

Erudite Manual

Introduction

by *Mariano Montone* (*marianomontone@gmail.com*)

This is a manual and reference of Erudite Documentation System.

Erudite documents are edited using a wiki-like markup with capabilities for linking and embedding Smalltalk code.

Main documentation is realized via books, a collection of documents organized in sections.

Books are read and edited via the reader and editor morphic applications.

Book reader and editor

To read a book, send open to a Book:

```
EruditeBook eruditeManual open
```

The book reader has editing capabilities, but the editor provides live preview and instant updates for editing.

To edit a book, send edit to a Book:

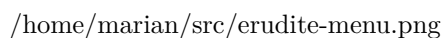
```
EruditeBook eruditeManual edit
```

Books are stored as methods in the class side of EruditeBook class and are marked with a pragma

```
<book: 'title'>
```

. To *remove* a book, just delete the method. To specify the *package* the book belongs to, just recategorize the method (**PackageName*).

Most of the actions are invoked from the context menu of the book *explorer* on the left of the editor:

/home/marian/src/erudite-menu.png

A new book can be created from the WorldMenu *Erudite...->New...* menu. Then enter a title and you are ready to go. Always remember to save your book before quitting the editing application. Books are saved as methods on the class side of *EruditeBook*.

Book editing can be invoked via the edit message on the book, or the WorldMenu *Erudite...->Edit...* menu.

First of all, a book *section* needs to be selected or created. Use the editor menu, *add section*. Then you can start editing the section content on the text panel on the right. Markup syntax is described in Syntax reference.

When you are done, save the book, from the explorer menu *save*.

List of menu actions

add section. Create a new section. If you want to create a section toplevel, then unselect all sections on the left panel before invoking this action. Otherwise, select the section you want the new section to be child of.

rename section. Rename the selected section.

remove section. Remove the selected section.

move up. Move the selected section up (before its left sibling).

move down. Move the selected section down (after its right sibling).

open. Open an *existing book* with the book reader.

save. Store the book being edited on a method on the class side of *EruditeBook*.

edit. Open the *current book* for editing with the book editor.

save as... Save the current book as a new book with a different title.

refresh. Invalidates rendering caches of the book reader/editor. Book documents are rerendered after that.

reload. Forgets the current changes and load the book from the storage.

reset variables. The book reader and editor manages Smalltalk variables the same as a Smalltalk Workspace does (see *#Syntax* test for examples). This sets already assigned variables to nil.

explore book. Open the current book with a Smalltalk explorer.

toggle view sources. Makes the text editing panel visible/invisible.

toggle live editing. By default, in the book editor, the markup renderer panel on the left is updated whenever you edit the markup sources on the right panel. The problem with this is that this can be expensive as the content grows and too much parsing and rendering is being done for each change. Toggling live editing off, prevents the left panel from being updated on each change, and update only

when markup changes are accepted (alt-S). *Tip:* toggle live editing off for big documents.

Syntax reference

Headings start with two or more exclamation marks:

```
!! heading
!!! subheading
!!!! subsubheading
```

Text between double *:

```
**bold text**
```

Text between double slashes:

```
//italics//
```

To prevent text formatting, enclose the text in triple back quotes.

Links have the following syntax:

```
{target::linkType|label}[options]
```

The parser and renderers (formatters) are designed to handle different types of links. The system is extensible. Links for handling Smalltalk code and references are provided in the basic package:

Links to sections in the same book:

Syntax:

```
{sectionName::section|optional label}
```

Example:

```
{Introduction::section}
```

```
==> Introduction
```

Links to sections in another book:

Syntax:

```
{bookName#sectionName::section|optional label}
```

Example:

```
{EruditeManual#Syntax test::section}
```

```
==> Erudite Manual#Syntax test
```

Syntax:

```
{selectorName::selector}
```

Example:

```
{at:put: ::selector}
```

=> at:put:

Syntax:

```
{className::class|optional label}[options]
```

Example:

```
{EruditeBook::class}
```

=> EruditeBook

Options:

- *embed*: embed the class source code.

Example:

```
{EruditeBook::class}[embed]
```

=>

EruditeBook

Syntax:

```
{Class>>selector ::method|optional label}[options]
```

Example:

```
{EruditeBook>>initialize: ::method}
```

=> EruditeBook»initialize:

Options:

- *embed*: embed the method source code.

Example:

```
{EruditeBook>>initialize: ::method}[embed]
```

=>

EruditeBook»initialize:

Syntax:

```
{HTTP url address ::url|optional label}
```

Example:

```
{http://cuis-smalltalk.org::url|Cuis Smalltalk}
```

=> Cuis Smalltalk

Images from files can be inserted via links:

```
{filePath ::image}
```

Images are serialized to the Erudite Document; there's no need to ship the images separately with the books.

Example:

```
/home/marian/Escritorio/smalltalk-logo.png
```

Code is enclosed between triple brackets, like:

```
[[[code]]]
```

. Only Smalltalk code is supported at the moment.

Syntax:

```
[[[code]]] action
```

. Where action is optional.

Actions:

- *doIt* : Renders a link besides the code with which to evaluate the code. - *exploreIt* : Renders a link besides the code with which to explore the result of code evaluation. - *inspectIt* : Renders a link besides the code with which to inspect the result of code evaluation. - *printIt* : Renders a link besides the code with which to print the result of code evaluation to the Transcript. - *printItHere* : Renders the result of code evaluation in place. - *embedIt* : The code is not shown. The result is printed to the document in place. - *doItWithButton* : Does not render the code. Instead, it makes a link with *label* that evaluates the code.

Syntax:

```
[[[code]]] doItWithButton: label.
```

IMPORTANT: label ends in a period. - *exploreItWithButton* : Same as above, but explore. - *inspectItWithButton* : Same as above, but inspect.

Examples:

Plain Smalltalk code:

```
Dictionary new at: #foo put: 'bar'
```

```
.
```

```
Dictionary new at: #foo put: 'bar'; yourself
```

```
.
```

```
Dictionary new at: #foo put: 'bar'; yourself
```

```
2 + 4 * 5
```

ImageMorph new

Literate Programming

Literate programming is a style of programming invented by Donald Knuth, where the main idea is that a program's source code is made primarily to be read and understood by other people, and secondarily to be executed by the computer.

This frees the programmer from the structure of a program imposed by the computer and means that the programmer can develop programs in the order of the flow of their thoughts.

A Literate program consists of explanation of the code in a natural language such as English, interspersed with snippets of code to be executed. This means that Literate programs are very easy to understand and share, as all the code is well explained.

Donald Knuth. "Literate Programming (1984)" in Literate Programming. CSLI, 1992, pg. 99.:

"I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: "Literate Programming."

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other."

According to Knuth, literate programming provides higher-quality programs, since it forces programmers to explicitly state the thoughts behind the program, making poorly thought-out design decisions more obvious. Knuth also claims that literate programming provides a first-rate documentation system, which is not an add-on, but is grown naturally in the process of exposition of one's thoughts during a program's creation. The resulting documentation allows the author to restart his own thought processes at any later time, and allows other programmers to understand the construction of the program more easily. This differs from traditional documentation, in which a programmer is presented with

source code that follows a compiler-imposed order, and must decipher the thought process behind the program from the code and its associated comments. The meta-language capabilities of literate programming are also claimed to facilitate thinking, giving a higher "bird's eye view" of the code and increasing the number of concepts the mind can successfully retain and process. Applicability of the concept to programming on a large scale, that of commercial-grade programs, is proven by an edition of TeX code as a literate program.

Literate programming is very often misunderstood to refer only to formatted documentation produced from a common file with both source code and comments -which is properly called documentation generation-; or to voluminous commentaries included with code. This is backwards: well-documented code or documentation extracted from code follows the structure of the code, with documentation embedded in the code; in literate programming code is embedded in documentation, with the code following the structure of the documentation.

This misconception has led to claims that comment-extraction tools, such as the Perl Plain Old Documentation or Java Javadoc systems, are "literate programming tools". However, because these tools do not implement the "web of abstract concepts" hiding behind the system of natural-language macros, or provide an ability to change the order of the source code from a machine-imposed sequence to one convenient to the human mind, they cannot properly be called literate programming tools in the sense intended by Knuth.

Literate Programming can be realized by the creation of Erudite books. In particular, via code embedding in its different forms. Contrary to other LP systems, there's no need for a tangling and weaving phase; the document source is kept separately from the source code, Smalltalk code is referenced and embedded from the documentation. That means that in spite of documentation and code being separate, the referenced code is always up to date. Finally, the possibility of evaluating Smalltalk code from the documentation makes Erudite very unique compared to the other LP systems.

Syntax test

bold

italics

unformatted ****unformatted**** //unformatted//

Smalltalk at: #Object

Smalltalk inspect

Smalltalk

Smalltalk

```
Smalltalk
2 * 3 + 5 / 34
ImageMorph new
ImageMorph new
z _ true
z
x := 22
x
Literate Programming.Read the introduction.
Erudite Manual#Syntax test
Object
labelled class
Object
Dictionary»at:
labelled method
Dictionary»at:
print
print
print
at:put:
http://www.cuis-smalltalk.org
Cuis Smalltalk
/home/marian/Escritorio/smalltalk-logo.png
```

Implementation

This section describes how Erudite is implemented. It is a simple example of how literate-programming can be realized with Erudite.

The model in Erudite consists of `EruditeDocument` and `EruditeBook` classes.

Erudite documents are piece of text meant to be formatted. Their content is Erudite source code.

Erudite books are formed of sections. Each section has a document and other subsections as children. Each subsection also has a document and other subsections as children, and so on, and so on.

The Erudite source code parser is implemented as a PEG parser using `PetitParser`. `EruditeMarkupGrammar` describes the grammar abstractly and `EruditeMarkupParser` is the actual parser (returns parse nodes). You can test the parser invoking the `parse:` message on it:

```
EruditeMarkupParser parse: '!!! A heading'
```

You can also test particular grammar rules:

```
EruditeMarkupParser new heading2 parse: '!!! A heading'
```

The Smalltalk parser parses Smalltalk comments into Erudite documents. This parser tries to interpret the different type of references in Smalltalk comments, like classes methods and selectors. This parser is used by the Erudite tools extensions.

There are two formatters implemented at the moment. A Morphic formatter, used by the Morphic book reader and editor. And a Latex formatter with which you can generate PDF documents.

The Morphic formatter is implemented in `MorphicEruditeDocRenderer`. It is implemented as a visitor that outputs attributed Morphic Text on each visit to a parse node.

Here is an example of how it is used:

```
|erudite|
erudite _ SmalltalkEruditeParser parse: 'This is an Object. Look at Object>>at:
Events are triggered via #triggerEvent:'.
```

```
(MorphicEruditeDocRenderer on: erudite) render edit
```

The Latex formatter is implemented in `LatexEruditeDocRenderer`.

For example, this is how to generate a Latex/PDF for Erudite manual. First generate the

```
.tex
```

file:

```
(LatexEruditeDocRenderer on: EruditeBook eruditeManual)
texFilePath: '/tmp/EruditeManual.tex';
render
```

Then process it with

```
pdflatex
```

command:

```
pdflatex -shell-escape /tmp/EruditeManual.tex
```

Erudite document links are parsed in a generic way. Their syntax is like this:

```
{target::linkClass|optionalLabel}
```

. That linkClass is then used to match a valid renderer. Links renderers are subclasses of DocLinkRenderer and are matched looking for the class named <linkClass>DocLinkRenderer (DocLinkRenderer classes with linkClass as prefix), and invoked via render: aDocLink in: aDocument on: aStream message.

Books are stored as serialized Smalltalk objects inside methods at the class side of EruditeBook:

```
EruditeBook%3E%3Estore
```

```
EruditeBook»storeOnMethod:
```

As you can see, it is done via the storeOn: method (it is indirectly call from WriteStream»store: method).

Invoking the serialized methods returns the book:

```
EruditeBook cuisManual
```

Methods with Books are tagged with a *book* pragma, that is used later for listing available books:

```
EruditeBook class»booksList
```

Extensions

Extensions to Smalltalk tools are shipped separately.

To install:

```
Feature require: 'EruditeToolsExtensions'
```

Markup and formatting in class comments

This is implemented.

Markup and formatting of code comments

TODO. Not implemented yet.

Extras

A PetitParser tutorial is included in the Erudite distribution.

To install:

Feature require: 'PetitParserTutorial'

Also, a book on Morhic, the Cuis UI system:

To install:

Feature require: 'MorphicBook'