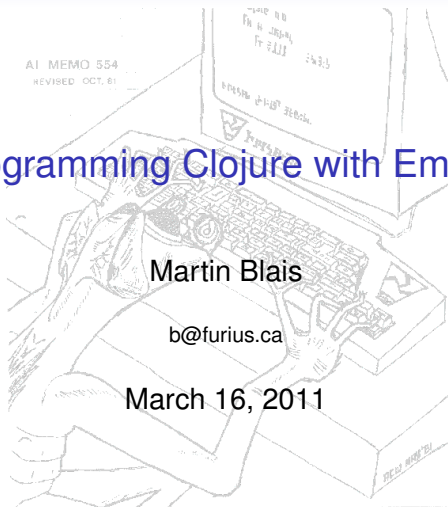


Programming Clojure with Emacs

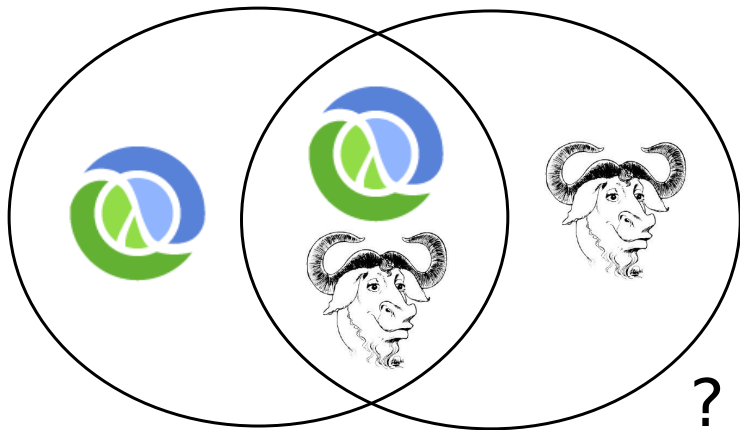
Martin Blais

b@furius.ca

March 16, 2011



Overview: Audience



Outline

Emacs Concepts

Fun & Tricks

The REPL Experience

How It Works

Setting Up

Using SLIME

clojure-test

CDT

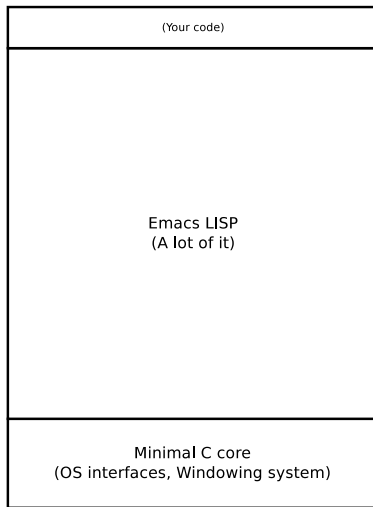
Appendix

Emacs is not an Editor

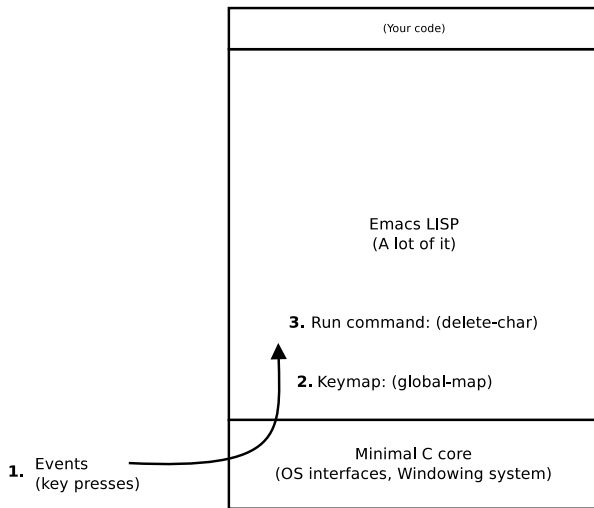
```
~/src/emacs-23.2$ sloccount
```

lisp:	964459	(78.48%)
ansic:	247701	(20.16%)
objc:	9978	(0.81%)
sh:	4407	(0.36%)
perl:	1254	(0.10%)
cs:	772	(0.06%)
python:	331	(0.03%)
xml:	43	(0.00%)
csh:	8	(0.00%)
sed:	4	(0.00%)

Emacs is not an Editor



Emacs is not an Editor



Elisp, “Emacs LISP”

- Is a LISP-2
- Dynamic scoping is everywhere by default
- No tail-call optimization
- Has a `cl` module which implements some Common-LISP
- Its REPL *is* Emacs itself, that’s the idea!

Most of Emacs is implemented in Elisp—even some low-level functionality— and you can redefine it all. That’s why there are so many extensions.

Elisp: Commands vs. Functions

A *command* is a function with an **(interactive)** call.

```
(defun kill-isearch-match ()  
  "Kill the current isearch and continue searching."  
  
  (kill-region isearch-other-end (point)))
```

This is how context is passed to functions:

```
(defun remove-text-properties-in-region (beg end)  
  
  (set-text-properties beg end nil))
```

Elisp: Commands vs. Functions

A *command* is a function with an **(interactive)** call.

```
(defun kill-isearch-match ()  
  "Kill the current isearch and continue searching."  
  (interactive)  
  (kill-region isearch-other-end (point)))
```

This is how context is passed to functions:

```
(defun remove-text-properties-in-region (beg end)  
  (interactive "r")  
  (set-text-properties beg end nil))
```

Screen Real Estate

The screenshot displays the Eclipse IDE interface for a Rails application. The main editor shows the `AccountController` class with the following code:

```

1 class AccountController < ApplicationController
2
3   def register
4     puts request.env["HTTP_REFERER"]
5     puts "### Session : #{@session[:user]}"
6     @account = Account.new
7   end
8
9   def create
10    @account = Account.new(params[:account])
11    if @account.save
12      flash[:notice] = "You've been successfully registered."
13      session[:user] = @account
14      redirect_to :controller => 'home'
15    else
16      render :action => 'register'
17    end
18  end
19
20  def login
21    @account = Account.authenticate(params[:account][:login], params[:account]
22    if @account
23      session[:user] = @account
24      go_to = session[:destination]
25      session[:destination] = nil
26      redirect_to go_to || '/home'
27    else
28      flash[:notice] = "Your login or password were incorrect, please retry."
29      redirect_to :action => 'register'
30    end
31  end
32 end
  
```

The file explorer on the left shows the project structure, including `controllers` and `views` directories. The outline view on the right shows the class structure, including `AccountController` and its methods. The bottom panel shows the `Class: Array` reference, which includes a description of arrays and a list of methods.

Class: Array

Arrays are ordered, integer-indexed collections of any object. Array indexing starts at 0, as in C or Java. A negative index is assumed to be relative to the end of the array—that is, an index of -1 indicates the last element of the array, -2 is the next to last element in the array, and so on.

Includes:

Enumerable(all?, any?, collect, detect, each_cons, each_slice, each_with_index, entries, enum_cons, enum_slice, enum_with_index, find, find_all, grep, include?, inject, map, max, member?, min, partition, reject, select, sort, sort_by, to_a, to_set, zip)

Class methods:

`[], new`

Screen Real Estate

The screenshot displays the Eclipse IDE interface for a Rails application. The main editor shows the `AccountController` class with the following code:

```

class AccountController < ApplicationController
  def register
    puts request.env["HTTP_REFERER"]
    puts "### Session: #{@session[:user]}"
    @account = Account.new
  end

  def create
    @account = Account.new(params[:account])
    if @account.save
      flash[:notice] = "You've been successfully registered."
      session[:user] = @account
      redirect_to :controller => 'home'
    else
      render :action => 'register'
    end
  end

  def login
    @account = Account.authenticate(params[:account][:login], params[:account][:password])
    if @account
      session[:user] = @account
      go_to = session[:destination]
      session[:destination] = nil
      redirect_to go_to || '/home'
    else
      flash[:notice] = "Your login or password were incorrect, please retry."
      redirect_to :action => 'register'
    end
  end
end

```

The IDE also shows a sidebar with the project structure, including controllers, helpers, models, and views. A tooltip or popup window is visible over the `session[:user] = @account` line, containing the text: "Multiple markers at this line - TODO: complete that. - TODO: complete that."

At the bottom of the IDE, the "Rails Plugins" panel is open, showing the "Array" class reference page. The page content includes:

Class: Array

Arrays are ordered, integer-indexed collections of any object. Array indexing starts at 0, as in C or Java. A negative index is assumed to be relative to the end of the array—that is, an index of -1 indicates the last element of the array, -2 is the next to last element in the array and so on.

Indexing:

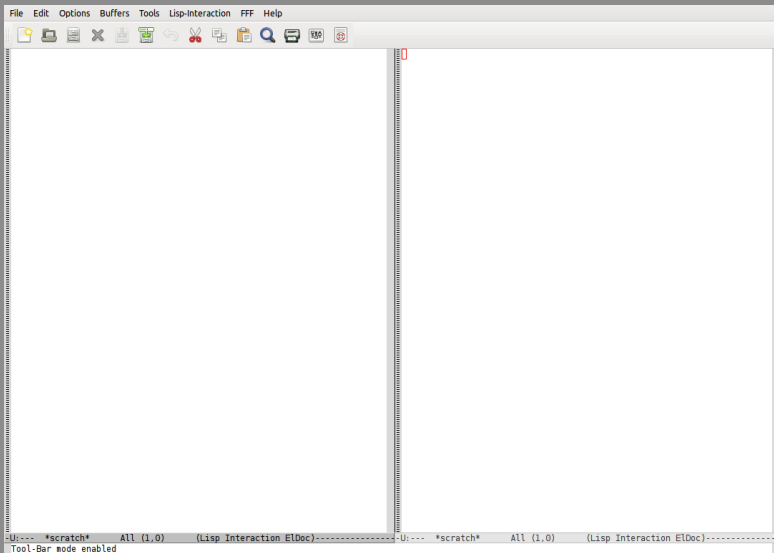
Enumerates over, any?, collect, collect_with_cons, each_slice, each_with_index, entries, inject, inject_with, map_slice, map_with_index, find, find_all, grep, include?, inject, map, map_slice, map_with_index, min, partition, reject, select, sort, sort_by, to_a, to_set, zip)

Class methods:

`[], new`

A large, semi-transparent watermark "23%" is overlaid on the bottom half of the image.

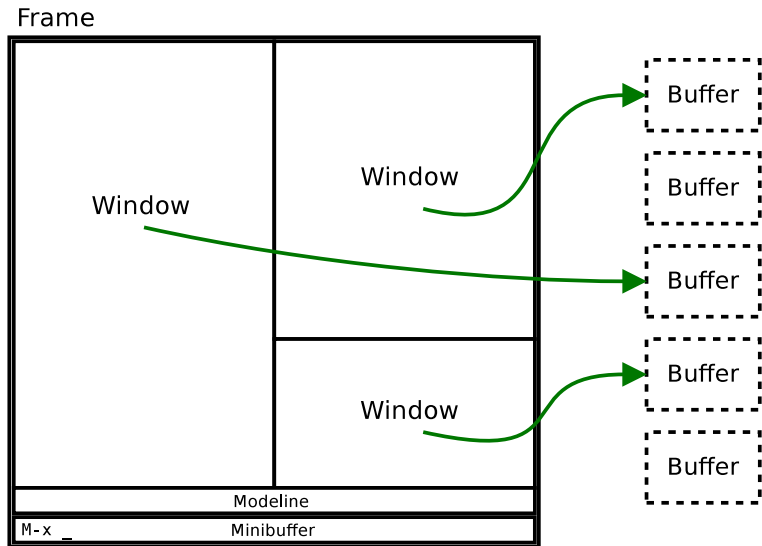
Screen Real Estate



Screen Real Estate



Emacs Anatomy



Emacs Anatomy

Buffer

```

(beginning-of-buffer)
  (clojure.core)

  (def unquote)
  (def unquote-splicing)

  (def
    ^ { :arglists '(($ items))
      :doc "Creates a new list containing the items."
      :added "1.0" }
    (persistent-list creator))

(mark)

(point)
  (added "1.0")
  (in (fn* fn ([form &opts & decl]
             (.withMeta "clojure.lang.IObj" (cons (fn* decl)
                                                  (.meta "clojure.lang.IMeta &form))))))

  (def
    ^ { :arglists '([coll])
      :doc "Returns the first item in the collection. Calls seq on its
            argument. If call is nil, returns nil."
      :added "1.0" }
    (first (fn first ([coll] (. clojure.lang.RT (first coll))))))

  (def
    ^ { :arglists '([coll])
      :tag clojure.lang.ISeq
      :doc "Returns a seq of the items after the first. Calls seq on its
            argument. If there are no more items, returns nil."
      :added "1.0" }
    (next (fn next ([x] (. clojure.lang.RT (next x))))))

  (def
    ^ { :arglists '([coll])
      :tag clojure.lang.ISeq
      :doc "Returns a possibly empty seq of the items after the first. Calls seq on its
            argument."
      :added "1.0" }
    )

(end-of-buffer)

```

filename
(maybe)

Emacs Modes

Emacs buffers are in a combination of modes:

- **Major mode:** usually depends on file type
C-c ... keys
Java, C, Python Clojure, HTML, Text, Fundamental
- **Minor mode:** optional features, extra flavour
Abbrev, Auto Fill, Auto Save, Electric,
Flyspell, Font-lock, Outline

```
\end{frame}
\begin{frame}[fragile]
-1:--- emaclj.tex      16% (171,2)  (LaTeX Fill)-----
Wrote /home/blais/emaclj/emaclj.tex
```

Finding Help on a Function

Keybinding

M-z

Function (or Variable)

zap-to-char

Info (Manual)

12.3 Other Kill Commands

=====

[...]

``M-z CHAR'`

Kill through the next occurrence of CHAR (`'zap-to-char'`).

[...]

The command ``M-z'` (`'zap-to-char'`) combines killing with searching: it reads a character and kills from point up to (and including) the next occurrence of that character in the buffer. A numeric argument acts as a repeat count; a negative argument means to search backward and kill text before point.

Doc. String

(zap-to-char ARG CHAR)

A version of zap-to-char that does not include the char we're searching for. This version deletes the char if it is right next to the cursor when the function is invoked.

Finding Help on a Function

C-h b



Keybinding

M-z

Function (or Variable)

zap-to-char

Info (Manual)

C-h i



12.3 Other Kill Commands

=====

[...]

`M-z CHAR'

Kill through the next occurrence of CHAR ('zap-to-char').

[...]

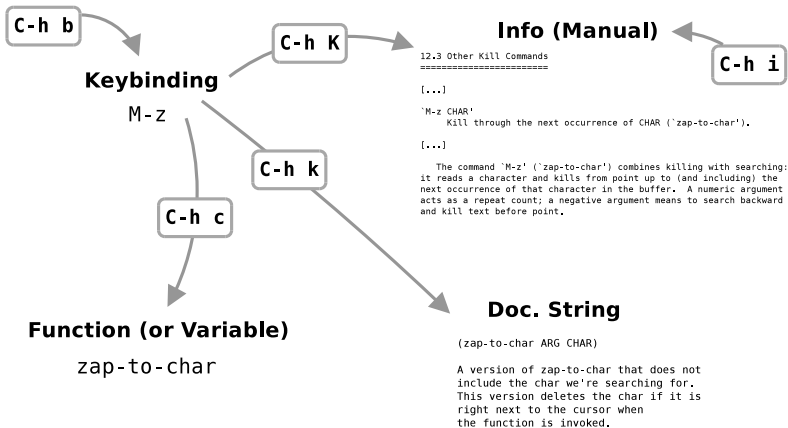
The command `M-z' ('zap-to-char') combines killing with searching: it reads a character and kills from point up to (and including) the next occurrence of that character in the buffer. A numeric argument acts as a repeat count; a negative argument means to search backward and kill text before point.

Doc. String

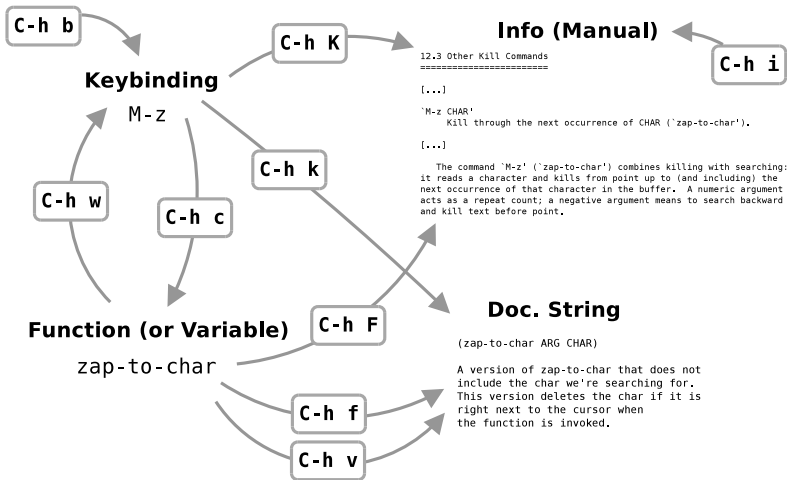
(zap-to-char ARG CHAR)

A version of zap-to-char that does not include the char we're searching for. This version deletes the char if it is right next to the cursor when the function is invoked.

Finding Help on a Function



Finding Help on a Function



Finding Help on a Function

And of course ... the source code itself:

```
(defun zap-to-char (arg char)
  "Kill up to and including ARGth occurrence of CHAR.
Case is ignored if 'case-fold-search' is non-nil in the current buffer.
Goes backward if ARG is negative; error if CHAR not found."
  (interactive "p\ncZap to char: ")
  ;; Avoid "obsolete" warnings for translation-table-for-input.
  (with-no-warnings
    (if (char-table-p translation-table-for-input)
        (setq char (or (aref translation-table-for-input char)
                       char))))
  (kill-region (point) (progn
                        (search-forward (char-to-string char)
                                       nil nil arg)
                        (point))))
```

Finding Help: Info

- The manual: Use `C-h i`
 - `texinfo` is your friend, learn to use it! (*Demo*)
 - `h`: how to browse info
 - `d`: go to the directory (the top node)
 - `u`, `TAB`, `RET`: move between nodes
 - Use interactive-search in the index! (`C-s`)
- Emacs vs. Elisp documentation
 - *“The Emacs Editor”*
 - *“An Introduction to Programming in Emacs Lisp”*
 - *“The Emacs Lisp Reference Manual.”*
(watch out: this installs separately)
- `/usr/share/emacs/23.1/lisp/*`
- Some distributions don't install Elisp source
(`.el` vs. `.elc`)

About Key Bindings

(Demo) General usage

The following commands require special mention:

- C-g : abort current operation
- C-] : abort recursive operation
- C-u *N* : repeat the next operation *N* times

A Few Tricks...

- **Use** emacsclient
- swap-strings
- dubious-paragraphs
- dabbrev
- **Region undo**
- align-regexp
- speedbar
- kill-isearch-match
- param-cycle-indentation
- filecache + **auto-adding files into filecache (+ hook)**
- iswitchb
- all

(Demo)

Registers

Stores “stuff” in named registers (by letter)

(Demo)

C-x r

- SPC - Positions
- s - Text
- r - Rectangles
- w - Window Configuration

C-x r i R - insert

C-x r j R - jump

Closing Balanced Expressions

- **paredit-mode.el**
Minor mode that auto-inserts matching parentheses.
- **close-matching.el**
“Smarter parenthesis” insertion.

```
(def { :foo [1, 2, 3]
      :bar [4, 5, 6] })
```

(Demo)

Closing Balanced Expressions

- **paredit-mode.el**
Minor mode that auto-inserts matching parentheses.
- **close-matching.el**
“Smarter parenthesis” insertion.

```
(def { :foo [1, 2, 3]
      :bar [4, 5, 6] })
```

(Demo)

Closing Balanced Expressions

- **paredit-mode.el**
Minor mode that auto-inserts matching parentheses.
- **close-matching.el**
“Smarter parenthesis” insertion.

```
(def { :foo [1, 2, 3]
      :bar [4, 5, 6] })
```

(Demo)

Closing Balanced Expressions

- **paredit-mode.el**
Minor mode that auto-inserts matching parentheses.
- **close-matching.el**
“Smarter parenthesis” insertion.

```
(def { :foo [1, 2, 3]
      :bar [4, 5, 6] })
```

(Demo)

LISP Editing Modes

Emacs-Lisp mode (& Lisp Interaction mode)

- Evaluate expressions directly into the running Emacs VM
- A `*Messages*` buffer captures the output



Superior Lisp Interaction Mode for Emacs

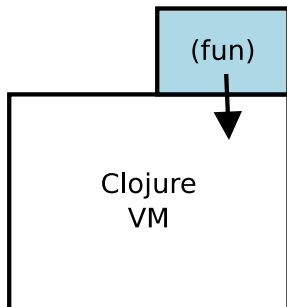
- Communicates expressions with a subprocess
- Provides a REPL in `*slime-repl clojure*`

How Emacs Interacts with Clojure

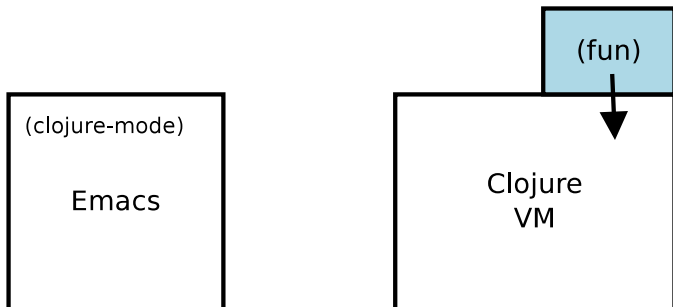


Clojure
VM

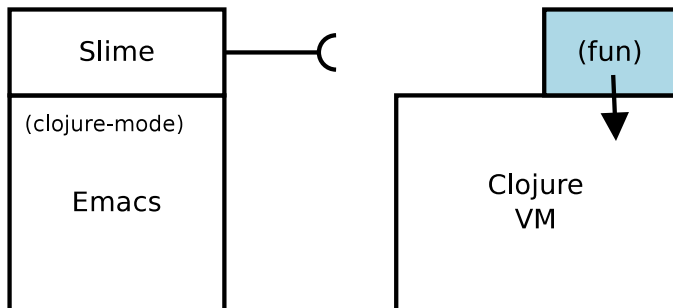
How Emacs Interacts with Clojure



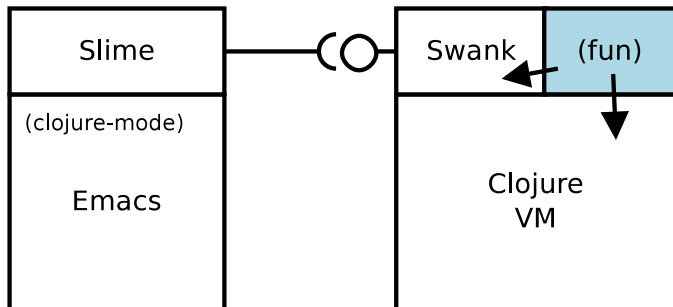
How Emacs Interacts with Clojure



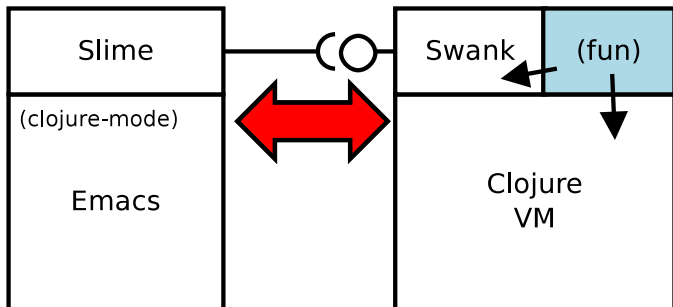
How Emacs Interacts with Clojure



How Emacs Interacts with Clojure



How Emacs Interacts with Clojure



Setting Up Emacs for Clojure/Slime

Setup Clojure:

1. Download swank-clojure

`https://github.com/technomancy/swank-clojure.git`

2. Start a Clojure VM with a Swank listener

Setup Emacs:

1. Download slime

`https://github.com/technomancy/slime.git`

2. Download clojure-mode

`https://github.com/technomancy/clojure-mode.git`

3. Load Emacs and configure

4. Connect to the Clojure VM

¡ACHTUNG! Use the Github forks from technomancy!

Setting Up Emacs for Clojure/Slime

Setup Clojure:

1. Download swank-clojure

`https://github.com/technomancy/swank-clojure.git`

2. Start a Clojure VM with a Swank listener

Setup Emacs:

1. Download slime

`https://github.com/technomancy/slime.git`

2. Download clojure-mode

`https://github.com/technomancy/clojure-mode.git`

3. Load Emacs and configure

4. Connect to the Clojure VM

¡ACHTUNG! Use the Github forks from technomancy!

Setting Up Emacs for Clojure/Slime

Setup Clojure:

1. Download swank-clojure

`https://github.com/technomancy/swank-clojure.git`

2. Start a Clojure VM with a Swank listener

Setup Emacs:

1. Download slime

`https://github.com/technomancy/slime.git`

2. Download clojure-mode

`https://github.com/technomancy/clojure-mode.git`

3. Load Emacs and configure

4. Connect to the Clojure VM

!ACHTUNG! Use the Github forks from technomancy!

Start Clojure with Swank: Manual Startup

Start Clojure manually:

```
java -cp clojure.jar:swank-clojure.jar  
      clojure.main -e "(use 'swank.swank) (start-repl)"
```

Using Leiningen:

```
cd project ; lein swank
```

Start Clojure with Swank: Using `streamlined`

"I told you I would come back, even if I am a bit streamlined."

```
tangerine:~/jars$ streamlined clojure/clojure-1.2
```

```
Classpath:
```

```
/home/blais/jars/clojure-1.2/clojure.jar
```

```
/home/blais/jars/clojure-1.2/clojure-contrib-1.2.0.jar
```

```
/home/blais/jars/clojure-1.2/swank-clojure-1.3.0-SNAPSHOT.jar
```

```
user=> Connection opened on local port 4005
```

```
#<ServerSocket ServerSocket[addr=localhost/127.0.0.1,port=0,localport=
```

- It is transparent about what it does (use `-v`)
- It is fast
- Sets the `*classpath*` (like Leiningen)
- Supports options for Clojure Debugging Toolkit
- Resolves collisions between versions

Load Emacs and Configure SLIME

Put this in your `.emacs`:

```
(add-to-list 'load-path "/path/to/slime"))

;; Load the slime support
(require 'slime)

;; Remove annoying warning
(eval-after-load 'slime
  '(setq slime-protocol-version 'ignore))

;; Load REPL support in slime
(slime-setup '(slime-repl))
```

Load Emacs and Configure clojure-mode

Put this in your `.emacs`:

```
(add-to-list 'load-path "/path/to/clojure-mode")  
  
;; Load editing support for Clojure  
(require 'clojure-mode)  
  
;; Load test support for Clojure (needs Slime).  
(require 'clojure-test-mode)
```

Connect Emacs to the VM

M-x slime-connect

That's it!

Notes:

- Doing it manually pays off in the long run ... keep it simple
- You do not need `swank-clojure.el`
- You can create a little Emacs function to connect with default parameters

SLIME Commands: Symbol Completion

- **C-c TAB** `slime-complete-symbol`
Complete symbol near cursor (from code)
- **C-c M-i** `slime-fuzzy-complete-symbol`
Complete with scoring and show all matches
- **M-/** `dabbrev-expand`
Complete from other buffers

Also see function template in minibuffer after typing `SPC`

SLIME Commands: Accessing Documentation

C-c C-d ... `slime-doc-map`

All documentation functions

- ... **d** `slime-describe-symbol`
Lookup docstring for symbol at point
- ... **a** `slime-apropos`
Search for regexp in all symbols
- ... **p** `slime-apropos-package`
Get the list of function of a package

SLIME Commands: Finding Source Code

- **M-.** `slime-edit-definition`
Go to the source
- **M-,** `slime-pop-find-definition-stack`
Pop location to the last source

SLIME Commands: Evaluating Expressions

Same bindings as for other LISP editing modes.

- **C-x C-e / C-M-x** `slime-eval-defun`
Evaluate last expression (or containing)
- **C-c C-p** `slime-pprint-eval-last-expression`
Pretty printed to a buffer
- **C-c C-r** `slime-eval-region`
Evaluate region
- **C-c :, C-c C-e** `slime-interactive-eval`
Asks for a string, evaluates it
- **C-c C-k** `slime-interactive-eval`
Asks for a string, evaluates it

SLIME Commands: Evaluating Expressions

Where Does My Output Go?

- Return values : minibuffer and `*Messages*` buffer
 - Stdout of main thread : `*slime-repl*` buffer
 - Stdout of other threads : to the VM console
-
- **C-c C-z** `slime-switch-to-output-buffer`
Displays the REPL buffer

SLDB Stack Trace

```
(let [c 1, d 2]
      (defn foo [b c] (swank.core/break) d))
(foo "foo" "bar"))
```

BREAK:

```
[Thrown class java.lang.Exception]
```

Restarts:

- 0: [QUIT] Quit to the SLIME top level
- 1: [CONTINUE] Continue from breakpoint

Backtrace:

```
0: demo_inspector$eval1954$foo__1955.invoke(NO_SOURCE_FILE:1)
  Locals:
    b = "foo"
    c = "bar"
    d = 2
    foo = #<demo_inspector$eval1954$foo__1955 demo_inspector$eval1
1: demo_inspector$eval2006.invoke(NO_SOURCE_FILE:1)
2: clojure.lang.Compiler.eval(Compiler.java:5424)
```

SLIME Commands: Expanding Macros

- **C-c C-m** `slime-macroexpand-1`
Expand macro at point, one level
- **C-c M-m** `slime-macroexpand-all`
Expand macro at point, recursively

No evaluation is performed—only macro expansion.

SLIME Commands: Misc.

Limited support is offered for fetching VM info:

- **C-c C-x c** `slime-list-connections`
List the live connections to VMs
- **C-c C-x t** `slime-list-threads`
List the threads in the VM

How to stop an infinite loop:

- **C-x C-b** `slime-interrupt`
Interrupt the subprocess (**does not always work**)

SLIME Commands: Misc.

Enumerate an object's fields:

- **C-c S-i** `slime-inspect`
Inspect the fields of something

Some support exists for tracing functions:

- **C-c C-t** `slime-toggle-trace-fdefinition`
Toggle tracing to stdout on a function

You can list who calls a function:

- **C-c C-w c** `slime-who-calls`
Show all the known callers

SLIME Commands: Unsupported

Commands that aren't supported by swank/Clojure:

- “Real” compilation
- Disassembling functions
- Undefining functions
- Most of the cross-referencing features
- The HyperSpec (should we link to clojuredocs.org?)

Using `clojure-test-mode`

Displays test results in an overlay:

```
(deftest test1
  (is (= 5 (+ 2 2)))
)
```

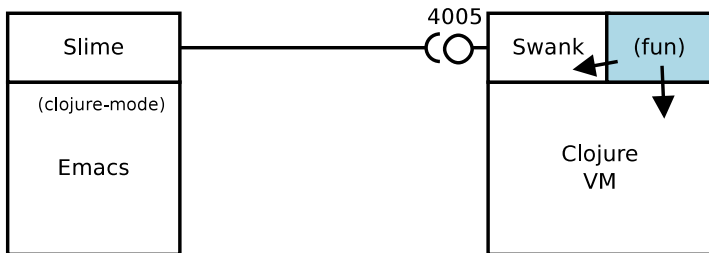
Provides three commands:

- `C-c` , runs the tests
- `C-c` ' shows the result detail
- `C-c` k clears the overlay

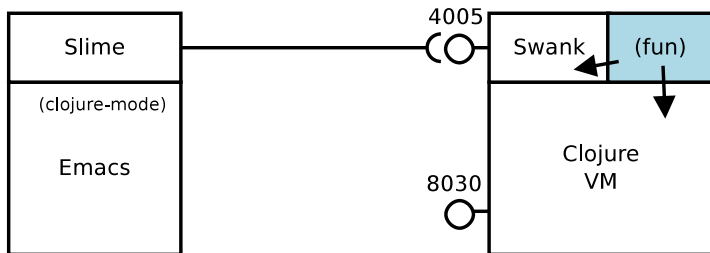
Error reports look like this:

```
FAIL in (test1) (test-mode-demo.clj:12)
expected: (= 5 (+ 2 2))
actual: (not (= 5 4))
```

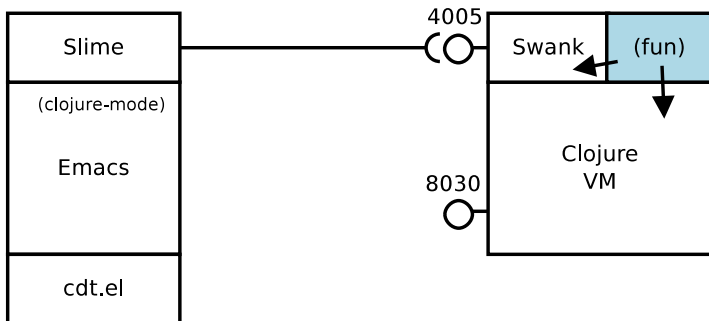
Clojure Debugging Toolkit



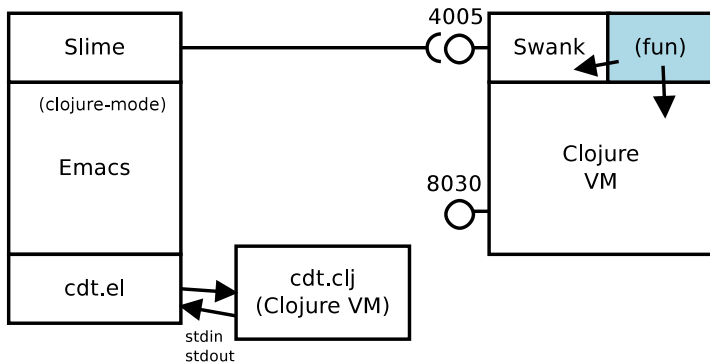
Clojure Debugging Toolkit



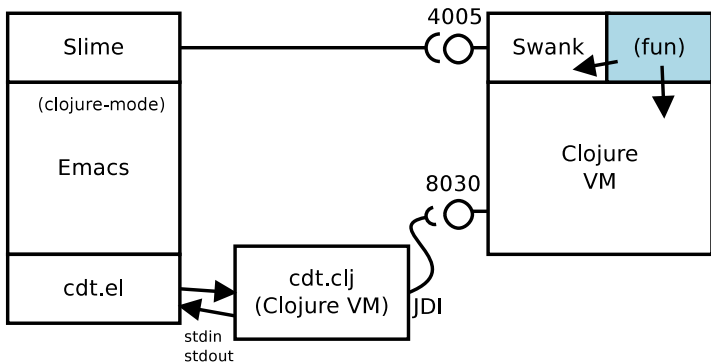
Clojure Debugging Toolkit



Clojure Debugging Toolkit



Clojure Debugging Toolkit



Related Projects

nREPL aims at replacing swank-clojure, to remove the dependency on Emacs/SLIME protocol (by Chas Emerick) and make a common library usable by other IDEs.

`https://github.com/clojure/tools.nrepl.git`

Slides available at: <http://furius.ca/tmp/emacs1j/>

Q&A

Learn from a Guru, Learn from Others

