Magritte Tutorial *

Lukas Renggli
renggli@iam.unibe.ch
www.lukas-renggli.ch

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

December 1, 2006

1 Getting Started

Fire up the prepared image and get yourself ready for the exercises. Make sure your Seaside server is running within the image by browsing the counter application at *http://localhost:8080/seaside/counter*.

The image you are using has been created from a fresh Squeak 3.9 image, by loading the following packages from SqueakMap: *Shout*, *eCompletion* and *Magritte*. Also a package called *Magritte-Tutorial* has been added, where you can put your code.

You can easily create your own development image by taking a fresh 3.7, 3.8 or 3.9 image and loading Magritte from SqueakMap yourself, note that your computer will be busy for a while because this will pull in a few other prerequisites (such as a Web server and Seaside).

The exercises start simple and with detailled instructions on how to perform the given tasks. Exercises marked with a star are a bit trickier, you might want to solve them later on. Sometimes you will probably not exactly know what class to use, what method to call or what parameters to pass, use the power of Smalltalk (senders, implementers, references, ...) to browse the source-code of Magritte, you might even discover some other features that were not presented during the lecture.

2 Juggle with Descriptions

Have a look at the source-code of the classes `MAPersonModel`, `MAAddressModel`, and `MAPhoneModel`, that have been presented during the lecture and that are

*The solutions are kindly provided by Damien Cassou, `damien.cassou@laposte.net`

pre-installed in your image. Browse to *http://localhost:8080/seaside/person-editor* and check if you can see all the features presented during the lecture. All the following exercises will be built upon this simple model.

Exercise 1 Add a new field that holds a *Comment* about the person, display it as a text-area field as the last element. Test it in the Web browser by starting a new session on the same application. If you are required to add more than one new method by hand you did something wrong.

Solution 1 In the `MADescription` hierarchy there is a class `MAStringDescription` which is used to describe a string. You can try to use it like this:

```
MAPersonModel class>>descriptionComment
    ^ MAStringDescription auto: 'comment' label: 'Comment' priority: 100
```

The form input rendered is a text-input whereas the subject asks you for a text-area (a text-input on more than one line). Have a look at `MAStringDescription` subclasses. You will see a `MAMemoDescription` class with a `#lineCount:` method. Use this class:

```
MAPersonModel class>>descriptionComment
    ^ MAMemoDescription auto: 'comment' label: 'Comment' priority: 100
```

A text-area is then displayed.

Exercise 2 Add a new field that holds the *Nationality* of the person, display it as a sorted drop-down box with a few selectable countries. Put it right after the *Address* field. The default choice for new objects should be *Switzerland*.

Solution 2 When you want to ask the user to choose one element in a list of multiple, you should use the `MASingleOptionDescription` class. This class is a subclass of `MAOptionDescription` which accepts an `#options:` method to specify the different choices. The default choice (displayed by magritte if no other choices have been selected) is chosen with the `#default:` method:

```
MAPersonModel class>>descriptionNationality
    ^ (MASingleOptionDescription auto: 'nationality' label: 'Nationality' priority: 55)
        options: #( 'Switzerland' 'France' 'Germany' );
        default: 'Switzerland';
        beRequired;
        yourself
```

You can use the `#beSorted` method to sort values in the list.

Exercise 3 Add a new field that holds the *E-Mail* of the person, display it as a required text-field. Add custom conditions to force the user to give a valid e-mail address. Don't allow addresses from the providers `hotmail.com` and `gmx.com`, `gmx.de`, `gmx.it`, etc. Test your code in the Web browser.

Solution 3 An *E-Mail* is basically a string. So, to start, you can just ask for a string:

```
MAPersonModel class>>descriptionEmail
    ^ (MAStringDescription auto: 'email' label: 'Email' priority: 95)
                beRequired;
                yourself
```

If you open a browser on `MADescription` class (protocol *validation*), you will see a `#addCondition:labelled:` method.

```
MAPersonModel class>>descriptionEmail
    ^ (MAStringDescription auto: 'email' label: 'Email' priority: 95)
        addCondition: [ :value | value matches: '#*@#*.#*' ]
            labelled: 'Please enter a valid email';
        addCondition: [ :value | (value matches: '#*@hotmail.com') not ]
            labelled: 'Hotmail users not allowed';
        addCondition: [ :value | (value matches: '#*@gmx.#*') not ]
            labelled: 'GMX users not allowed';
        beRequired;
        yourself
```

Exercise 4★ Reuse the description of the *Nationality* from `MAPersonModel` in the model of the address `MAAddressModel` by calling the appropriate description of the person and changing the label to *Country*. Make sure not to modify the original description by creating a copy. Test it in the Web browser.

Solution 4 Descriptions are created in methods. To reuse a description, you can just send the method:

```
MAAddressModel class>>descriptionCountry
    ^ MAPersonModel descriptionNationality copy
        label: 'Country';
        yourself
```

Question 5 Make a rough guess on how much code was saved by using Magritte compared to a manual approach in Seaside. From when on does it make sense to use a meta-model? What are the advantages? What are the disadvantages?

3 Integration into Seaside

Up to now we have been working with descriptions only, we didn't write any line of Seaside code. Moreover the model was not remembered somewhere and therefor was lost between different sessions. In this section we will concentrate on Seaside and build a simple user-interface that allows us to manage multiple persons. To get an idea of how the result could look like, see Figure 1.

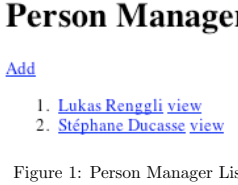

Figure 1: Person Manager List

Exercise 6 Start by creating a new sub-class of `WAComponent` called `MAPersonManager`. Add a class-instance-variable called `Persons` that will serve us as a simple place to keep the model objects. Initialize it with an empty `OrderedCollection`. Register the newly created class as a new Seaside entry point.

Create a method `#renderContentOn:` that displays the heading *Person Manager*. Test the setup of your new application in the Web browser.

Solution 6 Create this methods:

```
MAPersonManager class>>persons
    ^ Persons

MAPersonManager class>>persons: aCollection
    Persons := aCollection

MAPersonManager class>>initialize
    super initialize.
    self persons: OrderedCollection new.
    self registerAsApplication: 'personmanager'

MAPersonManager>>renderContentOn: html
    html heading: 'Person Manager'
```

Don't forget to execute `MAPersonManager initialize`. The `#rendererClass` method ask Seaside to use the new rendering technique.

Exercise 7 Create an action-method `#add`, that creates a new instance of `MAPersonModel`, calls the default Magritte editor and adds it to our collection. Note that Magritte will answer `nil`, if the user hits the cancel-button in the editor. Create a link in your page that will execute the `#add` method.

Solution 7 A Seaside component is created from a model using `#asComponent`. You can add buttons to save and cancel using `#addValidatedForm`. If you want to use another component instead of the current one, use the method `#call:`. This method returns what *answer*ed the component:

```
MAPersonManager>>add
    | person |
    person := self call: (MAPersonModel new asComponent
        addValidatedForm; yourself).
    person isNil
        ifFalse: [ self class persons add: person ]
```

Remember we are using the new rendering technique ? Let's see how one would add a link to a page with this technique:

```
MAPersonManager>>renderContentOn: html
    html heading: 'Person Manager'.
    html anchor
        callback: [ self add ];
        with: [ html text: 'add' ]
```

We will see how to make this call smaller in a latter exercise.

Exercise 8 Create an accessor `#persons` for the class-variable `Persons` on the instance-side. Render a list of all persons within `MAPersonManager`.

Solution 8

```
MAPersonManager>>persons
    ^ self class persons

MAPersonManager>>renderLineForPerson: aPerson on: html
    html text: aPerson firstName; space; text: aPerson lastName

MAPersonManager>>renderContentOn: html
    html heading: 'Person Manager'.
    html anchor
        callback: [ self add ];
```

```
        with: [ html text: 'add' ].
    html break.
    self persons
        do: [ :person | self renderLineForPerson: person on: html ]
        separatedBy: [ html break ]
```

Exercise 9 Render the list entries as links, so that existing persons can be edited.

Solution 9

```
MAPersonManager>>editPerson: aPerson
    self call: (aPerson asComponent
        addValidatedForm;
        yourself)

MAPersonManager>>renderLineForPerson: aPerson on: html
    html anchor
        callback: [ self editPerson: aPerson ];
        with: [ html text: aPerson firstName; space; text: aPerson lastName ]
```

Exercise 10 Render an additional link called *view*, as seen in Figure 1, to display the entry in a read-only view. You can turn a component into read-only mode by sending the message `#readonly:`.

Solution 10

```
MAPersonManager>>viewPerson: aPerson
    self call: (aPerson asComponent
        addForm: #( cancel );
        readonly: true;
        yourself)

renderLineForPerson: aPerson on: html
    html anchor
        callback: [ self editPerson: aPerson ];
        with: [ html text: aPerson firstName; space; text: aPerson lastName ].
    html space.
    html anchor
        callback: [ self viewPerson: aPerson ];
        with: [ html text: 'view' ]
```

4 Using Magritte Reports

Rendering a list like this is nice, but it doesn't scale well if you have many entries. As well it doesn't take advantage of the nice descriptive model we have. Luckily Magritte includes built in reporting facilities that we will be extending our application with. In this section we make our little application look like Figure 2.

Exercise 11 Add a new child component to `MAPersonManager` and initialize it, lazily or within the method `#initialize`, with `MAReport rows: self persons description: MAPersonModel description`. Render the report instead of the list.

Solution 11 Create a new `report` instance-variable.

```
MAPersonManager>>report
    ^ report
```


Figure 2: Person Manager Report

```
MAPersonManager>>report: aReport
    report := aReport

MAPersonManager>>children
    ^ OrderedCollection with: self report

MAPersonManager>>initialize
    super initialize.
    self report: (MAReport rows: self persons description: MAPersonModel description)

MAPersonManager>>renderContentOn: html
    html heading: 'Person Manager'.
    html anchor on: #add of: self.
    html render: self report
```

Exercise 12 Play with the report in your browser: sort the rows, add new persons... You will probably notice that the report doesn't update itself when adding new persons. It's because the report creates a copy. Call `#refresh` to do so.

Solution 12

```
MAPersonManager>>add
    | person |
    person := self call: (MAPersonModel new asComponent
        addValidatedForm; yourself).
    person isNil ifFalse: [
        self class persons add: person.
        self report refresh ]
```

Exercise 13★ Right now almost all the information about a person is displayed within the report, if you enter much data this gives a very wide Web page. Filter out all the descriptions except the ones for `#firstName`, `#lastName`, `#email`, and `#nationality`.

Solution 13

```
MAPersonManager>>initialize
    super initialize.
    self report: (MAReport
        rows: self persons
        description: (MAPersonModel description select: [ :each |
            #( firstName lastName email nationality )
                includes: each accessor selector ]))
```

Exercise 14 Add a new column to the report to hold commands to *view*, *edit* and *remove* entries. To get a hint on how to archive the desired behaviour, brows the references of `MACommandColumn`.

Solution 14

```
MAPersonManager>>initialize
    super initialize.
    self report: (MAReport
        rows: self persons
        description: (MAPersonModel description select: [ :each |
            #( firstName lastName email nationality )
                includes: each accessor selector ])).
    self report addColumn: (MACommandColumn new
        addCommandOn: self selector: #view:;
        addCommandOn: self selector: #edit:;
        addCommandOn: self selector: #remove:;
        yourself)
```

Exercise 15★ Implement a search functionality within your report. The user should be asked for a filter string that will be compared to all the string-fields within the persons. To get some pointers have a look at the implementors and senders of `#rowFilter:`, `#toString:` and `#readUsing:`, and `#matches:`. Add a *clear* link as well, to remove any filter.

Solution 15

```
MAPersonManager>>filter
    | filter |
    filter := self request: 'Enter a filter string'.
    self report rowFilter: [ :person |
        (self report columns collect: [ :col |
            col description in: [ :desc | desc toString: (person readUsing: desc) ] ])
            anySatisfy: [:value | value matches: filter ] ]

MAPersonManager>>clear
    self report rowFilter: nil

MAPersonManager>>renderContentOn: html
    html heading: 'Person Manager'.
    #( add filter clear )
        do: [ :symbol | html anchor on: symbol of: self ].
        separatedBy: [ html space ].
    html render: self report
```

5 Described Descriptions★

As experience shows, customers are often not completely satisfied with the features the application developers are giving to them. They want more and they want to be able to customize everything by themselves. Thanks to the reflective nature of Magritte – all the descriptions within Magritte are described with Magritte descriptions itself – we can easily provide the necessary functionality.

Question 16 Point your browser to *http://localhost:8080/seaside/example-browser* and select `MADescriptionEditor` in the first drop-down box. What is this? What could it be useful for?

Question 17 Browse the source code of `MADescriptionEditor`. As you can see for the preview and instance of `MAScaffolder` is created. This is a class that has only two instance-variables: one references a description-container, the other a dictionary mapping description-elements to actual values. Have a look at the methods `#readUsing:` and `#write:using:`, and their default implementations in `Object`. Why are those methods overridden in `MAScaffolder`?

Exercise 18 Add a class-instance-variable to `MAPersonModel` called `CustomDescription`. Initialize it with an empty instance of `MAContainer`. Create an accessor method. Override `#description` on the instance-side, call super and compose it with the `CustomDescription`. So far, you should see no difference in the behaviour of the application when testing it.

Solution 18

```
MAPersonModel class>>customDescription
    ^ CustomDescription

MAPersonModel class>>customDescription: aContainer
    CustomDescription := aContainer

MAPersonModel class>>initialize
    super initialize.
    self customDescription: MAContainer new

MAPersonModel>>customDescription
    ^ self class customDescription

MAPersonModel>>description
    ^ super description copy
        addAll: self customDescription;
        yourself
```

Execute `MAPersonModel initialize` and `MACachedBuilder default flush` to forget about the cached values.

Exercise 19 Add another link above your report to let the user edit the custom description. Unfortunately the default implementation of `MADescriptionEditor` has no way to go back to the caller. The simplest solution is to subclass `MADescriptionEditor` and render an additional button by overriding `#renderButtonsOn:` that answers the modified description.

Solution 19

```
MAPersonManager>>modifyPersonModel
    description := self call: (MADescriptionPersonEditor new
        description: MAPersonModel customDescription).
    description isNil
        ifFalse: [ MAPersonModel customDescription: description ]

MAPersonManager>>renderContentOn: html
    html heading: 'Person Manager'.
    #( modifyPersonModel add filter clear )
        do: [ :symbol | html anchor on: symbol of: self ]
        separatedBy: [ html space ].
    html render: self report

MADescriptionEditor subclass: #MADescriptionEditorWithSave
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Magritte-Tutorial-View'

MADescriptionEditorWithSave>>renderButtonsOn: html
    super renderButtonsOn: html.
    html submitButtonOn: #save of: self
```

The method `#save` is already defined in `MAExampleEditor`.

Exercise 20 Now you are able to add additional fields to your `MAPersonModel` class, unfortunately the editor doesn't work anymore. If you try to add or edit a person you get the error-message: *This message is not appropriate for this*

object.. Obviously the custom-descriptions that have been added through the web do not have an accessor and you probably don't want to let your customers care about those low-level things anyway. To solve the problem you need to override the methods `#readUsing:` and `#write:using:` similar to the way it is done in `MAScaffolder`.

Solution 20 Add an instance variable `values` into `MAPersonModel`.

```
MAPersonModel>>values
    ^ values ifNil: [ values := Dictionary new ]

MAPersonModel>>readUsing: aDescription
    ^ (MAPersonModel customDescription includes: aDescription)
        ifTrue: [ self values at: aDescription ifAbsent: [ nil ] ]
        ifFalse: [ super readUsing: aDescription ]

MAPersonModel>>write: anObject using: aDescription
    (MAPersonModel customDescription includes: aDescription)
        ifTrue: [ self values at: aDescription put: anObject ]
        ifFalse: [ super write: anObject using: aDescription ]
```

Question 21 Test your application toughly. How does it behave if you add, remove or change fields within living instances of `MAPersonModel`?

Figure 3: Person Editor Customized