Mewa: Meta-level Architecture for Generic Web-Application Construction

Adrian Lienhard, lienhard@student.unibe.ch
Software Composition Group
University of Bern
Bern, Switzerland

November 16, 2003

Abstract. Web-applications are very popular, lightweight applications that entirely run in web-browsers over the internet. In today's business, web-applications become more and more complex but they still need to be fast developed, flexible for changes and easy to maintain — conventional techniques often lack these properties. High-level, cleanly layered solutions open promising possibilities to overcome these difficulties. This paper presents a lightweight, object-oriented, metadata-driven approach to build better engineered and easier evolvable and maintainable web-applications.

Contents

1 Introduction **3**
- 1.1 Context . . . 3
- 1.2 Problem: Bad Separation of Concerns . . . 4
- 1.3 Mewa: A Meta-Layer Solution . . . 5
- 1.4 Implementation . . . 5

2 Design **7**
- 2.1 Overview . . . 7
  - 2.1.1 Related Work . . . 8
- 2.2 Metaobject and its Attributes . . . 8
  - 2.2.1 Metaobject . . . 9
  - 2.2.2 Attributes . . . 10
- 2.3 Generic Components . . . 12
  - 2.3.1 Viewer . . . 12
  - 2.3.2 Editor . . . 12

3 Example **13**
- 3.1 Metaobject creation and configuration . . . 13
- 3.2 Instantiating components . . . 15
- 3.3 Creating a Batched List of Persons . . . 16

4 Conclusion **18**

Chapter 1

Introduction

1.1 Context

Many web-applications provide the functionality of saving, persisting and presenting data. There are even applications that almost only have this task: Team- and project-managements or customer-relationship-managements (CRM) often involve only very simple business logic in their processes (e.g., some workflow rules).

These applications mainly consist of the following layers: persistency, domain-model (state and business-logic) and presentation (including application-logic). The presentation-layer can be mainly divided into *views* which present the data in desired ways and *editors* which allow the user to modify and add new data.

- The *domain-model-layer* is a net of *domain-objects* which represents or encapsulate an application's state and business logic and should be unaware of the presentation-layer. To make the model persistent, often RDBMS or OODBMS are used. Some application designs avoid the domain-model layer and access the persistency layer directly from the presentation layer because it is still a challenge to map object-oriented models to relational schemas. For complex applications, this is no solution because the layering and with that the separation of concerns gets lost which leads to a poor design.
- The *presentation-layer*:

  A *view* is a graphical representation (rendered as html) of a very specific and small subset of domain-objects. Normally they present an aspect of interest to the user (for example a customer and all products he has bought). The views are composed by subviews. At last, each domain-object has its own view.

  An *editor* acts similar to a view but lets the user edit or add a new domain-object to the model (rendered as html forms). Editors in-

clude the most application-logic: They have to render the appropriate fields, populate dropdown boxes, embed subforms etc. Furthermore, validation of the submitted data has to be done according to the specified business rules. When validation rules are broken, the form has to be redisplayed with the cached user input and appropriate error messages.

1.2 Problem: Bad Separation of Concerns

Conventional web-applications often include a lot of presentation and application-layer code (generation of html, processing of user requests etc.) which is tightly coupled with the domain-layer beneath. In the worst case there is no clear layering so that the user-interface code, the logic and persistency code is mixed up. Scripting and server-pages technologies such as ASP (e.g., with VBScript or JavaScript), JSP and similar techniques do not provide high-level abstractions on their own. The render-centric approach which forces developers to deal with HTTP request/response processing is not adequate for complex applications, because html generation often gets thightly coupled with the domain-model: A single script has responsibilities spanning several layers. The script must accept input, handle application logic, handle business logic and generate output (presentation) [KD02]. The coupling between the layers makes it very difficult or even impossible to develop, test and evolve the domain-model in isolation from the presentation.

> "Design decisions are very hard to track in a low-level implementation; as the application evolves, changes can easily lead to inconsistencies. Because Web-based applications tend to evolve quickly, with frequent updates and redesigns, poor maintainability is a critical problem." [GG99]

The main reason of missing suitable abstractions may be that the web implementation model which is based on low-level technologies and which originally was only supposed to act as an information medium (request-response paradigm) does not relate well to state of the art software development models. A key aspect is to decouple design decisions concerning the domain-model from those concerning presentation and user interaction. Meta information about the domain-model is very often implicitly used in the presentation-layer (e.g., hard-coded in the rendering by using scripts). And worse, this information is duplicated wherever a domain-object is being represented.

> Example: A product management application may implement a listing of products, views, editors and reports – each of those components needs to know how to access a domain-object "product", how to get a collection of producers and what business rules exist to add new products. Now, when the requirements for products change or when we have to add a web-shop, information is used

again in different places and can not easily be reused if we do not have an appropriate abstraction. This leads to poor design with high maintainability and evolution costs.

1.3 Mewa: A Meta-Layer Solution

The conventional, render-centric approaches which have the effect of bad separation of concerns lead to the assumption that a higher level of abstraction is needed. The meta-information about objects of the domain-model-layer has to be gathered in one and only one place: a meta-model. Its metaobjects describe the domain-model objects. With this information in hand, we are able to write *generic* components which act as viewers, editors etc. — for all domain-objects without the need to code each component from scratch.

Now, changes in the domain-model do not effect many different parts (especialy not on the presentation layer) of the application anymore since the meta-layer constrains the scope of reconfigurations. Only the meta-model has to be adapted. Since all the generic components only rely on the meta-model, these changes are reflected immediately.

General adaptability is achieved by dynamic adjustment of metaobject's configurations (at runtime). This means that we can adapt the presentation and behavior (e.g., validation rules) of domain-objects according to their context in the application without the need to code different views. Most adaptation concerning the look and feel of the user interface can be obtained by Cascading Style Sheets (CSS).

Moreover, this approach opens new possibilities to extend the framework with new features such as security (or user based capabilities), personalization, transactions, versioning and concurrency support, internationalization, (pdf) reports etc.

1.4 Implementation

Mewa has been implemented in Smalltalk with Squeak [Squ] using Seaside [Sea]. Seaside is a sophisticated web-application framework which already provides a very good abstraction from the underlying technologies. It models an entire user session as a continuous piece of code. With this very unique feature (e.g., compared to the servlet models) it abstracts the http request-response cycles away and makes a clean and simple object oriented design possible.

To gain real experience with the proposed approach, a team- and project-management application has been developed for a small company of the size of about 5-10 people. All their customers, projects, teams and members are managed through the application. It offers tasks, checklists, working time-logging, billing, addressbook etc. The data is presented in various perspectives (or aspects) such as "project" or "member". The perspective "project", for example, shows all tasks and checklists concerning a project whereas "member"

only shows items of interest to a specific member. In addition, there are several reports and statistics to visualize data. See figure 1.1 on page 6.

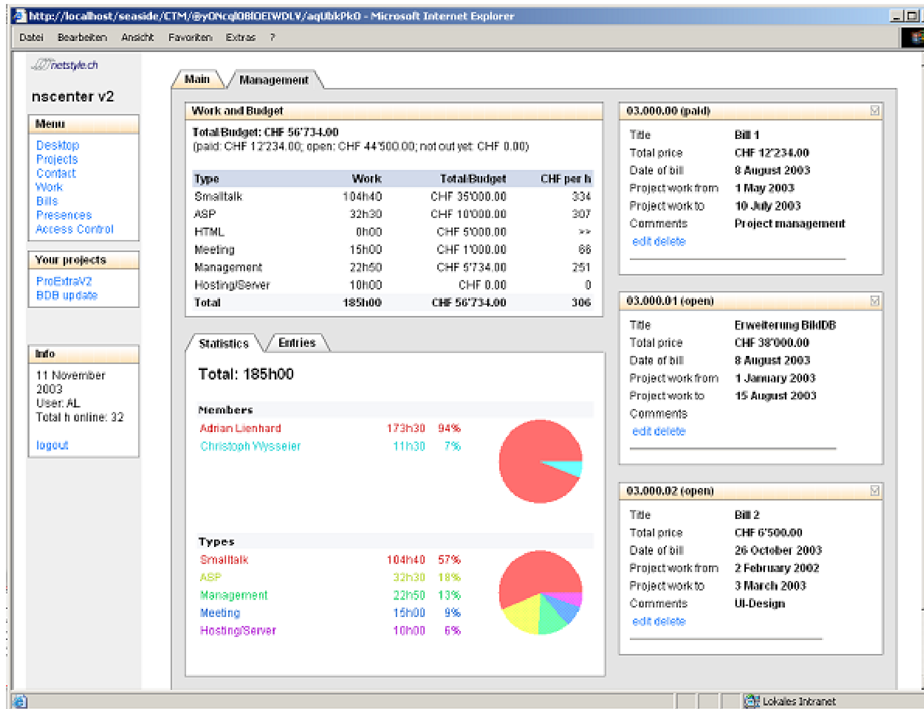

Figure 1.1: Screenshot on project view

Chapter 2

Design

2.1 Overview

The design of the proposed architecture is intended to be as simple and straight forward as possible and flexible enough to be easily extended to special needs.

The main idea is to provide a meta-layer which introduces a level of indirection between the domain-model and the presentation-layer. This makes it easy to write simple generic presentation components which only interact with the meta-layer. A metaobject reifies important parts of the structure and functionality of its baseobject (the domain-object) and provides an interface for services such as data caching and validation (for editors), formatting output etc. Metaobjects act mainly as descriptive entities with basic behaviour to interact with the baseobject they represent. This makes writing generic components which present or edit domain-objects significantly easier.

Each metaobject is linked to its baseobject. There may exist several metaobjects for the same baseobject in parallel because each presentation component has his own instance(s) of metaobject. Metaobjects are only created when needed. The presentation components access baseobjects only indirectly through the metaobject and should not need to access baseobjects directly.

Figure 2.1 shows the newly introduced meta-layer in between the two existing layers. With this architecture the presentation components get much slimmer and most of them can even be replaced by a generic component. This is possible because the logic to deal with a specific domain-object is shifted from the components to the meta-layer. The components themselves just have to understand the unified interface of metaobject and its attributes to be able to interact with domain-objects.

Components provide the presentation logic by using metaobjects. They are generic and applicable for all domain-objects which provide metaobjects. The current implementation includes components such as views (visual representation of baseobjects) and editors (input forms with validation, caching). But it

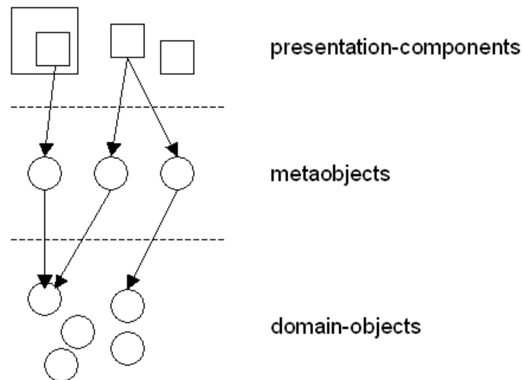

Figure 2.1: Introduced meta-layer: interaction between presentation components, metaobjects and baseobjects

is not limited to this selection at all.

For the design of the meta-model – component interaction, the visitor design pattern [ABW98] plays a key role: Components are visitors, which walk over the metaobject's attributes when rendered.

2.1.1 Related Work

In comparison to [Fer89], our metaobjects are much more specific to model our web-application specific domain. They do not provide real reflection in the common sense (no computational reflection). Much more, they provide descriptions and services we need for constructing web-applications. Our metaobjects could also be understood as proxies [ABW98] with added knowledge and the abilty to reason about the structure of their objects.

VisualWorks [Vis03] uses a related approach to specify the graphical user interface. Its so called window-specs define the layout, look and feel and the model of each ui-element. The information is stored in a class method and is used at runtime to create a window. Normally the specification is created automatically from the UIPainter (a WYSIWYG GUI-editor). In comparison to Mewa, this approach is very UI-centric: A specification is used for one and only one window whereas Mewa concentrates on describing the domain-model rather than the user interface. The meta-layer is still clearly decoupled from the presentation-layer which makes it possible to be used in very different contexts.

2.2 Metaobject and its Attributes

Default instances of metaobjects are created by the baseobject itself and can be adapted dynamically at runtime if needed. This dynamic configuration makes

type / *arity*	*"owned"*	*relationship*
single	{Text, Date, ...}Attribute (text-inputs, composition of input fields)	SingleRelationshipAttribute (dropdown-list, radio-buttons)
multiple	MultipleAttribute (embedded subforms with add/edit/delete - functionality)	MultipleRelationshipAttribute (checkboxes, multi-select listboxes)

Table 2.1: Different types of attributes (and their graphical representation in editors)

it possible to adapt the behavior and (visual) appearance of an object based on its state, on the context or on the current user (security, personalization).

Metaobjects consist of attributes which reify the structure of the baseobject. Attributes provide information about labels, types (e.g., text, date, single and multiple relationships etc.), validation and security rules, formating output etc. The attributes describe accessors and not instance variables since this is more flexible. Accessors may be simple setter and getter methods but may also simply return derived values. In this case they are marked as read-only by the corresponding attribute.

Focusing on baseobjects we can make the following decomposition of a domain-object: The internal structure of a baseobject is a composition of other baseobjects and non-baseobjects (strings, numbers, dates etc). We distinguish between single or multiple (implemented using collections) compositions and we say an object is "owned" by the baseobject or there exist a relationship to an external collection of (base)objects (this correlates to the notion of aggregation and composition in UML class diagrams).

This examination gives the table: Table 2.2 on page 9. The Figure 2.2 on page 10 shows the corresponding class hierarchy.

2.2.1 Metaobject

The following section presents the public protocol of *Metaobject*:

- **accessing-attributes**
  `addAttribute: anAttribute`
  `hideAttributeOf: aSelector`
  `attributeOf: aSelector`
  `attributes`
  `allSelectors`
  As said above, attributes are identified by selectors. It is assumed that the baseobject provides the method(s) named selector (and `<selector>: anObject` if it is not read-only). The order of adding attributes is preserved and used when accepting a visitor. This is important because we

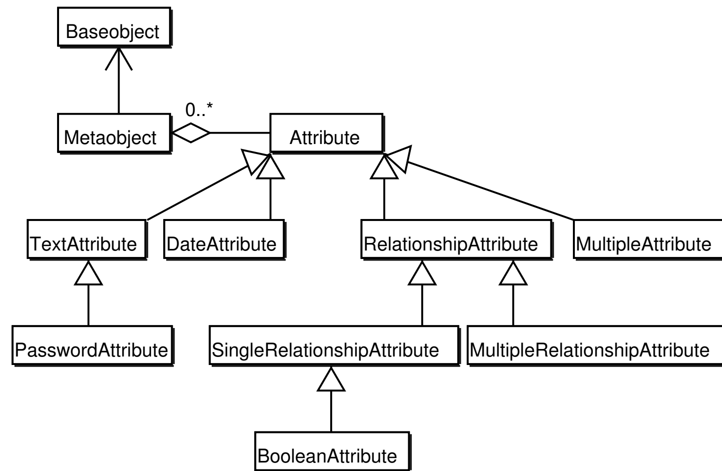

Figure 2.2: Metaobject and Attribute class hierarchy

want the items of the visual representation to be always in the same order. This could be adapted by reordering attributes if required. Often the ability to hide attributes in some situations is needed: For instance if we want to show a list of bills for a project, the bills should not show the project name again, since this would be unnecessary redundant information for the user.

- **services**
  `validate` - by iterating over the attributes, validation errors are collected and returned. If an empty collection is returned there are no validation rules broken.
  `refreshCache / commitCache` - refreshes/commits cache of each attribute from/to the baseobject.
  `accept: aVisitor` - starts the visitor on metaobject iterating over all attributes which are not hidden. The visitor is a presentation component (see 2.3) which is about to render the metaobject's baseobject.

2.2.2 Attributes

The following section presents the public protocol of Attribute and its concrete subclasses and some examples on how and why it is used:

- **configuration**
  `label: aString, formatWith: aBlock, hide, readOnly`
  A label sets the field name (e.g., "birthday"). Optionally a format block can be provided to adapt the standard behavior of sending asString to the rendered object. If an attribute is marked hidden, it is never visited but it remains accessible. This behavior can be used to create customized editors. For example to add a subform to the standard editor which handles the aspects of the hidden attribute and replaces its standard rendering. Read-only attributes are discarded by editors but not by viewers. They may be used for derived values like "age" of a person which depends on the birthday.

  The concrete subclasses of *Attribute* provide the following configuration protocol

  - *TextAttribute*: `maxLength: aNumber, multiLine: aBoolean`
  - *RelationshipAttribute*: `relationshipTo: aBlock` expects a block which returns a collection of items which the baseobject may have a single or multiple relationship to. Optionally `relationshipTo: aBlock formatWith: aFormatBlock` provides the possibility to format how the referenced baseobject is printed: aFormatBlock is a one parameter block which takes one item and returns a formated string. This may be used for instance if we want persons to be rendered with their initials rather than what the default implementation `Person>>asString` implements.
  - *SingleRelationshipAttribute*: `nilItemString: aString` - sets the label in case of no specified relationship i.e. the instance variable is nil. This so called "nil item" is rendered as first item of drop down boxes (strings may be 'unknown', 'none', 'not selected' etc.).
  - *MultipleRelationshipAttribute*: There are no further configuration methods than inherited from the superclass.
  - *MultipleAttribute*: `baseClass: aClass` - sets the class with which new "owned" instances are created.

- **accessing**:
  `value, formatedValue`
  `cache, cache: anObject`
  `refreshCache, commitCache`
  Value and formatedValue returns the current value for this attribute from the baseobject. The cache-methods are used by editors to proivide chaching of the entered data when validation errors occur.
  A special case are instances of DateAttribute which cache entered dates as arrays of size three. If the user input would be tried to be converted directly to date instances, invalid data could not be cached. The conversion takes place when submitting the cache.

- **validation**:
  `addValidationRule: aBlock errorString: aString`
  `addRequiredRule`
  `validateCache`
  Validation rules are added giving blocks which take one value and return a boolean (true if rule holds). The method validateCache returns the error-string of the first rule which does not hold or nil, if there are no errors. `addRequiredRule` is a convenience method to add the rule which assures that a value is set. For Text- or SingleRelationshipAttributes this means "not nil", for Multiple[Relationship]Attributes it means "not empty collection".

2.3 Generic Components

The currently implemented components are simple but very generic. They subclass from *Visitor* and implement methods such as `#visit{Text, Date, SingleRelationship, ...}Attribute: anAttribute`. To adapt to special needs (e.g., different rendering styles) the components can easily be subclassed.

2.3.1 Viewer

The standard viewer implements the following rendering-methods that can easily be overridden to change the rendering style: `renderContentOn: html` which is Seaside's the standard method to render components. The standard renderer renders a table and then calls `renderFieldsOn: html` which starts the visitor. Finally each field is rendered by calling `renderLabel: aString value: anotherString`. For an example of writing your own viewer see chapter 3.

2.3.2 Editor

The editor has to implement much more functionality than the viewer but is similar to adapt to a different rendering. The hook methods are `renderMainOn: html`, `renderLabel: aString control: aBlock`, `renderButtonsOn: html`, `renderValidationErrorsOn: html`. The main functionality which the standard editor implements are:

- Handling form fields: Text (text field), date (three text fields), single/multiple relationships (dropdown list, checkboxes) and multiple owned domainobjects (new, edit, delete actions with embedded sub-forms which are instances of editors and viewers themself). For an example of all these fields see Figure 3.1.
- Validation: According to the metaobject validation and chaching of user input is done. Rendering of validation errors.

Chapter 3

Example

The following example shows how simple it is to create fully functional views and editors. As we can make use of the standard components there is almost no rendering code and application logic needed.

Let us assume we want to model person with name, birthday, parents, several phone numbers and interests... First of all we create a class named *Person* which implements the desired accessors. We also need a domain-model class – the root of our model. Its (sole) instance holds persons in a collection and implements methods such as `persons`, `femalePersons`, `malePersons`, `addPerson: aPerson` etc.

In comparison to conventional aproaches we do not write a presentation component which is specific for our person domain-objects. Normally we would have to code how name, birthday etc. is rendered and accessed to display a person. When we write a component which shows a list of persons (see 3.3) we usually can't reuse this code because we want the representation to be slightly different. If we want to reuse, we have to add decision logic like 'if context = #list then renderPhoneNumbers else ...'. Creating an editor, again makes it almost impossible to reuse code we wrote for one of the views.

With Mewa, only configurations of metaobjects for each domain-object are needed, what we show in the following section 3.1. Then the standard components can be used. Figure 3.1 shows the result of our new person editor.

If we want to modify the look or behaviour of components we can subclass the standard viewer or editor. We will show this in section 3.2 where we create a batched list of persons.

3.1 Metaobject creation and configuration

Instances of *Person* should be able to return default metaobjects for themselves. The code for this default configuration may look something like the following listing:

Figure 3.1: Standard editor for a person baseobject

```
Person>>metaobject
	| metaobject |
	metaobject := Metaobject for: self.

	metaobject addAttribute: ((TextAttribute for: #firstname)
		label: 'Name';
		maxLength: 30;
		addRequiredRule).

	metaobject addAttribute: ((DateAttribute for: #birthday)
		label: 'Birthday';
		addRequiredRule).

	metaobject addAttribute: ((TextAttribute for: #age)
		label: 'Age';
		readOnly).

	metaobject addAttribute: ((BooleanAttribute for: #male)
		label: 'Sex';
		trueItemString: 'male' falseItemString: 'female').

	metaobject addAttribute: ((SingleRelationshipAttribute for: #father)
		label: 'Father';
		relationshipTo: [ Session currentSession domainmodel malePersons ]
			formatWith: [ :each | each firstname , ' (' , each sex , ')'];
		nilItemString: 'unknown').

	metaobject addAttribute: ((SingleRelationshipAttribute for: #mother)
		label: 'Mother';
```

```
                relationshipTo: [ Session currentSession domainmodel femalePersons ]
                        formatWith: [ :each | each firstname , ' (' , each sex , ')' ];
32              nilItemString: 'unknown').

        metaobject addAttribute: ((MultipleRelationshipAttribute for: #interests)
                label: 'Interests';
36              relationshipTo: [ self class interests ];
                addValidationRule: [ :collection | collection size > 1 ] errorString: 'Select at least two interest items').

        metaobject addAttribute: ((MultipleAttribute for: #phoneNumbers)
40              label: 'Phone Numbers';
                baseClass: PhoneNumber formatBlock: [ :each | each type , ': ' , each number ]).

        ^metaobject
```

The statements on line 5-12 add simple text- and date attributes. On line 14 we add a read-only attribute for age. It is read-only because we do not store the age in an instance variable but rather derive it from the birthday. Thus, age will be visible in viewers but not in editors.

Statements 22-26 add the attribute for #father. `Session currentSession` returns the current Seaside session from where we access the domain-model. The simplest solution is to use a single instance of domain-model (singleton pattern) which is shared for all users. There may be different designs needed depending on how persistency is implemented — but this is out of scope.

On line 34 we add a multiple relationship to a collection of possible interests. To illustrate the use of rules, we add a custom rule that assures that at least two items are selected.

At last we add a MultipleAttribute for phoneNumbers. PhoneNumber is a new domain-class we add. It simply holds the type of phone (private, business, mobile etc.) and its number. Similar as for Person we write `PhoneNumber>>metaobject`. The descriptions will be used by the editor rendering a person to add, edit or delete phone entries from person by nesting an editor for the phone domain-object.

3.2 Instantiating components

Creating an editor or a view for a person is simple now. We can just call the editor-/viewer component from another Seaside component:

```
newPerson := self call: (EditorVisitor model: Person new).
```

This creates an editor on a new instance of Person and calls it. It will return the newly created person when the user has successfully filled in the form or nil if he cancels. Figure 3.1 shows the editor.

Similar to the editor we create views for a person:

```
newPerson := self call: (EditorVisitor model: person).
```

3.3 Creating a Batched List of Persons

Now, we would like to create a list for our persons. Standard views render their object similar to editors with label-value pairs in a vertical list which is not suitable for us now. So we would like a new viewer which renders objects in a different way: They should be rendered in one line without labels and each item in a seperate table data tag. Figure 3.2 shows the desired result.

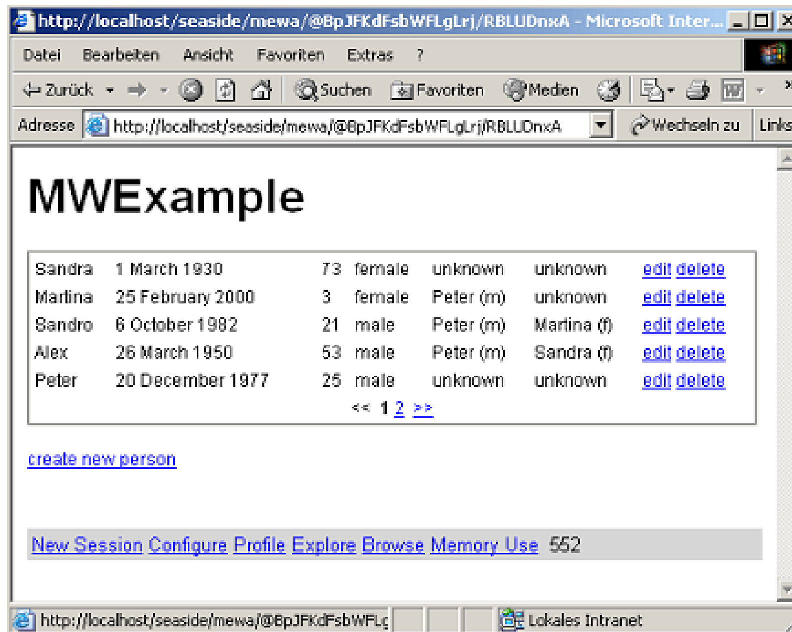

Figure 3.2: List with custom person viewers

We achieve the adaptation of the view by subclassing from `ViewerVisitor`. So we create a new class named `ListItemViewer` and override `renderContentOn: html` and `renderLabel: aString value: anotherString` as follows:

```
1   ListItemViewer>>renderContentOn: html
            self renderFieldsOn: html
```

The standard implementation puts a table tag around `renderFieldsOn: html` which starts the visitor (metaobject accepts the visitor which is the component itself!). Because we do not want a table to be rendered around each line in our list we remove it. To render each item of the object in a single table data tag without table row we do the following:

```
1   ListItemViewer>>renderLabel: aString value: anotherString
            renderer tableData: anotherString
```

Now, we are ready to write the component which will render a list of persons. We create a new class with the following methods:

```
1  MewaExample>>initialize
           | views |
           super initialize.
4          views := self session model persons collect: [ :each |
                   ListItemViewer metaobject: (
                           each metaobject
                                   hideAttributeOf: #interests;
8                                  hideAttributeOf: #phoneNumbers;
                                   yourself) ].
           batcher := WABatchedList new
                   items: views;
12                 batchSize: 5;
                   yourself
```

On initialization, a collection of ListItemViewer instances (one for each person domain-object) is created. Because we do not want to show the aspects 'interests' and 'phoneNumbers' we modify the metaobject of each person to hide them. Like this, we do not have to write any special rendering code which modifies the output! Line 10 creates an instance of WABatchedList, a standard Seaside component.

What follows is the rendering of a table with each viewer in a separate table row and the actions 'edit' and 'delete' on one row.

```
1  MewaExample>>renderContentOn: html
           html h1: self class.
           html cssClass: 'TABLE-LIST'.
4          html table: [
                   batcher batch do: [ :viewer | html tableRow: [
                           html render: viewer.
                           html tableData: [
8                                  html
                                           anchorWithAction: [ self edit: viewer metaobject baseobject ]
                                           text: 'edit'; space;
                                           anchorWithAction: [ self delete: viewer metaobject baseobject ]
12                                         text: 'delete' ] ] ].
                   html attributeAt: #align put: #center.
                   html tableRowWith: batcher span: 8 ].
                   html break.
16                 html anchorWithAction: [ self addNew ] text: 'create new person'.
```

The 'edit' action, for example, is implemented like this:

```
1  MewaExample>>edit: aPerson
           self call: (EditorVisitor model: aPerson)
```

That is all the code we need for our list component!

Chapter 4

Conclusion

We have shown that conventional approaches can lead to bad web-application designs and thus take more time to be developed and are harder to enhance and adapt to changing requirements. Mewa introduces a meta-level which is capable of describing the domain-model and so the presentation-layer can be decoupled from the domain-layer. The example has shown that many presentation components of web-applications that normally would have to be implemented can be replaced by a couple of generic components. Aside from less code Mewa makes it possible that evolution of the domain-model has much less effect on the presentation-layer. Furthermore the architecture opens promising possibilities to enhance applications with additional functionality like security or concurrency support.

The approach seems worthwhile to be considered for data-centric web-applications because they have a lot of views and editors. Though, for special types of applications which do not focus on data representation but much more on complex processes and business logic with many specialized input-forms, the approach may not fit well but will probably still be useful for a fast prototyping.

One of the strength of Mewa is its flexibility and simplicity. It can easily be enhanced to special needs that arise with concrete problems. For instance, we could consider improving the layout adaptability by introducing layout-managers which would make it possible to change the look of the generic components without the need to create separate subclasses.

Bibliography

[ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion.* Addison Wesley, 1998.

[Fer89] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989.

[GG99] Hans-W. Gellersen and Martin Gaedke. Objectoriented web application development. *IEEE INTERNET COMPUTING*, Jan, Feb 1999.

[KD02] Alan Knight and Naci Dai. Objects and the web. *IEEE SOFTWARE*, Mar, Apr 2002.

[Sea] Seaside. http://www.beta4.com/seaside2/.

[Squ] Squeak. http://www.squeak.org/.

[Vis03] Cincom Smalltalk, sep 2003. http://www.cincom.com/scripts/smalltalk.dll/.