{} descriptions Documentation

Release 0.1

Mariano Montone

April 21, 2014

CONTENTS

{} descriptions is a meta level descriptions library for Common Lisp.

It is inspired by Smalltalk Magritte, as well as Lisp On Lines.

CHAPTER

ONE

OVERVIEW

{} descriptions is a meta level descriptions library for Common Lisp.

It is inspired by Smalltalk Magritte, as well as Lisp On Lines.

CHAPTER

TWO

INSTALL

Download the source code from https://github.com/mmontone/descriptions and point *.asd* system definition files from `./sbcl/system (ln -s <system definition file path>)` and then evaluate:

```
(require :descriptions)
```

from your lisp listener.

You will also need to satisfy these system dependencies:

- *alexandria*
- *anaphora*
- *sheeple*
- *cxml* and *cl-json* for the serialization module
- *cl-ppcre* for the validation module

The easiest way of installing those packages is via Quicklisp

This library is under the MIT licence.

CHAPTER

THREE

GETTING STARTED

{} descriptions is a very dynamic meta-description library. Meta description of the application's domain model is done defining descriptions for the different aspects is desired to represent. This helps automate the generation of different views, editors, serialization schemas, validation procedures for the domain model objects and to avoid very repetitive and error-prone work when those objects change its shape.

Domain model meta descriptions consist of the definition of description objects for its different aspects (viewing, editing, validation, persistence, serialization, etc). Description objects are a collections of attributes, can inherit from each other, and are composable.

Let's go through an example to see how this works.

Say we have a *person* model object:

```
(defclass person ()
  ((id :accessor id)
   (username :initarg :username
             :accessor username)
   (fullname :initarg :fullname
             :accessor fullname)
   (email :initarg :email
          :accessor email)
   (password :initarg :password
             :accessor password)))
```

We would like to print instances of that model on the screen. So first we define a *{person}* description that just describes which attributes *person* model objecs possess and their "types".

```
(define-description {person} ()
  ((username =>string
             :reader #'username
             :writer #'(setf username))
   (email =>email
          :reader #'email
          :writer #'(setf email))
   (fullname =>string
             :reader #'fullname
             :writer #'(setf fullname))
   (password =>password
             :reader #'password
             :writer #'(setf password)))
  :documentation "A person description")
```

Descriptions names are enclosed between brackets `{ }` as a naming convention. Also, notice that the attribute types (`=>string`, `=>email`, `=>password`), begin with `=>`. Attributes types begin with `=>` as a naming convention.

Now we can use the information on the attributes of a person, and its types, to print a description of a person on the screen. To do that we define a new description that indicates which person slots we would like to print, and in which order.

```
(define-description {person-view} ({person})
  ((fullname =>view :label "Full name")
   (username =>view)
   (email =>view :label "E-mail")
   (password =>view :view nil)))
```

We can see that the password is disabled for viewing, that "Full name" and "E-mail" are used for labels, and that fullname, username, email and password are to be displayed in that order.

We can see that in action:

```
(display-object (make-instance 'person :username "mmontone"
                                       :email "mariano@copyleft.no"
                                       :password "lalala"
                                       :fullname "Mariano Montone")
                {person-view})
```

prints:

```
Full name: "Mariano Montone"
Username: "mmontone"
E-mail: "mariano@copyleft.no"

### Descriptions composition
```

Descriptions can be composed using inheritance. Multiple inheritance is supported and they are implemented on top of a prototype based object system, so they can be composed in run time on the fly.

3.1 Attributes composition

When inheriting from other descriptions, attributes with the same name are collapsed into one containing all the attribute properties.

For example, consider the basic *{person}* description we had before. We can ask for an attribute name and type, but asking if that attribute is going to be displayed throws an error, because that attribute property does not belong to the {person} description, but the *{person-view}* description we saw before.

```
{}> (attribute-name (get-attribute {person} 'fullname))
FULLNAME
{}> (attribute-type (get-attribute {person} 'fullname))
#<Object =>STRING {1005A0A0A3}>
{}> (attribute-view (get-attribute {person} 'fullname))
;; Error
```

But if we ask the same to the *{person-view}* description, we can access to all the attributes properties.

```
{}> (attribute-name (get-attribute {person-view} 'fullname))
FULLNAME
{}> (attribute-type (get-attribute {person-view} 'fullname))
#<Object =>STRING {1005A0A0A3}>
{}> (attribute-view (get-attribute {person-view} 'fullname))
T
```

This makes the approach layered, with each description describing different aspects of the same model and extending other more basic descriptions.

3.2 Descriptions composition

As we mentioned before, descriptions can be composed on the fly thanks to the prototype object system the library is implemented in.

For example, we can create new descriptions with the *make-description* function, passing the descriptions we want to compose as parents:

```
(let ((description (make-description :parents
                                     (list {person-validation} {person-repl-editing}))))
  (let ((person (make-instance 'person)))
      (edit-object person description)
      (validate-object person description)
      (describe person)))
```

A prettier way of doing it is using the description name as a function:

```
(let ((description ({person-validation} {person-repl-editing})))
    (let ((person (make-instance 'person)))
      (edit-object person description)
      (validate-object person description)
      (describe person)))
```

We can also choose not to compose descriptions, but work with them separately on the different aspects of the model objects:

```
(let ((person (make-instance 'person)))
  (edit-object person {person-repl-editing})
  (validate-object person {person-validation})
  (describe person)
  (print
   (with-output-to-string (s)
     (serialize-object person {person-serialization} s))))
```

CHAPTER

FOUR

IMPLEMENTATION

{} descriptions is implemented as a thin layer over Sheeple prototypes. As a consequence, descriptions and its attributes can be easily be combined in run time.

4.1 Why not just use metaclasses instead?

It is not possible to achieve quite the same with metaclasses. They do not allow to keep the metamodel independent of the actual implementation of the class. It should be possible to exchange descriptions on the fly, and even use multiple descriptions at the same time for the same underlying domain object.

CHAPTER

FIVE

REFERENCES

- http://code.google.com/p/magritte-metamodel/
- http://www.cliki.net/lisp-on-lines
- http://common-lisp.net/project/lisp-on-lines/repo/lisp-on-lines/doc/manual.html
- http://scg.unibe.ch/archive/papers/Reng07aMagritte.pdf

CHAPTER

SIX

API

Description external symbols documentation

function (**=>view** *&rest property-values*)
Create a =>VIEW attribute. Takes a plist of property values for the created attribute

variable (**=>view** *&rest property-values*)

function (**=>string** *&rest property-values*)
Create a =>STRING attribute. Takes a plist of property values for the created attribute

variable (**=>string** *&rest property-values*)
String attribute

variable {description}

function (**attribute-documentation** *attribute*)

function (**add-attribute** *description attribute &key reader writer accessor*)

function (**=>password** *&rest property-values*)
Create a =>PASSWORD attribute. Takes a plist of property values for the created attribute

variable (**=>password** *&rest property-values*)
Password attribute

function (**=>integer** *&rest property-values*)
Create a =>INTEGER attribute. Takes a plist of property values for the created attribute

variable (**=>integer** *&rest property-values*)
Integer attribute

function (**=>single-option** *&rest property-values*)
Create a =>SINGLE-OPTION attribute. Takes a plist of property values for the created attribute

variable (**=>single-option** *&rest property-values*)

function (**=>email** *&rest property-values*)
Create a =>EMAIL attribute. Takes a plist of property values for the created attribute

variable (**=>email** *&rest property-values*)
Email attribute

function (**=>boolean** *&rest property-values*)
Create a =>BOOLEAN attribute. Takes a plist of property values for the created attribute

variable (**=>boolean** *&rest property-values*)
Boolean attribute

function (**get-attribute** *description attribute-name*)

variable =>

function (**=>valued** *&rest property-values*)
Create a =>VALUED attribute. Takes a plist of property values for the created attribute

variable (**=>valued** *&rest property-values*)

macro (**define-description** *name parents attributes &rest options*)
Define a new description

macro (**define-attribute** *name parents properties &rest options*)
Define an attribute type

param name the attribute type name.

param parents the attribute type parents.

param properties list of properties.

param options options, like :documentation, etc

function (**=>to-many-relation** *&rest property-values*)
Create a =>TO-MANY-RELATION attribute. Takes a plist of property values for the created attribute

variable (**=>to-many-relation** *&rest property-values*)

function (**display-object** *sheeple:object description &optional stream*)

function (**attribute-properties** *attribute*)

function (**=>to-one-relation** *&rest property-values*)
Create a =>TO-ONE-RELATION attribute. Takes a plist of property values for the created attribute

variable (**=>to-one-relation** *&rest property-values*)

function (**description-attributes** *description*)
Obtain a description attributes. :param description: the description.

function (**make-attribute** *attribute-type &rest property-values*)
Create an attribute.

function (**make-description** *&key parents attributes*)

function (**=>multiple-option** *&rest property-values*)
Create a =>MULTIPLE-OPTION attribute. Takes a plist of property values for the created attribute

variable (**=>multiple-option** *&rest property-values*)

macro (**with-description-attributes** *attributes description &body body*)
Run body with description attributes bound

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *search*

Symbols

A

D

G

M

W