
cldomain Documentation

Release 0.18.1

Russell Sim

Apr 17, 2024

TOPIC AREAS

1	Installation	3
1.1	Requirements	3
1.2	Download	3
2	Configuration	5
2.1	Loading the extension	5
2.2	cl_systems	5
2.3	cl_packages	6
2.4	cl_debug	6
2.5	highlight_language	6
3	Output formats	7
4	Complete Example	9
5	Common Lisp docstrings	11
6	Don't include the docstring: :nodoc:	13
6.1	Code	13
6.2	Output	13
7	Cross-references	15
8	Hyperspec References	17
9	Packages	19
10	Variables	21
10.1	Code	21
10.2	Output	21
10.3	Code	21
10.4	Output	22
11	Functions	23
11.1	Code	23
11.2	Output	23
12	Macros	25
12.1	Code	25
12.2	Output	25

13 Generics	27
13.1 Code	27
13.2 Output	27
13.3 Disable listing specializers	27
14 Methods	29
14.1 Code	29
14.2 Output	29
14.3 Disable inheriting documentation	29
14.4 Disable linking to the generic	30
15 CLOS Class	31
15.1 Code	31
15.2 Output	31
15.3 Code	32
15.4 Output	33
16 0.18.1 - 2023-06-04	35
16.1 Minor Fixes	35
16.2 Documentation	35
17 0.18.0 - 2023-03-04	37
17.1 Features	37
17.2 Minor Fixes	37
17.3 Documentation	37
17.4 Cleanups	38
18 0.17.1 - 2023-01-18	39
18.1 Minor Fixes	39
19 0.17.0 - 2023-01-18	41
19.1 Features	41
19.2 Minor Fixes	41
19.3 Documentation	41
19.4 Build Tooling	41
20 0.16.2 - 2023-01-08	43
20.1 Minor Fixes	43
21 0.16.1 - 2023-01-08	45
21.1 Minor Fixes	45
22 0.16.0 - 2023-01-08	47
22.1 Features	47
22.2 Minor Fixes	47
22.3 Documentation	47
22.4 Tests	47
22.5 Build Tooling	48
22.6 Cleanups	48
23 0.15.3 - 2022-07-24	49
24 0.15.2 - 2022-07-24	51
25 0.15.1 - 2022-07-24	53

26	0.15 - 2022-07-23	55
27	0.14 - 2022-07-10	57
28	0.13 - 2015-09-06	59
29	0.12 - 2015-02-24	61
30	0.11 - 2014-12-30	63
31	0.10 - 2014-06-12	65
32	0.9 - 2014-02-10	67
33	0.8 - 2014-02-10	69
34	0.7 - 2013-06-12	71
35	0.6 - 2013-04-22	73
36	0.5 - 2013-04-20	75
37	0.4 - 2013-04-19	77
38	0.3 - 2013-04-16	79
39	0.2 - 2013-04-14	81
40	0.1 - UNRELEASED	83
	Index	85

CLDomain is a Common Lisp domain for [Sphinx Documentation Generator](#). Sphinx is a multi-language documentation tool. This project extends its functionality to cover Common Lisp. The aim is to support documentation with the same ease as Sphinx would a Python project.

Documentation is extracted from the various entity's documentation strings, loaded from ASDF systems and associated internal packages.

Hyperspec is a cross referencing extension that supports linking to the *Hyperspec*.

CLDomain is licensed under the [GPLv3](#). Please report any bugs in the [Mailing List](#).

This documentation contains example of the generated documentation but if you want a more serious example [cl-git](#) is using it too.

INSTALLATION

1.1 Requirements

- Sphinx
- roswell
- pygments-cl-repl

1.2 Download

Releases are available via [pypi](#) or as [git tags](#). The [source](#) is also available.

```
pip install sphinxcontrib-cldomain
```


CONFIGURATION

Configuring CLDomain involves two actions: (a) adding the extensions to the extension list, (b) telling CLDomain the systems and packages to load.

2.1 Loading the extension

To load the extension add `sphinxcontrib.cldomain` to the list of extensions, you might also want to load the `sphinxcontrib.hyperspec` extension if you want any COMMON-LISP package symbols to be directed to the `hyperspec`.

```
# Extensions: add 'sphinxcontrib.cldomain' and 'sphinxcontrib.hyperspec',  
# just like this example:  
extensions - [  
    'sphinx.ext.intersphinx',  
    'sphinxcontrib.cldomain',  
    'sphinxcontrib.hyperspec'  
]
```

2.2 cl_systems

To load symbols from an ASDF system, specify the system, the path and the list of packages to load.

```
from os.path import join, dirname, realpath, expandvars  
  
# --- CL domain customizations:  
#  
# cl_systems: The systems and packages from which to extract documentation:  
#  
# name - The name of the system to load.  
# path - The path to the system.  
# packages - A list of the packages to extract symbol information from.  
#  
# Note: This conf.py sits in a subdirectory below ("../"), relative to where  
# the "my-system.asd" system description file lives:  
cl_systems - [{"name": "my-system",  
              "path": join(dirname(realpath(__file__)), "../"),  
              "packages": ["my-package-1", "my-package-2"]}]
```

As each system will be loaded before any of the package information is loaded and passed to the Sphinx engine via JSON.

name

The name of the system to be loaded.

path

The path to the ASD file of the system. This is useful because often the path to the system might be relative to the documentation.

package

The packages from the system to load documentation and other symbol information.

2.3 cl_packages

The `cl_packages` variable contains a list of all the packages to load symbols for. It can be used instead of the system base object specified above. It's really mostly useful for loading the COMMON-LISP package if that is needed.

```
# cl_packages: A list of packages that already exist in the lisp image.  
cl_packages - ["common-lisp"]
```

2.4 cl_debug

Setting debug to True will output the JSON that is rendered when collecting LISP symbol information.

```
# For developer debugging only (and the curious, although, it did kill the cat!)  
# Currently ``True`` or ``False`` to output the JSON collected from cl-launch.  
cl_debug - False
```

2.5 highlight_language

It is also worthwhile setting the default highlighting language to Common Lisp if you don't want to have to specify the language for every source block.

```
# Ensure that the default highlighting language is CL:  
highlight_language = 'common-lisp'
```

OUTPUT FORMATS

Sphinx can output HTML, pdf, info

To test the info file you can open it in Emacs using `C-u C-h i <filename>`.

COMPLETE EXAMPLE

```
from os.path import join, dirname, realpath, expandvars

# Extensions: add 'sphinxcontrib.cldomain' and 'sphinxcontrib.hyperspec',
# just like this example:
extensions = [
    'sphinx.ext.intersphinx',
    'sphinxcontrib.cldomain',
    'sphinxcontrib.hyperspec'
]

# --- CL domain customizations:
#
# cl_systems: The systems and packages from which to extract documentation:
#
# name - The name of the system to load.
# path - The path to the system.
# packages - A list of the packages to extract symbol information from.
#
# Note: This conf.py sits in a subdirectory below ("./"), relative to where
# the "my-system.asd" system description file lives:
cl_systems = [{"name": "my-system",
               "path": join(dirname(realpath(__file__)), "./"),
               "packages": ["my-package-1", "my-package-2"]}]}

# cl_packages: A list of packages that already exist in the lisp image.
cl_packages = ["common-lisp"]

# Ensure that the default highlighting language is CL:
highlight_language = 'common-lisp'

# For developer debugging only (and the curious, although, it did kill the cat!)
# Currently ``True`` or ``False`` to output the JSON collected from cl-launch.
cl_debug = False
```


COMMON LISP DOCSTRINGS

CLDomain collects the documentation strings for the package-exported symbols in each system enumerated in the `cl_systems` configuration variable, which CLDomain appends to the symbol's signature. You can include additional documentation after the directive and it will also get included in the Spinx-generated output. The output template looks like:

type: signature

symbol-docstring

Any additional text described in the RST files.

For an example, follow [this](#) link or read on.

DON'T INCLUDE THE DOCSTRING: :NODOC:

By default documentation strings will propagate through from symbols declared in Common Lisp, sometimes you'd prefer to provide separate (non-docstring) documentation. That's what the `:nodoc` option does.

Argument lists and specializers will still be printed, but can be disabled using other configuration.

For example:

```
.. cl:macro:: example-macro
   :nodoc:

No documentation from the ``example-macro`` documentation string.
```

6.1 Code

```
(defmacro example-macro ((arg1 arg2) &body arg3)
  "The CL Domain will try and convert any uppercase symbols into
reference for example EXAMPLE-FUNCTION or a hyperspec link LIST. Any
unmatched symbols are converted to literals as is ARG1, ARG2 and ARG3.
Explicit package references will also help resolve symbol sources
COMMON-LISP:CDR. Keywords are also detected for example :TEST."
  arg3)
```

6.2 Output

Macro (`example-macro` (*arg1 arg2*) &body *arg3*)

No documentation from the `example-macro` documentation string.

CROSS-REFERENCES

You can cross reference Lisp entities using the following CLDomain Sphinx roles, which results in a hyperlinked reference to the matching identifier, if found:

:cl:function:

References a function, as in `:cl:function:`example-function`` (link: [example-function](#)).

:cl:generic:

References a generic function, as in `:cl:generic:`example-generic`` (link: [example-generic](#)).

:cl:method:

References a generic-specializing method, as in `cl:method:`example-generic method <sphinxcontrib.cldomain.doc:example-generic (sphinxcontrib.cldomain.doc:example-class (eq keyword:test1))>`` (link: [example-generic method](#)).

:cl:macro:

References a macro, as in `:cl:macro:`example-macro`` (link: [example-macro](#)).

:cl:variable:

References a variable, as in `:cl:variable:`*example-variable*`` (link: [*example-variable*](#)).

:cl:clos-class:

References a CLOS class, as in `:cl:clos-class:`example-class`` (link: [example-class](#)).

:cl:clos-slot:

References a CLOS slot, as in `:cl:clos-slot:`slot2 <sphinxcontrib.cldomain.doc:example-class sphinxcontrib.cldomain.doc::slot2>`` (link: [slot2](#)).

:cl:symbol:

References a symbol, such as `:cl:symbol:example-function` (link: [example-function](#)).

HYPERSPEC REFERENCES

Generating a reference is very easy (and you've probably noticed already if you've read the Common Lisp code snippets used to generate the examples). To generate a Hyperspec reference:

1. THE COMMON LISP SYMBOL NAME IS IN ALL CAPS, LIKE LIST OR FORMAT. (No, the documentation isn't shouting at you. It's the normal Lisp convention for symbols.
2. Prefix the symbol name with COMMON-LISP:, e.g., COMMON-LISP:CAR

The *cl:function: example* has an example of Hyperspec-ing in its example code.

PACKAGES

CLDomain, like Common Lisp, needs to know the current package when resolving symbols. The `:cl:package:` directive is the CLDomain equivalent of `(in-package ...)`. You can switch between packages at any time in the documentation file using this directive.

.. cl:package:: package

Use `package` as the package name when resolving symbols to documentation:

```
.. cl:package:: sphinxcontrib.cldomain.doc
```

For multi-package documentation in the same Sphinx documentation file:

```
.. cl:package:: sphinxcontrib.cldomain.doc  
documentation... documentation... documentation...  
.. cl:package:: org.coolness.my.code  
foo... bar... baz... lemon odor quux!!!
```


VARIABLES

`.. cl:variable:: symbol-name`

The `cl:variable` directive will resolve the arguments and documentation from the common lisp definition:

```
.. cl:variable:: *example-variable*
```

10.1 Code

```
(defvar *example-variable* "value"  
  "This is an example variable.")
```

10.2 Output

Variable `*example-variable*`

This is an example variable.

You can include additional text, which appears after the docstring (unless you use the `:nodoc:` option):

```
.. cl:variable:: *example-variable-2*
```

This variable requires more explanatory text after its docstring. Because, more text means more clarity and further explains **the** intent of **the** original software developer.

10.3 Code

```
(defvar *example-variable-2* "another value"  
  "This example has additional text.")
```

10.4 Output

Variable `*example-variable-2*`

This example has additional text.

This variable requires more explanatory text after its docstring. Because, more text means more clarity and further explains the intent of the original software developer.

FUNCTIONS

`.. cl:function:: symbol-name`
Outputs the function's signature (arguments):

```
.. cl:function:: example-function
```

11.1 Code

```
(defun example-function (arg1 arg2 &optional (arg3 #'sort) &key (kw *example-variable*))  
  "The CL Domain will try and convert any uppercase symbols into  
  reference for example EXAMPLE-FUNCTION, EXAMPLE-GENERIC or a hyperspec  
  link LIST. Any unmatched symbols are converted to literals as is  
  ARG1, ARG2 and ARG3. Explicit package references will also help  
  resolve symbol sources COMMON-LISP:CAR. Keywords are also detected  
  for example :KEYWORD."  
  (list arg1 arg2 arg3))
```

11.2 Output

Function (`example-function` *arg1* *arg2* &optional (*arg3* #'sort) &key (*kw* *example-variable*))

(setf (**example-function** *arg1* *arg2*) *value*)

The CL Domain will try and convert any uppercase symbols into reference for example `EXAMPLE-FUNCTION`, `EXAMPLE-GENERIC` or a hyperspec link `LIST`. Any unmatched symbols are converted to literals as is `ARG1`, `ARG2` and `ARG3`. Explicit package references will also help resolve symbol sources `CAR`. Keywords are also detected for example `:KEYWORD`.

MACROS

.. **cl:macro::** symbol-name

Emit the macro's signature and documentation:

```
.. cl:macro:: example-macro
```

12.1 Code

```
(defmacro example-macro ((arg1 arg2) &body arg3)
  "The CL Domain will try and convert any uppercase symbols into
  reference for example EXAMPLE-FUNCTION or a hyperspec link LIST. Any
  unmatched symbols are converted to literals as is ARG1, ARG2 and ARG3.
  Explicit package references will also help resolve symbol sources
  COMMON-LISP:CDR. Keywords are also detected for example :TEST."
  arg3)
```

12.2 Output

Macro (**example-macro** (*arg1 arg2*) &body *arg3*)

The CL Domain will try and convert any uppercase symbols into reference for example *EXAMPLE-FUNCTION* or a hyperspec link *LIST*. Any unmatched symbols are converted to literals as is *ARG1*, *ARG2* and *ARG3*. Explicit package references will also help resolve symbol sources *CDR*. Keywords are also detected for example *:TEST*.

GENERICIS

Generics will by default produce a list of methods that specialize them. Setf functions or methods will also be printed.

.. `cl:generic::` symbol-name

The `:cl:generic:` directive emits the documentation for a generic function and its specializers:

```
.. cl:generic:: example-generic
```

13.1 Code

```
(defgeneric example-generic (arg1 arg2 &optional arg3)
  (:documentation "A test generic function."))
```

13.2 Output

Generic (`example-generic` *arg1* *arg2* &optional *arg3*)

```
(example-generic (arg1 EXAMPLE-CLASS) (arg2 (eql :TEST)) &OPTIONAL ARG3)
(example-generic (arg1 EXAMPLE-CLASS) (arg2 (eql :TEST1)) &OPTIONAL ARG3)
(example-generic (arg1 EXAMPLE-CLASS) (arg2 (eql :TEST2)) &OPTIONAL ARG3)
(example-generic (arg1 EXAMPLE-CLASS) (arg2 T) &OPTIONAL ARG3)
(setf (example-generic (arg1 EXAMPLE-CLASS) (arg2 (eql :TEST))) (new-value T))
(setf (example-generic (arg1 EXAMPLE-CLASS) (arg2 T)) (new-value EXAMPLE-CLASS))
```

A test generic function.

13.3 Disable listing specializers

Generics will also list other specializing methods by default this behaviour can be disabled by passing the `::nospecializers::` option:

```
.. cl:generic:: example-generic
  :nospecializers:
```

The same generic that is listed in the *Generics* example will render like this with it's specializers disabled..

Generic (**example-generic** *arg1 arg2 &optional arg3*)

A test generic function.

METHODS

Methods can also be documented separate to the Generic they implement. This is for cases where the method might define vastly different behaviour to the generic. Or maybe you want to group all the methods that relate to a CLOS Class with that object's documentation rather than having users jump generics page to verify what types are specialized.

.. `cl:method::` symbol-name (specializer)

The `:cl:method` emits the documentation for generic method specializers:

```
.. cl:method:: example-generic example-class :test
```

For the time being, all specializing arguments that aren't in the current package must be qualified with a package, e.g., `common-lisp:t`

14.1 Code

```
(defmethod example-generic ((arg1 example-class) (arg2 (eql :test)) &optional arg3)
  "This is the first specialized version of example-generic."
  (list arg1 arg2 arg3))
```

14.2 Output

Method (`example-generic` (*arg1* *EXAMPLE-CLASS*) (*arg2* (eql :TEST)) &optional *arg3*)

(setf (`example-generic` (*arg1* *EXAMPLE-CLASS*) (*arg2* (eql :TEST))) (*new-value* T))

This is the first specialized version of example-generic.

See also: *example-generic*

14.3 Disable inheriting documentation

Note: The output for a specializing method will include its parent generic function's documentation string if there is no documentation for the method, i.e., specializing methods will inherit their parent generic's docstring. The `:noinherit:` option suppresses this behaviour and will result in no docstring:

```
.. cl:method:: example-generic example-class :test1
  :noinherit:
```

This will be useful if you want to specify a completely custom documentation string in the generated documentation. The output will look like.

Method (**example-generic** (*arg1* *EXAMPLE-CLASS*) (*arg2* (eql :TEST1)) &optional *arg3*)

See also: *example-generic*

14.4 Disable linking to the generic

By default all methods will contain a **See Also** section at the end the links back to the generic that they specialize.

This can be disabled by specifying the `::nolinkgeneric::` option:

```
.. cl:method:: example-generic example-class :test
   :nolinkgeneric:
```

The output will look like.

Method (**example-generic** (*arg1* *EXAMPLE-CLASS*) (*arg2* (eql :TEST)) &optional *arg3*)

(setf (**example-generic** (*arg1* *EXAMPLE-CLASS*) (*arg2* (eql :TEST))) (*new-value* *T*))

This is the first specialized version of `example-generic`.

CLOS CLASS

Documentation for CLOS Classes is configured like

.. **cl:clos-class::** symbol-name

The `:cl:clos-class:` directive emits Common Lisp Object System (CLOS) class documentation:

```
.. cl:clos-class:: example-class
```

The `:noinitargs:` option can be specified to exclude the class' list of `:initarg` initializers that are ordinarily included in the class' signature:

```
.. cl:clos-class:: example-class  
:noinitargs:
```

Note: There is no mechanism or directive to document individual slots at the moment.

15.1 Code

```
(defclass example-class ()  
  ((slot1 :initarg :slot1 :accessor slot1  
          :initform "default"  
          :documentation "the first slot.")  
   (slot2 :initarg :slot2 :accessor slot2  
          :documentation "the second slot."))  
  (:documentation "An example class."))
```

15.2 Output

CLOS class `example-class`

Superclass

[T]

Metaclass

standard-class

Initargs

- `:slot2 - slot2`
- `:slot1 - slot1`

An example class.

```
CLOS slot sphinxcontrib.cldomain.doc::slot2
```

Type

sb-mop:standard-direct-slot-definition

Initarg

:slot2

Reader

```
(sphinxcontrib.cldomain.doc::slot2 (object example-class))
```

Writer

```
(setf (sphinxcontrib.cldomain.doc::slot2 (object example-class)) (new-value T))
```

the second slot.

```
CLOS slot sphinxcontrib.cldomain.doc::slot1
```

Type

sb-mop:standard-direct-slot-definition

Initarg

:slot1

Reader

```
(sphinxcontrib.cldomain.doc::slot1-alt (object example-class))
```

Reader

```
(sphinxcontrib.cldomain.doc::slot1 (object example-class))
```

Writer

```
(setf (sphinxcontrib.cldomain.doc::slot1-alt (object example-class)) (new-value T))
```

Writer

```
(setf (sphinxcontrib.cldomain.doc::slot1 (object example-class)) (new-value T))
```

the first slot.

15.3 Code

```
(define-condition example-error (simple-error)
  ((message
    :initarg :message
    :accessor error-message
    :initform nil
    :documentation "Message indicating what went wrong."))
  (:documentation "An example condition"))
```

15.4 Output

CLOS class `example-error`

Superclass

[`T`]

Metaclass

`sb-pcl::condition-class`

Initargs

- `:message` – *message*

An example condition

CLOS slot `sphinxcontrib.cldomain.doc::message`

Type

`sb-pcl::condition-direct-slot-definition`

Initarg

`:message`

Reader

`(sphinxcontrib.cldomain.doc::error-message (object example-error))`

Writer

`(setf (sphinxcontrib.cldomain.doc::error-message (object example-error)) (new-value T))`

Message indicating what went wrong.

0.18.1 - 2023-06-04

16.1 Minor Fixes

- handle condition slot documentation strings
- rendering of slot's in pdf's

16.2 Documentation

- cleanup index page
- fix pdf and info links
- build latex after rendering

0.18.0 - 2023-03-04

17.1 Features

- add support for documenting CLOS object slots

17.2 Minor Fixes

- raise a runtime error if the package is missing
- remove nospecializers method option
- handle nil arguments correctly
- convert desc_sig_keyword to desc_clparameter
- add &body to potential lambda list keywords
- add missing sphinx dependency
- print the localised name of the object type
- add print-object methods to help with debugging
- add missing :name values for variables and classes
- cleanup conditional
- simplify xref logic
- handle classes with no slots
- cleanup formatting and fix tests

17.3 Documentation

- update documentation and restructure
- update makefile and doc building
- update theme to custom cldomain theme

17.4 Cleanups

- cleanup licensing dates

0.17.1 - 2023-01-18

18.1 Minor Fixes

- remove clear_doc method

0.17.0 - 2023-01-18

19.1 Features

- refactor object backend
- cleanup generic/method linking
- update the generic linking so it's less obtrusive
- self expander support

19.2 Minor Fixes

- cleanup specializer handling

19.3 Documentation

- changelog had the wrong title headings

19.4 Build Tooling

- add example envrc

0.16.2 - 2023-01-08

20.1 Minor Fixes

- add missing roswell file

0.16.1 - 2023-01-08

21.1 Minor Fixes

- add back files missing from dist

22.1 Features

- rename type to class

22.2 Minor Fixes

- fix method arguments in PDF output closes #7
- fix dictionary changed size during iteration
- rename type to class, in reality we are documenting classes, not types.
- bump pants to 2.14.0
- remove list_unused_symbols
- disable more warnings

22.3 Documentation

- add PDF and Info examples to documentation
- update changelog
- update bugtracker and documentation url
- fix sphinx url
- fix reference to pdf

22.4 Tests

- add tests for types, clos classes
- hookup lisp tests

22.5 Build Tooling

- migrate from pants to pyproject for building

22.6 Cleanups

- modernise system definition

0.15.3 - 2022-07-24

- assign *TRACE-OUTPUT* and *DEBUG-IO* to *ERROR-OUTPUT*

0.15.2 - 2022-07-24

- fix don't decode bytes before writing them

0.15.1 - 2022-07-24

- fix decode bytes before writing them

0.15 - 2022-07-23

- stop qualifying lambda list symbols with a package
- fix display of method specializer links #16
- fix labelling of link back to generic

0.14 - 2022-07-10

- convert to unix-opts, because i couldn't get clon to work
- strip packages from symbols if it's the current package, so CL-GIT::BODY would become BODY.
- add whitespace between method arguments so method (full-name (objectreference)) will print as method (full-name (object reference))
- symbols that a appear at the start of newlines are now correctly rendered, this might break CLISP, but will work in SBCL. The bug was introduced by trying to support CLISP, but i think valid rendering trumps multiplatform support for now.

CHAPTER
TWENTYEIGHT

0.13 - 2015-09-06

- updated com.dvlsoft.clon to net.didierverna.clon.

0.12 - 2015-02-24

- fixed argument generation bug.

0.11 - 2014-12-30

- support loading symbol information from multiple packages.

0.10 - 2014-06-12

- added back parentheses to parameter lists.
- added type information to parameter list of methods.
- added links to other methods from a method docstring.
- fixed bug with macro documentation strings.
- added better keyword detection in documentation strings.
- fixed bug where symbols at the end of documentation strings were ignored.

0.9 - 2014-02-10

- fixed problem with version number generation.

0.8 - 2014-02-10

- fixed bug with lisps argument.
- removed dependency on swank.
- remove specializers symbols package if it's the current package.

0.7 - 2013-06-12

- started to make internals more modular.
- print specialisation for methods.
- add links to method specializers.
- added methods to index.

0.6 - 2013-04-22

- added more documentation.
- added better error handling when json fails to parse.
- methods can now pull documentation from their generic.

0.5 - 2013-04-20

- inherit environment when calling subprocesses.
- better handling of symbols in doc strings.

0.4 - 2013-04-19

- fixed some packaging bugs.
- made the data model more tolerant to missing symbols.
- fixed symbol resolving bug.
- added output of unused symbols.

0.3 - 2013-04-16

- cleaned up specializer output.
- fixed bug when rendering specializers that have the form :KEYWORD SYMBOL.
- updated documentation.
- split out package code from lisp program.

0.2 - 2013-04-14

- link between generics and specializers.
- ignore symbols in documentation if they are in the arg list.
- better Quicklisp support.
- handling of symbols that boarder on punctuation.

0.1 - UNRELEASED

- initial prototype

Symbols

example-variable (*Lisp Variable*), 21
 example-variable-2 (*Lisp Variable*), 22

C

cl:clos-class (*directive*), 31
 cl:clos-class (*role*), 15
 cl:clos-slot (*role*), 15
 cl:function (*directive*), 23
 cl:function (*role*), 15
 cl:generic (*directive*), 27
 cl:generic (*role*), 15
 cl:macro (*directive*), 25
 cl:macro (*role*), 15
 cl:method (*directive*), 29
 cl:method (*role*), 15
 cl:package (*directive*), 19
 cl:symbol (*role*), 15
 cl:variable (*directive*), 21
 cl:variable (*role*), 15

E

example-class (*Lisp CLOS class*), 31
 example-error (*Lisp CLOS class*), 33
 example-function (*Lisp Function*), 23
 example-generic ((sphinxcontrib.cldomain.doc:example-class
 (eql keyword:test))) (*Lisp Method*), 29,
 30
 example-generic ((sphinxcontrib.cldomain.doc:example-class
 (eql keyword:test1))) (*Lisp Method*), 30
 example-generic (*Lisp Generic*), 27
 example-macro (*Lisp Macro*), 13, 25

M

message (*Lisp CLOS slot*), 33

S

slot1 (*Lisp CLOS slot*), 32
 slot2 (*Lisp CLOS slot*), 32