AdapForms: A Framework for Creating and Validating Adaptive Forms

Morten Bohøj[1], Niels Olof Bouvin[2], and Henrik Gammelmark[3]

[1] Alexandra Institute, Åbogade 34, DK-8200 Aarhus N
[2] Dept. of Computer Science, Aarhus University, Åbogade 34, DK-8200 Aarhus N
[3] Dat1, Åbogade 15, DK-8200 Aarhus N

Abstract. AdapForms is a framework for adaptive forms, consisting of a form definition language designating structure and constraints upon acceptable input, and a software architecture that continuously validates and adapts the form presented to the user. The validation is performed server-side, which enables the use of complex business logic without duplicate code. Thus, the state of the form is kept persistently at the server, and the system ensures that all submitted forms are valid and type safe.

1 Introduction

The World Wide Web has gained its remarkable success not only because it is a (relatively) simple and scalable publishing system, but also because it offers easy access for users to add new content or interact in other ways by filling on-line forms. This is done many times daily in search fields, e-shops, application forms, and so on. While many of these text fields and forms are straightforward, some are more challenging for the user to correctly interpret and fill. Likewise, an essential part of form handling at the developer's end is the checking and validation of the input, before it can be added to a database or used as basis for other calculations. These validation steps range from trivial tests that can be evaluated using e.g., string length over more complex rules checked with regular expressions, and finally input that can only be checked against the "business logic" of the particular application. In practical terms, this can range from checking whether a family name has been entered, whether an amount entered consists only of digits, or whether a desired vacation period is actually available. As many administrative processes move on-line, the associated forms follow. Forms may be simple or complex with opaque and non-trivial interrelationships that can be difficult to communicate effectively to the user. When is a form filled out correctly or sufficiently; and how do the choices in one field affect the rest of the form? Which parts of the form are relevant, and what can be safely ignored?

We describe in this paper AdapForms, a general solution to handling forms in a fashion that enables developers to clearly designate accepted types of input, reuse existing templates (e.g., sub-forms for postal addresses), use complex validation rules, allow users to resume filling out complex forms at a later point, and communicate the state of the form to the user unobtrusively. An open source

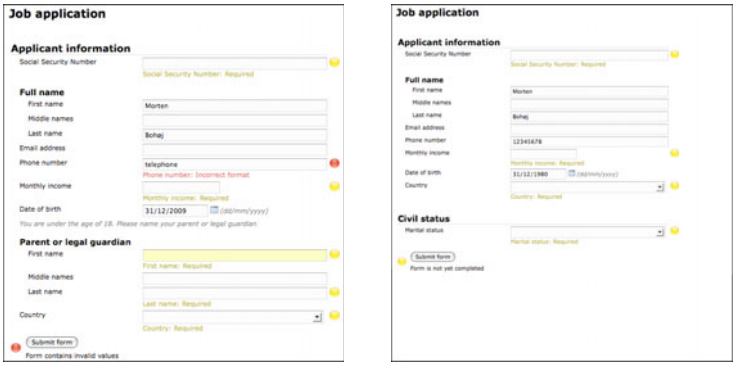
S. Auer, O. Diaz, and G.A. Papadopoulos (Eds.): ICWE 2011, LNCS 6757, pp. 105–120, 2011.
© Springer-Verlag Berlin Heidelberg 2011

demonstrator of the framework is available online[1] and an exhaustive description can be found in [6].

The paper is structured as follows: Our system is described from a functional point of view in Section 2 and an architectural one in Section 3, the system has been evaluated as described in Section 4, we describe related work in Section 5, and the paper is concluded in Section 6.

2 The AdapForms Framework

Validation of user submitted data on the Web is usually a two-step process—input is validated in the browser (usually through custom JavaScript code), and validated again when submitted to the server. Client-side validation has certain advantages, especially responsiveness as no network latency is involved. However, client-side validation cannot replace server-side validation, as the input sent back to the server may be subverted or corrupted accidentally or maliciously. Thus, server-side validation is still necessary—at the very least to thwart SQL injections and similar attacks. AdapForms is a server-centric framework, which



(a) Yellow and red signify missing resp. invalid data. The Parent form is due to the age of the user

(b) The applicant is of legal age, and thus Civil status is relevant

Fig. 1. AdapForms screenshots

allows direct code access to the form and data structures, described further in Section 3. This also means that all validation is done on the server instead of the client, which provides a higher degree of safety regarding validity of the input. An advantage of having the state constantly up-to-date server-side, is that it allows direct transparent integration with the host application instead of relying on additional server calls for adaptation and validation, as described below. Furthermore, business logic is not exposed in JavaScript accessible in the browser, but is kept on the server.

[1] `http://adapforms.gammelmark.eu/`

While the AdapForms framework is focused on the underlying handling of form structure and data and not on the user interface to the form, an example XHTML/Ajax based prototype implementation of the UI has been created for testing and demonstration purposes. This example UI could also have been created using HTML5 and taking advantage of the new form elements. HTML 5 thus provides a richer user experience without replacing the AdapForms framework. The web UI is used as example throughout this paper and two screen shots are shown in Figure 1(a) and 1(b). The screen shots (discussed in more detail in Section 2.2) illustrate how forms are validated and adapted to the user's input. The UI is not part of the framework and UIs for multiple platforms could be created, without changing the document XML definition or server side validation. Implementing different UIs also allows for changing the rendering of how feedback is given to the users.

Listing 1.1. Sample XML form

```
<adapforms>
 <include file="templates.xml"/>
 <form title="Example Form">
  <defaults readonly="false" write-roles="applicant"/>
  <template name="anAddress">
   <group>
    <text id="street" label="Street">
     <validator criteria="lower-case(.) = ." message="Must be all lower-case"/>
    </text>
    <text id="city" label="City" pattern="[a-z]*"/>
    <integer id="zip" label="ZIP/postal code"/>
   </group>
  </template>
  <group id="info" label="Personal information">
   <use template="anAddress" id="address" label="Current address"/>
   <integer id="age" label="Age" minValue="0"/>
   <toggle id="parental" label="Parental accept" relevant="../age &lt;= 18"
           write-roles="parents"/>
   <text id="parentName" label="Name of parent" relevant="../parental/@relevant"/>
  </group>
  <bean id="payment" label="Payment details" type="myapp.model.PaymentInfo"/>
  <repeat id="kids" label="Children" entryLabel="Child" minRepeats="0">
   <text id="name" label="Childs name"/>
  </repeat>
 </form>
</adapforms>
```

2.1 Defining Forms

The form structure in AdapForms is defined in a XML document, which is parsed by the host application and forms the basis for the internal representation of the form. An example is seen in Listing 1.1. This sample is meant to demonstrate features rather than show a typical form. The XML form can contain the following elements representing different types of data:

`Label` Read-only text label.
`Text` Regular text element, single- or multi-line.
`Secret` Text field for masked input.
`Integer` & `Decimal` Ranged or unranged integer or decimal respectively
`Toggle` True or false.
`Choice` Single selection among fixed, mutually exclusive, choices.

`Multi-choice` Multiple selections among fixed choices.
`Date` Single date selection.
`HelpText` Text to help the user fill out the form.
`Group` Logical grouping of elements in the form. `Group` can contain the elements described above and group them together. This grouping allows for an easier overview of the form and allows for making an entire section e.g., `relevant` or `required`. An example of the `group` element is shown in line 14 of Listing 1.1.
`Repeat` Repeats are a collection of form elements that can be repeated in a list-like manner. This can e.g., be relevant for allowing users to add a variable number of children, without knowing how many elements to create beforehand. The number of repeats allow can be limited or unlimited and both as a required minimum or limited maximum. A small example of a `repeat` element is shown in line 22 of Listing 1.1.
`Bean` A bean is a `group` generated from a JavaBean—see below.

Each of these form elements has a number of possible attributes. Some of these attributes are specific to a particular form element, while others are shared among all elements. Most are optional and only `id` and `label` are required. The common attribute set is as follows:

`id` A locally unique ID. The ID must be unique among siblings in the XML tree. The ID is required for all form elements.
`label` A textual label explaining the element's purpose. The label attribute is optional for the *group* and *bean* elements and required for the rest.
`relevant` Boolean value indicating the relevance for the user. Relevance can e.g., be used to hide irrelevant elements from the user.
`required` Indicates whether or not the element is required.
`read-roles` & `write-roles` List of user roles allowed to read or write, respectively, the specific element. The user roles are handled by the host application when loading the form. The use of roles is not required. User roles may be used to allow more rights to some users, such as administrators.
`readonly` Indicates that no user is allowed to change the element value.
`uiflags` Text string which may hold UI specific information.

If an optional attribute is not present in an element, the value is inherited from the parent node. If no parent node is present, the value is taken from the nearest `defaults` element (see line 4 in Listing 1.1). The sole purpose of the `defaults` element is to provide default values, and if no default value is found, the framework defaults apply. When designing forms, one might experience defining the same element structure more than once. This could e.g., be elements of an address section. To allow for reuse of such reoccurring structures, the framework has a template option. Templates can be defined within the same XML document as the rest of the form, as in line 5–13 of Listing 1.1, with the template used in line 15, or they can be placed in a separate XML document and included in the form using the `include` element, as in line 2 in Listing 1.1. This supports the reuse of common structures, such as addresses, across forms. Similar to the

use of templates, the framework allows for the use of JavaBeans. A bean is included in the form using the `bean` element as shown in line 21 of Listing 1.1. The use of JavaBeans results in a group node in the form structure, which contains the public available properties. The framework supports properties with the following Java types: `String` (Represented as `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). The framework does not currently support beans within beans. The individual properties can be annotated in the Java class using `@AdapFormsProperty` to support e.g., renaming the label or designating required attributes. Annotating the bean class itself defines defaults for all properties.

2.2 Adaptation

One of the core features of AdapForms is the ability to adapt the form according to user input. These adaptations could be changing/setting element values based on input in other fields, or hiding/showing parts of the form that may be irrelevant/relevant based on user input. An example of the latter could be to hide the workplace address field, if the user indicates she is currently unemployed. Figure 1(a) and Figure 1(b) shows an example of this adaptation, where in Figure 1(a) the date of birth is set to 2009, thus giving an age under 18 and therefore requiring a parental section. In Figure 1(b), the age is above 18 and the parental section is replaced by the question of marital status. By hiding fields not relevant to the user, we eliminate some potential confusion for the user, so she can concentrate on the fields that are relevant to her. Likewise, by filling out some fields based on either entered information or previous knowledge, we minimise the typing needed by the user, as she can merely verify the information.

XPath. One way to perform form adaptation is to do it directly in the XML document using AdapForms' integration of XPath 2.0[1] expressions. An example of the use of XPath expressions is shown in line 17 of Listing 1.1. Here the relevance of the toggle element `parental` is dependent on the value of the sibling element `age` being less than or equal to 18. XPath is very expressive and has the advantage of placing the validation rules close to the adaptation targets. As XPath is often used in connection with XML, the developer is also more likely to be familiar with it than having to learn a completely new language. While XPath is powerful, it has some limitations in this setting, e.g., it can only affect a single state parameter as it only has one output value. It can also only look at the information typed in the form and not access previously known information.

Form Hooks. The adaptations mentioned above are pre-coded adaptations meaning that they are based on pre-coded rules and anything learnt from other users filling out the same form can not be taken into account. This is where the AdapForms goes beyond using XForms and XPath. AdapForms supports more complex adaptations through the use of form hooks. A hook is defined

by the Java interface `FormHook` which defines three methods `onValueChange`, `onRepeatEntryAdd` and `onRepeatEntryRemove`. Each hook is registered with a specific form path, described in Section 3.2, before the form instance is initialised. The same hook may be registered with more than one form path or no path, if, for some reason, the developer wants a single hook to receive all change events generated. The hook is notified through the aforementioned methods, when a change occurs at the registered path. `onValueChange` is called on value changes, whereas `onRepeatEntryAdd` and `onRepeatEntryRemove` is called for changes on repeat entries. Changes on form-level, such as submission of the form, is handled through the `InstanceCallback` interface, and here only a single callback can be registered to a form instance. Because hooks are created using Java, it allows for more options with regards to adaptation, as a hook can manipulate several values or look up previously entered information in a database. Hooks also make it possible to base the adaptations on arbitrary business logic, or more complex computations using machine learning, if these solutions are Java based. Machine learning would e.g., allow for the form to change a default choice in a multiple choice to reflect the choice of the majority of users.

2.3 Validation

When receiving data through web forms, it is essential to validate the input prior to processing. This validation can be anything from checking whether a field is a valid date to checking that a supplied user name is known. Often frameworks, such as PowerForms[4], perform this validation client-side in order to provide quick user feedback, but this means that the same validation must be done at the server as well to make sure no invalid data slips through. AdapForms performs the validation server-side, while continuously providing feedback to the client. This adds some time for validation, but ensures that the form instance on the server holds the correct information. The validation is done asynchronously, so the latency is usually not noticed.

The framework operates with three different kinds of problem definitions: `Error`, `Warning` or `Required`. `Error` indicates that the input is invalid. This could be entering a string where a number is expected. `Warning` indicates that while the input is not invalid it may be problematic. This could e.g., be if the user have put the year 1010, where it is more likely that he meant 2010. `Required` indicates that an element that is required is missing some input. These problem classifications lead to a form instance being able to be in the following states:

`Uninitialised` The form instance has been created, but is not yet initialised and can thus not handle input

`Initialising` The initialising process is ongoing

`Invalid` The form contains one or more problems. In Figure 1(a) this is highlighted by displaying an error message next to the submission button.

`Incomplete` The form is valid except for errors of the `Required` kind. Figure 1(b) shows how this could be shown to the user by a warning message next to the submission button.

`IncompleteWithWarnings` The form may hold `Required` or `Warning` errors
`ValidWithWarnings` The form only holds errors of the `Warning` type.
`Valid` The form holds no errors

On submission of the form, the host application may choose to accept or reject the form based on the form status. The host application can use the form status to perform different actions based on the status, e.g., saving incomplete forms for later completion instead of submitting it.

The validation in AdapForms takes place on three different levels: basic framework validation, validation rules, and host application. The final form status is based on the union of the errors on these three levels.

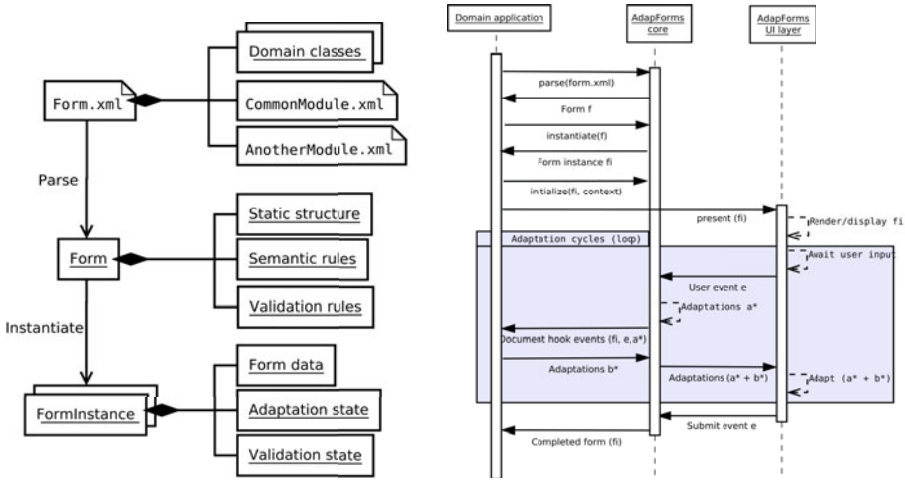
The basic framework validation includes:

Required field validation Checking that required fields contain a value. This could be shown as in both Figure 1(a) and Figure 1(b) with yellow warnings.
Type check Checks that e.g., integer elements only holds integers or date elements contains only dates.
Range check Integer elements may be upper- or lower-bounded, as shown in line 16 of Listing 1.1, as can the length of text elements
Format check Text elements may define regular expressions the input must uphold as in line 10 of Listing 1.1

This built-in validation will trigger when the form is loaded and when values are changed. The host application may not override these mechanisms to avoid inconsistencies. The XML form itself may also contain more complex validation rules by using XPath expressions. These can be placed in `validator` elements, which is then placed as children of the element it validates. Each `validator` element has a criteria, which contains an XPath expression, and a message to give the user in case of an error. An example of a validator is shown in line 8 of Listing 1.1, where it checks that the street is written only in lower-case letters. Validators are mostly used to validate values, but are not limited to this. Validators are evaluated on every change of values in the form, and not just on change of the element the validator is attached to. As a third option of validation, the host application may set or remove validation rules for individual form elements. This is most easily done using `FormHook`s, attached to the element that needs checking. Hooks would typically be used to perform more elaborate validation that is not possible with the built-in validation, such as comparing input to values in a database. The host application may decide to perform validation any time and is not limited to the use of hooks.

3 Architecture

The framework is built using the Model-View-Controller pattern, where the core framework essentially comprises the Model and Controller 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



(a) Parsing and instantiation of a form (b) Interaction of the major components

Fig. 2. AdaptForm in action

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.

3.1 Form Life Cycle

The software architecture of AdapForms can best be understood by considering the life cycle of an adaptive form, so that the various phases can be related to concrete software elements. The Instantiation and Initialisation phases can be thought of as a single phase in the abstract sense.

Definition. The form syntax and semantics are defined in a XML file external to the framework.

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 2(a) for a visual illustration of the process.

Instantiation. A concrete instance of the form is created. Using the static form as a basis, the instance will hold 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`.

Initialisation. The form instance is initialised 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 (i.e., XPath and hooks) are triggered, causing the form to be pre-adapted as much as possible before the user encounters the form.

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. Each user interaction triggers a new adaptation cycle.

Adaptations may be scripted with XPath expressions and queries. To facilitate complex domain-specific validation and reasoning, the host application may register hooks with the instance, which are triggered whenever a given form element changes its value or state.

Submission. At submission, the instance is checked for any consistency problems and validation errors, and the event is then passed (along with the instance) to a callback handler, specified by the host application.

3.2 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. Examples include look-up of form elements (form structure tree), addressing nodes in the internal state tree, defining semantic rules etc. A path resembles the hierarchical file names, and is heavily inspired by XPath and by h-maps [8]. 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 sub-forms, where the user may specify any number of occurrences of a given data type. In these cases, an extra annotation level to the instance paths is introduced, so the paths become `/applicant/jobs[1]/company` and `/applicant/jobs[2]/company` for the first and second repeat, respectively. This syntax is also similar to 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.

3.3 Element State Tree

The form instance holds state for each form element present in the instance. The actual state is held in a hierarchical data structure with form paths as keys.

This is also inspired by how FormGen [5] stores state, but with the major difference that the host application need not be implemented against specifically generated tree classes. Thus, any host application can load any adaptive form and process it at run-time, although it may of course not always be able to do

anything sensible with the entered data besides saving or forwarding it. The trade-off of 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: The unique form path and a `Relevant` state parameter, specifying whether the element is currently visible to the user. In addition, any form element capable of holding a value, also holds the following state: Current value, `ReadOnly` state parameter, `Required` state parameter (for validation), list of associated hooks, and the validation state. In Figure 2(a), 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 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, allowing direct API manipulation.

As mentioned above, the tree is partially inspired by hierarchical maps [8]; a data structure intended 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 represents a level in the tree, starting from the root. In addition to holding values, a leaf may have registered one or more message handlers, which are invoked in order, when a message is destined for the corresponding tree path. Due to the tree structure, serialisation to and from XML is straightforward.

In contrast 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 indicated above. Thus, every update to the structure is checked against the form structure. This serves several 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 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 centralised 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.

While each state tree node has its own set of parameters, these may be overruled at run-time 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. Inheritance rules are out of scope for this article, but are described in [6].

Adaptation Cycles. All adaptations within a form instance are 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 reception 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 confusion.

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

Figure 2(b) 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. A single user-initiated adaptation may trigger further adaptations and form hooks.

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.
- Populate the form elements with data and sending it to the core component.
- Adapting the presented form, as indicated by the core component.
- Highlighting validation errors within the presented form, based on feedback from the core component.
- Submitting the form, when the user deems it to be completed.

Note, that we are not restricted to graphical interfaces. Any interface, e.g., audible or tactile, could theoretically be implemented, although the mapping is not necessarily straightforward. A proof-of-concept implementation for a XHTML/ Ajax web platform has been implemented, as mentioned previously.

4 Evaluation

A framework such as AdapForms must be viewed from at least three perspectives, when considering its fitness as a tool, namely the perspectives of the developer (creating the forms), the form filler (the user filling in information), and the form handler (responsible for handling the input). We have conducted some initial user feedback studies on each of the three perspectives.

Form Filler Feedback. As mentioned above, a prototype user interface for AdapForms has been implemented The form filler feedback was conducted on a version prior to the demonstrator web UI included in the current framework version. Changes were only made to the demonstrator UI and not to the handling framework. The screen shots in Figure 1(a) and 1(b) are from the new version. In the version used for the test, the framework indicated correctly filled fields with green icons, and indicated errors and warnings with red and yellow icons respectively. Error and warning messages were only available when hovering over the icons next to the field.

In order to get feedback from form fillers, the web UI prototype was used for experimentation. Three users were presented with a form produced with the AdapForms framework, using the example UI, and a similar form implemented in standard static HTML. In order to do a reasonable comparison, the static form performed the same validation on submission as the adaptive form did during completion and gave the same messages at the top of the document.

One of the major issues discovered with the adaptive form was with the validation feedback given to the user. The coloured icons indicating validation status next to the fields were largely ignored by the users, and the users were puzzled as to why they could not submit the form. When the users noticed the icons, it was unclear to them what the different icons signified. The fact that the error messages were available by hovering over the icons was not immediately clear to the users and one user tried to click the icon. On the other hand, the users had no problems with the validation error messages shown at the top of the traditional web form after submission. This is probably because this seems to be the norm for displaying error messages, but it would also indicate that error messages should be displayed clearly in order for the user to notice and react to them. This feedback has resulted in changes in the web implementation to display the error messages directly beneath the field the error concerns, as can be seen in Figure 1(a). The green icons indicating correct fields have also been removed to avoid too much visual overload.

Another issue identified during the user testing was how to indicate the expected format. The tested version did not indicate the expected format, which especially caused problems with date fields, as dates can be formulated in a number of different ways. This observation resulted in showing the expected format next to the date field in the current version.

An important observation during the user testing was how the form was filled out top to bottom, under the assumption that the fields already filled out would not change. This is as expected, but is important to remember when defining

forms and avoid creating rules that will change fields already filled out "on the way down". The lessons from this evaluation are mainly about the construction of the UI, which is not directly the focus of the framework. The evaluation does however show the usefulness of the idea behind the framework and the use of adaptive forms. A majority of users in the evaluation liked the adaptive forms after having adjusted themselves to the different mindset of getting instant feedback and not having to wait for submission validation.

Developer Feedback. As a way of getting some feedback from developers who were to use the framework in their development work, the framework was given to a developer to evaluate. The developer was given the framework including the UI web implementation and was asked to develop a web form. The developer found the learning curve of using the web implementation quite steep due to the many Ajax HTTP calls exchanged between the client and the server. It needed a different way of thinking about control in the servlet environment and a change to letting the hooks do much of the work. This would indicate that the control flow should be made more clear in the future along with some more detailed tutorials. The work with defining the forms using XML worked well and the structure was quickly grasped. The main complaint was the lack of a more convenient way of verifying the correctness of the form, prior to instantiation. This could be solved by including a tool in the framework that would control the form against the XML Schema available and at the same time ensuring that referenced JavaBeans actually exist. As for the use of the validation rules, the simple rules were easily understood and after looking into XPath, which was unknown to the developer beforehand, this was also easily understood. One complaint was however how this would fit into already defined business logic without having to write the same thing twice. One way of avoiding defining business rules more than once could be to rely mainly on JavaBeans and then reuse these beans throughout. The general response to the use of the framework, was that after passing the initial hurdle of learning the work flow of the framework, it can save a developer a lot of time, compared to having to writing the validation of client and server side by hand. One of the authors (not the original developer of AdapForms) have utilised the AdapForms framework in the context of the eGov+ project, as reported in [2,3]. Our experience with the framework closely resembles those reported above.

Form Handler Feedback. The framework has also been used in connection with the parental leave case of the eGov+ project and included in the prototype developed in connection with this case and described in [2,3].

In connection with the development of this prototype, the concept of adaptive documents was introduced to the caseworkers set to handle the incoming forms. They were generally positive about adaptive documents and some effort was made trying to annotate existing physical documents, in order to establish relationships between fields.

The caseworkers found it challenging to analyse the existing documents and identify fields which were e.g., dependent on former input or pairs of fields depended on each other. This shows that the change from creating one static form to handle all potential users and their input and then designing a form which is capable of adapting based on input such as gender or user role is non-trivial. This challenge is something that has to be worked out between the developers in charge of developing the form and the people with the domain knowledge specific to the form in question.

5 Related Work

Form validation is a intrinsic part of form processing, and has been researched both on the desktop and in web-based settings. FormGen [5] is a Java GUI tool, which can generate a dynamic form based on data structured as a context-free grammar (CFG). FormGen uses this CFG to generate Java classes for both data model and GUI components. As in AdapForms, the data model in FormGen is a tree, and as the user adds data to the form, new nodes are added to the tree.

Dynamic Forms [7] is an example of a validating form generator using the Form Descriptive Language (FDL). FDL can be edited with an interactive editor written in FDL. When users are interacting with the finished form, Dynamic Forms provides feedback of completion and validity to the user by colour coding. The idea of colour coding is also known from other frameworks and is also used in the example implementation of the web user interface for AdapForms.

XForms [9] is a W3C alternative to the common HTML form. XForms is, as AdapForms, XML based, and differs from regular HTML forms by separating data and markup. Thus, one form can be specified and used several different places with different markup. XForms extends the standard HTML form with validation and adaptation. The validation can be performed using XML Schema. Types are defined using XML Schema, and input can later be validated against these types. XForms can also mark input as read-only, required, or relevant. Adaptation of the document can be accomplished by changing the value of a field's relevance at run-time. The GUI may hide all irrelevant fields as not to confuse the user. AdapForms have the same capabilities for in-document validation and adaptation, but extends these capabilities by allowing more complex validation and adaptation within the framework, such as using database look-ups and other server-side techniques, as described in Section 2.2 and 2.3.

JavaScript is widely used to make regular static HTML forms more responsive and adaptive. Such JavaScript can however be tedious and error prone to develop, which is addressed by PowerForms [4] by supporting client side validation for user input in HTML forms. It generates an interactive form based on a HTML static form and an PowerForms specification. The validation is continuous, showing the validity as the form is filled. The status is again indicated by red, yellow or green. The rules are constructed using a simple `if-then-else` structure, or using regular expressions. Regular expressions are used to validate the format of the text being entered. The rules are transformed into a finite-state machine

expressed in JavaScript, which validates the form input. The `if-then-else` constructs can be chained to make the validity of fields depend on other fields.

6 Conclusion

Forms are an integral part of modern living, yet the basic form has changed little over the decades—it may these days reside on a web page, but the general form remains the same. Validation of forms is usually either a lengthy process (fill-out, submit, review error messages, revise, resubmit). or relatively simple-minded using specialised JavaScript code. We have in this paper presented a general framework to define forms, designate acceptable field values, the inter-relationships between fields, as well as an architecture to transparently validate, report state of, and adapt a form as the user is filling it out. Depending on the needs of the developer, the validation can be simple (type and range checks), XPath based, or advanced such as integrating server-side business logic through JavaBeans. Depending on the user's input, the form can adapt itself, freeing the user from having to deal with an over-general form covering all possible permutations. Our initial evaluation leaves us hopeful that this is a valid approach, though there certainly still are many aspects of effectively communicating expected values and form state to the user that need to be explored. Likewise, the prototype implementation, while suited for our experiments has room for improvement, notably in terms of scalability.

Acknowledgments

The eGov+ project is financed by a grant from NABIIT, the Danish strategic research programme for nano-, bio-, and IT-sciences.

References

1. Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., Siméon, J.: XML path language (XPath) 2.0, W3C recommendation. Tech. rep., The World Wide Web Consortium, W3C (2007)
2. Bohøj, M., Borchorst, N.G., Bouvin, N.O., Bødker, S., Zander, P.O.: Time collaboration. In: Proceedings of the 28th International Conference on Human Factors in Computing Systems. ACM, Atlanta (2010)
3. Bohøj, M., Bouvin, N.O.: Collaborative time-based case work. In: Proceedings of the Hypertext Conference 2009, pp. 141–146. ACM, New York (2009)
4. Brabrand, C., Møller, A., Ricky, M., Schwartzbach, M.: Powerforms: Declarative client-side form field validation. World Wide Web 3(4), 205–214 (2000)
5. Brandl, A., Klein, G.: FormGen: A Generator for Adaptive Forms Based on EasyGUI. In: Proceedings of HCI International Human-Computer Interaction: Ergonomics and User Interfaces, vol. 99, pp. 22–26 (1999)

6. Gammelmark, H.: Adaptive Forms. Master's thesis, Department of Computer Science, Århus, Denmark (April 2009), http://adapforms.gammelmark.eu/files/adapforms-thesis.pdf
7. Girgensohn, A., Zimmermann, B., Lee, A., Burns, B., Atwood, M.: Dynamic forms: An enhanced interaction abstraction based on forms. In: Proceedings of Interact 1995, pp. 362–367. Chapman & Hall, Boca Raton (1995)
8. Ørbæk, P.: Programming with hierarchical maps. Tech. rep., Aarhus University, PB-575 (2005), http://www.daimi.au.dk/publications/PB/575/PB-575.pdf
9. W3C: XForms 1.1 (2007), http://www.w3.org/TR/xforms11/