Contents

1 Minor Modes

Like Emacs, StumpWM has the concept of minor modes. While minor mode definition and usage is documented in the manual, this document is intended to be a more fine grained and detailed, a walkthrough if you will.

2 Background Information

2.1 Scopes

Before anything can be said about minor modes themselves, first we must discuss minor mode *scopes*. Emacs has the benefit of having one core datatype - the buffer - which all things deal with. However StumpWM has several core datatypes - the window, the frame, the group, the head, and the screen. So while Emacs minor modes can simply be enabled in the current buffer (or all buffers if they are global) StumpWM minor modes must be scoped to a specific core datatype, referred to as a *scope object*.

While one can be rather precise with minor mode scopes, for the time being we will deal only with the predefined scopes [1]. In summation, minor modes are enabled in a *scope object* (a window, frame, group, etc) determined by the minor modes *scope designator*.

[1] see the manual §10.2 for a list of predefined minor mode scopes

2.2 The Nature of Minor Modes

As mentioned in the manual, minor modes are mixins, which are dynamically added and removed from their *scope objects*. For the unfamiliar, a mixin is a class which defines methods and slots which may be used in a plethora of other classes. Oftentimes mixins stand outside the regular hierarchy of the classes they are mixed in to.

Because a minor modes are mixins they are enabled by defining a new class on the fly which contains the minor mode as a mixin and calling `CHANGE-CLASS` upon the scope object the minor mode is to be enabled in.

2.3 The Benefit of Mixins

By defining minor modes as mixins we get the benefit of being able to override and augment generic functions that specialize upon the object class the minor mode is scoped to. Take, for instance, a minor mode scoped to windows which adds some form of tracking information. When a window is then added to or removed from a group it makes sense to update this tracking information. This becomes trivial when using mixins - one simply defines a before, after, or around method upon the generic functions `GROUP-ADD-WINDOW` and `GROUP-DELETE-WINDOW` which specializes upon the mixin type.

2.4 Enabling and Disabling Minor Modes

Minor modes are enabled and disabled explicitly, but when they are global they can also be enabled and disabled implicitly, for example when creating a new object. This implicit enabling and disabling is referred to as autoenabling and autodisabling. Autoenabling and autodisabling is how the minor mode is enabled or disabled in a specific object; a call to `enable-minor-mode` will call `autoenable-minor-mode` on every relevant scope object for that minor mode. This may be a single object or every object of a specific class, for example it may be called on the current group, or if the minor mode is global, on every group. If a minor mode is global it is added to a list tracking the enabled global minor modes, and this list is consulted when creating an object or changing the class of an object to find new minor modes to autoenable in the object.

3 Defining Minor Modes

Moving on to the meat of this document, how *does* one define and work with minor modes? To answer this, lets take a look at a reimplementation of the `swm-gaps` module using minor modes.

3.1 `DEFINE-MINOR-MODE`

We begin with the form `DEFINE-MINOR-MODE`, which, as its name implies, defines a minor mode. This will define a class and a set of functions, methods, and hooks related to it.

```
(define-minor-mode gaps-mode ()
  ((gaps-x :initform 10
           :accessor gaps-mode-x
           :allocation :class)
   (gaps-y :initform 10
           :accessor gaps-mode-y
           :allocation :class)
   (gaps-for-head :initform nil
                  :accessor gaps-mode-head-gaps
                  :allocation :class)
   (inc-hints :accessor gaps-mode-inc-hints
              :allocation :class))
  (:scope :tile-window)
  (:global t)
  (:lighter "GAPS")
  (:interactive t))
```

This looks rather like a `DEFCLASS` form, doesnt it. We have our class name (`GAPS-MODE`), a list of superclasses, a set of slots, and finally a set of options.

While this minor mode is simplistic enough not to warrant any broken-out behavior, more complex minor modes may wish to define their behavior in a separate class and define a minor mode that is its subclass. Of note when supplying a superclass list is that the minor mode should descend from the class `MINOR-MODE`. When no superclasses are provided this is done automatically, but when superclasses are provided the class `MINOR-MODE` should appear as a direct or indirect superclass.

The slots are all normal class slots. Here, because `GAPS-MODE` is global, we define the the slots as class allocated. This keeps the gap sizes consistent from window to window.

The additional options are where minor modes have much of their behavior defined. Here we see that the minor mode is scoped to tile windows (`(:scope :tile-window)`), and in addition it is global (`(:global t)`). As such, when enabling this minor mode it will not only be enabled in the current window (when the current window is a tiling window), but will be enabled in *every* tiling window. In addition, as new windows are created, if they are tiling windows they will have this minor mode enabled in them.

The next option defines a lighter for the minor mode (`(:lighter "GAPS")`). While this option can be a function, we have used a string as we dont need to generate different strings.

Finally, we instruct the macro to generate a command which toggles the minor mode on and off (`(:interactive t)`). This command will have the same name as the minor mode because we supplied `T` as the argument to this option.

3.1.1 Macro Expansion

Now lets take a look at the expansion of this macro. As you can see theres a good deal of code generated. All of the following takes place within a `progn`.

First we validate the minor modes superclasses and scope, signalling an error if they are invalid. The function `validate-minor-mode-superclasses` ensures none of the superclasses for the minor mode descend from a regular stumpwm class. The function `validate-scope` ensures that if a superclass of the minor mode is itself a minor mode that the scopes are compatible.

```
(validate-minor-mode-superclasses '(minor-mode))
(validate-scope :tile-window '(minor-mode))
```

Next we define some keymap variables. This could be prevented by adding the option (`(:expose-keymaps nil)`). These keymap variables hold keymaps created from the `:root-map` and `:top-map` options, which are both nil in this example.

```
(defvar *gaps-mode-root-map*
  (make-minor-mode-keymap nil)
  "The root map for GAPS-MODE")
(defvar *gaps-mode-top-map*
  (make-minor-mode-top-map nil '*gaps-mode-root-map*)
```

```
  "The top map for GAPS-MODE")
```

Now we generate the actual class itself. We add one slot to the slots we provided the `define-minor-mode` macro with, which holds a reference to the keymap variable.

```
(defclass gaps-mode (minor-mode)
  ((#:gkeymap819 :initform '*gaps-mode-top-map* :reader
                 gaps-mode-keymap :allocation :class)
   (gaps-x :initform 10 :accessor gaps-mode-x :allocation :class)
   (gaps-y :initform 10 :accessor gaps-mode-y :allocation :class)
   (gaps-for-head :initform nil :accessor gaps-mode-head-gaps
                  :allocation :class)
   (inc-hints :accessor gaps-mode-inc-hints :allocation :class))
  (:default-initargs))
```

Once weve defined the class we start defining methods. We start with the global-p method, which specializes on the minor mode name and just returns `T`.

```
(defmethod minor-mode-global-p ((mode (eql 'gaps-mode))) t)
```

The lighter method funcalls a generated function and conses it up with the results of the next method. As such this generic function returns a list of all minor mode lighters. If the user provided a function to this option it would be called in place of our generated function.

```
(defmethod minor-mode-lighter ((#:gmode818 gaps-mode))
  (cons (funcall (lambda (mode) (declare (ignore mode)) "GAPS") #:gmode818)
        (call-next-method)))
```

The scope method is similar to the global-p method, as it specializes on the minor mode name and returns the scope designator.

```
(defmethod minor-mode-scope ((#:gmode818 (eql 'gaps-mode)))
  (declare (ignore #:gmode818))
  :tile-window)
```

After that, we generate a set of hooks. The enable and disable hooks are run when autoenabling/autodisabling a minor mode, while the plain hook and destroy hook are run when explicitly enabling/disabling a minor mode. In addition to this a set of generic functions are defined which return and set the minor mode hooks. (note that the commas here are inserted for org mode, and are not present in the actual macroexpansion)

```
(defvar *gaps-mode-enable-hook*
  nil
  "A hook run when enabling GAPS-MODE, called with the mode symbol and the scope
object.")
(defvar *gaps-mode-disable-hook*
  nil
  "A hook run when disabling GAPS-MODE, called with the mode symbol and the
scope object. This hook is run when GAPS-MODE is disabled in an object, however
if an object goes out of scope before a minor mode is disabled then this hook
will not be run for that object.")
(defvar *gaps-mode-hook*
  nil
  "A hook run when explicitly enabling GAPS-MODE, called with the mode symbol
and the scope object.")
(defvar *gaps-mode-destroy-hook*
  nil
  "A hook run when explicitly disabling GAPS-MODE, called with the mode symbol
and the scope object.")
(defmethod minor-mode-enable-hook ((mode (eql 'gaps-mode)))
  (declare (ignore mode))
  *gaps-mode-enable-hook*)
(defmethod (setf minor-mode-enable-hook) (new (mode (eql 'gaps-mode)))
  (declare (ignore mode))
  (setf *gaps-mode-enable-hook* new))
(defmethod minor-mode-disable-hook ((mode (eql 'gaps-mode)))
  (declare (ignore mode))
  *gaps-mode-disable-hook*)
(defmethod (setf minor-mode-disable-hook) (new (mode (eql 'gaps-mode)))
  (declare (ignore mode))
  (setf *gaps-mode-disable-hook* new))
(defmethod minor-mode-hook ((mode (eql 'gaps-mode)))
  (declare (ignore mode))
  *gaps-mode-hook*)
(defmethod (setf minor-mode-hook) (new (mode (eql 'gaps-mode)))
  (declare (ignore mode))
  (setf *gaps-mode-hook* new))
(defmethod minor-mode-destroy-hook ((mode (eql 'gaps-mode)))
  (declare (ignore mode))
  *gaps-mode-destroy-hook*)
(defmethod (setf minor-mode-destroy-hook) (new (mode (eql 'gaps-mode)))
```

```
  (declare (ignore mode))
  (setf *gaps-mode-destroy-hook* new))
```

Next we have the keymap method, which like the lighter methods returns a list of all minor mode keymaps for a given object.

```
(defmethod minor-mode-keymap ((#:gmode818 gaps-mode))
  (cons (slot-value #:gmode818 '#:gkeymap819) (call-next-method)))
```

The enable-when method is used as an explicit user check to see if a minor mode should be enabled. If the `:enable-when` option is not provided this method defaults to always returning `T`.

```
(defmethod enable-when ((mode (eql 'gaps-mode)) (obj tile-window)) t)
```

Finally we have the autoenable/autodisable methods, of which three are defined. If the minor mode is already enabled in the object, then a condition is signalled. This method doesnt call `error` because we dont want to break to the debugger when we try to autoenable an already enabled minor mode, but we do want to provide a way for people to do something in such situations if they so please (via `handler-bind` and company). (Note that this condition type is subject to change - it should be changed to the type `minor-mode-autoenable-error`).

```
(defmethod autoenable-minor-mode ((mode (eql 'gaps-mode)) (obj gaps-mode))
  (signal 'minor-mode-enable-error :mode 'gaps-mode
                                   :object obj
                                   :reason 'already-enabled))
```

Up next is the actual autoenable method, which specializes upon the minor mode class name and the minor modes scopes class type (in this case a `tile-window`). This method first calls the `enable-when` generic function, and if the scope has a *filter type* defined checks the object against it using `typep`. The *filter type* is an optional type for minor mode scopes, and is useful because the autoenable methods cannot specialize upon arbitrary types - only upon classes can they specialize. Finally we mix the minor mode into the object, and then run the `*gaps-mode-enable-hook*`. If a hook was not defined we invoke the `continue` restart.

```
(defmethod autoenable-minor-mode ((mode (eql 'gaps-mode)) (obj tile-window))
  (when (and (enable-when mode obj))
```

```
    (prog1 (dynamic-mixins:ensure-mix obj 'gaps-mode)
      (handler-bind ((minor-mode-hook-error
                      (lambda (c)
                        (let ((r (find-restart 'continue c)))
                          (when r (invoke-restart r))))))
        (run-hook-for-minor-mode #'minor-mode-enable-hook 'gaps-mode obj)))))
```

Finally we have the autodisable method, which specializes upon the minor mode class name and the minor mode itself. This method first runs the hook `*gaps-mode-disable-hook*` (invoking the continue restart if the hook is not found) and then unmixes the minor mode from the object in question.

```
(defmethod autodisable-minor-mode ((mode (eql 'gaps-mode)) (obj gaps-mode))
  (handler-bind ((minor-mode-hook-error
                  (lambda (c)
                    (let ((r (find-restart 'continue c)))
                      (when r (invoke-restart r))))))
    (run-hook-for-minor-mode #'minor-mode-disable-hook 'gaps-mode obj t))
  (dynamic-mixins:delete-from-mix obj 'gaps-mode))
```

At the end of this all we have the command to toggle the minor mode, as well as the minor mode command definer.

The command to toggle the minor mode is self explanatory - if we provide a non-nil argument then we enable the minor mode, if we provide a nil argument we disable the minor mode, otherwise if the minor mode is enabled we disable it and if it is not enabled we enable it.

```
(defcommand gaps-mode
    (&optional (yn nil ynpp))
    ((:y-or-n))
  (flet ((enable ()
           (enable-minor-mode 'gaps-mode))
         (disable ()
           (disable-minor-mode 'gaps-mode)))
    (cond (yn (enable)) (ynpp (disable))
          ((minor-mode-enabled-p 'gaps-mode) (disable)) (t (enable)))))
```

The command definer defines a command which is only active if the minor mode is also active. It generates a call to `defcommand` with the proper arguments, and binds the special variable `*minor-mode*` to the current scope object for the minor mode in question.

```
(defmacro define-gaps-mode-command
    (name (&rest args) (&rest interactive-args) &body body)
  (multiple-value-bind (body decls docstring)
      (parse-body body :documentation t)
    `(defcommand (,name ,'gaps-mode)
         ,args
         ,interactive-args
       ,@(when docstring (list docstring))
       ,@decls
       (let ((*minor-mode* (find-minor-mode ',,'gaps-mode (current-screen))))
         ,@body))))
```

Finally, at the end of everything, we call `sync-keys` in case a keymap has been changed.

```
(sync-keys)
```

3.2 Implementing Gaps

As implied by the `:scope` of the minor mode defined above, we are implementing gaps by modifying a function that is called upon windows; we are modifying `geometry-hints`. (This presupposes that `geometry-hints` is defined as a generic function). The core of this is an around method which specializes upon `gaps-mode`. Here we adjust the x, y, width, and height values returned, and ensure that the parent window sticks to the size of the window. Through trial and error, `wx` and `wy` were determined to be best returned as 0.

```
(defmethod geometry-hints :around ((win gaps-mode))
  (multiple-value-bind (x y wx wy width height border center)
      (call-next-method)
    (declare (ignorable wx wy center))
    (values (+ x (gaps-mode-x win))
            (+ y (gaps-mode-y win))
            0
            0
            (- width (* 2 (gaps-mode-x win)))
            (- height (* 2 (gaps-mode-y win)))
            border
            t)))
```

After the core is set up, we do need to do some bookkeeping for the minor mode. We do most of this by defining after methods for `autoenable-minor-mode` and `autodisable-minor-mode`. As these methods will often access an objects slots, it is considered best practice to define a separate function, and call that function from within the after method. This is because enabling/disabling minor modes involves calling `change-class`[2].

The autoenable after method will check if the object has the slot `inc-hints` bound, and if so store the value of `*ignore-wm-inc-hints*` within it and set `*ignore-wm-inc-hints*` to `T`. This is done because gaps look best when all windows are the same size. Additionally, this method will maximize the window to ensure the gaps take effect.

```
(defun gaps-mode-after-enable (object)
  (unless (slot-boundp object 'inc-hints)
    (psetf *ignore-wm-inc-hints* t
           (gaps-mode-inc-hints object) *ignore-wm-inc-hints*))
  (maximize-window object))

(defmethod autoenable-minor-mode :after ((mode (eql 'gaps-mode)) object)
  (gaps-mode-after-enable object))
```

The autodisable method simply checks if the object is a tile window, and if so maximizes it, ensuring the disabling of gaps (visually) takes effect.

```
(defun gaps-mode-after-disable (object)
  (when (typep object 'tile-window)
    (maximize-window object)))

(defmethod autodisable-minor-mode :after ((mode (eql 'gaps-mode)) object)
  (gaps-mode-after-disable object))
```

Finally, when gaps mode is explicitly disabled, we want to reset `*ignore-wm-inc-hints*` to whatever it was set to before, and make the slot `inc-hints` unbound so that `*ignore-wm-inc-hints*` gets saved the next time gaps mode is enabled. To do this we hang a hook function on the destroy hook for gaps mode. This hook is run once when the minor mode is explicitly disabled, and it is run with the very first object the minor mode will be disabled in. It is run before autodisabling the first object.

[2] `http://www.lispworks.com/documentation/HyperSpec/Body/f_chg_cl.htm`

```
(defun update-wm-increment-hints-for-destroy-hook (mode obj)
  (declare (ignore mode))
  (setf *ignore-wm-inc-hints* (gaps-mode-inc-hints obj))
  (slot-makunbound obj 'inc-hints))

(add-hook *gaps-mode-destroy-hook* 'update-wm-increment-hints-for-destroy-hook)
```

3.3 The Full Implementation

Here is the full implementation of gaps mode (all 48 lines of it). Do note that this presupposes that `geometry-hints` has been defined as a generic function.

```
(in-package :stumpwm)

(define-minor-mode gaps-mode ()
  ((gaps-x :initform 10 :accessor gaps-mode-x :allocation :class)
   (gaps-y :initform 10 :accessor gaps-mode-y :allocation :class)
   (gaps-for-head :initform nil :accessor gaps-mode-head-gaps :allocation :class)
   (inc-hints :accessor gaps-mode-inc-hints :allocation :class))
  (:scope :tile-window)
  (:global t)
  (:lighter "GAPS")
  (:interactive t))

(defmethod geometry-hints :around ((win gaps-mode))
  (multiple-value-bind (x y wx wy width height border center)
      (call-next-method)
    (declare (ignorable wx wy center))
    (values (+ x (gaps-mode-x win))
            (+ y (gaps-mode-y win))
            0
            0
            (- width (* 2 (gaps-mode-x win)))
            (- height (* 2 (gaps-mode-y win)))
            border
            t)))

(defun gaps-mode-after-enable (object)
  (unless (slot-boundp object 'inc-hints)
```

```
    (psetf *ignore-wm-inc-hints* t
           (gaps-mode-inc-hints object) *ignore-wm-inc-hints*))
  (maximize-window object))

(defmethod autoenable-minor-mode :after ((mode (eql 'gaps-mode)) object)
  (gaps-mode-after-enable object))

(defun gaps-mode-after-disable (object)
  (when (typep object 'tile-window)
    (maximize-window object)))

(defmethod autodisable-minor-mode :after ((mode (eql 'gaps-mode)) object)
  (gaps-mode-after-disable object))

(defun update-wm-increment-hints-for-destroy-hook (mode obj)
  (declare (ignore mode))
  (setf *ignore-wm-inc-hints* (gaps-mode-inc-hints obj))
  (slot-makunbound obj 'inc-hints))

(add-hook *gaps-mode-destroy-hook* 'update-wm-increment-hints-for-destroy-hook)
```