Magritte

Lukas Renggli

renggli@iam.unibe.ch
Software Composition Group
University of Bern

Who am I?

- Academics
  - PhD Student, University of Bern
- Industrial
  - Software Engineer, netstyle.ch
- Communities
  - Author of Magritte and Pier, and some other open-source projects
  - Contributor to Seaside and Squeak

Agenda

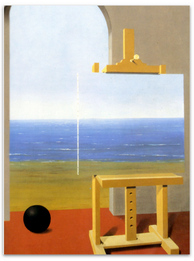
- Introduction
- Examples
- Implementation
- Customization

- Hands-on Exercises

Magritte

Introduction

Describe once,
Get everywhere



What is it useful for?

- Introspection
- Reflection
- Documentation
- *Viewer building*
- *Editor building*
- Report building
- *Data validation*
- Query processing

- Object persistency
- Object indexing
- Object setup
- Object verification
- *Object adaption*
- *Object customization*

and much more

Why is it useful?

- Describe once, get everywhere.
- Automatically build views and editors, process queries and store objects.
- Extensibility of classes is ensured.
- Fully customizable, e.g., it is possible to replace any automatically generated view with a modified or customized one.

Why is it cool?

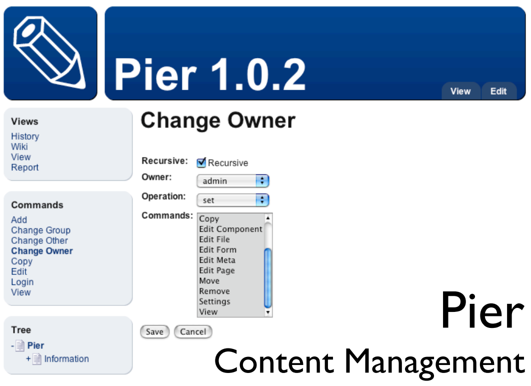
- Describe once, get everywhere.
- Be more productive.
- Lower coupling in software components.
- Do more, with less code.
- Do more, with less hacking.

What is it used for? (1)

- Pier – a meta-described collaborative web-application framework.
- Aare – a proprietary workflow definition and runtime engine with integrated document management system.
- Conrad – a conference registration and management system.

What is it used for? (2)

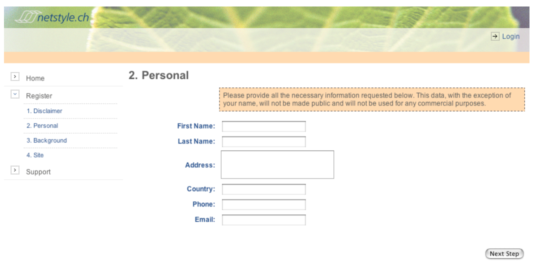
- Seaside-Hosting – free hosting service for non-commercial Seaside applications.
- DigiSens – a proprietary monitoring system for high precision sensors.
- cmsbox – the next generation of a content management system.



Change Owner

Recursive: ✓ Recursive
Owner: admin
Operation: set
Commands:

Pier
Content Management

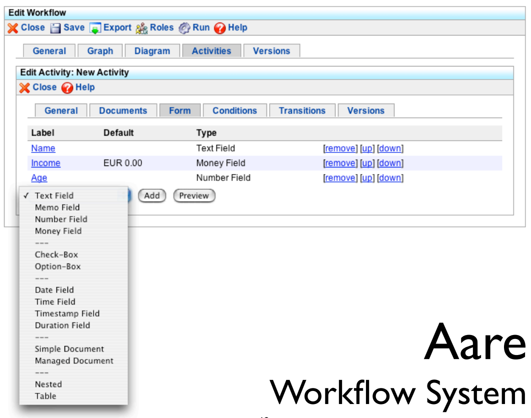


2. Personal

Please provide all the necessary information requested below. This data, with the exception of your name, will not be made public and will not be used for any commercial purposes.

First Name:
Last Name:
Address:
Country:
Phone:
Email:

Seaside-Hosting
Hosting Application



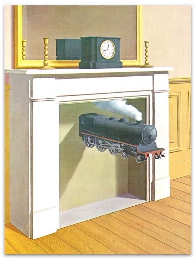
Label	Default	Type	
Name		Text Field	[remove] [up] [down]
Income	EUR 0.00	Money Field	[remove] [up] [down]
Age		Number Field	[remove] [up] [down]

Aare
Workflow System

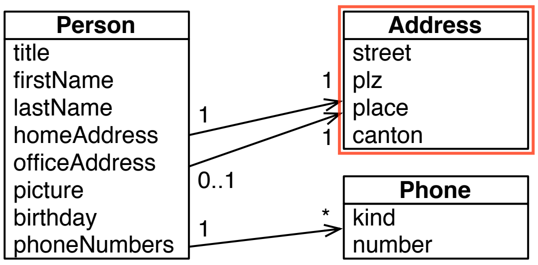
Magritte

Examples

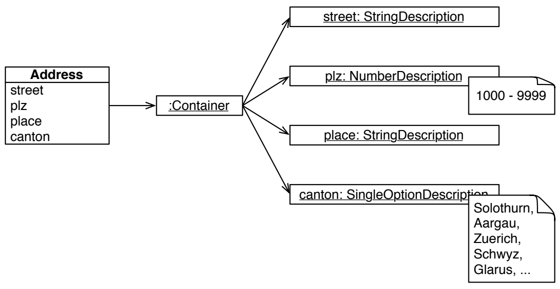
Describe once,
Get everywhere



Address Book



“Describing” the Address



Defining Descriptions

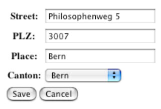
- A object is described by adding methods named #description* (naming convention) to the *class-side* answering different description-entities.
- All descriptions will be *automatically collected* and put into a container description when sending #description to the object.
- Descriptions can be built programmatically.

Describing the Address

```
MAAddressModel class>>descriptionStreet
    ^ MAStringDescription auto: 'street' label: 'Street' priority: 10.

MAAddressModel class>>descriptionPlz
    ^ (MANumberDescription auto: 'plz' label: 'PLZ' priority: 20)
        min: 1000 max: 9999;
        yourself.

MAAddressModel class>>descriptionPlace
    ^ MAStringDescription auto: 'place' label: 'Place' priority: 30.

MAAddressModel class>>descriptionCanton
    ^ (MASingleOptionDescription auto: 'canton' label: 'Canton' priority: 40)
        options: #( 'Bern' 'Solothurn' 'Aargau' 'Zuerich' 'Schwyz' 'Glarus' ...);
        reference: MAStringDescription new;
        beSorted;
        yourself.
```

Seaside Interface

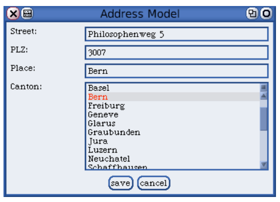
```
result := self call: (aModel asComponent
    addValidatedForm;
    yourself).
```



Street:	Philosophenweg 5
PLZ:	3007
Place:	Bern
Canton:	Bern

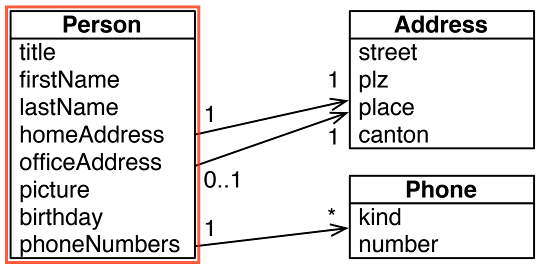
Morphic Interface

```
result := aModel asMorph
      addButtons;
      addWindow;
      callInWorld.
```



Street: Philosophenweg 5
PLZ: 3007
Place: Bern
Canton: Basel, Bern, Freiburg, Geneve, Glarus, Graubunden, Jura, Luzern, Neuchatel, Schaffhausen

Address Book



Describing the Person (1)

```
MAPersonModel class>>descriptionTitle
    ^ (MASingleOptionDescription auto: 'title' label: 'Title' priority: 10)
        options: #( 'Mr.' 'Mrs.' 'Ms.' 'Miss.' );
        yourself.

MAPersonModel class>>descriptionFirstName
    ^ (MAStringDescription auto: 'firstName' label: 'First Name' priority: 20)
        beRequired;
        yourself.

MAPersonModel class>>descriptionLastName
    ^ (MAStringDescription auto: 'lastName' label: 'Last Name' priority: 30)
        beRequired;
        yourself.
```

Describing the Person (2)

```
MAPersonModel class>>descriptionHomeAddress
    ^ (MAToOneRelationDescription auto: 'homeAddress' label: 'Home Address')
        classes: (Array with: MAAddressModel);
        beRequired;
        yourself.

MAPersonModel class>>descriptionOfficeAddress
    ^ (MAToOneRelationDescription auto: 'officeAddress' label: 'Office Address')
        classes: (Array with: MAAddressModel);
        yourself.

MAPersonModel class>>descriptionPicture
    ^ (MAFileDescription auto: 'picture' label: 'Picture')
        addCondition: [ :value | value isImage ] labelled: 'Image expected';
        yourself.
```

Describing the Person (3)

```
MAPersonModel class>>descriptionPhoneNumbers
    ^ (MAToManyRelationDescription auto: 'phoneNumbers' label: 'P. Numbers')
        classes: (Array with: MAPhoneNumber);
        default: Array new;
        yourself.

MAPersonModel class>>descriptionBirthday
    ^ MADateDescription auto: 'birthday' label: 'Birthday'.

MAPersonModel class>>descriptionAge
    ^ (MANumberDescription selector: #age label: 'Age')
        beReadonly;
        yourself.
```

Recapitulation

- Put your descriptions on the *class-side* according to the *naming-convention*.
- Ask your object for its *description-container* by sending #description.
- Ask your object for an *User-Interface* by sending #asComponent or #asMorph.

Magritte

Implementation

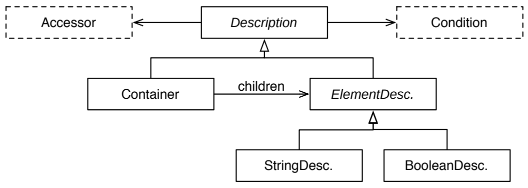
Describe once,
Get everywhere

Descriptions

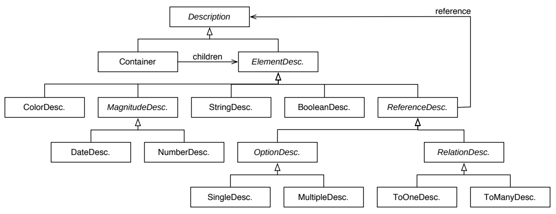
- Problem
  - Smalltalk classes are all very different and require different configuration possibilities.
- Example
  - `Boolean` and `String` are not polymorphic, therefor different code for printing, parsing, serializing, editing, comparing, querying, etc. is necessary.
- Solution
  - Introduce a descriptive hierarchy that can be instantiated, configured and composed.

Descriptions

a composite pattern
to describe model-classes/-instances



Descriptions

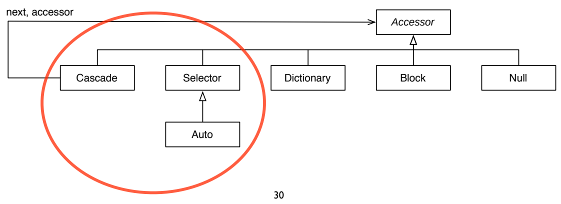


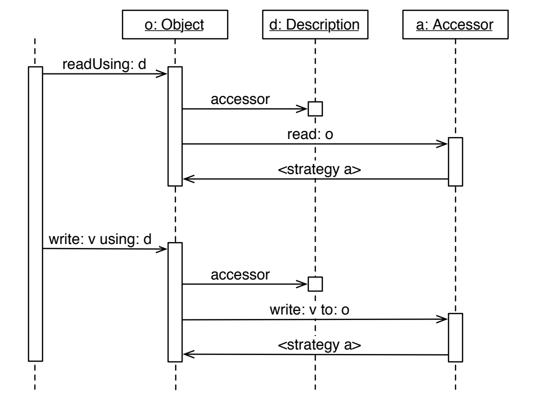
Accessors

- Problem
  - In Smalltalk data can be stored and accessed in very different ways.
- Examples
  - Accessor methods, chains of accessor methods, instance-variables, dictionaries, blocks, etc.
- Solution
  - Provide a strategy pattern to be able to access the data trough a common interface.

Accessors

a strategy pattern
to access model-entities



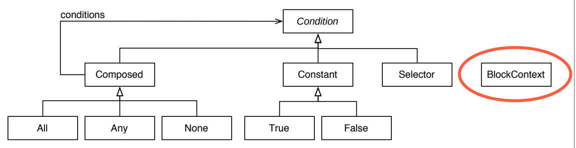


Conditions

- Problems
  - End users want to visually compose conditions.
  - Instances of BlockContext can be hardly serialized.
- Solution
  - Introduce condition objects that can be composed to describe constraints on objects and data.

Conditions

a composite pattern
to model constraints

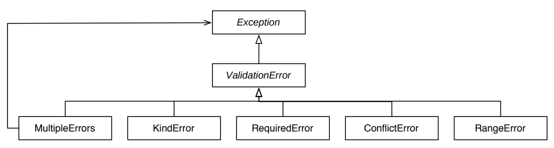


Exceptions

- Problems
  - Actions on the meta-model can fail.
  - Objects might not match a given meta-model.
  - Software would like to avoid errors.
  - End users want readable error messages.
- Solution
  - Introduce an exception hierarchy knowing about the description, the failure and a human-readable error message.

Exceptions

a composite pattern
of smalltalk exceptions

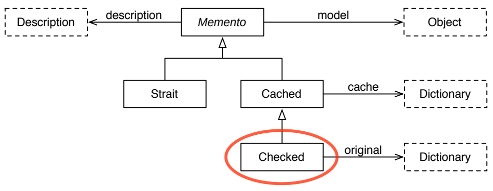


Mementos

- Problems
  - Editing might turn a model (temporarily) invalid.
  - Canceling an edit shouldn’t change the model.
  - Concurrent edits of the same model should be detected and (manually) merged.
- Solution
  - Introduce mementos that behave like the original model and that delay modifications until they are proven to be valid.

Mementos

a proxy pattern
to cache model-entities



Magritte

Customization

Describe once,
Get everywhere

Dynamic Descriptions

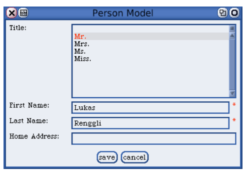
- Problem
  - Instances might want to *dynamically* filter, add or modify their descriptions.
  - Users of a described object often *don't need all the available descriptions*.
- Solution
  - Override #description on the instance-side to modify the default description-container.
  - Add other methods returning different *filtered* or *modified* sets of your descriptions.

Building Descriptions Dynamically

```
" select descriptions "
MAPersonModel>>descriptionPrivateData
    ^ self description select: [ :each |
        #( title firstName lastName homeAddress )
            includes: each accessor selector ].

" add another description "
MAPersonModel>>descriptionWithEmail
    ^ self description copy
        add: (MAStringDescription auto: 'email' label: 'E-Mail' priority: 35);
        yourself.

" modify existing description "
MAPersonModel>>descriptionWithRequiredImage
    ^ self description collect: [ :each |
        each accessor selector = #picture
            ifTrue: [ each copy beRequired ]
            ifFalse: [ each ] ].
```

Using Dynamic Descriptions

```
model := MAPersonModel new.

" get a morph "
morph := model descriptionPrivateData
    asMorphOn: model.

" get a component "
component := model descriptionPrivateData
    asComponentOn: model.
```


Person Model

Title: Mr. / Mrs. / Ms. / Miss.

First Name: Lukas *

Last Name: Renggli *

Home Address:

Custom Validation

- Problem
  - A lot of *slightly* different validation strategies leads to an explosion of the description class-hierarchy.
- Example
  - A number must be in a certain *range*.
  - An e-mail address must match a *regular-expression*.
- Solution
  - Additional validation rules can be added to all descriptions.

Validation Rules

- Use #addCondition:labelled: to add additional conditions to descriptions that will be *automatically checked* before committing to the model.
- The first argument is a block taking one argument, that should return true if the argument validates.
- Using a *block-closure* is possible, but you will loose the possibility to serialize the containing description. Send it the message #asCondition before adding to parse it and keep it as serialize-able *AST* within the description.

43

Validation Examples

```
(MANumberDescription selector: #age label: ‘Age’)
      addCondition: [ :value | value isInteger and: [ value between: 0 and: 100 ] ]
      labelled: ‘invalid age’;
      ...

(MAStringDescription selector: #email label: ‘E-Mail’)
      addCondition: [ :value | value matches: ‘#*@#*.#*’ ]
      labelled: ‘invalid e-mail’;
      ...

(MADateDescription selector: #party label: ‘Party’)
      addCondition: [ :value | self possiblePartyDates includes: value ]
      labelled: ‘party hard’;
      ...
```

44

Custom Description

- Problem
  - In some cases it might happen that there is no description provided to use with a model class.
- Example
  - Money: amount and currency.
  - Url: scheme, domain, port, path, parameters, etc.
- Solution
  - Create your own description.

45

Your own Description

- 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.
  - #validateSpecific: to validate.
- Create a view, if you want to use it for UI building.

46

Tips for Builders

- Have a look at existing descriptions.
- Carefully choose the right superclass.
- Reuse the behaviour from the superclass.
- Parsing, printing and (de)serialization is implemented in vistiors:
  - MAStringReader, MAStringWriter
  - MABinaryReader, MABinaryWriter

47

Custom View

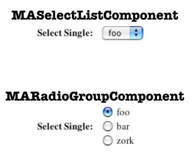
- Problems
  - Custom descriptions mostly need a *new view*.
  - Applications might need a *special view* for existing descriptions to adapt a better user experience.
- Example
  - Money: an input-field for the amount and a drop-down box to select the currency.
- Solution
  - Choose a different view or create your own.

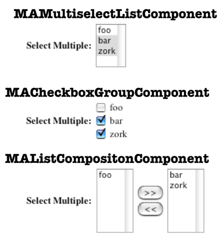
48

Different Views

Single Option

Multiple Option





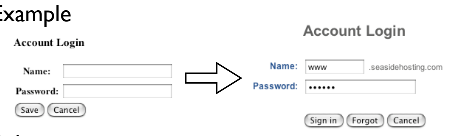
aDescription componentClass: aClass

Your own View

- 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 (there is no clean way to do that right now, Pragmas would help).

Custom Rendering

- Problem
  - Automatic built UIs are often not that user-friendly, and they all look more or less the same.
- Example



- Solution
  - Use CSS and customize the rendering of your UI.

Possibility 1

- Create a subclass of WAComponent.
- Create an i-var holding onto the automatically built component:
  dialog := aModel asComponent
- Don't forget to return it as a child!
- Implement your own rendering code, accessing the magritte sub-views by calling:
  dialog childAt: aModel class descriptionFoo
- Commit your model by sending:
  dialog commit

Possibility 2

- Create a new subclass of MAComponentRenderer.
- Implement the new visitor to get the layout you need.
- Override the method #descriptionContainer in your model like this:

```
MyModel class>>descriptionContainer
   ^ super descriptionContainer
        componentRenderer: MyRendererClass;
        yourself.
```

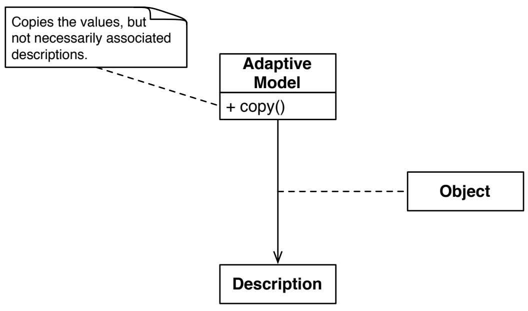
Possibility 3

- Create a new subclass of MAContainerComponent.
- Override the method #renderContentOn: to get the layout you need (avoiding the vistor).
- Override the method #descriptionContainer in your model like this:

```
MyModel class>>descriptionContainer
   ^ super descriptionContainer
        componentClass: MyComponentClass;
        yourself.
```

Adaptive Model

- Problem
  - End users require quick changes in their software.
  - End users want to customize and build their own meta-models on the fly.
- Example
  - Add additional fields to an address database.
- Solution
  - Magritte is self described.


Copies the values, but not necessarily associated descriptions.
Adaptive Model
+ copy()
Object
Description

Adaptive Model 1

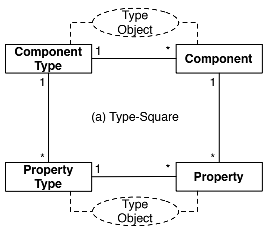
- Create a subclass of MAAdaptiveModel
- Create an editor for the adaptive descriptions: anAdaptiveModel description asComponent

Adaptive Model 2

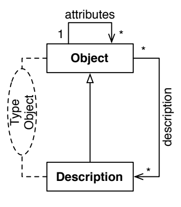
- Add an instance-variable description to your object and override #description.
- Add an instance-variable values to your object that is initialized with a Dictionary.
- Override two methods with something like:

```
AdaptiveModel>>readUsing: aDescription
        ^ values at: aDescription ifAbsent: [ aDescription default ]

AdaptiveModel>>rwrite: anObject using: aDescription
        values at: aDescription put: anObject
```

Type-Square



(a) Type-Square



(b) Magritte

Conclusion

- Describe once, get everywhere.
- Ensure extensibility and maintainability.
- Automate boring tasks, like building and validating GUIs.
- Be adaptive.

Magritte

Meta Described Web Application Development

http://www.iam.unibe.ch/~scg/Archive/Diploma/Reng06a.pdf