Meta data at Work with Magritte: A Tutorial

Stéphane Ducasse

September 21, 2017
master @ 06a94db

Copyright 2017 by Stéphane Ducasse.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:
http://creativecommons.org/licenses/by-sa/3.0/

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.

Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):
http://creativecommons.org/licenses/by-sa/3.0/legalcode

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Magritte in a nutshell

Many applications consist of a large number of input dialogs and reports that need to be built, displayed and validated manually. Often these dialogs remain static after the development phase and cannot be changed unless a new development effort occurs. For certain kinds of application domains such as small businesses, changing business plans, modifying workflows, etc., it usually boils down to minor modifications to domain objects and behavior, for example new fields have to be added, configured differently, rearranged or removed. Performing such tasks is tedious.

Magritte is a meta-data description framework. With Magritte you describe your domain objects and among other things you can get Seaside components and their associated validation for free. Moreover Magritte is self-described, enabling the automatic generation of meta-editors which can be adapted for building end-user customizations of application.

In this chapter we describe Magritte, its design and how to customize it. Now be warned, Magritte is a framework supporting meta-data description. As any framework, mastering it takes time. It is not a scaffolding engine, therefore Magritte may imply some extra complexity and you have to understand when you want to pay for such complexity.

1.1 Basic principles

In this section we present the key principles. With such a knowledge you can get 80% of the power of Magritte without knowing all its possible customizations. The key idea behind Magritte is the following: given one object with a set of values, and a description of this information, we will create automatically tools that treat such information and for example automatically create Seaside components.

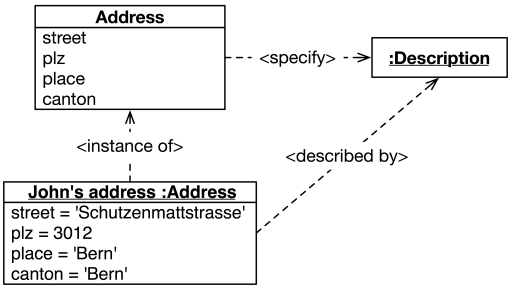

Figure 1-1 An object is described by a description which is defined on its class.

Figure 1-1 shows that a person address', John's address, instance of the class Address, is described by a description object which is attached to the class `Address`. A program (i.e., database query, generic UI, seaside component builder) will interpret the value of the instance by using its descriptions.

Here are the basic description assumptions:

- An object is described by adding methods named `description` (naming convention) to its class. Such description methods create different description entities. The following `MAAddress` class method creates a string description object that has a label `'Street'`, a priority and two accessors `street` and `street:` to access it.

```
MAAddress >> descriptionStreet
  <magritteDescription>
  ^ MAStringDescription new
    accessor: #street;
    label: 'Street';
    priority: 10;
    yourself
```

Note that there is no need to have a one to one mapping between the instance variables of the class and the associated descriptions.

- All descriptions are automatically collected and put into a container description when sending `magritteDescription` to the object (see Figure 1-2).
- Descriptions are defined programmatically and can also be queried. They support the collection protocol (`do:`, `select:`...).

Now once we have described an instance, Magritte can build for us different representations. An interesting one in the context of web development is the automatic creation of Seaside components.

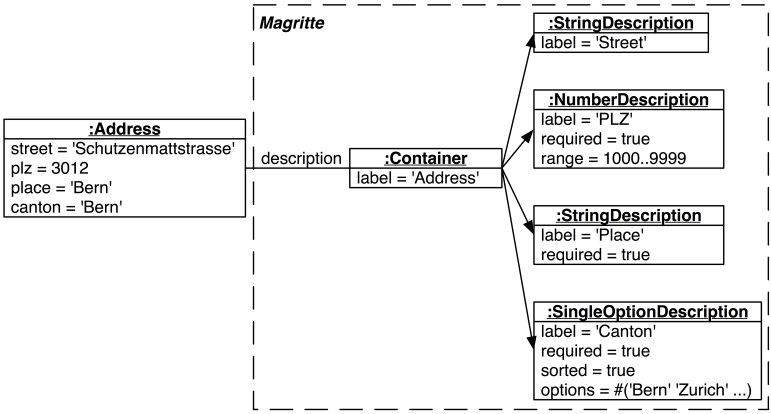

Figure 1-2 Describing an Address.

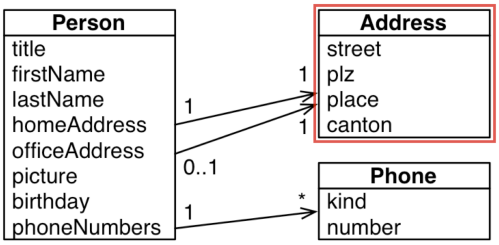

Figure 1-3 anAddress asComponent produces a Seaside component.

1.2 Obtaining a component

Once an object is described, you can obtain a Seaside component by sending to the object the message `Object >> asComponent`. For example to get a component for an address: `anAddress asComponent` as shown in Figure 1-3.

It is often useful to decorate the component with buttons like cancel and Ok. This can be done by sending the message `WAComponent >> addValidatedForm` to the component. Note that you can also change the label of the validating form using `addValidatedForm:` and passing an array of associations whose first element is the message to be sent and the second the label to be displayed.

```
anAddress asComponent addValidatedForm.
```

```
anAddress asComponent addValidatedForm: { #save -> 'Save'. #cancel
    -> 'Cancel' }.
```

A Magritte form is generally wrapped with a form decoration via `WAComponent >> addValidatedForm.`. Magritte forms don't directly work on your domain objects. They work on a memento of the values pulled from your object using the descriptions. When you call `save`, the values are validated using the descriptions, and only after passing all validation rules are the values committed to your domain object by the mementos via the accessors.

A description container is an object implementing collection behavior (`collect:`, `select:`, `do:`, `allSatisfy:`, ...). Therefore you can send the normal collection messages to extract the parts of the descriptions you want. You can also iterate over the description or concatenate new ones. Have a look at the `MAContainer` protocol.

You can also use the message `MAContainer >> asComponentOn:` with aModel to build or select part of a description and get a component on a given model. The following code schematically shows the idea: two descriptions are assembled and a component based on these two descriptions is built.

```
((anAddress descriptionStreet , anAddress descriptionPlace)
  asComponentOn: anAddress) addValidatedForm
```

1.3 Summary

While Magritte is really powerful, using a meta-data system will add another indirection layer to your application, so this is important to understand how to use such power and when to just use normal development techniques. A good and small example to learn from is the Conrad conference management system developed to manage the ESUG conference available at http://www.squeaksource.com/Conrad.html.

Here Magritte is used to describe the different forms and all the data. The Pier content management system is a more complex example of using Magritte, it can be downloaded at http://www.piercms.com.

CHAPTER 2

First concrete example

Let us go over a simple but complete example. We want to develop an application to manage person, address and phone number as shown in Figure 2-1.

2.1 Getting started

Get a Seaside image from http://www.seaside.st. Normally Seaside is shipped with Magritte.

```
MCSmalltalkhubRepository
  owner: 'Magritte'
  project: 'Magritte3'
  user: ''
  password: ''
```

Make sure your Seaside server is running within the image by browsing the

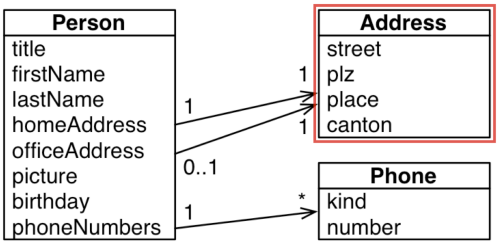

Figure 2-1 Our address design.

counter application at http://localhost:8080/examples/counter

If it is not, use the Seaside control panel from Tools menu of the world menu to check the status of the web server.

2.2 Defining the address class

We define a class `MAAddress` with four instance variables and their corresponding accessors.

```
Object subclass: #Address
  instanceVariableNames: 'street place plz canton'
  classVariableNames: ''
  category: 'MAAddress'
```

```
MAAddress class >> example1

  | address |
  address := self new.
  address plz: 1001.
  address street: 'Sesame'.
  address place: 'DreamTown'.
  address canton: 'Bern'.
  ^ address
```

Then we add the descriptions to the `MAAddress` class as follows: the street name and the place are described by a string description, the PLZ is a number with a range between 1000 and 9999, and since the canton is one of the predefined canton list (our address is for Switzerland so far), we describe it as a single option description.

```
MAAddress >> descriptionStreet
    <magritteDescription>

    ^ MAStringDescription new
      accessor: #street;
      label: 'Street';
      priority: 10;
      yourself
```

```
MAAddress >> descriptionPlz
    <magritteDescription>

    ^ MANumberDescription new
      accessor: #plz;
      label: 'PLZ';
      priority: 20;
      min: 1000;
      max: 9999;
      yourself
```

```
MAAddress >> descriptionPlace
    <magritteDescription>

    ^ MAStringDescription new
        accessor: #place;
        label: 'Place';
        priority: 30;
        yourself
```

```
MAAddress >> descriptionCanton
    <magritteDescription>

    ^ MASingleOptionDescription new
      options: #( 'Zurich' 'Bern' 'Luzern' 'Uri' 'Schwyz'
    'Unterwalden' 'Glarus' 'Zug' 'Freiburg' 'Solothurn' 'Basel'
    'Schaffhausen' 'Appenzell' 'St. Gallen' 'Graubunden' 'Aargau'
    'Thurgau' 'Ticino' 'Vaud' 'Valais' 'Neuchatel' 'Geneve' 'Jura' );
      reference: MAStringDescription new;
      accessor: #canton;
      label: 'Canton';
      priority: 40;
      beSorted;
      yourself
```

2.3 First magritte program

Now we can start manipulating the descriptions. Inspect the description object of the address object:

```
| address |
address := Address example1.
address magritteDescription inspect.
```

We can iterate over the descriptions and get the values associated with the descriptions of our address model:

```
| address |
address := Address example1.
address magritteDescription do: [ :description |
  Transcript
    show: description label; show: ':'; tab;
    show: (description toString: (address readUsing: description));
    cr ]
```

Executing the second code snippet outputs the following in the Transcript:

```
Street:    Sesame
PLZ:       1001
Place:     DreamTown
Canton:    Bern
```

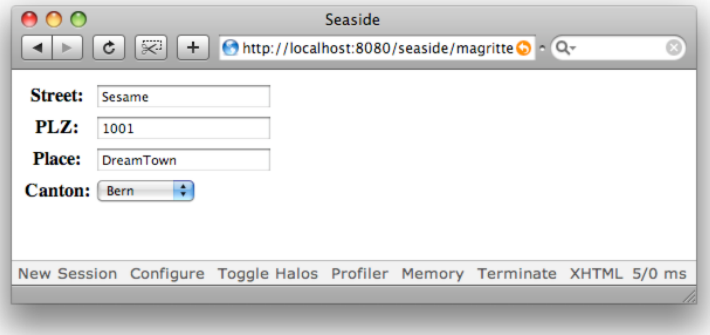

Figure 2-2 Address example1 asComponent.

2.4 Creating a Seaside editor.

Now we are ready to create a Seaside component automatically in a similar manner.

```
WAComponent subclass: #MyContactAddress
  instanceVariableNames: 'addressForm'
  classVariableNames: ''
  category: 'MAAddress'
```

```
MyContactAddress >> initialize
  super initialize.
  addressForm := MAAddress example1 asComponent
```

```
MyContactAddress >> children
  ^ Array with: addressForm
```

The method `asComponent` sent to the address object automatically builds a Seaside component for us. The resulting editor is displayed in Figure 2-2.

2.5 Decoration

To enable validation and add buttons to save and cancel the editing process is a matter of adding a decoration. The message `addValidatedForm` decorates the component with the necessary rendering and behavior.

```
MyContactAddress >> initialize
  super initialize.
  addressForm := MAAddress example1 asComponent.
  addressForm addValidatedForm
```

As a result we get a complete address editor, as seen in Figure 2-3.

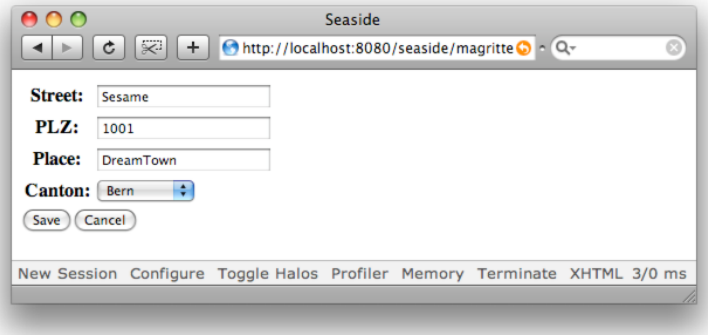

Figure 2-3 Same Magritte generated component with buttons and validation.

2.6 Conclusion

In summary Magritte is easy to use with Seaside. You put your descriptions according to a naming-convention. You can then ask your model objects for their descriptions by sending the message `description` or alternatively you directly ask Magritte to build a Seaside editor for you by sending the message `asComponent`.

CHAPTER 3

Magritte main entities

In this chapter we present some of the key elements in the Magritte architecture.

3.1 Descriptions

Descriptions, as we have seen in the above examples, are naturally organized in a description hierarchy. A class diagram of the most important descriptions is shown in Figure **??**. Different kinds of descriptions exist: simple type-descriptions that directly map to Pharo classes, and some more advanced descriptions that are used to represent a collection of descriptions, or to model relationships between different entities.

Descriptions are central to Magritte and are connected to accessors and conditions that we present below.

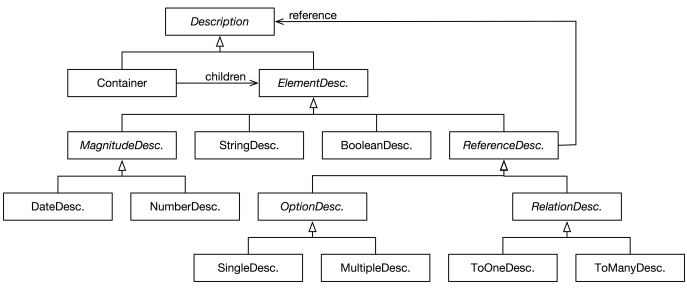

Figure 3-1 Description hierarchy.

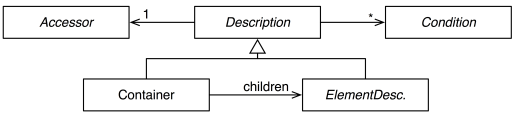

Figure 3-2 Descriptions are a composite and connected via accessors.

1. **Type Descriptions.** Most descriptions belong to this group, such as the `ColorDescription`, the `DateDescription`, the `NumberDescription`, the `StringDescription`, the `BooleanDescription`, etc. All of them describe a specific Smalltalk class; in the examples given, this would be Color, Date, Number and all its subclasses, String, and Boolean and its two subclasses True and False. All descriptions know how to perform basic tasks on those types, such as to display, to parse, to serialize, to query, to edit, and to validate them.

1. **Container Descriptions.** If a model object is described, it is often necessary to keep a set of other descriptions within a collection, for example the description of a person consists of a description of the title, the family name, the birthday, etc. The `ContainerDescription` class and its subclasses provide a collection container for other descriptions. In fact the container implements the whole collection interface, so that users can easily iterate (`do:`), filter (`select:`, `reject:`), transform (`collect:`) and query (`detect:`, `anySatisfy:`, `allSatisfy:`) the containing descriptions.

1. **Option Descriptions.** The `SingleOptionDescription` describes an entity, for which it is possible to choose up to one item from a list of objects. The `MultipleOptionDescription` describes a collection, for which it is possible to choose any number of items from a predefined list of objects. The selected items are described by the referencing description.

1. **Relationship Descriptions.** Probably the most advanced descriptions are the ones that describe relationships between objects. The `ToOneRelationshipDescription` models a one-to-one relationship; the `ToManyRelationshipDescription` models a one-to-many relationship using a collection. In fact, these two descriptions can also be seen as basic type descriptions, since the `ToOneRelationshipDescription` describes a generic object reference and the `ToManyRelationshipDescription` describes a collection of object references.

3.2 Exceptions

Since actions on the meta-model can fail. In addition, objects might not match a given meta-model. Finally it is often important to present to the end

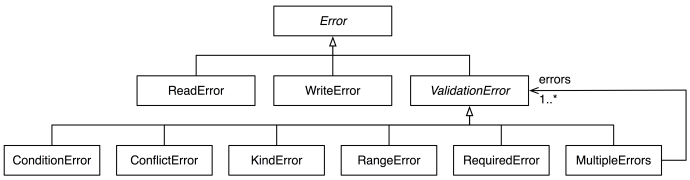

Figure 3-3 Exceptions.

users readable error messages. Magritte introduces an exception hierarchy which knows about the description, the failure and a human-readable error message as shown in Figure 3-3.

3.3 Adding Validation

Descriptions can have predicates associated with them to validate the input data. Magritte calls such validation conditions associated with a description. A condition is simply a block that returns true or false. A simple one is the single field condition which is passed the pending value directly and can be attached directly to your descriptions. It is as simple as the following block which checks the length of the value once the blanks are removed.

```
aDescription addCondition: [ :value | value withBlanksTrimmed size >
    3 ]
```

There are advantages to having your rules outside your domain objects, especially if you're taking advantage of Magritte as an Adaptive Object Model where users can build their own rules. It also allows the mementos to easily test data for validity outside the domain object and gives you a nice place to hook into the validation system in the components. Still you have to pay attention since it may weaken your domain model.

Multiple Field Validation. When we did the Mini-reservation example in previous chapters we wanted the starting date to be later than today. With Magritte we can express this point as follows:

```
MAAddress >> descriptionStartDate
  <magritteDescription>

  ^ MADateDescription new
    accessor: #startDate;
    label: 'Start Date';
    addCondition: [ :value | value > Date today ];
    beRequired;
    yourself
```

```
MAAddress >> descriptionEndDate
  <magritteDescription>

    ^ MADateDescription new
       accessor: #endDate;
        label: 'End Date';
        addCondition: [ :value | value > Date today ];
        beRequired;
        yourself
```

Now to add a validation criterion that depends on several field values, you have to do a bit more than that because you cannot from one description access the other. Suppose that we want to enforce that the endDate should be after the startDate. We cannot add that rule to the endDate description because we cannot reference the startDate value. We need to add the rule in a place where we can ensure all single field data exists but before it's written to the object from the mementos.

We need to add a rule to the objects container description like this

```
MAAddress >> descriptionEndDate
  <magritteDescription>
    ^ super descriptionContainer
        addCondition: [ :object |
            (object readUsing: self descriptionEndDate) >
                (object readUsing: self descriptionStartDate)]
        labelled: 'End date must be after start date';
        yourself
```

This simply intercepts the container after it is built, and adds a multi field validation by accessing the potential value of those fields. You get the memento for the object, and all the field values are in its cache, keyed by their descriptions. You simply read the values and validate them before they have a chance to be written to the real business object. Multi field validations are a bit more complicated than single field validations but this pattern works well.

3.4 Accessors and Mementos

Accessors. In Pharo, data can be accessed and stored in different ways. Most common data is stored within instance variables and read and written using accessor methods, but sometimes developers choose other strategies, for example to group data within a referenced object, to keep their data stored within a dictionary, or to calculate it dynamically from block closures. Magritte uses a strategy pattern to be able to access the data through a common interface, see Figure 3-4.

By far the most commonly used accessor type is the `SelectorAccessor`. It can be instantiated with two selectors: a zero argument selector to read, and

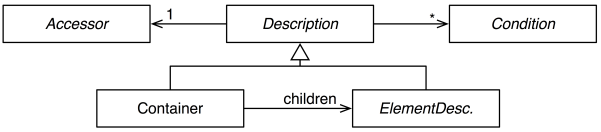

Figure 3-4 Accessors in Magritte.

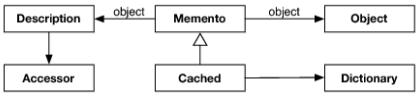

Figure 3-5 Mementos (simplified version).

a one argument selector to write. For convenience it is possible to specify a read selector only, from which the write selector is inferred automatically.

The `MADictionaryAccessor` is used to add and retrieve data from a dictionary with a given key. This access strategy is mainly used for prototyping as it allows one to treat dictionaries like objects with object-based instance variables.

When a memento writes valid data to its model, it does so through accessors which are also defined by the descriptions. `MAAccessor` subclasses allow you to define how a value gets written to your class.

Mementos. Magritte introduces mementos that behave like the original model and that delay modifications until they are proven to be valid. When components read and write to domain objects, they do it using mementos rather than working directly on the objects. The default memento is `MACheckedMemento`. The mementos give Magritte a place to store invalid form data prior to allowing the data to be committed to the domain object. It also allows Magritte to detect concurrency conflicts. By never committing invalid data to domain objects, there's never a need to roll anything back. So mementos are good since editing might turn a model (temporarily) invalid, canceling an edit shouldn't change the model and concurrent edits of the same model should be detected and (manually) merged. `mementoClass` is a class factory that you can specialize to specify your own memento. But normally you should not need that.

3.5 Custom Views

Components control how your objects display. Some descriptions have an obvious one to one relationship with a UI component while others could easily be shown by several different components. Figure 3-6 shows some components. For example, an `MAMemoDescription` would be represented by a

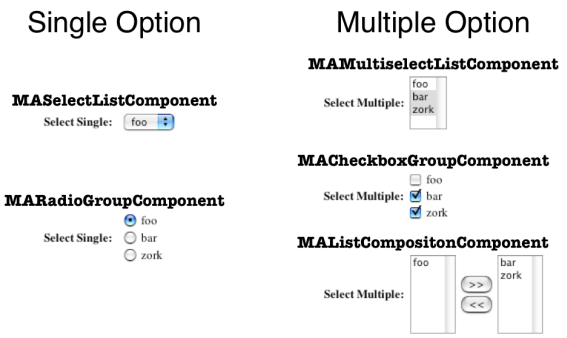

Figure 3-6 Some Magritte specific widgets.

text area, but a `MABooleanDescription` could be a checkbox, or a drop down with true and false, or a radio group with true and false.

Each description defaults to a component, but allows you to override and specify any component you choose, including any custom one you may write using the message `Description >> componentClass:`.

```
aDescription componentClass: aClass
```

You can also define your own view by following the steps:

- Create a subclass of `MADescriptionComponent`.
- Override `renderEditorOn:` and/or `renderViewerOn:` as necessary.
- Use your custom view together with your description by using the accessor `componentClass:`.
- Possibly add your custom view to its description into `defaultComponentClasses`.

3.6 Custom Descriptions

In some cases it might happen that there is no description provided to use with a class. If your domain manipulates money (amount and currency) or URLs (scheme, domain, port, path, parameters) you may want to define your own descriptions (or load an extension package that already provides these descriptions) to take advantage of Magritte.

Extending Magritte is simple, create your own description but remember that Magritte is described within itself so you have to provide certain information.

- Create a subclass of `MAElementDescription`.

- On the class-side override: `isAbstract` to return false, `label` to return the name of the description. On the instance-side override: `kind` to return the base-class, `acceptMagritte:` to enable visiting and `validateSpecific:` to validate.
- Create a view, if you want to use it for UI building.

We suggest you have a look at existing descriptions. In addition, carefully choosing the right superclass can already provide you part of what you are looking for. Parsing, printing and (de)serialization is implemented by the following visitors: `MAStringReader`, `MAStringWriter`, `MABinaryReader` and `MABinaryWriter`.

CHAPTER 4

Magritte Katas

This chapter presents some exercises about Magritte. They are based on the Magritte Tutorial written by Lukas Renggli (the original designer and implementor of Magritte). We thank him for Magritte and this set of exercises. The exercises has been ported to Magritte 30 and Seaside 3.1. Magritte30 supports instance-based declarations while Magritte20 was only supporting class-based declarations. The exercises have been enhanced and are maintained by Stéphane Ducasse and Damien Cassou.

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 Pharo (senders, implementers, references, ...) to browse the source-code of Magritte, you might even discover some other features that were not presented during this tutorial.

4.1 Getting Started

Get an image and bring the Configuration Browser and install the stable version of MagritteContactManager.

In case you need to know where the code of the tutorial is saved. It is in the following repository:

```
MCSmalltalkhubRepository
  owner: 'Magritte'
  project: 'Magritte3'
  user: ''
  password: ''
```

Make sure your Seaside server is running within the image by browsing the counter application at http://-local-host:8080/examples/-counter

If it is not, the seaside control panel from Tools menu of the world menu.

4.2 Juggle with Descriptions

Have a look at the source-code of the classes `CMPerson`, `CMAddress`, and `CMPhone`. In essence a person is described by different attributes such as it first name, name email, birthday... In addition it in relation with an address object and it has a list of phones.

Here are the class definitions. Note that the class names are a bit overkill with the Model prefix. In future versions, we may drop it.

```
Object subclass: #CMPerson
    instanceVariableNames: 'title firstName lastName homeAddress
    officeAddress picture birthday phones password'
    classVariableNames: ''
    package: 'Magritte-ContactManager'
```

```
Object subclass: #CMPhone
    instanceVariableNames: 'kind number'
    classVariableNames: ''
    package: 'Magritte-ContactManager'
```

```
Object subclass: #CMAddress
    instanceVariableNames: 'street plz place canton nationality'
    classVariableNames: ''
    package: 'Magritte-ContactManager'
```

All the following exercises will be built upon this simple model.

Browse to http://localhost:8080/personeditor and check if you can see all the features presented during the lecture.

You should get an application as shown in Figure 4-1.

4.3 Exercise 1: Adding a simple field

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.

Add the following instance variable and accessor.

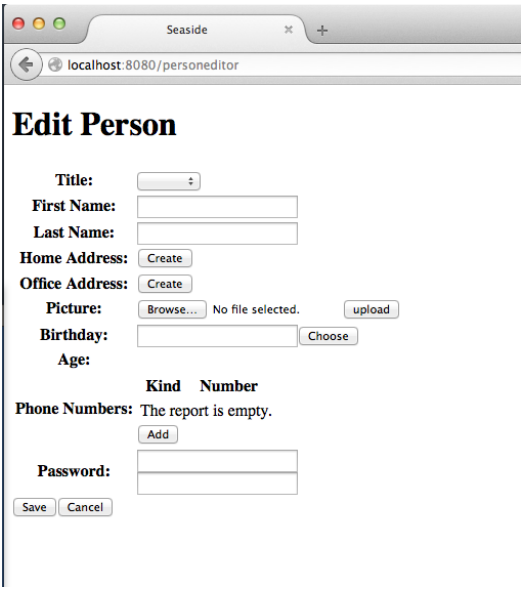

Figure 4-1 A person editor.

```
CMPerson >> comment
    ^ comment
```

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

```
CMPerson >> descriptionComment
    <magritteDescription>
    ^ MAStringDescription new
            accessor: #comment;
            label: 'Comment';
            priority: 100 ;
            yourself
```

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 as follows:

```
CMPerson >> descriptionComment
    <magritteDescription>
     ^ MAMemoDescription new
                accessor: #comment;
                label: 'Comment';
                priority: 100 ;
```

```
                yourself
```

A text-area is then displayed.

4.4 Exercise 2: Adding Nationality as a single selection

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.

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 sub-class of `MAOptionDescription` which accepts the `options:` message to specify the different choices. The default choice (displayed by Magritte if no other choices have been selected) is chosen with the `default:` message:

```
CMPerson >> descriptionNationality
    <magritteDescription>
     ^ (MASingleOptionDescription new
            accessor: #nationality;
            label: 'Nationality';
            priority: 55;
            yourself)
            options: #( 'Switzerland' 'France' 'Germany' );
            default: 'Switzerland';
            beSorted;
            beRequired;
            yourself
```

You can use the `beSorted` message to sort values in the list. You can add also the `beRequired` to make sure that the user cannot choose an empty item in the list.

4.5 Exercise 3: Adding email

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.

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

```
CMPerson >> descriptionEmail
    <magritteDescription>
     ^ MAStringDescription new
            accessor: #email;
            label: 'Email';
            priority: 95;
            yourself
```

If you open a browser on `MADescription` class (protocol validation), you will see that we can send the message `addCondition:labelled:` to a description

```
CMPerson >> descriptionEmail
    <magritteDescription>
    ^ (MAStringDescription new
        accessor: #email;
        label: 'Email';
        priority: 95;
        yourself)
        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
```

4.6 Exercice 4: Reusing Descriptions

You can also reuse a description. For example, we will reuse nationality description from `CMPerson` in the address `CMAddress` 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.

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

```
CMAddress >> descriptionCountry
    <magritteDescription>
    ^ CMPerson new descriptionNationality copy
        label: 'Country';
        yourself
```

Note that here we create a new person because there is no reference from `CMAddress` to `CMPerson`.

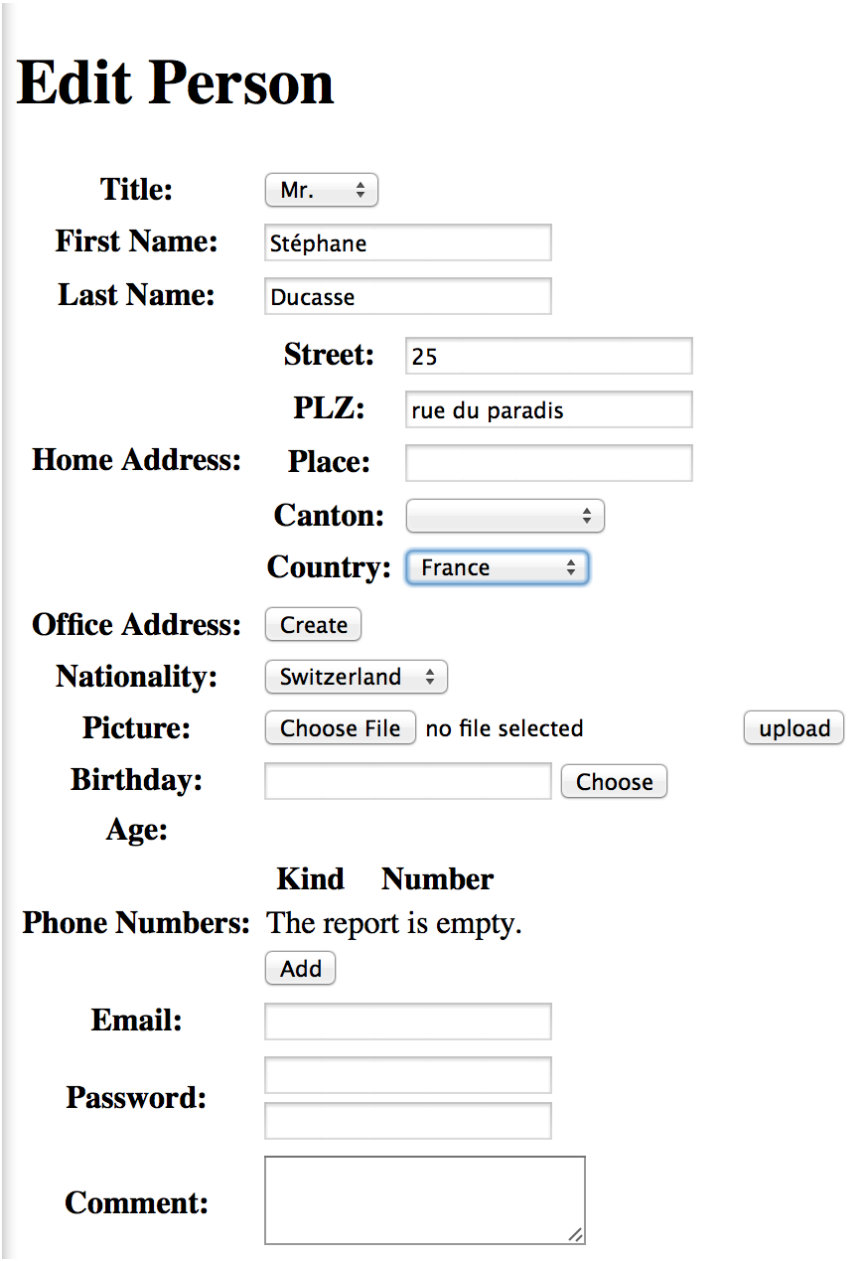

Figure 4-2 The Nationality description is reused by the `CMAddress`.

4.7 Exercise 5: Food for Thought

Before starting to use Seaside in combination with Magritte it is worth to step back and ask ourself some questions:

- 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 / disadvantages?

CHAPTER 5

Seaside integration katas

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 therefore 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.

5.1 Exercise 6: Introducing the Person Manager Application

Start by creating a new subclass 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 named `personmanager`. Create a method `renderContentOn:` that displays the heading 'Person Manager'.

Test the setup of your new application in the Web browser.

Solution.

```
WAComponent subclass: #MAPersonManager
    instanceVariableNames: ''
    classVariableNames: 'Persons'
    package: 'Magritte-ContactManager'
```

```
MAPersonManager class >> initialize
    "self initialize"
```

```
    super initialize.
    self persons: OrderedCollection new.
    WAAdmin register: self asApplicationAt: 'personmanager'
```

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

5.2 Exercise 7: Adding a contact

In the `MAPersonManager` class, define the method `add`, that creates a new instance of `CMPerson`, calls the default Magritte editor and adds it to our collection. Note that Magritte will answer `nil`, when the user hits the cancel button in the editor. Create a link in your page that will execute the `add` method.

Solution.

A Seaside component is created from a model using the message `asComponent`. Around a component, you can add buttons to save and cancel using the message `addValidatedForm`. If you want to use another component instead of the current one, use the message `call:`. This method returns what answered the component:

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

Let's see how one add a link to a page:

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

5.3 Exercise 8: Rendering the contacts

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

Solution 8.

We define a simple accessor either using the class side message or directly accessing the variable.

```
MAPersonManager >> persons
    ^ self class persons
```

or

```
MAPersonManager >> persons
    ^ Persons
```

We modify the rendering method to display the persons.

```
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 ]
```

For this we introduce a dedicated method:

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

5.4 Exercise 9: Enhancing the list

Improve the list entries rendering so that existing persons can be edited.

Solution 9.

First we define a new method named `editPerson:` which presents its argument as a form component.

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

Second we modify the `renderLineForPerson:on:` method to add a link that invokes the previous method.

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

Figure 5-1 Persons with edit and view.

```
                space;
                text: aPerson lastName ]
```

5.5 Exercise 10: Readonly

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:` with a boolean stating the read-only status.

Solution.

First we define a method that will show the values of a component.

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

```
MAPersonManager >> 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' ].
```

CHAPTER 6

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. For example professional applications such as the one of Quuve are fully developed with Magritte and based on Magritte report. In this section we make our little application look like Figure 2.

6.1 Exercise 11: A first report

Add an instance variable that will refer to the report. Add accessors. Add a new child component to `MAPersonManager` and initialize it, lazily or within the method `initialize`, with the following expression `MAReport rows: self persons description: self persons magritteDescription`. Pay attention that the `children` message should return all the subcomponent of the receiver. Once done, render the report instead of the list.

Solution 11.

```
WAComponent subclass: #MAPersonManager
    instanceVariableNames: 'report'
    classVariableNames: 'Persons'
        category: 'Magritte-ContactManager'
```

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

We initialize the report following the

```
MAPersonManager >> initialize
    super initialize.
    self report: (MAReport rows: self persons description: self
    persons first magritteDescription)
```

Now we define the method `children` to make sure that the report component is correctly returned.

```
MAPersonManager >> children
    ^ OrderedCollection with: self report
```

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

6.2 Exercise 12: Handling adding refresh

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 the message `refresh` on the report to fix this problem.

Solution 12

We change the definition of the `add` method.

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

6.3 Exercise 13: Filtering information

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` and `email`.

Solution 13.

```
MAPersonManager >> filteredDescriptionsFrom: aPerson

    ^ aPerson magritteDescription select: [ :each |
```

```
        #(firstName lastName email) includes: each accessor
    selector ]
```

```
MAPersonManager >> initialize
    super initialize.
    self report: (MAReport rows: self persons description: (self
    filteredDescriptionsFrom: self persons first))
```

6.4 Exercise 14: Adding extra column

Add a new column to the report to hold commands to view and edit entries. To get a hint on how to achieve the desired behaviour, browse the references of `MACommandColumn`.

Solution 14.

We define an instance of `MACommandColumn` to group two commands. Note that we use two variants to show that we can control the text displayed from the action performed.

```
MAPersonManager >> initialize
  super initialize.
  self report: (MAReport rows: self persons description: (self
    filteredDescriptionsFrom: self persons first)).
  self report addColumn: (MACommandColumn new
            addCommandOn: self selector: #viewPerson:;
            addCommandOn: self selector: #editPerson: text: 'Edit';
    yourself)
```

6.5 Exercise 15: Adding a search facilities

We will now 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.

The method `rowFilter:` expects a block returning a boolean telling if the row should be shown.

```
MAPersonManager >> filter
  | filter |
  filter := self request: 'Enter a filter string'.
  self report rowFilter: [ :person |
```

```
    (self report columns collect: [ :col |
      col magritteDescription in: [ :desc | desc toString: (person
    readUsing: desc) ] ]) anySatisfy: [:value | value matches:
    filter ] ]
```

The solution does not compare the search to only string fields but to all the values of the receiver converted to strings. The following expression

```
(self report columns collect: [ :col |
  col description in: [ :desc | desc toString: (person readUsing:
    desc) ] ])
```

returns a collection of strings of the value held by the receiver (here a person).

Clearing the filter is simply to set its argument to `nil`.

```
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
```

SD: check because when pressing on editView I got a ”requiresMultipart-Form” is nil

CHAPTER 7

Giving access to end-users

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 to offer flexibility to end-users.

7.1 Question 16

Point your browser to http://localhost:8080/seaside/magritte/editor. You should get a browse navigating the different descriptions themselves. What you see is that a description is an object having multiple fields. In addition since a description is also described using magritte we can get easily a editor to edit descriptions. Browse the class `MADescriptionEditor`: it is the one implementing the description example editor itself.

The class `MADescriptionEditor` uses an instance of `MAAdaptativeModel`. This class is just a mock to act as a described object. It 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 this class? Because this class can represent any object but the object field values are not stored the same way as a normal object.

7.2 Question 17: not sure that we will keep it.

Add a class variable to `CMPerson` called `CustomDescription`. Initialize it with an empty instance of `MAContainer`. Create an accessor method. Override `magritteDescription` 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 17.

```
CMPerson class >> customDescription
  ^ CustomDescription

CMPerson class >> customDescription: aContainer
  CustomDescription := aContainer
```

```
CMPerson class >> initialize
  super initialize.
  self customDescription: MAContainer new

CMPerson >> customDescription
  ^ self class customDescription
```

```
CMPerson >> magritteDescription
  ^ super magritteDescription copy
    addAll: self customDescription; yourself
```

Execute: `CMPerson initialize`

Note that we add a customise element shared by all the instances but we could do the same at the instance level. Now when working with table it is mandatory that all the rows have the same description.

7.3 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

```
MADescriptionEditor subclass: #MAPersonDescriptionEditor
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Magritte-Tutorial'
```

```
MAPersonManager >> modifyPersonModel
  | description |
  description := self call: (MAPersonDescriptionEditor new
    magritteDescription: CMPerson customDescription).
  description isNil
    ifFalse: [ CMPerson 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
```

```
MAPersonDescriptionEditor >> renderButtonsOn: html
  super renderButtonsOn: html.
  html submitButton
    on: #save of: self
```

Now you can add and save a new and specific description representing a new object.

SD: Should check with the version of magritte that does not break on XML...