previously mentioned. This means that every update to the structure is checked against the form structure. This serves numerous purposes:

- Type safety is guaranteed, as only data of the type dictated by the corresponding form element can be written. When retrieving data from the structure, there is thus no need for type checking, and any type errors are caught early. Automatic data type conversion is applied, where possible and applicable.
- Upon querying or updating a value in the structure, the action is checked against the role permissions of the element. This guarantees that no data is accidentally overwritten by a user not having the correct role.
- It is guaranteed that no "phantom" values are inserted, that are not logically linked to the form due to path typos, logic errors or similar.

Although most of this should already be implicitly guaranteed by the UI layer implementation, it serves as a final centralized sanity check and guards against both programming errors and attempts to craft malicious input. Furthermore, the host application may manipulate parts of the data structure directly, and once again this therefore guarantees a sound state of the structure at all times.

4.3.1 Inheritance

Even though each state tree node has its own set of parameters, these may be overruled at runtime due to the semantic inheritance. Consider the case of two nested form elements; the logical group */foo* and its contained text element */foo/bar*. If the *relevant* flag of the former element is changed to *"false"* this value will propagate to the latter as well.

Mechanisms are provided for the form semantics logic to query and modify both the stored (possibly overruled) and the active value, but regardless of how the parameters are manipulated, the inheritance overrule remains in effect.

Parameter	**Inheritance rules**
Relevant	Only negative relevance values are inherited. This means that if an element is hidden (i.e. is not relevant), so will its children as in the example above. When the flag is reset to *"true"*, this is not inherited to the children, as this would result in the entire subtree becoming relevant, irrespective of their previous setting. Instead, the override is merely no longer in effect, and whatever *Relevant* flags the child elements have will again be active. Explicitly setting a child to *"true"* will have no effect until the override is cleared.
ReadOnly and *Required*	Not inherited. There are both pros and cons to this approach depending on the desired effect; the form author may want to mark an entire subtree as being required, but that would overrule any deliberate non-required flags set in reusable templates etc[2]. It is unclear how to implement this inheritance, as there will always be special cases in which the default is not desired; hence the decision to not support inheritance.

Parameter	Inheritance rules
Validation status	Not inherited per se. However, the framework provides a means of determining if a certain sub-tree of the form[3] has validation errors, and if so, what the collective severity is. Validation is covered in chapter 7. If a concrete UI implementation should choose to do so, this mechanism can be used to implement the *implicit checklist* used by the Dynamic Forms framework (cf. section 3.2.2 on page 12) by exploiting this feature.

4.4 Adaptation cycle

All adaptations within a form instance is initiated by a user action. This means that the form cannot suddenly change in front of the user, unless he or she has triggered the adaptation by changing one or more values in the form, submitting it, or have otherwise actively performed an action.

The term *adaptation cycle* denotes a complete cycle from the receipt of user input (value change, or submission request), and until the user experiences the adaptations. Although the adaptation cycle is always triggered by a single user action, the triggering of form hooks or semantic rules may cause a cascade of adaptations to take place, before the cycle ends. The host application can perform changes to the form instance at any given time. However, these adaptations will be accumulated in a buffer, and will only be presented to the user at the next adaptation cycle to avoid confusing the user.

An *adaptation* is conceptually everything from a value change, validation status change, relevance change, read-only status change to very volatile events such as displaying a message to the user and more UI-specific adaptations, such as redirecting a web user to another website, closing a Swing window or playing a sound.

Figure 4.2 on the next page illustrates the interaction sequences and the role of the iterative adaptation cycles within a control flow context. Each iteration of the outer loop represents a complete adaptation cycle. The inner loop illustrates the fact that a single user-initiated adaptation may trigger further adaptations and form hooks.

Chapter 6 goes into greater detail about what happens in the adaptation cycle.

4.5 Initialization sequence

The intialization sequence of a form instance is responsible for building the internal state tree, and setting/calculating default values and state parameters. However, the sequence is not that straightforward due to the many aspects that can influence the state even during the initialization.

The primary concern is the existance of form hooks, that may be triggered even during initialization.

Three following main approaches have been considered:

- **Initialize the form iteratively, performing evaluations as they are triggered**
  The easiest way to initialize the form is doing it iteratively, dealing with hooks and state evaluations as they are encountered during the sequential traversal of the form.
  This will result in many invoked hooks basing their decisions on values or state that has not yet been initialized, if they depend on other elements than simply the one triggering them.

[2] For example a "name" group may be marked as required, but the nested *middle name* element should still be optional

[3] Not necessarily a proper subtree, as the entire form can also be checked in the same manner. This is in fact used to determine the overall form status

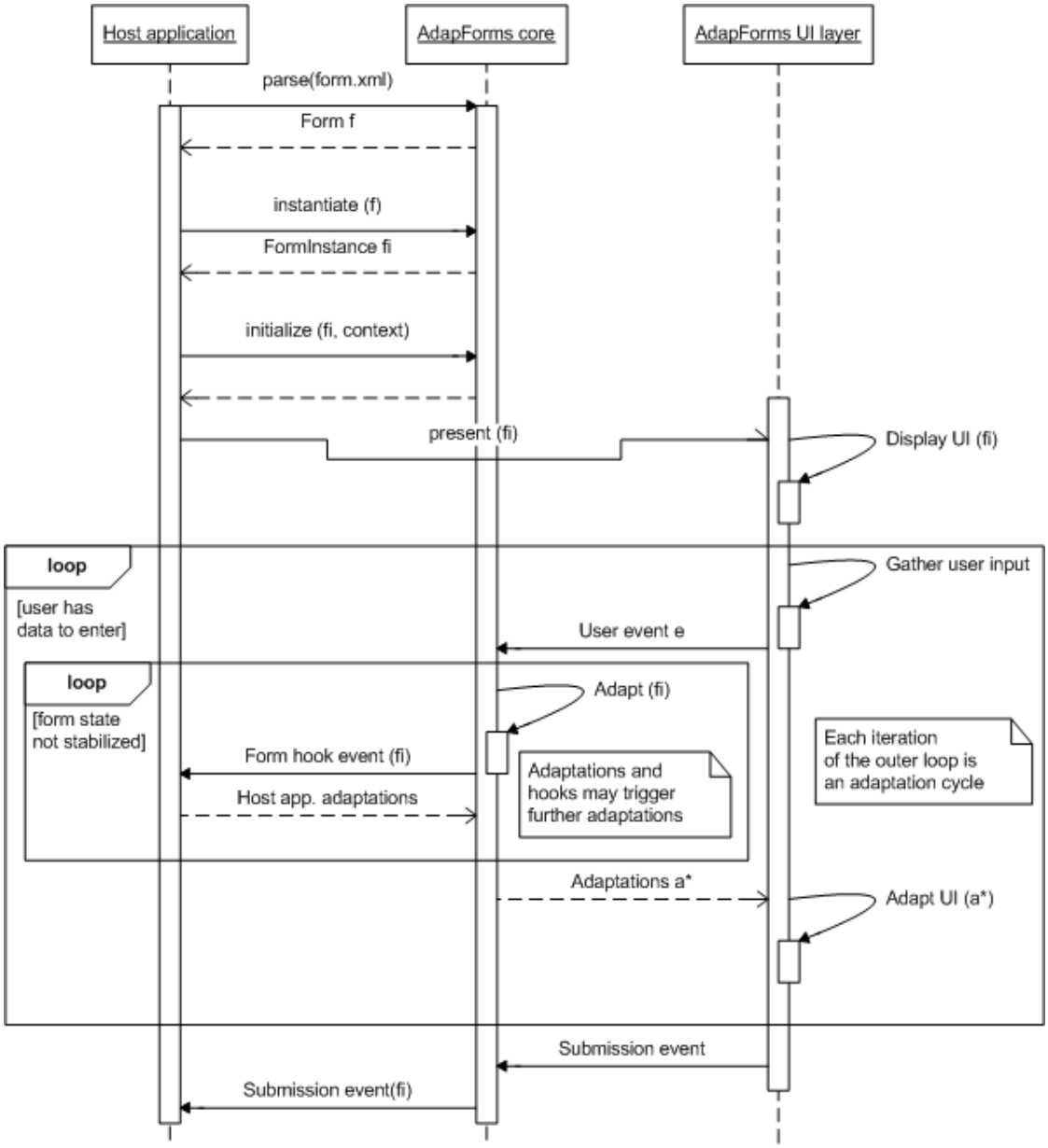


Figure 4.2: Interaction of the major system components

- **Initialize all defaults first, then call hooks and adapt**
  This guarentees that no hooks or adaptation rules are triggered before the entire instance has been at least initialized with the default (or previously entered) values. However, when the first evaluations occur, they may still occur on a wrong basis (outcome produced by other evaluations yet to take place).
  Although the hooks may cause changes to the values in the instance, it is assumed that this will lead to less wrong evaluations than if the hooks where called before all elements where given an initial value.
- **Disallow form hooks to trigger during initialization**
  The obviously simplest solution will be to completely disallow that hooks are called before the

form has been fully initialized. However, this complicates matters for the developer, since he then must actively review all form elements of interest after initialization, to determine if an adaptation should take place
This is especially an issue when loading forms that have been previously filled out partially; the same adaptation should (usually) take place whether the user just entered the value, or the value was just reloaded from storage. A trivial example is to determine whether element *X* should be relevant, based on the value in element *Y*.

Disallowing hooks was not an option based on the discussion above, so the obvious choice is to initialize as much of the form as possible and then invoke the hooks, thus minimizing the *"chicken or the egg"* dilemma. The sequence is described in execution order below:

1. **Create all state tree nodes**
   At creation, each node is initialized with the framework defaults, to make sure they have a value for each state parameter (such as *ReadOnly*, *Relevant*, *Required*, etc). If the form definition supplies a static parameter value (which is often the case), this value will be used instead. The rationale is that this presumably closely resembles the truth, even if they may be changed later by form hooks.
2. **Input values**
   The default element values (or values previously entered and now re-loaded) are inserted into the form sequentially. In the process, only the basic validation code is triggered for each element, updating the validation status for the element.
3. **Evaluate dynamic state parameters**
   All dynamic state parameters are evaluated and any complex validation rules are triggered. This is described in greater detail in section 6.3 on page 40. The rationale for waiting with these evaluations is, that the expression will often depend on the form values entered, and thus should be re-evaluated later anyway.
4. **Execute relevant hooks**
   For each element that has been assigned a value in the previous step, any associated hooks are triggered in registration order.
   At this point in time, we have initialized as many values as possible, providing the best possible premise for the hook to execute. Notice that hooks yet to be called, may still change some of the state the current hook is reacting upon, and the problem has thus not been completely eliminated.
   Whenever a hook has performed changes to the instance, the dynamic state parameters are re-evaluated to keep them updated.
   Hooks may choose to respond differently whether the form is being initalized or is performing a normal adaptation cycle.
5. **Repeat entries are initialized**
   If the form contents were loaded from previous stored data, any repeat entries that were entered, are now initialized by recursively running this procedure on each entry.

Whenever the user adds a new repeat entry to the form instance, the above sequence is also used, as the addition essentially creates an entirely new subtree in the state tree.

4.6 UI-specific view component

The UI layer responsible for interfacing between the user and the core framework is only defined abstractly in the general architecture, as a component with the following primary responsibilities:

- Presenting a pre-adapted form instance to the user.
- Collecting data to enter into the form elements and sending them to the core component.
- Performing adaptations on the presented form, based on decisions of the core component.
- Pointing out validation errors within the presented form, based on feedback from the core component.
- Providing a mechanism for submitting the form, when the user deems it to be completed.

Note that there is no restriction that it must be a graphical interface (i.e. GUI). Any interface, be it audible or tactile for instance, could theoretically be implemented, although the mapping is not necessarily straightforward. The architecture of a proof-of-concept implementation for a web platform is presented in section 8.3 on page 49.

4.7 Internationalization (i18n)

Mechanisms have been added to support multiple locales and languages. When a form is instantiated by the host application, it can optionally pass a *Localization* object to the framework. A localization factory is provided, capable of producing a default (English) localization, and its Danish counterpart. If none is explicitly specified, the default will be used.

A localization provides one or more of the following:

- **Mapping from phrase-keys into human-readable sentences**
  This is used when outputting framework-generated message to the client. For example *"UI_READONLY"* maps to *"Read-Only element. You cannot change this value"* by default. The localization may overwrite any number of these definitions.
- **Mapping from form element labels to their translated versions**
  When rendering form element labels as text, they are attempted converted by looking for a mapping. If no mapping is found, they are outputted in their raw format. This allows the host application to translate its forms into multiple languages without having to change the actual form definition. The same mapping is used for developer-supplied validation problem messages.
- **Java Locale**
  A *Locale* object used, among other things, to display calendars in the proper language (i.e the human-readable names for the months and days of the week), as well as determining whether the first day of a week is Monday or Sunday etc.

CHAPTER 5

Representation of data and meta-data

One of the main areas of the thesis is to find an efficient and easy-to-use way to represent adaptive forms as files that can be easily transmitted and reused (in part or entirety) by various host applications. This chapter is devoted to devise a proposition for such a language. Depending on view, the form description may be thought of as meta-data for the actual input data (element values), which is what the user (or host application) inputs to the form.

5.1 Choice of language class

Based on the discussion in section 3.1 on page 9, two basic declarative approaches for the choice of a form description language has been considered; either use a very specific definition language (the classic approach), or build a language upon XML (the modern web approach).

The advantages of using XML include:

- It is widely adopted and well-known. Most developers are familiar with XML, so no new syntax needs to be learned; only specific tags and attributes (and their semantic meaning) are unknown.
- There are a large number of generic tools available for XML parsing and processing, including XPath, XSLT, XQuery etc.
- The tree-structure of XML documents lend itself to easily structuring of data, and reuse of sub-forms

The advantages of using a specifically designed language syntax include:

- Often, it feels more natural to write in a language that is designed for the purpose, as opposed to a strict generalized syntax such as XML.
- It may be quicker to use a more precise syntax of a specialized language, and the code may be more compact and easily readable.

Based on the above, and the fact that OIOXML primarily encourages XML languages, the XML approach has been chosen.

5.2 Representing form structure

Form structure refers to the static (non-adapted) ordered and nested list of form elements, that collectively make up the available input fields, headlines etc. The structure is built upon the hierachical identification scheme described in section 4.2 on page 23.

Each element is declared as a seperate XML node with a number of attributes. There is a common set of attributes that each element must or can specify, as well as specific attributes depending on the element type in question.

The common attribute set is as follows. Note that the bottom three attributes are applicable only if the element is capable of holding a user-supplied value:

Attribute	Description
id (Required)	The locally unique ID of the element. Uniqueness is required only in the local group. E.g. */address/name* and */person/name* both have the local ID *"name"* but the uniqueness is not violated as their fully qualified form paths are unique.
label (Required)	A textual label that will be displayed next to the element to briefly describe its purpose. For a *Group* or *Bean* element, the label is optional, and if omitted the grouping will be only structural, but no headlines or other visual clues are displayed.
relevant (Optional)	Boolean value indicating the relevance status of the element upon initialization.
uiflags (Optional)	A string value interpreted by the specific UI layer in use. The core framework does not make use the value. An example usage is to specify whether a *choice* element should be rendered as a dropdown list or as radio buttons in the web module; this information is irrelevant for the rather abstract core framework.
read-roles (Optional)	Comma-seperated list of roles that are allowed to read the value of the element.
write-roles (Optional)	Comma-seperated list of roles that are allowed to change the value of the element.
readonly (Optional)	Boolean value indicating if no users are allowed to change the value of the element, i.e. if it is for display/feedback purposes only.
required (Optional)	Boolean value indicating if the element must hold a non-empty value to be valid. The precise definition of "empty" depends on the element type, but usually means either the empty string or *null*.

The name of the XML node must be one of the element types listed in section 2.2.1 on page 7, written in lowercase. An example declaration of a *Text* element looks like this:

```
<text id="firstname" label="First name" required="true" />
```

5.2.1 Attribute inheritance

For each of the optional attributes listed above, the value is inherited from the parent node, if none is explicitly specified. If no parent node exists (the element is on the outer-most layer of the form structure), the value is set according to the most recently encountered <*defaults*> node. This node type has the sole purpose of specifying the default values for the listed optional attributes. For example, to make all following form elements required fields, and only modifiable by the role *"applicant"*, the following can be used:

```
<defaults required="true" write-roles="applicant" />
```

If an attribute on an outer-most element is not specified, and no recent <*defaults*> node assigning a default value for the attribute in question exist, the framework default will be used to ensure that all attributes have a value.

5.2.2 Templates and reuse

One of the design goals was to experiment with supporting reuse of partial forms, and the definition of more complex data-types consisting of multiple simple types. In fact, these can be reduced to the same problem, and solved by allowing the developer to define a *template*, and reference it from various points.

To keep things as simple as possible and avoid overcomplicating the language, a template is basically the definition of a form element, with the exception that it lacks the *id* and *label* attributes. It can thus contain any arbitrarily complex structure (a *Group* containing more elements), or be a simple alias for a refined data type such as an email address. To make the process explicit, the following terms are used:

- **Template definition**
  The template is defined as a seperate entity without structure affiliation, identified by a unique name. It can be considered the definition of a complex data type. Definition is done using the <*template*> tag.
- **Template instantiation**
  The template is applied (or instantiated) at one or more positions within the form stucture (or other template definitions) and is thus given stuctural affiliation, a label and a unique ID. Instantiation is accomplished by the <*use*> tag.

The following example shows two template definitions and one instantiation of each:

```
<template name="simple-address">
  <group>
    <text id="street" label="Street name"/>
    <text id="city" label="City"/>
    <integer id="zip" label="ZIP code"/>
  </group>
</template>

<template name="two-char-string">
  <text maxLength="2" minLength="2" required="true"/>
</template>

<use template="address" label="Delivery address" ref="simple-address" />
<use template="country" label="Country code" ref="two-char-string" />
```

Templates can be defined either in-line in the form definition, or in external module files that can be referenced by path in the form definition, thus allowing both local and global reuse. When instantiated, the elements of the template will have any non-specified optional parameters supplied with default or inherited values, as discussed above, according to the context in which they are used.

A more code-centric reuse method, using JavaBeans, is introduced in section 5.4 on the next page.

Recursion

Recursively defined data structures may pose a problem in the case where a template either contains an instantiation of itself, or another template that in turn references the first template. Both of these conditions would cause an endless recursive definition of the form structure, only possibly terminated by a negative *relevant* state (causing all subsequent instances to be hidden) or a *Repeat* element, that must manually be expanded by the user.

However, this would still require the static representation of the form to handle the recursion, and a logical error in the form semantics could lead to infinite unfolding of the structure, causing overflows. Due to this - and the fact that I can think of only limited example uses for this feature - the language does not allow this kind of recursion.

One possible solution to enforce this would be to use a stack and try to unfold the structure, raising an error if the same template is seen twice. I found it to be simpler, though, to simply disallow forward referencing, so that only templates declared earlier in the input file (or includes) may be referenced, much like how C compilers work. This is also easier to implement, as the parsing can be done in a single pass.

Definition prior to instantiation is therefore a restriction in the language, but there is no loss of generality compared to the aforementioned approach, since the form author can just reorder the template definitions: Any pair of templates *X* and *Y* can be re-arranged to allow for the desired instantiation pattern, since they cannot both depend on the other. By moving either *X* or *Y*, other templates (dependent on the moved template) may need to be moved also, but the same argument can be applied iteratively.

5.2.3 Proposed solution

The complete language is defined formally in appendix F, in shape of a XML Schema Definition (XSD). The parser will reject any forms that does not comply with the schema or contains references to modules that does not.

The namespace of both forms and modules is *“http://adapforms.gammelmark.eu/ns”*, and the *version* attribute must have the value *“0.1”* to be compatible with the definition at the time of writing.

5.3 Representing and handling input data

5.3.1 Internal representation

Internally the data is stored in the state tree, as discussed in section 4.3 on page 25.

There is a special situation, in which the data entered by the user does not match the data in the internal data structure; namely, if the user has entered an invalid value, that cannot be converted to the

proper expected format. For instance, the user has entered a date in the 13th month, or entered a string where an integer was expected.

In this case the framework is unable to store the value internally because of the type safety. This is not deemed to be a problem though, since the value is not valid, and the user will be notified via a validation problem, and any existing value will be cleared. Semantic rules or validators will never be exposed to the wrongly typed value, while that would most likely cause a runtime error.

5.3.2 External representation

Due to the hierachical nature of the data structure, serialization to and from XML is straightforward, thus allowing the host application to easily save an entire instance in textual form to disk, and reload it at a later time, whether the form has been completed or not.

Furthermore, it allows XML-based applications easier integration with the framework, as the XML nodes can be moved directly between application and framework (and thereby indirectly the user interface). Because the developer is free to design the form structure (and thus the data model) as needed, within the boundaries of the form language, in many cases no XML transformation is needed, simplifying the integration. In practice this access is provided by superimposing the Java DOM API interfaces upon the state tree, essentially making the state tree behave like an XML document. This feature is covered in detail in section 6.3 on page 40.

For legacy host applications or to ease integration with HTML, SQL or similar technologies that do not natively support rich hierachical data models, mechanisms are provided to query and update the form data as (key, value) pairs.

5.4 JavaBean integration

Apart from manually defining form elements in the description language, and reading/writing the value of each in turn, another mechanism is supported in the form of JavaBeans[1]. This is inspired by the way the *scaffold* mechanism works in Ruby on Rails (cf. section 3.3.3 on page 18).

To include a bean in the form structure, you simply add a <*bean*> tag, as in the following example:

```
<bean id="address" label="Address" type="myapp.model.Address" />
```

The result is a group node in the form structure with the id of *“adress”*, which contains an element for each public property of the bean. *String* properties become *Text* elements, etc. The supported types are: *String* (Text element), *int* or *long* (Integer element), *float* or *double* (Decimal element), *boolean* (Toggle element), *java.util.Date* (Date element) and *enum* types (Choice element).

When inserting values into the form, the host application may simply pass a bean with the relevant data. Conversely, the host application may read the values entered by the user into a bean with a single call.

Beans consisting of other beans are not supported at this time, however useful it might be. This would raise a number of recursion issues and complicate the reflection mechanisms, but is a topic worth investigating as “dotting” through Java domain classes could map directly to a hierachical data model, if type reference loops are disallowed.

[1] A bean is basically a Java class with a collection of matching get- and set-methods with each pair controlling a stored value, and a public zero-parameter constructor. A typical bean does not have any (or only limited) logic code, but merely holds a set of data fields

5.4.1 Annotating beans

A downside to using beans is that the usual attributes cannot be specified. For instance, it is not possible to name the element something else than what is inferred from the get- and set-methods, and it is not possible to specify whether the entry is required or optional.

To compensate for this, the most common attributes can be specified, by using Java Annotations applied to either the bean class (to define defaults) or to the property get-methods. Consider the following example:

```
@AdapFormsProperty(label="ZIP Code", required=false)
public int getZIP() { ... }
public void setZIP(int zip) { ... }
```

Here, the property *"ZIP"* has been annotated, so that when encountered by the form parser and a form element of type *Integer* is created, the element will have a more human-friendly label, and it is specified to be optional.

Adding annotations like this allows the developer to control how the beans are interpreted by the framework, without changing the actual host application code considerably by renaming or adding methods and interfaces.

5.4.2 Implications on data consistency

As there are obvious benefits for the form author in terms of automatically generating parts of the form, and in essence tie user input directly to the object model of the host application, there is a downside: Any changes to the object model will, in many cases, render any stored completed forms useless (at least if they are stored in their raw XML form), since there will be a mismatch between the expected form structure, and the actual structure.

The use of implicitly inferred forms also means that it will become difficult to store the forms and their data for archival purposes, if the data model changes over time.

I have not been able to find any obvious solution to these issues, except for creating a mechanism for "compiling" a form, that will output its equivalent with the bean elements expanded. This could be coupled to a versioning system of sorts. For archival purposes this will do fine, but if the forms are to be reused at a later time, there may be mismatches still.

CHAPTER 6

Semantics and adaptations

This chapter is devoted to discussing the subject of making a form dynamic and adapt itself based on input values. Different techniques are presented, most notably of which is the application of XPath expressions to describe complex adaptation semantics.

6.1 Form hooks

As briefly mentioned earlier, the host application may interfere with the adaptation cycle via a hook mechanism. The hook is defined by the Java interface *FormHook*, and consists of the following event receivers:

```
void onValueChange(FormInstance instance, FormPath path);
void onRepeatEntryAdd(FormInstance instance, FormPath path, int entryID);
void onRepeatEntryRemove(FormInstance instance, FormPath path, int entryID);
```

onValueChange is called whenever a value changes, whereas *onRepeatEntryAdd* and *onRepeatEntryRemove* are used to monitor addition and removal of repeat entries. The precise semantics of the event methods can be found in the JavaDoc documentation.

Each hook object must be registered with a specific form path in the instance, before the instance is initialized. In this way, the hook receives only the events relevant to it. However, because each hook method is called with parameters designating the instance and path for which the event was triggered, the same hook may be registered at multiple paths. In the rare event that the developer wants the same hook to receive events for a large number of form elements, the hook may be registered with no path, allowing it to receive all events.

Form-level events, like submission of the form, is handled by the callback defined in the *InstanceCallback* Java interface. As opposed to the hooks, only a single callback may be defined for the instance, which must be defined before initialization.

At each triggered event, the host application is free to modify the values and state of all form elements in the instance, thus triggering adaptations.

6.1.1 Redistribution

If the form is to be distributed among multiple parties, it is no longer sufficient to just distribute the XML file containing the form definition. All the hooks and their registrations also need to be passed along. To make things easier, the interface *FormBehaviour* is supplied. A class implementing this interface can be shipped with the XML file and could contain all hooks and related logic. The interface defines two basic event receivers:

```
void preInitialize(FormInstance instance);
void postInitialize(FormInstance instance);
```

The XML file can then specify the name of the class, which will be loaded and executed upon form instantiation. This approach has the added benefit that the secondary host application does not need to know about the hooks, and where to register them, since this is done transparently.

6.1.2 Downsides

Even though form hooks can perform any possible combination of adaptations and complex calculations[1], a number of downsides or limitations have been identified.

First of all, they are tedious to implement, as they must first be implemented as a (possibly anonymous) class implementing the proper interface, implement the desired logic imperatively, and then finally they must be registered at the proper form paths to ensure that they are triggered. This produces a lot of code for something that may conceptually be a very simple operation.

As stated above, the hooks must be distributed along with the XML file, which both limits the transparency of what is going on inside the code, and also makes it impossible to instantiate the form on any other platform than Java, unless multiple authoring is applied, violating the vision of the thesis.

Finally, there is no obvious way to apply hooks transparently to the templates discussed in section 5.2.2 on page 33, since the hooks need to be registered to each explicit path, at which the template is applied.

6.2 Language for semantic rules

One of the main aims of the thesis is to provide an easy, yet powerful, way to define semantic rules in the form, automating the form adaptation. So far only form hooks has been discussed, and although they are sufficient in terms of expressivity, they do suffer the downsides discussed above.

The goal is to find a small script-like language that allows the form author to express adaptation semantics with minimal effort and syntax usage, preferably platform-independent. The general idea is to be able to specify an evaluable expression in place of constants in the XML form definition. For example, the *Relevant* state parameter could have an expression attached to it, that will be re-evaluated whenever the premises for the expression changes[2].

[1] Since arbitrary Java code can be executed
[2] Dependencies on other element values and/or state parameters

6.2.1 Rule engines

The original idea was to use a business rule engine to infer the dynamic state parameters, taking advantage of the powerful engines already implemented. The Java community has established a Java Specification Request for an industry-standard API for rule engines, denoted JSR-94 [1].

Generally a rule engine works by storing a set of *facts*, collectively known as its knowledge base. To this comes a set of *rules*, which make up the logic. The idea is, that one presents the engine with known facts (element values and state parameters in this case), and whenever a fact changes, the relevant rules are triggered, possibly yielding a series of changes to the knowledge base (carried over directly to a list of adaptations).

Jess

I first looked at the JSR-94 framework called Jess [30]. Jess provides both forward and backward chaining and has a very versatile rule- and fact-description language with LISP-like syntax. Also supported is a XML version of the language denoted JessML, which is basically the same language wrapped in XML elements. Both languages are rather cumbersome to learn, however, and thus requires some study before the most basic rules can be defined properly.

Consider the following example rule from the benchmark form, that simply causes the *display-state* fact to reflect whether the value of *country* is *"USA"* or not. The value of the fact would then be mapped to the *relevant* state parameter of the *state* element.

```
(defrule display-state-only-if-usa
  ?fact <- (basic-facts)
  =>
  (bind ?isUS (eq ?fact.country USA))
   (modify ?fact (display-state ?isUS))
)
```

The rule further assumes that the mentioned fact has been defined and bound. Even though some of the fact generation may be automated by the framework, this is obviously much too complicated for simple tasks, and requires that the form author learns about rule engines and the Jess language[3].

Drools

Drools [18] is another example of a rules engine, also adhering to the JSR-94 standard. Besides the obvious advantage of being free to use, it also supports direct integration with Java, treating objects as facts.

The Drools language is XML-based, but interestingly it allows its host applications to superimpose a domain-specific language (DSL) on the existing language. Although this improves the learning curve by partially hiding the rule syntax behind domain concepts, it still requires knowledge about the way facts and rules are treated, in order to create more than basic rules.

Based on these experiments, I have decided not to use the a rule engine for the purpose. Although they theoretically fit the task nicely by mapping facts to state parameters and yielding change events transparently, in practice it seems too complicated to write the rules to be of much use.

[3] Granted that this could probably be written somewhat more intelligently, after some further study of the language, the point remains

6.2.2 Scripting language

I briefly considered using a simple scripting language for defining semantic rules, possibly using an open source ECMAScript/JavaScript interpreter. However, this approach suffer many of the same symptoms as using a rule engine, in that it requires the developer to learn another language (although presumably simpler and more familiar than the rule engine syntax), and the rule code is not very compact and human-readable. Although a very simple language may be found, this contradicts the recommendations made in section 3.1 on page 9, suggesting that a declarative approach should be chosen.

6.2.3 XPath

Since form paths essentially are simple XPath expressions, as described in section 4.2 on page 23, the use of XPath may be expanded to also serve as the language for defining semantic rules. It is assumed that this syntax will be more simple, and indeed more familiar to the developer, than any rule-based system. In fact, the form path scheme is a proper subset of the XPath language.

The semantic rules in shape of XPath expressions defined this way, will be less general than form hooks, in that they can affect a single state parameter only (only one output value), and have less expressive power, but they are hoped to facilitate the majority of the adaptation semantic rules a form author will encounter the need for.

Another advantage of using XPath is, that it is specifically designed to work on hierachical tree-structures, as opposed to the knowledge base of a rule engine. This means that expressions can be made relative to a certain node, allowing the same expression to be used in multiple places, making it ideal for defining semantic rules even on templates that are reused at different locations within the form.

Based on the above, XPath has been chosen as the semantic language of the framework. This closely resembles the way XForms uses XPath expressions, cf. section 3.2.3 on page 13.

6.3 Applying XPath expressions to the state tree

A XPath engine needs a data structure to operate on. The Java SDK (JDK) includes APIs for a range of DOM [11] tools, including XPath engines, which I will be using in the framework.

For this to work, a mapping from the state tree onto a DOM document is needed. The naive approach would be to run through the state tree, building a document in parallel, and when completed, evaluate the XPath expressions on the new document. This is easy to implement, as only existing features are used, but it has the significant downside that the tree should be rebuilt every time a value or state parameter changes. An obvious improvement would be to keep the two state trees synchronized at all times, but this requires a lot of extra bookkeeping when updating the structure from multiple sources and contexts.

The solution chosen is to actually make the state tree become a DOM document implementation itself. In practice this means that the state tree implements the *org.w3c.dom.Document* interface, while the tree nodes implement *org.w3c.dom.Element*. The DOM API is quite complex with a large number of methods being part of the standard. However, it is possible, with limited efforts to create a read-only document, throwing “not supported” DOM exceptions where appropriate.

The end result is a state tree that for all intents and purposes acts as a read-only DOM document, allowing XML tools such as XPath and XQuery to operate directly on it. As a bonus, the host

application has gained a more direct way of accessing the tree, supplementing the *ElementState* interface and XML serialization dicussed previously. Indeed, if desired, the state tree can be written as an XML file to disk using standard DOM libraries.

With little extra effort, the tree nodes have been modified to allow updates of the same parameters as the *ElementState* exposes, with the exception of validation problems. In essense this means that the host application may navigate to a DOM node and directly manipulate (naively or using XML tools such as XSLT) the attributes of the node, corresponding to state parameters. Recall that the DOM element is simply an interface built on top of the state tree node, so the same data conversion and integrity checks are performed and adaptations are triggered in the usual manner transparently. The DOM structure itself cannot be modified, as it is dictated by the static form structure.

If serialized to XML, an element state looks like the following. Notice that the element value can be accessed in two ways, depending on preference:

```
<firstName required="true" readonly="false" relevant="true" relevantRaw="true" value="John"
    hasvalue="true" isvalid="true">John</firstName>
```

Some of the parameters deserves a note. *relevant* is the inherited relevance (the value “in effect”) whereas *relevantRaw* is the actual value stored in the node. *isvalid* indicates if there are any validation problems associated with the element. *Date* elements have an additional parameter called *datevalue*. This is the value of the stored date (if any) in the format expected by XPath. When using date- and duration-related XPath functions, this parameter should always be used to ensure correct localization-neutral date parsing.

Container elements (*Group* etc.) are nested in the obvious way, since they cannot hold values.

A special situation occurs when working with *Repeat* elements, since we need a way to address both the actual repeat element (e.g. at path */repeat*) and its concrete repeat enties (e.g. */repeat[1]* and */repeat[2]*). However, recall that the XPath expression */repeat* yields *all* elements named *repeat*, including all the repeat entries. Therefore, to address the element itself, one should use */repeat[1]*, and the entries will have their entry ID incremented, so the first and second entries is addressable by the */repeat[2]* and */repeat[3]* expressions, respectively.
Caution should be taken when making decisions based on the number of entries, since the value will be off by one if done naively. To obtain the correct count, either subtract one (i.e. *count(/repeat) -1*), or use one of the nested elements to count instead (i.e. *count(/repeat/text-element)*), since the repeat element itself has no children, but provides only state parameters for inspection.

The XPath as defined for DOM documents, denote absolute paths from the root of the form, including the root node. This does not go well with the way paths are specified within the framework; there can be only one root node of a wellformed XML document, and thus all paths would have to be prefixed with the name of that node. *“/applicant/name”* and *“/jobs[2]/title”* would have to become *“/form/applicant/name”* and *“/form/jobs[2]/title”*, respectively.

To circumvent this when evaluating XPath expressions, the DOM document simulates having multiple root nodes, in part by acting as a *document fragment*[4].

6.3.1 Incorporation into form definition

Because the expressions are usually very compact, and affect only a single state parameter, they may be incorporated directly into the XML nodes in place of the constant boolean values. For instance, consider the following example from the benchmark form:

[4] A fragment is basically a collection of well-formed DOM trees, used to temporarily hold or move sets of nodes between documents. The fragment itself is not a wellformed DOM tree, since it does not have a document/root element

```
<group id="spouse" label="Spouse" relevant="../marital = 'Married'">
<!-- Group contents left out for brevity -->
</group>
```

Here, the *relevant* state parameter is not a constant, but a short XPath expression that will be *"true"* iff the value of the element *marital* (Marital status choice) holds the value *"Married"*. Notice the use of relative paths, which makes it possible to use in templates too.

When evaluating on expression, the XPath engine is asked to produce a string from the expression. If the string is either *"true"*, *"false"*, *"yes"* or *"no"* (case insensitive), this value is converted to the corresponding boolean counterpart. If a different or no string is returned, the XPath engine is asked to re-evaluate the expression as having a boolean return value. If the evaluated value does not make sense as a boolean (e.g. if the expression selects a range of XML nodes or a string), the returned boolean will depend on the situation. For instance, *"true"* if at least one node or non-empty string was selected. Refer to [3] for details.

The full feature set of XPath 2.0 [3] is supported when designing AdapForms semantic rules. For more examples of what can be accomplished with XPath, consider the following rather peculiar XML form snippet, whose sole purpose is to illustrate the expressive power of the semantic rules:

```
<repeat id="numbers" label="Enter some numbers between 1 and 10" entryLabel="A number">
  <integer id="aNumber" label="Pick one" minValue="1" maxValue="10"/>
</repeat>

<helptext id="text-highavg" label="The average is quite high!"  relevant="avg(../numbers/
    aNumber) > 7.5"/>

<text id="foo" label="Foo" readonly="../text-highavg[@relevant = 'true']" />
<text id="bar" label="Bar" readonly="../foo/@readonly" relevant="contains(upper-case(../foo),
    'TEST')"/>
<text id="baz" label="Baz" required="count(tokenize(concat(../foo, ../bar) , ' ')) >= 3"
    readonly="@required = 'false'"/>

<helptext id="help" label="Something is wrong with one of the numbers entered above"
  relevant="/numbers/aNumber[@isvalid = 'false']" />

<date id="date" label="Date" />
<helptext id="date-in-past" label="The specified date is more than 20 days in the past"
  relevant="../date/@hasvalue='true' and days-from-duration(current-date() - xs:date(../date/
      @datevalue)) &gt; 20" />
```

The example explained:

- *text-highavg* is displayed iff the numerical average of the integers entered in the *numbers* repeat list is greater than 7.5.
- The *readonly* attribute of both */foo* and */bar* are copied over from another attribute. The expressions represent two ways of doing the same thing (the only semantic difference being that they copy their value from different attributes); the first is a proper boolean expression, and the second is a string that happens to be either *"true"* or *"false"*, in which case the framework will treat it as a boolean.
- *bar* is relevant iff *foo* contains the substring *"test"* (casing does not matter).
- *baz* is required iff the concatenation of *foo* and *bar* combined contains at least 2 spaces (which tokenizes into 3 strings). It is read-only iff it is not required.
- *help* is relevant iff one of the */numbers/aNumber* instances have validation problems.
- *date-in-past* is relevant iff a date is specified in the *date* element, which is more than 20 days in the past, counting from today.

6.4 Adaptation cycles revisited

Unfortunately, since the framework merely uses an external library to perform the XPath evaluations, there is no way to tell whether a given adaptation causes any of the semantic rules to be outdated, and thus in the need of being re-evaluated. Because of this, all semantic rules are re-evaluated every time an adaptation occurs. If one of the rules trigger an adaptation itself (evaluates to a new value), the process is restarted to propagate changes to all expressions (and form hooks).

In very large forms with many semantic rules or hooks, this may be quite a time-consuming process, and if the framework was to be useful in practice this probably needs to be improved, possibly in shape of a native XPath interpretation engine that can make informed decisions on which expressions may have had their evaluation premises altered. This can presumably only be a conservative estimation, but it could potentially improve the performance in case of many cascading adaptations. Given that XPath 2.0 is not Turing complete[5], there is certainly a basis for further investigation.
In practice however, a single user action will likely not cause more than a few adaptations, but the general case is another story.

6.4.1 Termination

A single adaptation may evidently trigger a cascade of other adaptations. Since the host application can carry out complex calculations in the Turing-complete Java language, a single cycle potentially involves recursion or loops. Thus, by the *Halting Problem* [12, pp. 414-416], it cannot be proved that all adaptation cycles will terminate. If the developer is not careful, endless loops may occur, causing the executing thread to terminate due to a stack frame overflow or simply never return.
This can be visualized as the inner loop of figure 4.2 on page 28 never terminating, because the form state is constantly changing.

The addition of XPath expressions only makes matters worse, and indeed two thoughtlessly written semantic rules alone can cause a loop to occur. That being said, this kind of interdependency probably indicates an overly complicated form design (or an error in its definition), and can for the most part be avoided by the developer putting some thought into the collective semantics of the form, and thoroughly testing the form with various inputs or use cases.

Heuristics may be employed to detect and break this kind of loop; for instance if the same form elements are alternating between similar states, the cycle may be forcefully broken by a runtime exception or timeout. However, this could potentially compromise the integrity of either the form instance or the host application, and I have therefore chosen not to do this. I deem that a definitive crash (or unresponsiveness) of the instance is easier to spot and remedy than a possibly unnoticed data inconsistency, if exceptions are silently ignored within a hook method or merely logged for future reference. This could potentially have severe consequences when the data is later applied to business logic.

[5] Although interestingly it has been proved that XSLT 1.0, heavily relying on XPath, is Turing complete [31]

CHAPTER 7

Validation and feedback

Input validation is a vital part of the framework, as it ensures that the host application receives valid data on form submission. Part of this task is to let the user know when something is filled out incorrectly. These, and related, issues are the subject of this chapter.

7.1 Concepts

A validation state is associated with each form element. The state consists of a (possibly empty) list of validation problems that refer to the specific element in question.

Each validation problem is assigned one of the following severity designations: *Error*, *Warning* or *Required*. Besides the severity designation, each validation problem holds a short human-readable text message describing the problem.

7.1.1 Validation severity

The severity *Error* indicates that something is wrong with the element; for example that the entered value is not valid in the context, has an incorrect format or similar. A *Warning* indicates a potential problem or acts as a "helper" to the user; for example a warning telling the user that he probably meant year 2009 instead of the entered 1009.

The validation severity denoted *Required* has been introduced to distinguish between a regular warning and if an element simply does not hold a value yet, even if it is required. This allows the host application to distinguish between a form that contains actual problems (erroneous or dubious input) and a form that simply has not been filled out entirely yet, but holds only valid data.

The collective severity of a single form element is the "worst" severity (ordered as listed above) of all the problems reported for the element. For instance, if an element has one *Error* and one *Required* problem, the collective severity is *Error*.

7.1.2 Form status

An adaptive form can be in one of a number of states, depending on the current validation status. Each form instance has two status codes, to distinguish between the complete form status, and the status for the part visible (defined by the user role). Even though part of the form is hidden from the user (because of the user role), the entire form is loaded and processed within the state tree. This was chosen to make sure that all semantic rules still are valid and in effect. The user is only presented with the validation status of the visible part of the form.

The form status is inferred after each adaptation cycle, and is determined by considering the cumulate validation severity of each form element that is currently relevant (i.e. not hidden).

Form status	**Explanation**
Uninitialized	The instance has been created, but it has not yet been initialized, and thus cannot handle user input yet.
Initializing	The initialization process is currently taking place.
Invalid	The form is currently in an invalid state, meaning that it contains one or more validation errors.
Incomplete	The form is partially completed. Some element values still needs to be supplied but no validation problems remain (except for *Required* problems).
IncompleteWithWarnings	The form is partially completed. Some element values still needs to be supplied but no validation *Error*s remain.
Valid	The form is currently in a valid state, meaning that its contents conform to all rules and validation criteria.
ValidWithWarnings	The form is currently in a valid state, meaning that its contents conform to all rules and validation criteria. However, one or more warnings are associated with the form.

On form submission, the host application may choose to accept or reject the submission request depending on the current form status. In most cases, only *Valid* forms would be accepted, but the host application may choose to e.g. store partially completed forms or similar. Also, the host application is notified every time the status changes via the instance callback.

7.2 Detecting validation problems

Validation problems can be indicated primarily by three sources, or *validation levels*: basic framework validation, validation rules and the host application. The final list of problems associated with a form element is the union of these disjoint collections. The two mechanisms do not interfere with each other, and can thus internally be thought of as two separate validation systems, although externally they appear as one.

7.2.1 Basic framework validation

The basic validation occurs at the element level, according to the attributes specified in the form definition. This includes, but is not limited to, the following:

- **Required field validation**
  For each required element, it is checked that the element actually contains a value.
- **Type checks**
  Integer elements must contain integers, Date elements must only accept dates etc.
- **Range checks**
  Valid integer inputs may be upper- or lower-bounded, as may the length of text element values. Choice elements can only take the value of one of its listed choices, entered dates must actually exist etc.
- **Format checks**
  Text elements may dictate regular expression patterns, dates must be supplied in the correct format etc.

All the built-in validation rules will trigger when the form is first loaded, and whenever the value of the element is changed. To avoid inconsistencies, it is not possible for the host application to influence or overwrite these mechanisms. Some examples of basic validation are listed below:

```
<text id="foo" label="Foo" maxLength="50" />
<integer id="bar" label="Bar" minValue="0" maxValue="20" />
<text id="baz" label="Baz" required="true" pattern="[a-z]{1,10}([^/]*)?" />
```

7.2.2 Complex validation rules

Since mechanisms for handling boolean XPath expressions are already supported (as discussed in section 6.3 on page 40), it is natural to reuse this feature to define validation rules. Instead of simply adding a new XML parameter called *validation*, taking an XPath expression, each value-holding element can be assigned a list of validators. The primary reason for this, is that users will not benefit from simply knowing that something is wrong; they need to be informed *what* is wrong, so a problem description must be supplied.

More than one value calls for a new XML node <*validator*> to logically group them, and the expansion from one validator to multiple is straightforward. In addition to the mentioned parameters, a validator can also be supplied with a severity designation, which can take any of the values previously described, although *Required* is probably not of much use in this context.

A demonstration of the validator syntax is found below. Note that the example combines both basic validation attributes and XPath validators:

```
<text id="foo" label="Foo" maxLength="50">
  <validator criteria="lower-case(.) = ." message="Must be all lower-case" />
  <validator criteria="@hasvalue='false' or contains(. , 'hello')" message="Must contain the
      substring hello" />
</text>

<integer id="bar" label="Bar" minValue="0" maxValue="20">
  <validator criteria=". &lt;= 10" severity="Warning"  message="Value should normally not be
      above 10" />
</integer>
```

Notice that, like the semantic rules there are no restrictions on what can be expressed by the rules. Although validators in most cases validate the value of the element they belong to, this need not be the case. An obvious deviation of this is a validator that makes sure a “start” date is chronologically earlier than its corresponding “end” date, or a validator that makes sure that two *Secret* elements have the same value (common typo avoidance technique). This also means, that all validators are triggered each time the state of the form changes, regardless of whether the changed element is the one to which the validator is attached[1].

The suppiled problem message is attempted translated via the current *Localization* using the same rules that govern element labels.

7.2.3 Domain validation

As a last line of defence, the host application may set or remove validation problems for each individual form element. This is an optional addition to the internal framework validation, and provide means of more complex validation that may rely on external logic, database lookups etc.

As opposed to traditional light-weight validation frameworks, such as Apache Commons Validator [22], validation is not performed by assigning a collection of validation rules/objects to each form element, and checking them sequencially when the value changes. The reason being that most of the basic validation rules (such as range checks and *Required* validation) is already present in the framework. What the host application needs to validate, is presumably often more involved and may involve the value or state of multiple elements at the same time.

Therefore, It is up to the host application itself to decide and control when to perform its validation checks. An obvious solution would be to add form hooks to the relevant elements, and perform the validation whenever one of their values change, but the framework makes no such restriction. In fact, the validation status can be changed for any element at any time; whether the element just had its value changed does not matter.

7.3 UI feedback

At the end of each adaptation cycle, for all form elements that had their validation state changed during the cycle, the new state (and list of problems) is given to the UI layer in shape of validation adaptations.

How the UI presents the problems to the user depends on the implementation selected, but an obvious choice is to display it visually using icons next to the form element. See section 8.3 on page 49 for a concrete suggested solution for the web platform.

The host application is capable of changing the *submit action title* at runtime, providing the user with additional indirect feedback about the form status. As an example, the title of the submit button[2] can be changed to e.g. *“Store partial form”* or *“Submit completed form”*, depending on the current form status.

Additional feedback methods may be available depending on the UI layer in use.

[1] This aligns with the arguments made in section 6.4 on page 43 with respect to the semantic rules being re-evaluated often

[2] Assuming the media presents the submit-capability as a button

CHAPTER 8

Implementation

8.1 Purpose and current status

The purpose of the implementation is to create a codebase for experimentation, and as such emphasis is not on security and performance, but rather on functionality and proof-of-concepts. However, all non-temporary code is written with practical use and maintainability in mind, so that it can be refined and improved if needed.

The protoype is written in Java 5.0.

8.2 Components

The implementation currently comprises the following major assemblies, technically represented as individual Eclipse [25] projects:

Project	Purpose and contents
adapforms-core (Mandatory)	Contains all the core generic components of the framework. This includes internal form representation, state tree, parsers, adaptation engine etc. The component is UI-independent and compiles into a JAR (Java Archive) file which must always be deployed when working with AdapForms.
adapforms-web (Optional)	Web-specific components. This allows adaptive forms to be applied to Java-enabled websites. The components are responsible for rendering the form to a web browser, and mediating user events and form adaptations between clients and the core framework residing on the server. It compiles into a JAR file that can be deployed to J2EE web applications along with adapforms-core.
adapforms-test (Development)	Unit tests and classes related to testing and experimentation.
adapforms-examples (Development)	Example Eclipse Dynamic Web Project, featuring complete JSP and Servlet examples. Compiles into a WAR (Web Archive) file that can be deployed directly on a J2EE application server like Apache Tomcat [23]. The project also includes a collection of example XML forms.

8.3 Web-based interface

The purpose of the web component is, as previously mentioned, to expose the adaptive forms to web browsers.

The renderer outputs an XHTML 1.1 input form in an appropriately formatted table. Each simple form element is transformed into a label and a XHTML input control.

It is a problem that XHTML does not have a concept of nested form elements; it merely supports a flat structure with named elements. To remedy this, the elements are graphically rendered with an indentation level which matches its depth in the structure tree, and technically the ID of a XHTML form element is simply the complete form path to the element, with some character ranges being escaped, to maintain XHTML compliance. This closely resembles the way the same issue is solved in FormEncode, cf. 3.2.4 on page 16.

Figure 8.1 shows the interface in action. The form can be styled either by the default CSS (Cascading Style Sheet) file served by the servlet, or by a user-supplied stylesheet. Apart from this, there are no means of influencing the look-and-feel of the output, besides creating another UI implementation for the core framework. Please note that the screenshot reflects the GUI as originally designed; section 9.1.2 on page 59 proposes some changes following end-user feedback.

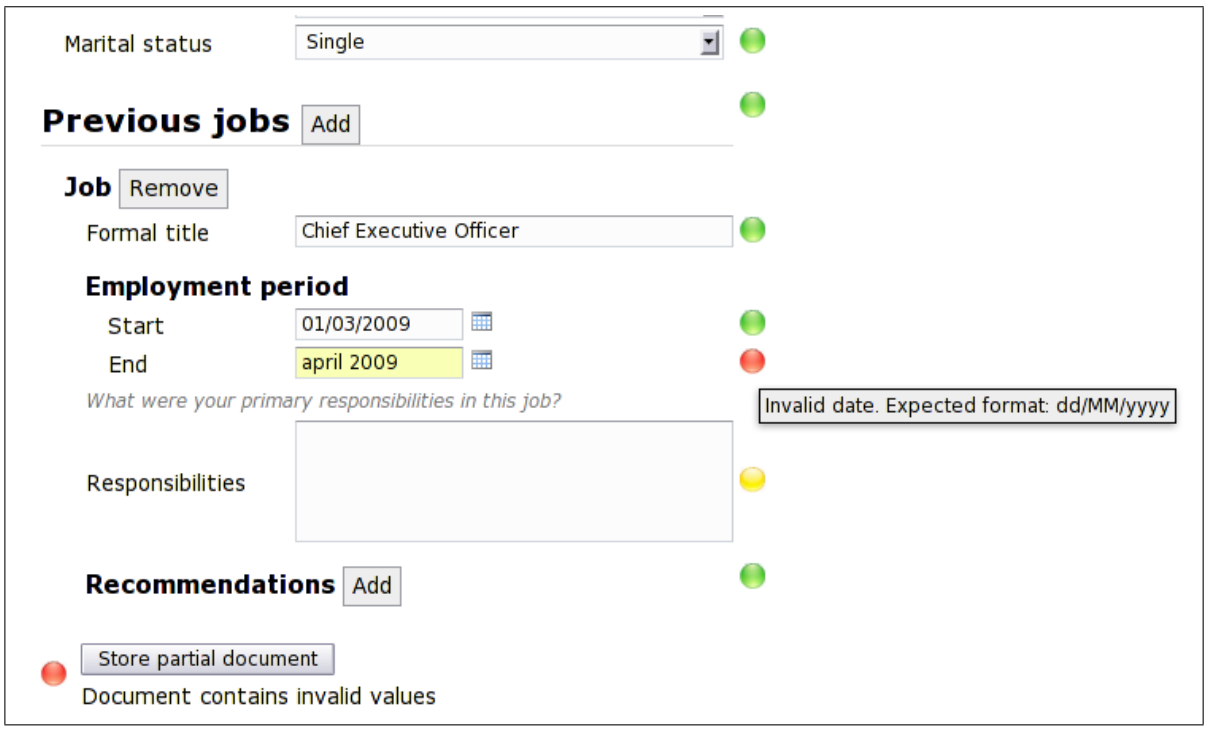


Figure 8.1: Partial screenshot of XHTML interface (adapforms-web), prior to end-user tests

The icons visible in figure 8.1 are used to indicate the status of the form element they are placed next to. The lock indicates that the element is currently flagged as being read-only. The yellow and red lights indicate validation warnings and errors respectively. By hovering the mouse over the lights, a text message with all validation problems for the element is displayed. The green light indicates that everything is fine, and the element needs no further attention. The red/green color-coding is inspired by the similar use in Dynamic Forms discussed in section 3.2.2 on page 12, although I am displaying a

more neutral icon as opposed to coloring the entire row. As opposed to Dynamic Forms, AdapForms does not support collapsible headers[1], but headlines may exists on different levels, so I have decided not to use the *implicit checklist* approach, as more colors or icons would probably just confuse the user.

Due to the nature of the adaptation cycle, the user will not see the status icons change until focus has been moved away from the element, signifying that the user believes the element to be filled out correctly.

When any of the controls have their value changed by the user, an event is sent to the server using Ajax[2], and the server responds by letting the browser know which adaptations (if any) is to be performed, along with any relevant validation problem messages. The interaction model is depicted in more detail in figure 8.2 on the next page. Please note that the sequence diagram has been simplified to not include messages to and from the core framework, which was illustrated in figure 4.2 on page 28.

When the form is submitted, the framework is notified by another Ajax call and the host application is notified by the registered callback. If the host application accepts the form (depending on e.g. validation status), the user can optionally be redirected to another web page, a message can be shown or similar. Notice that these adaptations are not limited to use in case of submission requests; such requests trigger an adaptation cycle in the usual sense.

To allow for the most flexibility, another web-specific adaptation has been added, capable of executing arbitrary JavaScript code in the user browser as response to an adaptation cycle[3].

8.3.1 Fallback for limited browsers

At the moment there is not implemented any fallback mechanism for browsers that does not support JavaScript or Ajax. However, if the browser does not understand JavaScript, a warning will be displayed right above the form. Similarly, if JavaScript is enabled, but Ajax is not supported, a warning will be displayed.

A fallback could be implemented by considering the form to be static, and at each form submit simply output the adapted form with any validation errors. This could be confusing and potentially frustrating for the user, but it would allow older browsers the ability to complete the form, despite of their technical shortcomings.

[1] At least not in the current UI implementation, other implementations may choose to support this

[2] Asynchronous JavaScript and XML [8]

[3] At first glance, this may look like a security risk, but in fact it is semantically no different than if the script would have been included in the rendered form to begin with. The only difference is, that the user cannot review all the script code before allowing JavaScript in the web browser, but that scenario is very hypothetical anyway

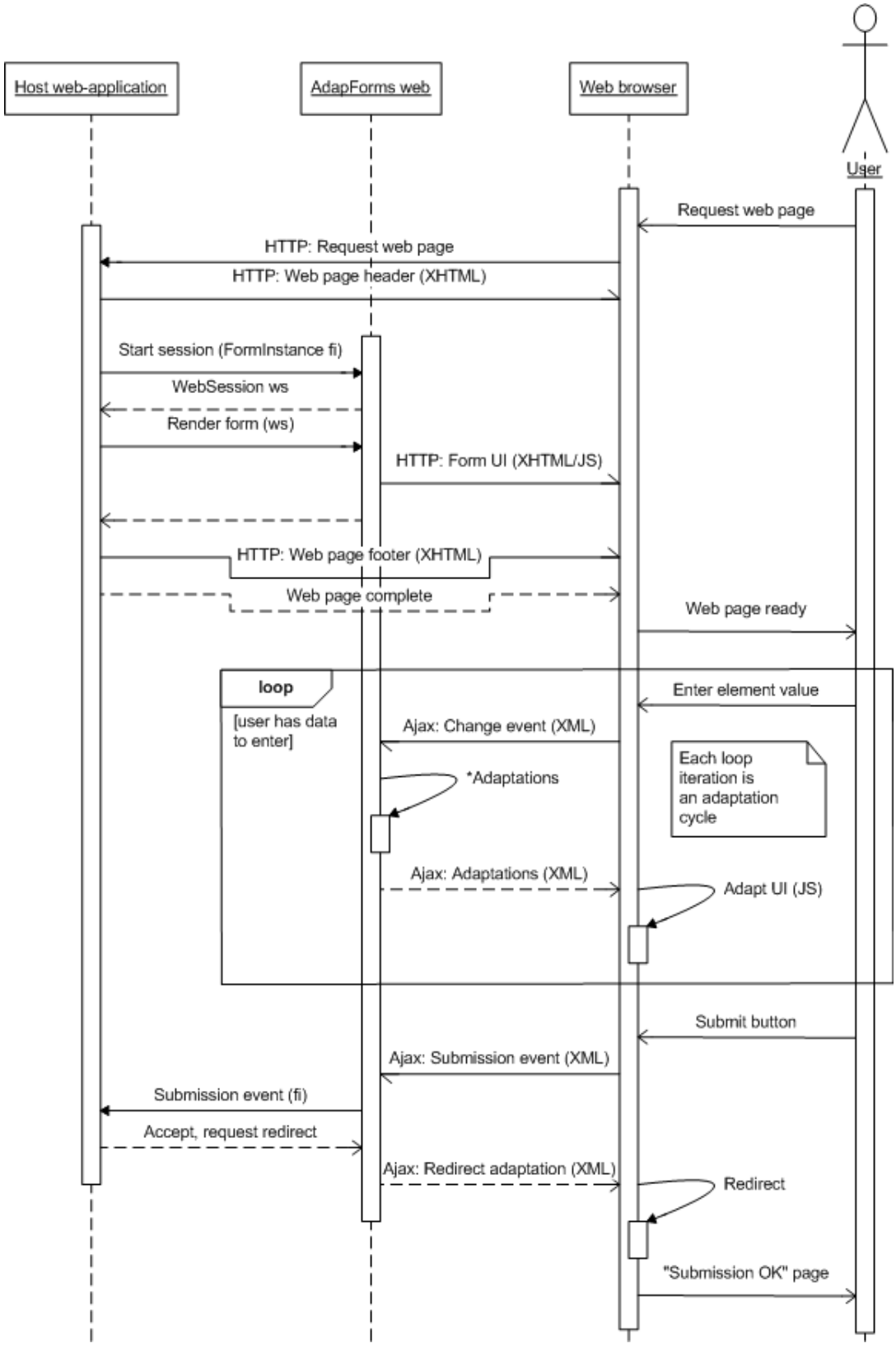


Figure 8.2: adapforms-web interaction model

8.4 Obtaining and using the framework

AdapForms is available under the BSD open source license, which basically means it is free to use for all purposes, cf. appendix H. Notice that not all dependencies may be compatible, as discussed in section 8.4.1.

Source code, binaries, documentation and some live demos of adapforms-core and adapforms-web can be found at the website accompanying the thesis: `http://adapforms.gammelmark.eu`

For a quick-start guide to using the implementation, refer to the following brief tutorials:

- Appendix C: Defining an adaptive form in XML.
- Appendix D: Getting started with adapforms-core.
- Appendix E: Getting started with adapforms-web.

As for technical documentation, it is available as JavaDoc on the website or can be generated locally with the provided Ant [20] build script (*doc* target).

8.4.1 Dependencies

The following libraries are prerequisites for the AdapForms framework to function properly. Notice that not all the libraries are needed, depending on your use scenario, and that some are already included within the builds. Obviously, a Java SE 5.0 SDK is also required.

Library	**License**	**Obtainable from**
log4j 1.2.15 (adapforms-core - Logging)	Apache License v. 2.0	Web reference [24]
Saxon-B XSLT processor 9.1 (adapforms-core - XPath 2.0 engine)	Mozilla Public License v. 1.0	Web reference [29]
Apache Commons Lang 2.4 (adapforms-web - Encoding tools)	Apache License v. 2.0	Web reference [21]
MooTools Core 1.2 (adapforms-web - JavaScript framework)	MIT License	Integrated into adapforms-web. Web reference [34]
vlaCal 2.1 (adapforms-web - JavaScript calendar)	Creative Commons Attribution-NonCommercial 3.0	Integrated into adapforms-web. Web reference [36]
JUnit 4.5 (adapforms-test - Unit testing)	Common Public License v. 1.0	Web reference [14]

The listed versions are the ones used during development and known to work. Substituting with older or newer versions may or may not work. The Saxon-B XML library may be replaced with little effort with

a compatible implementation if desired, as long as it supports the Java XML APIs.

Notice that the use of vlaCal dictates that adapforms-web cannot be used commercially. If this is desired, an alternative to this JavaScript calendar should be used instead, or vlaCal should simply be disabled.

8.5 Beyond prototyping

If the framework was to be deployed in real-life situations, a number of issues should be addressed, but these are irrelevant for the current proof-of-concept implementation. Some of them are listed here:

- **Security**
  The current implementation does not deploy any security features. In particular adapforms-web should undergo a security audit if used on the internet or other untrusted network, since nothing prevents malicious users from interfering with the web sessions of other users.

  Also, the behaviour classes and document hooks should be run inside a Java sandbox, to prevent malicious code distributed along with an adaptive form from interfering with the domain application or the operating system itself.

- **Concurrency**
  The existing code base is built upon the assumption that only one thread operates at a given time. Although the purpose is to create a new disjoint form instance for each user, there may still be more than one thread tampering with the instance: Interference from the host application, multiple concurrent Ajax requests on slow network connections etc. Currently this leads to undefined results.

- **Performance**
  Emphasis has been on creating a functional framework, using the least error-prone approaches possible. As a result, performance has suffered on the behalf of easier or more naive implementations of certain internal algorithms and data structures than what is optimal. Whether this is a problem in practice is not known.

  The client-server interaction may be simplified in some circumstances by using XForms or client-side validation code, and thus eliminating the need to query the server constantly. This is of course only possible, if no form hooks or semantic rules act on the form elements in question.

- **Robustness**
  adapforms-web produces valid XHTML 1.1 output, follows the W3C conventions for XML DOM [11] manipulation etc. However, certain browsers and platforms (such as Microsoft Internet Explorer) does not render and adapt the forms correctly under certain circumstances.

  More error tolerance is generally required throughout the framework, and many more unit tests should be implemented.

PART III

REFLECTIONS

Reflecting upon the designed framework, user studies and final conclusions

CHAPTER 9

Framework evaluation

9.1 End-user feedback

In order to get some external feedback on using the framework in practice, I have enlisted the help of a few volunteers. The aim is not to carry out an extensive usability survey, but rather perform an informal study of the impressions the framework make on average end-users, and their thoughts about the user experience.

The end-users were subjected to a traditional web page, on which a slightly modifed version of the benchmark form has been implemented using a standard HTML web form with no adaptations, and no validation before submission (see figure 9.1(a) on the next page). As for the list of jobs, requiring a *Repeat* element in AdapForms, two static entries were added, plus a third free-text element to enter any additional jobs into. I realize that this is probably not what one would usually encounter in "the wild", but I had to find a balance between what is possible to develop and what a developer could and would do in practice; In theory, all features of a given adaptive form could be implemented by hand, but that would render the comparison useless.

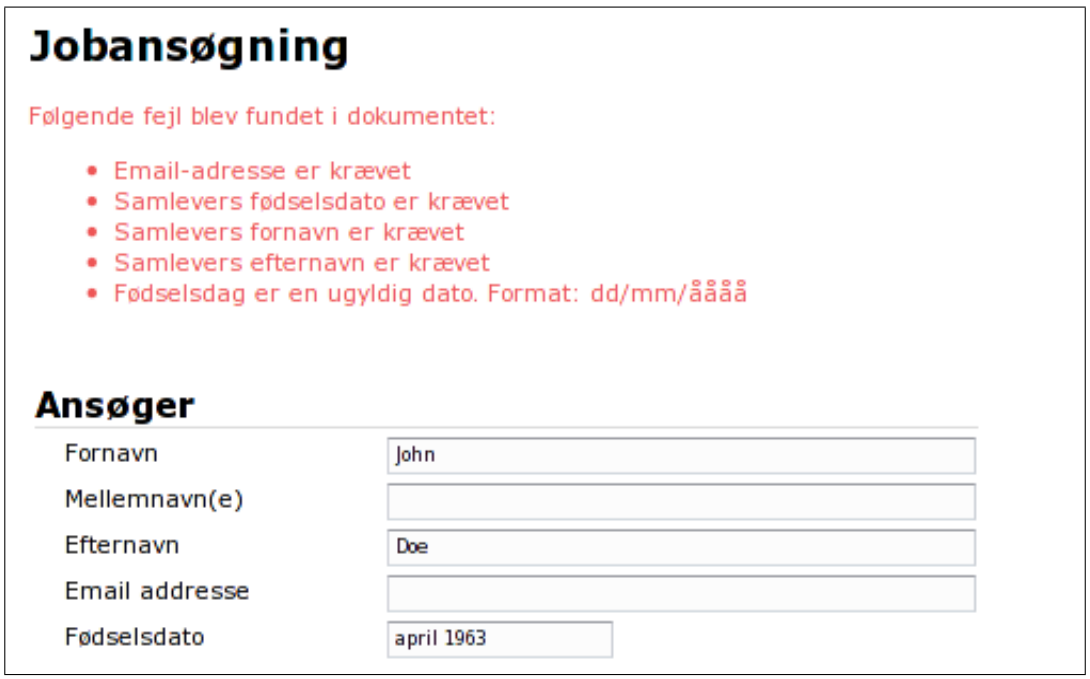
Secondly, the users were subjected to the same form as generated with AdapForms (figure 9.1(b)). For fairness, the overall layout and graphical style of the simple form resemble the output of the AdapForms version as much as possible (but without the dynamic buttons, validation icons etc).

All trials were performed with Danish testers, and thus both forms were in Danish to prevent a language barrier. Most user reactions were captured by screen capture software and the subjects were instructed to think aloud while interacting with the forms.

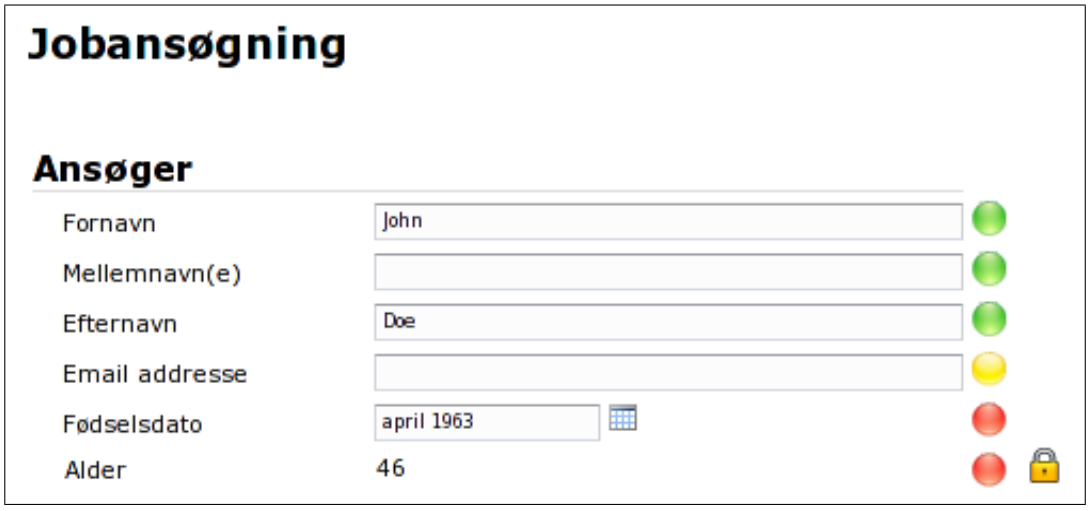
9.1.1 Findings

Validation problems

One of the major issues discovered with the AdapForms implementation is that the validation icons are largely overlooked when filling out the form. Even though the icons are placed right next to each form element, only one out of the three testers noticed them before having completed the form, and wondering why it refused to submit. Even when the icons were noticed, it was not clear how they should be interpreted, and it took all three some time to find out that the actual validation problem



(a) Traditional implementation



(b) AdapForms implementation

Figure 9.1: Screenshots of user feedback interfaces

messages were available by hovering the mouse over the icon. One person was guessing what the error could be, while another attempted to click the icons.

In contrast, all users were familiar with the way validation problems were displayed in the traditional implementation (figure 9.1(a)) and noticed them immediately. This is probably because they are listed very visually and are not simply existing icons that switch colors. One user suggested, that as soon as a problem is detected, a popup should be presented to be sure the user noticed it. While this forces the user to notice the message, it may be annoying when jumping around the form, especially if the *"Required element"* warnings should be presented as popups too.

One user deliberately entered text into the *ZIP code* element (expecting an integer) of both forms, and was surprised that the traditional implementation accepted the input, as if he expected immediate feedback. Surprisingly, the same person did not notice the validation icons on the AdapForms implementation until at the very bottom of the form. Upon submission of the traditional form and seeing the list of errors, he exclaimed *"It's not very smart that it didn't tell me that before"*, referring to the 4 dates all entered in a wrong format.

All users agreed, that once they understood the icons (and discovered the collective form status at the bottom of the page), that it is a very convenient feature to be able to see the problems in real-time, but it takes some time to understand. From these experiences, It is clear that another way of displaying the problems, or perhaps some help text above or next to the form, is needed to make it more usable. Alternatives include displaying a clearly visible error box, pointing at either the current element, or one for each element with validation problems, or coloring the entire elements to draw focus.

Date formats

Two of the users consistently entered dates in an incorrect format throughout both forms, where the third discovered the problem in the AdapForms implementation when the first icon turned red. In an extreme case, one of the users, not having discovered the problem message yet, attempted a total of 5 different formats before the right one was found. Although the expected format is presented to the user when the validation problem occurs, it may be beneficial to either print the expected format next to the element or perhaps replace the text field with three dropdowns: day, month and year. However, the latter may not be convenient when entering multiple dates since it takes longer.

Generally the data format expected was not always clear. For instance, one user entered *"no"* in the element titled *"Children under 18 years of age"*, when in fact it was expecting an integer. On the other hand, this is probably more a form design issue, than a framework issue and could be fixed by phrasing the element label more carefully. A possible improvement would be to completely disallow the entry of non-digit characters into the integer fields, with the help of a client-side script, but whether the confusion would disappear or grow is probably individual.

Other observations

Only one of the users seemed to notice the *ReadOnly* icon next to the *Age* element, which is dynamically calculated when their birthday is entered. The others simply skipped the element since there was no field to enter data into.

It seemed natural to all three, that additional elements showed up when they entered a value. The most notable example is the expansion of the *Spouse* group depending on the marital status. The only reaction was one user expressing *"something just happened"*, and then moved on to enter the spouse information. Another explanation may be, that the average users only focuses on the active element, and does not notice what is going on in other areas of the page. A possible improvement for this would be to use a JavaScript library for making the transition smooth and gradual, possibly slowly fading the elements in or out, so that user has time to notice the adaptation.

All users filled out the form in a top-down manner, starting with the very first element and proceeding downwards until they reached the submission button. As soon as they were finished with an element they proceeded to the next, expecting the above elements to remain like they left them. Although this was expected, it serves as a reminder that forms should avoid defining semantic rules that make changes "in the past", forcing the user to return to an already covered area of the form.

9.1.2 Aftermath

Following these observations the framework has been altered to address some of the raised issues. The primary focus was on the validation feedback technique with the colored icons. Realizing that there may have been too many icons which were not used anyway, I removed the green icons to make the remaining icons more visible. In addition to this, the textual problem description that was previously hard to find, is now printed directly beneith the affected form element in a color matching the icon (i.e. yellow or red). This is illustrated in figure 9.2. Not only is the problem description now directly accessible, but it also appears directly in the area where the user is expected to look for the next element to fill in.

The date-format problem has been mitigated by displaying the expected input format directly to the right of the date field as a help text. This is likewise illustrated in figure 9.2.

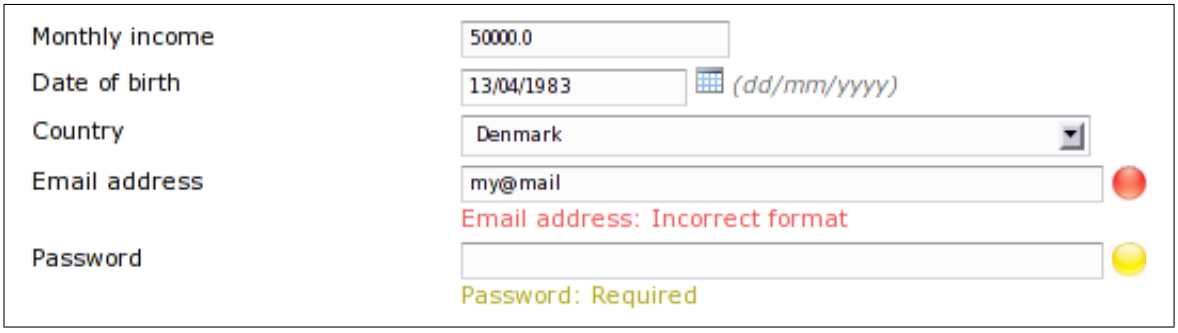


Figure 9.2: Revised user interface after end-user feedback

No further studies have been conducted after these changes, apart from one of the testers informally commenting that this is definately an improvement on usability.

9.2 Developer feedback

Almost from the point where the prototype implementation was able to visually display simple forms, the framework has been used within the *parental leave* application prototype [4] of the eGov+ project (mentioned in section 1.4 on page 4). During the following months the application prototype was further developed, and in the process, bugs and feedback about annoyances and inconveniences in AdapForms were reported. New ideas also came to life from this collaboration, such as the feature to display text messages to the user and localization of the form. Other ideas were found to be either out of scope of this thesis, or listed as *Future work* (chapter 11).

To get a second opinion, and because the aforementioned application prototype was finished before the framework, a series of short trials were designed and given to another Java web developer. Each trial or assignment is specified to be solved both with and without using the framework. The trials are listed in appendix G.

Obviously the developer trials are biased by definition, in the sense that I do not ask the developer to carry out tasks that the framework was not intended to support or use scenarios that I did not anticipate, but hopefully the diverse tasks will trigger any annoyances that may occur and have not yet been addressed. Just as interesting is whether the framework does not fit the mindset of the developer, or is overkill for some tasks.

9.2.1 Findings

The following is paraphrased from observations and comments made by Claus, while testing the nearly-finished framework prototype, with the addition of a few improvement suggestions of my own.

The learning curve for adapforms-web is quite steep, and requires the web developer to think differently about the flow of control between the web browser, host application and AdapForms. Concretely, the fact that a lot of Ajax HTTP traffic is exchanged outside the control of the enclosing webpage or servlet, magically invoking hooks and callbacks, caused a great deal of confusion. In retrospect, the control flow is not explicitly made clear, and if other developers were to use the framework, more explanatory tutorials would be needed, possibly accompanied by a more complete implementation example that demonstrate the intertwined event mechanisms, and illustrates how data is transferred to and from an instantiated form.

Definition of forms as XML works well, and the syntax is fairly straightforward to learn and one quickly gets familiar with the basics. The XML approach also makes it easy to exchange forms among different systems. However, when defining forms, especially in the beginning, the framework lacks a way of quickly validating them. As of now, the only means of verifying the correctness of the form is to actually write some code that attempts to parse the form, and then react on any exceptions that may be thrown in the process. Alternatively, one can use an XML editor that supports XSD validation, but the schema can only do so much; it cannot check that the bean types supplied actually exist and can be used in the context. Neither can it detect local name clashes between elements, when using externally referenced templates etc. For this purpose, Claus suggested implementing a simple tool (commandline or Ant task) that can be used to parse and instantiate a form, statically checking it for errors and detecting trivial unhandled exceptions in form hooks or XPath expressions[1].

As for simple validation logic it is intuitive and easy to use, but it is unclear how the validation should be coupled to the business logic potentially already present in the host application. Defining the validation rules two places is not optimal, but to some extend probably unavoidable. Synchronizing or reusing two validation schemes is not a trivial task. Part of the answer may lie in relying on validation logic built into the JavaBeans used by both the form and the business domain. In the simplest form, this could be done by directly calling the *set*-methods of the associated bean (or a prototype hereof) when the user supplies data, and converting any exceptions thrown into validation problems in an automated manner.

Having never experimented with XPath previously, semantic rules caused a few problems. Although the concept was easily understood, the expressions themselves were not quite as intuitive. Concretely, it was not clear how boolean expressions are interpreted and processed by the framework; although the DOM parameters *relevant*, *readonly* etc. theoretically always hold boolean values, they are in fact strings (which just happen to be either *"true"* or *"false"*), and as such cannot be applied directly to boolean operators and functions.

Moreover, an invalid XPath expression does not always result in an error. If the expression is not well-formed, the form parser will reject it with an error, but if non-existing elements are referenced within the expression, no error is raised; the expression will simply not behave like expected. Actually supporting this kind of static expression checking would require the implementation of a dedicated XPath interpreter which, cf. section 6.4 on page 43, is out of scope for this thesis. However, a debugging tool could be created, allowing the developer to inspect the value of each expression in a sandbox, executing arbitrary queries on the loaded form to see the raw results of sub-expressions, and reporting errors in a more developer-friendly manner.

In conclusion, Claus summarizes, that although it takes time to get into the proper mindset and get the base framework running, it is well worth it to invest the time needed to learn it properly, because it most

[1] Recall that during the initialization sequence, most hooks and semantic rules are triggered at least once, with the exception of hooks and rules associated with *Repeat* elements, that will not be initialized before the user adds an instance of them. This of course could also be automatically tested by the tool, by expanding all available repeats

likely will save time in the end. One of the clear advantages of using a framework like AdapForms is that validation and other trivial tasks are handled with little or no extra effort; tasks that one usually has a tendency to skip, when implementing forms from scratch.

9.3 Benchmark form

An interesting experiment is to see if the benchmark form defined in appendix A can be defined efficiently using the proposed framework, preferably without having to resort to expressing semantics using Java.

It may come as no surprise that this is indeed possible, since the benchmark form has been kept in mind during the entire design phase. Defining the structure is straightforward; the most challenging aspect being the list of jobs, which is gracefully handled by the *Repeat* element. The relevance and read-only adaptations are handled exclusively by XPath rules.

What remains is the validation logic. Originally the only validation logic envisioned was the simple built-in validation (such as required elements, regex patterns etc.) and the remaining validation was left to the host application to handle. When formally defining the benchmark form it became obvious that this was not convenient, as even the simplest validation rules should be implemented in Java external to the form definition. A trivial example is checking whether the *start* date is chronologically earlier than the corresponding *end* date.

As another case example consider the validation rule *"only one current employment is allowed"*. When implemented in Java this resulted in approximately 20 lines of code, iterating over all jobs, counting the number of *Currently employed* flags set to *"true"* and marking all of these with a validation error, if the total count was more than 1. By introducing XPath validation rules (discussed in section 7.2.2 on page 46), this was reduced to a single XPath expression, added to the *Currently employed* element:

```
<validator criteria=". = 'false' or count(/jobs/period/current[@value='true']) &lt;= 1"
    message="You can have at most one current job" />
```

The complete form definition encapsulating the structure, validation and adaptation semantics of the benchmark form is listed in appendix B. A live demonstration of the benchmark form is available at the thesis website.

9.4 Challenges

The three main challenges presented in the problem statement (section 1.2 on page 2) are used here to evaluate the framework as designed and implemented. It is explained at a glance, how each challenge was met (or not) by the framework.

9.4.1 Form and semantics definition

***Challenge**: "Which way is the best to represent the forms and their adaptive semantics?"*

The *"best"* way is obviously subjective and depends on context. Chapter 5 presented an XML-based language for representing the structure of a form. The language is declarative and abstract in the sense that UI specific details are not present in the definition, which in turn allows for *single authoring* and *multireification*; the technique of deriving multiple concrete forms from the same abstract declaration. The result is a language that is fairly simple to use and understand, at the expense of having only

limited control over how the form is presented to the user, which can be good or bad depending on the situation.

A specific goal was the support of repeatable sub-forms; lists of arbitrary length with each list item having a complex compound data structure itself. This is incorporated into the language fairly simply by the <*repeat*> form element, but presents some challenges for the runtime system, as there no longer exist a unique 1-to-1 mapping between the abstract structure and the runtime structure. Nested repeat elements are also supported.

The language encourages reuse, supported by splitting often used sub-forms into separate XML files that may be included by all the forms using them. This templating mechanism conceptually works by creating templates that are defined and named, and are instantiated in each document by unique name reference. Due to reasons discussed in section 5.2.2 on page 33, recursively defined forms or templates is not allowed.

The adaptation semantics of a form was the topic of chapter 6 and are defined in two ways: By defining rules directly in the XML definition using XPath expressions, and by *form hooks* that allow the host application to respond to framework events.

The XPath expressions are embedded directly into the definition of each form element in the shape of XML attributes. They exhibit significant expressive power, but can by no means accomplish all computational tasks. XPath expressions were chosen as a trade-off between ease-of-use and language complexity, as it was deemed that most everyday challenges could be expressed relatively simple, with limited understanding of programming and boolean logic.

9.4.2 Data representation and transfer

***Challenge**: "How can user input and context metadata be represented, to allow for easy transfer between host application, framework and a client (for instance a web browser)"*

Internally all form data and state is stored in the state tree; a hierachical data structure, which was the topic of section 4.3 on page 25. The tree enforces strict type checking (and conversion) to ensure that the tree is always in a valid state.

Data and state is constantly synchronized between the central state tree and the UI layer currently in use. The user agent holds only a visual[2] representation of the state in the tree, and the tree is considered the "master copy", which at all times is what adaptations are made upon, and the data the host application accesses. In fact, with little extra work AdapForms could be extended to allow for collaborative work between multiple clients (possibly using different UI implementations). This would allow e.g. a telephone supporter to see the exact same form (and in the same state) as the end-user calling for help.

The framework transparently bridges the gap between the rich data model of the host application, and the often simplistic model of the UI layer. As an example, every value in a complex multi-element form is considered a string by the web browser, but the developer need not worry about this.

The host application can access and manipulate the data and state in various ways at runtime. Traditional web applications that operate on *(key, value)* pairs (useful for relational databases for instance), can query each element individually by supplying the relevant form element path. More advanced host applications may use the power of XML to save or reload entire forms, or perform complex calculations on the DOM representation of the state tree.

To enhance the integration even more, forms may include JavaBean declatations using the <*bean*> element type. This provides a direct mapping from a Java class to form elements and back, without the need for specifying more than the name of the class and its form path. This was the topic of section 5.4 on page 35.

[2] Assuming a graphical interface, which need not be the case

9.4.3 Data integrity

***Challenge**: "Which is the best way to ensure valid data, both in individual fields, but also that the form in its entirety has been filled in correctly?"*

Validation techniques were the subject of chapter 7. AdapForms provides means of performing simple validation of the individual form elements in isolation. These include type checking, range or interval checks, regular expressions and *required* flags. Immediately after the value of an element is changed, the validation status of the element is updated according to all relevant validation rules (including more complex XPath validators). Finally, the host application may choose to specify additional warnings and errors for an element, to allow for more complex validation.

The exact way in which a user is notified of these problems depends on the UI implementation in use. In the prototype implementation, problems are indicated by yellow and red lights next to the troubled elements accompanied by a descriptive message. As already discussed, the original approach was not a success in practice, so this particular issue still remains open although a few improvements and further suggestions have been made.

The form can be filled out in any order desired, and it is always possible to go back and correct a mistake. There are no restrictions on the scope of semantic rules, so changing a value towards the bottom of a form may trigger adaptations earlier in the form. As this is inconvenient for the users who most likely completes the form top-down, it is strongly suggested that forms be designed to accomodate the top-down approach.

9.5 Comparison to XForms

Since the framework closely resembles XForms (described in section 3.2.3 on page 13) on a number of points, a brief comparison of the two is in order.

- **Purpose and scope**
  The purposes of the two frameworks differ in that AdapForms is very specifically targeting the creation and processing of data-intensive forms in the sense that one usually thinks about their paper equivalent. In contrast, XForms has a much wider scope and aims at providing general web forms for any and all purposes, including search fields, login boxes etc. As stated earlier, XForms is scheduled to replace traditional (X)HTML forms in the future XHTML 2.0 standard.
- **Architecture**
  One of the design goals of XForms is to move as much of the processing and data handling as possible to the client, which then just submits the gathered data as XML to the server using the well-known HTTP protocol, leaving the server with the decision of how to process and interprete the data.

  AdapForms is designed to have all the major functionality as close to the host application as possible (which would be at the server, when used in a client-server manner). This is done to make the data gathering as transparent as possible to the host application. The tasks of the UI layer is kept at a bare minimum.
- **Form structure**
  Both employ the use of XML for the definition of a hierachical data model with type information associated with each element for basic type safety. Arbitrary long lists of repeated instances of a substructure is supported by both, but neither supports recursive structures. AdapForms further supports the reuse of substructures (complex data types) via shared module inclusion and direct inference of elements from domain objects.

- **UI independence**
  Where the structure of the user interface is more or less given in AdapForms, XForms allows the form author to tailor the interface to any level of detail, limited only by the boundaries of the host language. XForms is designed to be UI- and device-independent, but it does not support multireification directly, since an XForms document is not a stand-alone entity in the same way that an adaptive form is. Although the data model and most of the semantics of the XForms document can be reused directly on another platform without modification, the document is not complete and usable until it is embedded into a host technology such as XHTML, and coupled explicitly to the UI elements.

  An adaptive form on the other hand is completely UI independent, but is very limited in the ways the interface can be customized. This was of course not an option when designing XForms, since they needed to support vastly more different use scenarios.

- **Adaptation semantics**
  Adaptation of state parameters such as *readonly*, *required* and even *relevant* is supported by both frameworks, and can be changed at runtime by XPath expressions, although in XForms they are defined indirectly (external to the data model) and then bound to the element(s) the expression applies to, whereas in AdapForms they are applied directly in the element declaration. XForms (1.0 and 1.1) supports XPath 1.0, while AdapForms supports XPath 2.0.

  In addition to using XPath expressions, the host application may interfere directly using e.g. form hooks. In XForms this requires additional manual implementation, such as using Ajax calls.

- **Host application integration**
  XForms is essentially decoupled from the host application in that it gathers the required information and then ships it as plain XML to the server. Even though this is an improvement over traditional HTML forms, the server should still check that all data is valid (never trust the client), and parse the XML nodes into the proper data types before use. AdapForms, on the other hand, has direct API interaction with the host application by design, and all data is guaranteed to be valid and properly type-converted as the validation logic is placed locally. The host application may interact with the form at any time using the local Java objects acting as a proxy for the actual form, regardless of how and where it is currently being executed, providing UI and communication transparency.

Chapter 10

Conclusion

Throughout this thesis a framework have been designed and continously refined, providing means of formally defining and interacting with UI-independent adaptive input forms.

A specific XML language was tailored for this purpose, in which the form author defines the data model of the form implicitly by laying out for structure of the form using abstract form element types such as *Text*, *Choice* and *Date*. Logical and visual grouping of the elements is supported, allowing for rich XML data models. Sub-forms and often used complex data types may be reused across multiple form definitions by a templating mechanism, by externally referencing AdapForms module files, defined using the same XML language. Arbitrarily nested elements are supported, and sub-structures may be repeated an arbitrary number of times (by user initiative), fullfilling the need for entering lists of complex data structures.

Adaptation semantics and validation logic is embedded directly into the XML language by using XPath queries, providing a compact, yet powerful, language for expressing basic logic and computations. Whenever the end-user changes a value in the form, an adaptation cycle is triggered, causing all rules and validators to be re-evaluated. The culmination of the adaptation cycle is a list of adaptations sent to the UI layer, which in turn uses it to adapt the rendered form and notify the user of any validation problems.

After loading a form into the framework, the form can be instantiated using a range of parameters. The form may be localized to a specific language and the user may be assigned a role, determining which form elements are available for reading and writing, respectively. In the presense of a previously saved (possibly partially completed) form, its entered values may be loaded into the new instance and user-interaction may resume, analogue to picking up a paper form and continuing from where if was left.

The entire state of the instantiated form is held by the runtime state tree, on which all adaptations are performed. The structure of the tree is dictated by the static form structure which cannot be altered at runtime, with the exception of repeated elements. The host application has access to the tree via a type safe API or may choose to interact with it via the DOM API, making it possible to execute XML tools such as XQuery or XSLT on the tree itself. All interactions with the tree are checked against the form structure, data type designations and semantic rules defined, and entered values will be automatically type-converted, so that the tree will always be in a valid state, regardless of how it is manipulated.

To further enhance the integration, parts of the form structure can be inferred reflectively from domain classes, and data may be transferred between the host application and the framework using these models directly, limiting both the amount of needed integration code and the amount of XML to be written.

10.1 Practical use

The complete AdapForms XML language is formally defined as an XML Schema, and a reference prototype implementation is built in Java. Although the implementation supports the full feature set of the proposed framework, it is only a prototype and thus not considered to be production-ready, mainly due to performance and security considerations.

If the implementation was to be refined or ported to other platforms, the framework could potentially be used anywhere a traditional form is now implemented, provided that the structure fits into the AdapForms language and the graphical appearance of the form need not be specifically tailored. More realistically, the framework could be applied in more or less its current form to Java-enabled web applications and replace the traditional (X)HTML-based input forms with an adaptive alternative. Actually using the framework for submitting applications to government bodies digitally, as demonstrated in the eGov+ prototype [4] using AdapForms, will very doubtfully take place in practice.

A general problem with using it at larger scale, is that it is traffic-intensive. Each time an end-user enters a value into the form, a round-trip to the core/server component is executed, triggering an adaptation cycle. Although this is one of the theoretical strenghts of the framework, in practice it may be a limiting factor instead. A more coordinated approach would be needed, where some of the trivial adaptations and validations could be done at the client also, and changes could be sent to the server in batches. This is not possible with the current approach of using XPath internally though.

Informal end-user experiments were conducted, suggesting that adaptive forms does have certain usability advantages over traditional static forms, granted that extra refinements are needed on the UI part.
All in all, I believe that AdapForms can be considered a succes; experiments by third party developers showed that the framework is indeed practical and worth the effort of learning, even though it requires thinking in new ways and has a rather steep learning curve.

10.2 Thesis work

During the course of the thesis work, the vision of single-authored adaptive forms progressively transformed from loose ideas into concrete concepts and finally from theory into a prototype implementation. The proposed framework solves most of the issues raised and features suggested in the problem statement, even though some aspects turned out radically different than originally envisioned. There have been both positive and negative examples of this.

Based on loose ideas from the eGov+ project, part of the vision was to tightly integrate the form with known data about the user and the context in which the form was viewed. This was achieved to some extend, but in a more decoupled and indirect way than I set out to, mainly because of the huge challenge of linking arbitrary context data to a very rigid form definition in a systematic manner.

On a personal note, I enjoyed working on an academic project of this scale, and will take both good and bad experiences, *dos and don'ts*, with me to future projects. I am very pleased with the outcome and might continue the development of the framework on a smaller scale in the future, focusing on making the prototype eligible for actual use.

Henrik Gammelmark, Aarhus

CHAPTER 11

Future work

Below is listed some issues either not addressed in this thesis or considered only briefly, as well as some ideas for further improvements of the work carried out.

11.1 Layout management

In the proposed framework there are limits to which ways the developer may influcence how the form is displayed to the end user. This stems from the design requirement, that the forms must be UI neutral, and thus only abstract layout features, such as logical grouping of elements are natively supported. The chosen UI layer is free to render the form in any way it sees fit, as long as the form structure is recognizable in some way.

In practice most UI implementations will presumably look fairly similar and use a graphical WIMP[1] interface. Although this need not be the case always, the number of remaining practical techniques are negligible. Based on this it may be beneficial to provide the developer with more efficient tools to structure and visually design the form, while still staying GUI-neutral, if not entirely UI-neutral.

The only means of manipulating the form visuals at present is through the *uiflags* parameter, which may be interpreted differently depending on the UI layer in effect. A concrete topic of interest could be to superimpose another XML-language onto the neutral language that adds a new level of GUI customization. Some of the spatial adaptation techniques from [10] may be applied in this context.

In many cases the forms will be used with a single UI only, so even though it is against the design goals of the framework, one may investigate how to make a mapping between the adaptive form and a specific markup language like XHTML. Possibly by mixing the two languages in the same XML file using the respective namespaces, so that the adaptive form can still be read and understood by other UI engines, who will simply ignore the unsupported XHTML markup.

A light version of this UI-specific customization is already possible while using the web UI, through the application of Cascading Style Sheets (CSS), but this only provides means of customizing the look-and-feel of the individual visual elements; the general graphical structure is strictly enforced by the XHTML rendering engine. The stylesheets are also not embedded into the form definition file, so they they need to be distributed manually along with the form.

[1] WIMP: Windows, Icons, Menus and Pointing devices - the classic GUI approach

11.2 Usability survey

The usability studies carried out were very limited and informal. An obvious area of research is to investigate how end-users perceive adaptive forms, and perform studies as to how user interfaces best support the common view(s) of a form. Perhaps radically new ways of presenting and conceptualizing the forms are needed.

As discussed in section 9.1.1 on page 56, a number of usability issues have already been identified, such as the way validation problems and data types are visualized, which may serve as the basis for further research.

It has been mentioned numerous times throughout the thesis that the defined forms are UI-independent. Although this has been one of the design criteria, no experiments with alternative user interfaces have been conducted. A potentially very interesting research area would be to investigate other devices and interfaces, e.g. mobile phones which are becoming increasingly powerful and very capable of handling distributed applications.

11.3 Better exploitation of XPath

The proposed framework uses XPath expressions as semantic rules to automatically adapt certain state parameters. Inspired by XForms, the use of the XPath expressions may be expanded to also setting the value of form elements by expression evaluation. For instance, automatically selecting the gender when the person enters his/her social security number.

Although this is simple in theory, given the existing XPath support, an implementation needs to address how to handle situations where the user has manually entered a different value, or changes it afterwards. This will lead to some inconsistencies between what the user entered and what the framework has computed; both may be correct depending on the situation. A possible workaround would be to only allow value computations in this manner, if the element is flagged as *ReadOnly*, but this may not always be desired.

11.4 Improved i18n support

The internationalization features described in section 4.7 on page 30 does not scale well with many languages, since these need to be distributed and loaded separately. An obvious improvement would be to include the translations in the actual form itself for easy redistribution and central maintenance. As an added bonus, this would allow the parser to reflectively list all translations of the language, allowing the host application to more transparently select the language to use based on user preferences for instance.

Reuse among forms may also be beneficial, as a range of forms may well contain similar concepts, and thus translations. Continuing on the idea of inlining translations into the form definition, key phrases could be included in AdapForms module files similar to templates.

11.5 Digital signing

Recall that one of the use cases of the framework was the use in government-citizen interaction in the eGov+ project. Every citizen of Denmark already holds (or can request for free) a government-issued digital certificate, usable across a large number of online goverment self-service web applications. An improvement of the framework would be to support the use of certificates to sign the contents of a form, before submitting it to the relevant authority. Because of the *role* mechamism of the framework, it will be beneficial if multiple users (the citizen, the case handler, other government bodies etc.) can each sign parts of the form, whether completed or not.

11.6 Implementation improvements

An obvious point of improvement, although not an academic exercise, would be to improve the implementation, so that it may be applied to applications in the wild. A few areas needing attention is listed in section 8.5 on page 53, and include improving the security, robustness and performance of the implementation and allowing for better concurrency handling.

Another UI implementation for a different type of user interface may also either support the framework architecture or identify general issues. Further improvements of the existing adapforms-web component is also an obvious choice, since it for instance does not support Internet Explorer correctly at present.

The XPath implementation (cf. section 6.4 on page 43) could also use some attention. Instead of merely applying a black-box XPath engine, a more specific engine could dramatically improve performance if it were to deploy conservative heuristics as to when re-evaluation of each expression is needed.

Bibliography

[1] BEA Systems Inc . Alex Toussaint. Java rule engine api jsr-94. Technical report, Java Community Process, 2003.

[2] Norman Walsh Ashok Malhotra, Jim Melton. Xquery 1.0 and xpath 2.0 functions and operators, w3c recommendation. Technical report, The World Wide Web Consortium (W3C), 2007.

[3] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. Xml path language (xpath) 2.0, w3c recommendation. Technical report, The World Wide Web Consortium (W3C), 2007.

[4] Morten Bohøj and Niels Olof Bouvin. Collaborative time-based case work. In *HYPERTEXT 2009, Proceedings of the 20th ACM Conference on Hypertext and Hypermedia*, Turin, Italy, 2009.

[5] John M. Boyer. Xforms 1.1, w3c candidate recommendation. Technical report, The Forms Working Group, The World Wide Web Consortium (W3C), 2007.

[6] Alfons Brandl and Gerwin Klein. Formgen: A generator for adaptive forms based on easygui. In *Proceedings of HCI International (the 8th International Conference on Human-Computer Interaction) on Human-Computer Interaction: Ergonomics and User Interfaces-Volume I*, pages 1172–1176, Hillsdale, NJ, USA, 1999. L. Erlbaum Associates Inc.

[7] Per de Place Bjørn. Oioxml som obligatorisk, åben standard - uddybende vejledning. Technical report, IT- og Telestyrelsen, Datastandardiseringskontoret, 2008.

[8] Jesse James Garrett. Ajax: A new approach to web applications. Technical report, Adaptive Path, 2005.

[9] Andreas Girgensohn, Beatrix Zimmermann, Alison Lee, Bart Burns, and Michael E. Atwood. Dynamic forms: An enhanced interaction abstraction based on forms. In *Proceedings of Interact '95*, pages 362–367. Chapman& Hall, 1995.

[10] Mikko Honkala. *Web User Interaction - a Declarative Approach Based on XForms*. Helsinki University of Technology, 2007.

[11] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document object model (dom) level 3 core specification. Technical report, The World Wide Web Consortium (W3C), 2004.

[12] John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, Inc., New York, NY, USA, 2002.

[13] Peter Ørbæk. Programming with hierarchical maps. Technical report, University of Aarhus, Daimi PB-575, 2005.

Web references

[14] Junit.org: Resources for test driven development.
http://www.junit.org
(Accessed November 2008).

[15] Alexandra Instituttet A/S. egov+ projektet.
http://www.egovplus.dk
(Accessed October 2008).

[16] Multiple authors. *XForms Tutorial and Cookbook*. WikiMedia Foundation, 2008).
http://en.wikibooks.org/wiki/XForms
(Accessed November 2008).

[17] Ian Bicking and Contributors. Formencode.
http://www.formencode.org
(Accessed February 2009).

[18] JBoss Community. Drools.
http://www.jboss.org/drools
(Accessed January 2009).

[19] Progeny Systems Corporation. Formfaces.
http://www.formfaces.com
(Accessed January 2009).

[20] Apache Software Foundation. The apache ant project.
http://ant.apache.org
(Accessed March 2009).

[21] Apache Software Foundation. Apache commons lang.
http://commons.apache.org/lang
(Accessed December 2008).

[22] Apache Software Foundation. Apache commons validator.
http://commons.apache.org/validator
(Accessed February 2009).

[23] Apache Software Foundation. Apache tomcat.
http://tomcat.apache.org
(Accessed October 2008).

[24] Apache Software Foundation. Log4j 1.2, logging services.
http://logging.apache.org/log4j/1.2
(Accessed November 2008).

[25] The Eclipse Foundation. Eclipse platform, 3.4 ganymede edition.
http://www.eclipse.org
(Accessed October 2008).

[26] David Heinemeier Hansson. Ruby on rails: Web development that doesn't hurt.
http://rubyonrails.org
(Accessed February 2009).

[27] Orbean Inc. Orbeon forms: Web forms for the enterprise done the right way.
http://www.orbeon.com
(Accessed January 2009).

[28] larsw joernt. Chiba home.
http://www.chibaxforms.org
(Accessed January 2009).

[29] Michael H. Kay. Saxon-b xslt and xquery processor.
http://saxon.sourceforge.net
(Accessed April 2009).

[30] Sandia National Laboratories. Jess, the rule engine for the javatm platform.
http://www.jessrules.com
(Accessed November 2008).

[31] Bob Lyons. Universal turing machine in xslt.
http://www.unidex.com/turing/utm.htm
(Accessed April 2009).

[32] IT og Telestyrelsen. Offentlig information online xml.
http://www.oio.dk
(Accessed November 2008).

[33] IT og Telestyrelsen. Offentlig information online xml, infostructurebase.
http://isb.oio.dk/repository
(Accessed November 2008).

[34] Valerio Proietti. Mootools: a compact javascript framework.
http://mootools.net
(Accessed January 2009).

[35] PalCom project. Palpable computing.
http://www.ist-palcom.org
(Accessed December 2008).

[36] R. Schoo. Vista-like ajax calendar version 2.
http://dev.base86.com/scripts/vista-like_ajax_calendar_version_2.html
(Accessed February 2009).

PART IV

APPENDIX

Supplementary information relevant, but not vital, to the thesis

APPENDIX **A**

Benchmark form

This form is not intended to be of any practical use or even make sense, but aims at a benchmark for comparing and testing the various technologies and ideas in the thesis.
Elements are required and may only be written to by users actings as applicants, unless otherwise specified.

- Social Security Number
- Name of applicant:
  - First name
  - Middle name *(optional)*
  - Last name
- Email address *(must have valid syntax or be empty)*
- Phone number *(5-12 digits, possibly prefixed by "(+countrycode)")*
- Monthly income *(in the interval 0-50.000)*
- Date of birth
- Name of parent/guardian *(only if the person is not yet 18 years of age)*:
  - First name
  - Middle name *(optional)*
  - Last name
- Country *(finite options)*
- State *(only if country is USA)*
- Marital status *(single, living with significant other or married - only if at least 18 years)*
- Name of spouse *(only if married)*:
  - First name
  - Middle name *(optional)*
  - Last name

- List of former and current employments *(at least one is required)*:
  - Company
  - Formal job title
  - Description of primary job responsibilities
  - Employment start date
  - Employment end date or "Still employed" *(end date cannot be later than start date, only one current employment is allowed)*
  - List of recommendations *(if any)*
    - Recommended by
      - First name
      - Middle name *(optional)*
      - Last name
    - Recommendation text *(at least 50 characters)*
  - Job confirmed by HR *(checkbox, only available to HR personnel)*
- Assigned case handler *(only available to HR personnel)*
  - First name
  - Middle name *(optional)*
  - Last name
- Application approved *(only available to HR personnel, readonly until all jobs have been individually confirmed)*

APPENDIX **B**

Benchmark form using AdapForms

The following lists the two files *benchmark.xml* and *benchmark-common.xml* which are the benchmark form from appendix A as an AdapForms XML definition, and a module holding some shared templates, respectively.

The form can be pre-adapted using one of the two user roles:

- **applicant**
  Person filling out the job application.
- **human-resources**
  Person from the Human Resources (HR) department handling the application.

```
<?xml version="1.0" encoding="UTF-8"?>

<adapforms xmlns="http://adapforms.gammelmark.eu/ns" version="0.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://adapforms.gammelmark.eu/ns adapforms.xsd">

  <include file="benchmark-common.xml"/>

  <form title="Job application">
    <defaults
      required="true"
      readonly="false"
      write-roles="applicant"
      read-roles="applicant,human-resources"
    />


    <!-- Basic applicant credentials -->

    <group id="applicant" label="Applicant information">
      <text id="ssn" label="Social Security Number" maxLength="10" minLength="9"/>
      <use template="person-name" id="name" label="Full name"/>
      <use template="email" id="email" label="Email address" required="false" />
      <text id="phone" label="Phone number" pattern="(\(\+\d{1,3}\) *)?\d{5,12}" />
      <decimal id="income" label="Monthly income" minValue="0" maxValue="50000" />
      <date id="dob" label="Date of birth"/>

      <helptext id="minor" label="You are under the age of 18. Please name your parent or
          legal guardian"
```

```
29          relevant="days-from-duration(current-date() - xs:date(../dob/@datevalue)) &lt; 18 *
                365.25" />
30
31        <use template="person-name" id="guardian" label="Parent or legal guardian" relevant="../
              minor/@relevant" />
32
33        <choice id="country" label="Country" allowBlankChoice="true">
34          <item value="DK">Denmark</item>
35          <item value="US">United States of America</item>
36          <item value="Other">Other</item>
37        </choice>
38
39        <choice id="state" label="State" default="Other" relevant="../country = 'US'">
40          <item value="NY">New York</item>
41          <item value="CA">California</item>
42          <item value="FL">Florida</item>
43          <item>Other</item>
44        </choice>
45      </group>
46
47
48      <!-- Marital information -->
49
50      <group id="status" label="Civil status" relevant="/applicant/minor/@relevant = 'false'">
51        <choice id="marital" label="Marital status" allowBlankChoice="true">
52          <item>Single</item>
53          <item>Lives with significant other</item>
54          <item>Married</item>
55        </choice>
56
57        <group id="spouse" label="Spouse" relevant="../marital = 'Married'">
58          <date id="dob" label="Date of birth"/>
59          <use template="person-name" label="Name" id="name"/>
60        </group>
61      </group>
62
63
64      <!-- List of current and previous jobs -->
65
66      <repeat id="jobs" label="Positions held during the past 10 years" entryLabel="Job"
            minRepeats="1">
67        <text id="company" label="Company"/>
68        <text id="title" label="Formal job title"/>
69
70        <group id="period" label="Employment period">
71          <date id="start" label="Start"/>
72          <toggle id="current" label="Still employed">
73            <validator criteria=". = 'false' or count(/jobs/period/current[@value='true']) &lt;=
                  1" message="You can have at most one current job" />
74          </toggle>
75          <date id="end" label="End" relevant="../current = 'false'">
76            <validator criteria="xs:date(@datevalue) &gt;= xs:date(../start/@datevalue)" message
                  ="Job cannot have negative duration" />
77          </date>
78        </group>
79
80        <helptext id="help" label="What were your primary responsibilities in this job?"/>
81        <text id="responsibilities"  label="Responsibilities" uiflags="type:multiline" required=
              "true"/>
82
83        <repeat id="recommendations" label="Recommendations" entryLabel="Recommendation">
84        <text id="description" label="Recommendation" minLength="50" uiflags="type:multiline" />
85          <use template="person-name" id="by" label="Recommended by" />
86        </repeat>
87
88        <toggle id="confirmed" label="Job confirmed by HR" read-roles="human-resources" write-
              roles="human-resources" />
```

```
89      </repeat>
90
91
92      <!-- Book-keeping by the HR department -->
93
94      <defaults
95        write-roles="human-resources"
96        read-roles="human-resources"
97      />
98
99      <group id="human-resources" label="Reserved for HR department">
100       <use template="person-name" id="handler" label="Assigned case handler" />
101       <toggle id="approved" label="Application approved" readonly="count(/jobs/confirmed[
              @value='false']) > 0" />
102     </group>
103   </form>
104 </adapforms>
```

Listing B.1: Benchmark form: benchmark.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <adapforms-module xmlns="http://adapforms.gammelmark.eu/ns" version="0.1"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xsi:schemaLocation="http://adapforms.gammelmark.eu/ns adapforms.xsd">
6
7    <!-- Name of a person split into parts -->
8
9    <template name="person-name">
10     <group>
11       <text id="first" label="First name"/>
12       <text id="middle" label="Middle names" required="false"/>
13       <text id="last" label="Last name" />
14     </group>
15   </template>
16
17   <!-- Regex validated email address -->
18
19   <template name="email">
20     <text maxLength="255"
21       pattern="[a-zA-Z0-9!#$%&amp;'*+/=?^_`{|}~-]+(?:\.[a-zA-Z0-9!#$%&amp;'*+/=?^_`{|}~-]+)*@
             (?:[a-zA-Z0-9](?:[a-zA-Z0-9-]*[a-zA-Z0-9])?\.)+[a-zA-Z0-9](?:[a-zA-Z0-9-]*[a-zA-Z0
             -9])?"/>
22   </template>
23 </adapforms-module>
```

Listing B.2: External referenced module: benchmark-common.xml

APPENDIX C

Tutorial: Defining an adaptive form in XML

This brief tutorial explains how to create an adaptive form as a XML file. The XML language is described in chapters 5 and 6. The precise syntax of the XML language is described in appendix F.

For a complete example form refer to appendix B.

1. **Create an empty XML file**
   The first step is to create a basic empty XML file with the correct namespace, schema definition and root node. Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<adapforms xmlns="http://adapforms.gammelmark.eu/ns" version="0.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://adapforms.gammelmark.eu/ns adapforms.xsd">

  <!-- Actual form goes here -->

</adapforms>
```

2. **Define the data model**
   The recursive structure (i.e. the collection of form elements) of the form defines the data model, while also describing abstractly how the elements should be displayed to the user.

   Each element must at least have an *id* and in most cases a *label* attribute. The available set of elements are listed in section 2.2.1 on page 7. See section 5.2 on page 32 for more common parameters and their use. Other attributes are available depending on the type of element. Structures may be reused among multiple forms, cf. section 5.2.2 on page 33.

   The structure is placed within a <*form*> tag inside the <*adapforms*> root tag. Example:

```
<form title="My Form">
  <integer id="foo" label="Foo goes here"/>
  <group id="bar" label="Bar!">
    <text id="baz" label="Baz in group bar"/>
    <date id="foobar" label="Foo bar!"/>
  </group>
</form>
```

   To ease the integration with the host application, a direct mapping to and from any JavaBean type can be used, by which part of the form structure can be automatically inferred . See section 5.4 on page 35 for details. Example:

```
<bean id="address" label="Address" type="adapforms.examples.domain.Address" />
```

3. **Define the semantics**
   The adaptive semantics of the form can be defined in two ways (both may be used in combination). Firstly, XPath expressions can be used directly in the XML file. For instance, the relevance (visibility) or *ReadOnly* flag of an element may depend on the state or value of other elements. As an example, the */bar/baz* element can be made *ReadOnly* if the value of the */foo* equals *"42"*, and the entire */bar* group only visible if the length of the value in */foo* is less than 4:

```
<group id="bar" label="Bar!" relevant="string-length(/foo) &lt; 4>
  <text id="baz" label="Baz in group bar" readonly="/foo = '42'"/>
  <date id="foobar" label="Foo bar!"/>
</group>
```

   This is covered in more detail in section 6.3 on page 40.

   Secondly, Java code can be associated with the initialization of the form, allowing automatic adding of form hooks, that may respond directly on user input, and manipulate the instantiated form. This is done be specifying a class of type *adapforms.form.FormBehaviour* on the <*form*> tag. Example:

```
<form title="My Form"  behaviour="adapforms.webtest.TestBehaviour">
```

   Be sure to ship the specified behaviour along with the XML when exchanging forms between applications.

4. **Define validation rules**
   Data integrity is ensured through the use of validation rules. In the simplest case these are simple XML properties associated with the element. Different validators are supported depending on the element type. A few examples follow:

```
<integer id="foo" label="Foo" minValue="-128" maxValue="127"/>
<text id="bar" label="bar" required="true" pattern="[a-z]*"/>
```

   More complex rules can be defined using XPath expressions, associated with an error message that is displayed when the defined criteria evaluates to *"false"*. Any number of validators can be listed as child nodes of the form element:

```
<text id="foo" label="Foo">
  <validator criteria="lower-case(.) = ." message="Must be all lower-case" />
  <validator criteria="@hasvalue='false' or contains(. , 'hello')" message="Must
      contain the substring hello" />
</text>
```

APPENDIX **D**

Tutorial: Getting started with adapforms-core

This brief tutorial explains how to load and use an adaptive form in the general case; that is, regardless of the media used to interact with the user. For more information on how to apply the framework to a web project, refer to appendix E.

1. **Obtain a form**
   The first step is to obtain or create an adaptive form, represented as an XML file. Appendix C provides an introduction for this purpose.
2. **Load and parse the form**
   In order to interact with an adaptive form, it first has to be loaded from a XML file and parsed. Example:

```
Form form = FormParser.parseForm(new File("myform.xml"));
```

3. **Instantiate the form**
   In order to use the form, it must be *instantiated*; that is, it must be customized to the context. You may localize the form to a given language or locale, cf. section 4.7 on page 30. At the same time, a *callback* must be specified, which will be notified about various form events; most interestingly when the form is being submitted. If you make use of the *role* mechanism, you must also specify the role to use. Example:

```
InstanceCallback callback = new AbstractInstanceCallback() {
  // Callback methods go here – for instance process the form when submitted
}

FormInstance instance = form.createInstance(callback, LocalizationFactory.createDanish
    ());
instance.setRole("applicant");
```

   The form instance will hold the current contents and validation status of all form elements at any given time.
4. **Add any manual hooks**
   The host application may be interesting in responding to specific element events, in order to act on user input on the fly. For instance if the user enters a phone number, the applicant may want to look up the address from a customer database, or perform some complex validation that requires the involvement of business logic. To achieve this, the host application may register hooks that are triggered whenever the relevant element changes its value. Example:

```
FormHook hook = new AbstractFormHook() {
  // Hook methods go here
}

instance.addHook("/path/to/element", hook);
```

Any hooks thare are defined in a *FormBehaviour* instance associated with the form, will be registered automatically upon instantiation.

5. **Initialize**
   The form instance is now ready to be initialized, and in the process populated with default values and preform the pre-adaptation:

```
instance.initialize();
```

   If the form has been previously saved (in either complete or partial state), the previously entered values may be supplied via the overloaded method *initialize(FormData)* instead.

6. **Present form to the user**
   The form should now be presented to the user in one shape or the other, allowing the user to fill out the form. The steps taken here depends on the media. See appendix E for use in a web context.

7. **Retrieve form data**
   When the form has been filled out and submitted, the *callback* defined when instantiating the form is invoked, and the data can be extracted.

APPENDIX **E**

Tutorial: Getting started with adapforms-web

This brief tutorial explains how to apply adaptive forms to a J2EE web platform, using the *adapforms-web* component.

I assume the use of the Eclipse Platform with Web Tools Platform (WTP) extensions, but no steps or code depend on Eclipse functionality to work.

1. **Create a web project**
   Create a new *Dynamic web project* (or open an existing), and configure it to run on your local J2EE servlet container (e.g. Tomcat).
2. **Add the AdapForms libraries**
   Copy the *adapforms-core.jar* and *adapforms-web.jar* to the *"WEB-INF/lib"* directory, to make them part of the project classpath. Be sure to also include the dependencies. See section 8.4.1 on page 52 for details.
3. **Bind the AdapFormsServlet to a URL pattern**
   Most framework web interaction is mediated indirectly through a single servlet: *AdapFormsServlet*. For the framework to work, this servlet must be bound to an URL range in *WEB-INF/web.xml*. Example:

```
<servlet>
  <description>Handles interaction with Adaptive Forms (adapforms-web)</description>
  <display-name>AdapForms</display-name>
  <servlet-name>AdapForms</servlet-name>
  <servlet-class>adapforms.web.AdapFormsServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>AdapForms</servlet-name>
  <url-pattern>/__adapforms/*</url-pattern>
</servlet-mapping>
```

   If the base URL differs from the default *"/__adapforms/"*, the framework must be notified about where to find the servlet, by calling the static *adapforms.web.WebRuntime.setServletPath(String)* method. You may start the server, navigate to the base url and the servlet should respond by letting you know it is present. The default is assumed for the rest of this tutorial.
4. **Define one or more forms**
   Create or copy in the adaptive forms for your application, as well as any Java beans they may reference. It is recommended to add it under the *"WEB-INF"* directory to make sure the raw forms are not directly accessible from a web browser See appendix C for a tutorial on how to create a form.

5. **Create or locate a JSP page/servlet to include the form in**
   The form will be included in a JSP page or a Java Servlet within your own web contents. Create a new web page or choose an existing one and begin a code block where the form should be placed. In this block, you should start by placing the form loading/initialization code, described in appendix D.

6. **Define a visual style (optional)**
   You may wish to just use the default stylesheet supplied by the framework, or you may design your own. To use a custom stylesheet, you should invoke *adapforms.web.WebRuntime.setUseDefaultCSS(false)* and include your own in the webpage instead. To create your own, you can retrieve the default via the relative local URL */__adapforms/res/adapforms.css* and customize it to your needs.

7. **Render the form to the browser**
   When the form is instantiated, you create a *WebSession*, and write the form to the client. Example:

```
// Create instance of the form
FormInstance instance = ...:     // Assumed loaded and initialized

// Begin web session
WebSession ws = WebRuntime.startSession(instance);

// Write form to browser
ws.writeToClient(request, response);
```

   You may output any desired XHTML before and after the *writeToClient* call. All further web interactions will be handled by the *AdapFormsServlet* on behalf of the host application.

   A number of XHTML header elements (script includes etc.) are output in the process. It is preferable to output these in the <*head*> tag of the page. To do this, place the following code within the <*head*> tag:

```
ws.writeHeadersToClient(request, response);
```

   If you do not do this, the headers will simply be output by the *writeToClient* method instead.

8. **Wait for user interaction**
   All interaction with the user occurs via the *AdapFormsServlet*, and no explicit steps should be taken here by the web application, except possibly responding to framework events, via registered hooks and callback.

9. **Receive submitted form data**
   When the form has been submitted, the server-side callback will be invoked in the usual manner. It will often be beneficial to redirect the user to another web-page to indicate succes. Example:

```
ws.redirectClient("submit-ok.jsp");
```

APPENDIX **F**

XML Schema for Adaptive Forms

The following XML Schema Definition (XSD) file formally specifies how adaptive forms are declared in an XML file.

The schema basically handles two different types of XML documents:

- **Form**: Definition of an adaptive form. The root node is <*adapforms*>.
- **Module**: Definition of complex types shared among multiple forms. The root node is <*adapforms-module*>.

```
1  <?xml version="1.0"?>
2  <!--
3     XML Schema definition for Adaptive Forms (AdapForms framework).
4
5     Visit http://adapforms.gammelmark.eu for more information.
6  -->
7
8  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
9    targetNamespace="http://adapforms.gammelmark.eu/ns"
10   xmlns="http://adapforms.gammelmark.eu/ns"
11   elementFormDefault="qualified">
12
13   <!--  Value holding element superclass -->
14
15   <xs:complexType name="value-holding-element">
16     <xs:sequence minOccurs="0" maxOccurs="unbounded">
17       <xs:element name="validator" type="validator" />
18     </xs:sequence>
19     <xs:attributeGroup ref="element-attr-modifiers"/>
20   </xs:complexType>
21
22   <xs:complexType name="validator">
23     <xs:attribute name="criteria" type="xs:string" use="required"/>
24     <xs:attribute name="message" type="xs:string"  use="required"/>
25     <xs:attribute name="severity" default="Error">
26       <xs:simpleType>
27         <xs:restriction base="xs:string">
28           <xs:enumeration value="Warning"/>
29           <xs:enumeration value="Error"/>
30           <xs:enumeration value="Required"/>
31         </xs:restriction>
32       </xs:simpleType>
```

```
33      </xs:attribute>
34    </xs:complexType>
35
36
37    <!-- Text element type -->
38
39    <xs:complexType name="element-text-type">
40      <xs:complexContent>
41        <xs:extension base="value-holding-element">
42          <xs:attribute name="pattern" type="xs:string"/>
43          <xs:attribute name="default" type="xs:string" default=""/>
44          <xs:attribute name="minLength" type="xs:nonNegativeInteger"/>
45          <xs:attribute name="maxLength" type="xs:nonNegativeInteger"/>
46        </xs:extension>
47      </xs:complexContent>
48    </xs:complexType>
49
50    <xs:complexType name="element-text">
51      <xs:complexContent>
52        <xs:extension base="element-text-type">
53          <xs:attributeGroup ref="element-attr-ident"/>
54        </xs:extension>
55      </xs:complexContent>
56    </xs:complexType>
57
58    <!-- Integer element type -->
59
60    <xs:complexType name="element-integer-type">
61      <xs:complexContent>
62        <xs:extension base="value-holding-element">
63          <xs:attribute name="minValue" type="xs:long"/>
64          <xs:attribute name="maxValue" type="xs:long"/>
65          <xs:attribute name="default" type="xs:long"/>
66        </xs:extension>
67      </xs:complexContent>
68    </xs:complexType>
69
70    <xs:complexType name="element-integer">
71      <xs:complexContent>
72        <xs:extension base="element-integer-type">
73          <xs:attributeGroup ref="element-attr-ident"/>
74        </xs:extension>
75      </xs:complexContent>
76    </xs:complexType>
77
78    <!-- Decimal element type -->
79
80    <xs:complexType name="element-decimal-type">
81      <xs:complexContent>
82        <xs:extension base="value-holding-element">
83          <xs:attribute name="minValue" type="xs:double"/>
84          <xs:attribute name="maxValue" type="xs:double"/>
85          <xs:attribute name="default" type="xs:double"/>
86        </xs:extension>
87      </xs:complexContent>
88    </xs:complexType>
89
90    <xs:complexType name="element-decimal">
91      <xs:complexContent>
92        <xs:extension base="element-decimal-type">
93          <xs:attributeGroup ref="element-attr-ident"/>
94        </xs:extension>
95      </xs:complexContent>
96    </xs:complexType>
97
98    <!-- Group element type -->
99
```

```
<xs:complexType name="element-group-type">
  <xs:sequence minOccurs="1" maxOccurs="unbounded">
    <xs:group ref="form-element-instance"/>
   </xs:sequence>
  <xs:attributeGroup ref="element-attr-modifiers"/>
</xs:complexType>

<xs:complexType name="element-group">
  <xs:complexContent>
    <xs:extension base="element-group-type">
      <xs:attributeGroup ref="element-attr-ident-optionallabel"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Repeat element type -->

<xs:complexType name="element-repeat-type">
  <xs:sequence minOccurs="1" maxOccurs="unbounded">
    <xs:group ref="form-element-instance"/>
   </xs:sequence>
  <xs:attributeGroup ref="element-attr-modifiers"/>
  <xs:attribute name="minRepeats" type="xs:nonNegativeInteger"/>
  <xs:attribute name="maxRepeats" type="xs:nonNegativeInteger"/>
  <xs:attribute name="entryLabel" type="xs:string"/>
</xs:complexType>

<xs:complexType name="element-repeat">
  <xs:complexContent>
    <xs:extension base="element-repeat-type">
      <xs:attributeGroup ref="element-attr-ident"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Bean element type -->

<xs:complexType name="element-bean-type">
  <xs:attributeGroup ref="element-attr-modifiers"/>
  <xs:attribute name="type" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="element-bean">
  <xs:complexContent>
    <xs:extension base="element-bean-type">
      <xs:attributeGroup ref="element-attr-ident-optionallabel"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Choice element type -->

<xs:complexType name="element-choice-type">
  <xs:sequence minOccurs="1" maxOccurs="unbounded">
    <xs:choice>
      <xs:element name="item">
        <xs:complexType mixed="true">
          <xs:attribute name="value"/>
        </xs:complexType>
      </xs:element>
      <xs:element type="validator" name="validator" />
    </xs:choice>
  </xs:sequence>
  <xs:attributeGroup ref="element-attr-modifiers"/>
  <xs:attribute name="default" type="xs:string"/>
  <xs:attribute name="allowBlankChoice" type="xs:boolean"/>
</xs:complexType>
```

```
167
168   <xs:complexType name="element-choice">
169     <xs:complexContent>
170       <xs:extension base="element-choice-type">
171         <xs:attributeGroup ref="element-attr-ident"/>
172       </xs:extension>
173     </xs:complexContent>
174   </xs:complexType>
175
176   <!-- Multichoice element type -->
177
178   <xs:complexType name="element-multichoice-type">
179     <xs:sequence minOccurs="1" maxOccurs="unbounded">
180       <xs:choice>
181         <xs:element name="item">
182           <xs:complexType mixed="true">
183             <xs:attribute name="value"/>
184           </xs:complexType>
185         </xs:element>
186         <xs:element type="validator" name="validator" />
187       </xs:choice>
188     </xs:sequence>
189     <xs:attributeGroup ref="element-attr-modifiers"/>
190     <xs:attribute name="default" type="xs:string"/>
191   </xs:complexType>
192
193   <xs:complexType name="element-multichoice">
194     <xs:complexContent>
195       <xs:extension base="element-multichoice-type">
196         <xs:attributeGroup ref="element-attr-ident"/>
197       </xs:extension>
198     </xs:complexContent>
199   </xs:complexType>
200
201   <!-- Template element type -->
202
203   <xs:complexType name="element-template-use-type">
204     <xs:attributeGroup ref="element-attr-modifiers"/>
205     <xs:attribute name="template" type="xs:string" use="required"/>
206   </xs:complexType>
207
208   <xs:complexType name="element-use-template">
209     <xs:complexContent>
210       <xs:extension base="element-template-use-type">
211         <xs:attributeGroup ref="element-attr-ident"/>
212       </xs:extension>
213     </xs:complexContent>
214   </xs:complexType>
215
216   <!-- Toggle element type -->
217
218   <xs:complexType name="element-toggle-type">
219     <xs:complexContent>
220       <xs:extension base="value-holding-element">
221         <xs:attribute name="default" type="xs:boolean" default="false"/>
222       </xs:extension>
223     </xs:complexContent>
224   </xs:complexType>
225
226   <xs:complexType name="element-toggle">
227     <xs:complexContent>
228       <xs:extension base="element-toggle-type">
229         <xs:attributeGroup ref="element-attr-ident"/>
230       </xs:extension>
231     </xs:complexContent>
232   </xs:complexType>
233
```

```
<!-- Label element type -->

<xs:complexType name="element-label-type">
  <xs:complexContent>
    <xs:extension base="value-holding-element">
      <xs:attribute name="default" type="xs:string" default=""/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="element-label">
  <xs:complexContent>
    <xs:extension base="element-label-type">
      <xs:attributeGroup ref="element-attr-ident"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Generic element type -->

<xs:complexType name="element-generic-type">
  <xs:attributeGroup ref="element-attr-modifiers"/>
</xs:complexType>

<xs:complexType name="element-generic">
  <xs:complexContent>
    <xs:extension base="element-generic-type">
      <xs:attributeGroup ref="element-attr-ident"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Date element type -->

<xs:complexType name="element-date-type">
  <xs:complexContent>
    <xs:extension base="value-holding-element" />
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="element-date">
  <xs:complexContent>
    <xs:extension base="element-date-type">
      <xs:attributeGroup ref="element-attr-ident"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>


<!-- Form element attribute groups -->

<xs:attributeGroup name="element-attr-ident">
  <xs:attribute name="id" type="xs:string" use="required"/>
  <xs:attribute name="label" type="xs:string" use="required"/>
</xs:attributeGroup>

<xs:attributeGroup name="element-attr-ident-optionallabel">
  <xs:attribute name="id" type="xs:string" use="required"/>
  <xs:attribute name="label" type="xs:string" use="optional"/>
</xs:attributeGroup>

<xs:attributeGroup name="element-attr-modifiers">
  <xs:attribute name="read-roles" type="xs:string"/>
  <xs:attribute name="write-roles" type="xs:string"/>
  <xs:attribute name="required" type="xs:string"/>
  <xs:attribute name="readonly" type="xs:string"/>
  <xs:attribute name="uiflags" type="xs:string"/>
```

```
301     <xs:attribute name="relevant" type="xs:string"/>
302   </xs:attributeGroup>
303
304   <!-- Misc reused types and groups -->
305
306   <xs:group name="form-element-instance">
307     <xs:choice>
308       <xs:element name="integer" type="element-integer"/>
309       <xs:element name="decimal" type="element-decimal"/>
310       <xs:element name="bean" type="element-bean"/>
311       <xs:element name="text" type="element-text"/>
312       <xs:element name="secret" type="element-text"/>
313       <xs:element name="group" type="element-group"/>
314       <xs:element name="toggle" type="element-toggle"/>
315       <xs:element name="date" type="element-date"/>
316       <xs:element name="label" type="element-label"/>
317       <xs:element name="repeat" type="element-repeat"/>
318       <xs:element name="choice" type="element-choice"/>
319       <xs:element name="multichoice" type="element-multichoice"/>
320       <xs:element name="use" type="element-use-template"/>
321       <xs:element name="helptext" type="element-generic"/>
322     </xs:choice>
323   </xs:group>
324
325   <xs:group name="form-element-type">
326     <xs:choice>
327       <xs:element name="integer" type="element-integer-type"/>
328       <xs:element name="decimal" type="element-decimal-type"/>
329       <xs:element name="bean" type="element-bean-type"/>
330       <xs:element name="text" type="element-text-type"/>
331       <xs:element name="secret" type="element-text-type"/>
332       <xs:element name="group" type="element-group-type"/>
333       <xs:element name="toggle" type="element-toggle-type"/>
334       <xs:element name="date" type="element-date-type"/>
335       <xs:element name="label" type="element-label-type"/>
336       <xs:element name="repeat" type="element-repeat-type"/>
337       <xs:element name="choice" type="element-choice-type"/>
338       <xs:element name="multichoice" type="element-multichoice-type"/>
339       <xs:element name="use" type="element-template-use-type"/>
340       <xs:element name="helptext" type="element-generic-type"/>
341     </xs:choice>
342   </xs:group>
343
344   <xs:complexType name="form-defaults">
345     <xs:attributeGroup ref="element-attr-modifiers"/>
346   </xs:complexType>
347
348   <xs:complexType name="template-definition">
349     <xs:sequence>
350       <xs:group ref="form-element-type"/>
351     </xs:sequence>
352     <xs:attribute name="name" type="xs:string" use="required"/>
353   </xs:complexType>
354
355
356   <!-- Definition of a form -->
357
358   <xs:complexType name="form">
359     <xs:sequence minOccurs="1" maxOccurs="unbounded">
360       <xs:choice>
361         <xs:element name="defaults" type="form-defaults"/>
362         <xs:group ref="form-element-instance"/>
363       </xs:choice>
364     </xs:sequence>
365
366     <xs:attribute name="title" type="xs:string"/>
367     <xs:attribute name="behaviour" type="xs:string"/>
```

```
368   </xs:complexType>
369
370
371   <!-- Definition of an adapforms file -->
372
373   <xs:element name="adapforms">
374     <xs:complexType>
375       <xs:sequence>
376
377         <!-- Includes -->
378
379         <xs:sequence minOccurs="0" maxOccurs="unbounded">
380           <xs:element name="include">
381             <xs:complexType>
382               <xs:attribute name="file" type="xs:anyURI" use="required"/>
383             </xs:complexType>
384           </xs:element>
385         </xs:sequence>
386
387         <!-- Templates -->
388
389         <xs:sequence minOccurs="0" maxOccurs="unbounded">
390           <xs:element name="template" type="template-definition"/>
391         </xs:sequence>
392
393         <!-- Form definition -->
394
395         <xs:element name="form" type="form"/>
396       </xs:sequence>
397
398       <xs:attribute name="version" fixed="0.1" use="required"/>
399     </xs:complexType>
400   </xs:element>
401
402
403   <!-- Definition of an adapforms module file -->
404
405   <xs:element name="adapforms-module">
406     <xs:complexType>
407
408       <!-- Templates -->
409
410       <xs:sequence minOccurs="0" maxOccurs="unbounded">
411         <xs:element name="template" type="template-definition"/>
412       </xs:sequence>
413
414       <xs:attribute name="version" fixed="0.1" use="required"/>
415     </xs:complexType>
416   </xs:element>
417
418 </xs:schema>
```

APPENDIX **G**

Developer trials

The following are the trials/assignments presented to the developer testing the AdapForms framework. The actual trials were given in Danish, but translated here into English:

1. **Simple web input**
   Create a simple webpage capable of reading a *String* from a text field and writing it to *System.out*.
2. **Simple web output**
   Expand the page so that it can print the text to the browser instead of *System.out*.
3. **Date input**
   Expand the page to also read a date (*java.util.Date*) from the browser, and write it to *System.out*. Make sure that only valid dates are entered, and present an error message otherwise.
4. **Bean mapping**
   Create a Java class *Foo* containing a *java.util.Date*, an *int* and a *String* with corresponding getters and setters.
   Create a webpage capable of reading these values from the web user and instantiate an instance of the class *Foo*, and possibly print it to *System.out*.
5. **Adaptation**
   Create a webpage on which the user can only enter a value into a given field, if a checkbox has been checked first.
   The field must be filled in if the box is checked, otherwise an error must be presented (required field).
6. **Server-side logic**
   Create a webpage with a text field, which presents an error to the user if a given Java method returns *"false"*, when given the field value as input.
   The purpose is to simulate complex server-side business logic, but for this trial the validation may be as simple as checking if the field has the value *"42"*.
7. **Localization**
   Transform one of the webpages so that the user may choose between a Danish and an English version (for instance via the query string *?lang=en*).
   Any error messages should also be translated.
8. **Lists**
   Create a webpage on which the user can input a series of books, in the form of (*title*, *author*) pairs.

9. **Roles**
   Create a webpage with two fields and only allow the user to enter a value in the second field, if he has the user role *"administration"*. The role may be set, for instance, via the query string *?role=administration*.

10. **Setup and grouping of elements**
    Create a webpage with a form containing a range of different fields of varying types, logically grouped. For instance, one group with a headline for address input, one for leasure activities etc. You may attach various validation constraints to the fields.

11. **Playground**
    Play around with other features of interest and possibly try to abuse the framework in order to provoke errors or undesired functionality.
    Perhaps make up a few everyday scenarios and apply the framework to them. This is a good way to discover limitations in the framework, as well as usage scenarios I have not myself anticipated.

APPENDIX **H**

AdapForms software license (BSD)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.