GWML: The Generic Window-manager in ML
Version 0.05

Fabrice Le Fessant
(Email: Fabrice.Le_Fessant@inria.fr)
INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France

June 19, 2020

Contents

Chapter 1

Overview

Gwml is a window-manager inspired from Gwm (Generic Window Manager). While Gwm was configurable through a Lisp dialect (WOOL), Gwml can be confugured in *Objective-Caml*. Thus, you neither need to learn a new language nor a special configuration file format. Instead, you will be able to configure your desktop simply by writting *Objective-Caml* files. If you don't know *Objective-Caml*, you will learn a very useful and powerful language (see http://pauillac.inria.fr/ocaml).

This ability makes Gwml the only window-manager, which is written in the same language as it is configurable with. As a consequence, you will be able to interact deeper within your window-manager than with any other one. Moreover, you will be able to integrate your configuration into the window-manager sources to speedup your desktop management and make other persons benefit from your configuration.

Since *Objective-Caml* is a powerful language, you will be able to use Threads, Unix system calls, objects, lexers and parsers (to read your old fvwm config file!) and many other features in your configuration files. In particular, the static typing of Caml will allow you to check your configuration files before restarting your window-manager (yes, it was awful to start Gwm with a buggy configuration).

Chapter 2

Usage

2.1 Installation

Gwml is part of the Efuns package, which can be found on **http://pauillac.inria.fr/~lefessan**. The Efuns package contains several parts:

- **Xlib**: An implementation of the Xlib library in *Objective-Caml*. The whole X protocol is implemented, plus some other extensions (Icccm, XShape, Xpm, Zpixmap, Xrm, etc ...). Moreover, it contains an emulation of the Graphics library of *Objective-Caml* (called XGraphics).
- **Dynlink**: An implementation of the Dynlink library (which enables programs to dynamically load bytecode files) which works for native programs.
- **Efuns**: Efuns is an Emacs clone. Like Gwml, it is configurable in *Objective-Caml* instead of Elisp. Thus, you can program better editing modes (in particular, by using ocamllex lexers) than with Emacs.
- **Gwml**: Gwml is the Generic Window-manager in ML.
- **Toplevel**: A library to evaluate *Objective-Caml* strings in a program (even native). Include the dynlink library. Look at toplevel/dyneval.mli for details.

To install Gwml, you need *Objective-Caml* version 0.05 installed.

e the configure script instead. Edit `Makefile.config` and modify the following variables:

- `OCAMLLIB`: directory where *Objective-Caml* is installed, and where the Xlib and Asmdynlink libraries will be installed (def: /usr/local/lib/ocaml)
- `OCAMLVERSION`: The version of *Objective-Caml* currently installed on your system.

- `INSTALLBIN`: directory where "efuns" and "gwml" executables will be installed. (def: /usr/local/bin)
- `EFUNSLIB`: directory where Efuns files will be installed (def: /usr/local/lib/efuns)
- `GWMLLIB`: directory where Gwml files will be installed (def: /usr/local/lib/gwml)
- `BYTE_THREADS`: should the bytecode version use threads (def: threads)
- `OPT_THREADS`: should the native code version usr threads (def: nothreads)
- `OCAMLSRC`: The absolute path of *Objective-Caml* sources if you want to compile the Toplevel library.
- `DYNLINK`: Set to `toplevel` if you want to compile the toplevel library, `dynlink` otherwise.
- `GWML_DYNLINK`: Should Gwml use the `dynlink` library or the `toplevel` library.
- `EFUNS_DYNLINK`: Should Efuns use the `dynlink` library or the `toplevel` library.

1. Compile in bytecode: "make byte".
2. Compile in native code: "make opt".
3. Install the bytecode version: "make install".
4. Install the native version: "make installopt".
5. If you want to avoid clashes between *Objective-Caml* versions, you shoud save *Objective-Caml* files used by Gwml and Efuns in `EFUNSLIB` and `GWMLLIB` directories. If you upgrade your *Objective-Caml* version, Efuns and Gwml will not be bothered. For this: "make version".

2.2 Known Bugs

These are bugs, missing features and advices from users. To add a line to this list, mail to **Fabrice.Le_Fessant@inria.fr**.

- Problems with XView (try to resize the window to get it mapped correctly).
- Shaped windows are not well decorated.
- Not really ICCCM 2.0 compliant (need more colormap support).

2.3 Invocation

Gwml can be started using either "gwml" (native version) or "gwml.byte" (bytecode version).

2.3.1 Command line options

The following parameters can be specified in the invocation command:

- **−c <file.ml>**: compile a configuration file. If Gwml has been build with the Toplevel library, it will compile the file itself (to prevent version clashes with ocamlc), else it will use ocamlc.
- **−d <display>** or **−−display <display>**: Use given display name instead of the default display (DISPLAY shell variable).
- **−I <dir>**: Add the given directory to the path where config files will be searched. By default, the minimum path is:
  $(GWMLPATH) plus
  $(HOME):$(HOME)/.gwml:$(HOME)/.pixmaps:. plus
  GWMLLIB:$(CAMLDIR):OCAMLLIB

  where HOME, GWMLPATH and CAMLDIR are shell envirronment variables and GWMLLIB and OCAMLLIB the directories specified in the Makefile.config during the installation.
- **−M** : map all windows (This is useful when some windows have been erroneously unmapped).
- **−r** : Make Gwml infinitely retry to decorate a screen if another window-manager is currently controlling that screen.
- **−f <Module>** : Make Gwml load this module (after Gwmlrc).
- **−q** : Prevent Gwml from loading gwmlrc.cmo.

Other anonymous parameters (not preceded by −) don't generate errors, they are queued in `Gwml.args` for latter use by the user configuration files.

2.3.2 Shell variables

Gwml will look for the following shell variables:

- **DISPLAY**: should be set to the name of the display (unless the -d option is specified).
- **HOME**: should be set to the HOME directory of the user (normally correctly set by the shell).
- **GWMLPATH**: path (semicolon separated list of directories) on which to search for bytecode modules and pixmap files.
- **CAMLDIR**: directory where *Objective-Caml* modules can be found.
- **XUSERFILESEARCHPATH** and **XFILESEARCHPATH** (or **XENVIRONMENT** or **X11ROOT**: the files and paths to look for X ressources.

2.3.3 Files

Gwml will look for the Gwmlrc module (gwmlrc.cmo) in its path, and load it unless the -q option is specified.

Gwml will also load ressources from the file specified by the XUSERFILE-SEARCHPATH (or " /.Xdefaults" at least) and the "Gwml" file in the XFILE-SEARCHPATH path plus the XENVIRONMENT directory, X11ROOT/lib/X11/app-defaults/ and /usr/X11/lib/X11/app-defaults/.

Chapter 3

Configuration

3.1 Overview

The default config is specified in the module Stdgwmlrc. This config can be copied to start a new personal configuration.

A configuration is installed by adding a new hook to `screen_opening_hooks`, which will be executed for each screen, with the screen wob as parameter, when Gwml takes the control of all screens. This screen hook must add a wob hook to the screen wob. This wob hook must intercept the `WobNewClient` message, which is sent when a new client must be decorated. The following code is given as example:

```
let my_window_conf screen_wob event =
   match event with
     WobNewClient client_desc ->
         ... client initialisation stuff ...
   | ... other messages handlers ...
   | _ -> ()

let _ =
  screen_opening_hooks := (fun screen_wob ->
    ... screen initialisation stuff ...;
    screen_wob.w_hooks <- my_window_conf :: screen_wob.w_hooks;
     ) :: !screen_opening_hooks
```

When a new client appears, a message, called `WobNewClient`, is sent to the screen wob. One of the hooks installed by the user configuration must intercept this event, create the description of the decoration of the window, and call `Client.decorate` with this decoration.

3.2 Default configuration

The Stdconfig module defines the following configuration:

3.2.1 Key bindings

Personally, the "Windows" key of my keyboard is binded on modifier 5 (using Xmodmap). Thus, I use this modifier for all window-manager bindings. You can change this by specifying other modifiers in your Gwmlrc module, for example with `Stdconfig.wm_modifiers := controlMask + mod1Mask` which will use the (Control + Alt) modifiers for all bindings.

Bindings for windows

```
Windows-l : lower the current window
Windows-i : iconify the current window
Windows-k : kill the client
Windows-delete : destroy the window
Windows-space : maximize the current window vertically
Windows-w : maximize the current window horizontally
Windows-t : toggle stay on screen in virtual moves

Windows-button-1 : move the current window
Windows-button-2 : iconify the current window
Windows-button-3 : resize the current window
```

Bindings for icons

```
button-1 : de-iconify the associated window
```

Bindings for the root window

```
Windows-p: print informations on all clients
Windows-(up,down,left,right): move virtual screens
Windows-r : restart gwml
Windows-x : start an xterm
Windows-e : start an Efuns
Windows-m : map all windows
Windows-z : de-iconify last iconified windows
Windows-d : debug mode

button-1 : open the first menu (Stdconfig.popup1)
button-2 : open the window menu
```

3.2.2 Focus modes

Stdconfig is in auto-raise and auto-colormap modes. You can change this in your Gwmlrc by setting `Stdconfig.auto_raise := false` or `Stdconfig.auto_colormap := false`.

3.2.3 Decorations

Screen

The screen contains a pager on the lower right corner. Icons are placed in the upper left corner. Virtual moves occur when the mouse touches the edges of the screen.

Windows

At the top, a bar contains a mini pixmap in the left corner, a null window in the middle and the name of the window on the right. Nothing in other sides. The pixmap in the corner can be specified in the ressource files by, for example:

```
Efuns*title_pixmap: mini-edit.xpm
Emacs*title_pixmap: mini-edit.xpm
```

XBuffy,CPUStateMonitor,Clock and XConsole are not decorated. See the last lines of the Stdconfig module.

Icons

In the center, a pixmap and the icon name under the pixmap. The pixmap can be specified in the ressource files by, for example:

```
XTerm*icon_pixmap: xterm.xpm
Efuns*icon_pixmap: textedit.xpm
Emacs*icon_pixmap: textedit.xpm
```

Icons are placed in the upper left corner of the screen (in three rows of 10 icons).

Chapter 4

Gwml modules

4.1 Gwml core

4.1.1 Eloop

This module implements the event dispatch table: After its initialisation, Gwml calls the `Eloop.event_loop` function. This function infinitely waits for events from each display. When such an event is received, the destination window is searched for in the windows table. If found, the corresponding event handler is triggered. Else, a default event handler is used.

`Eloop.add_display` is used to add a new display. A default event handler is specified for that display. `Eloop.add_window` is used to add a new window to the window table for a given display, with a corresponding event handler. `Eloop.remove_window` is used to remove windows from the window table.

`Eloop.known_window` is used to query if a window is already present in the window table. This function is used to discover if a window being mapped is already known, or if it is a new client which must be decorated. Therefore, it is extremely important that useless windows are removed from the window table (`Eloop.remove_window`)

The `display` object which must be provided for these calls can be found in the `screen_desc` record, label `s_scheduler`. From a given wob `w`, it can be addressed by `w.w_screen.s_scheduler`.

4.1.2 Env

This module implements some kind of dynamic scoping with static typing. Each wob contains an envirronment, where monomorphic variables can be put. Use `Wob.getenv` and `Wob.setenv` to access these envirronments. Use `Env.new_var ()` to define new monomorphic variables (see Stdconfig) for examples.

4.1.3 Gwml

This module defines the most useful types and values of Gwml.

Wobs

Wobs (as in Gwm) are the basic objects to build window decorations. Wobs have the following types:

```
type wob = {
    mutable w_window : window;
    w_parent : wob;
    w_top : wob;
    w_screen : screen_desc;
    w_first_hook : hook;
    w_last_hook : hook;
    mutable w_hooks : hook list;
    w_geometry : rect;
    mutable w_env : Env.t;
    w_info : wob_info;
  }
```

`w_window` is the X window corresponding to that Wob. It can be noWindow if the window has not been created yet (often happen with `WobInit, WobGetSize and WobResize` events). `w_parent` is the wob of the parent window, `w_top` the wob of the corresponding top window (Top windows are special wobs, defined in the Top module, which are direct root childs). For top wobs, `w_parent` is the wob of the root window and `w_top` is the wob itself. `w_screen` is the description of the screen of the wob window. `w_geometry` is the geometry of the window (`x,y` to its parent, `width` and `height`). `w_env` is the wob local envirronement, which should be manipulated using `Wob.getenv` and `Wob.setenv`.

Wobs are not directly in touch with X events. Instead, X events are translated by Gwml in other events, called `wob_event`. The `w_first_hook`, `w_last_hook` and `w_hooks` fields of wobs are handlers to wob events. When a wob event is received, `w_first_hook` is first triggered. Then, `w_hooks` are triggered with the same event. Finally, `w_last_hook` is triggered with the event. Nothing can prevent `w_first_hook` from execution. However, an exception raised in one of the `w_hooks` will prevent other hooks left and `w_last_hook` from being triggered. Each hook receives the wob as first argument and the wob event as second argument.

Wob events

```
type hook = (wob -> wob_event -> unit)

type wob_event =
| WobInit
```

```
| WobGetSize
| WobCreate
| WobResize
| WobKeyPress of Xtypes.Xkey.t * string * Xtypes.keycode
| WobButtonPress of Xbutton.t
| WobUnmap
| WobMap
| WobEnter
| WobLeave
| WobDestroy
| WobNewClient of client_desc
| WobMapRequest
| WobRefresh
| WobIconifyRequest of bool
| WobDeiconifyRequest of bool
| WobWithdrawn
| WobButtonRelease of Xbutton.t
| WobPropertyChange of atom
| WobSetInputFocus
| WobDeleteWindow
| WobInstallColormap of bool
| WobMessage of string
```

These events have the following meanings (* Note that predefined wobs already implement special behaviors for some of these events, in particular `WobGetSize`, `WobResize`, `WobCreate` and `WobDestroy`).

- **WobInit**: A wob receive this event normally before any other event. It is the good time to add any local variable needed during the normal behavior of the wob. The wob `w_window` and the ones of its parents are not normally initialized (thus, they are equal to noWindow).

- **WobGetSize**: This event is received when the parent wob wants to known the requested size of the wob. The wob should modify its geometry `w_geometry` with its requested `width` and `height`.

- **WobResize**: This event is received when the geometry of the wob has been modified. The `w_geometry` contains the new geometry. If the `w_window` has been initialized, the wob should resize and move its X window according to the new geometry.

- **WobCreate**: This event is received when the wob must create its X window. Before receiving this event, the wob should have receive `WobInit`, `WobGetSize` and `WobResize` to ensure its current geometry has been correctly computed. Once the X window has been created, it must be registered in the Eloop scheduler with an associated X event handler (* Note that Wob.xevent can be used as a default handler for standard windows).

- **WobKeyPress (key_event, key_string, keycode)**, **WobButtonPress button_event** and **WobButtonRelease button_event**: This events are received when keys or buttons are pressed in the wob, or when a grabbed key or button is pressed in a child wob.

- **WobUnmap** and **WobMap**: This events are received when the wob must be unmapped or mapped.

- **WobEnter** and **WobLeave**: This events are received when the pointer enters or leaves the wob.

- **WobDestroy**: This event is received when the wob must be destroyed. The wob must destroy its X window and remove it from the Eloop.scheduler to avoid that Gwml ignore windows with the same UID.

- **WobNewClient client_description**: This event is received by the screen wob when a new client must be decorated. The description of the client is provided as a parameter of the event. Since several hooks can receive this event, each hook must verify that the client has not been decorated yet by a previous hook. **This event can only be received by screen wobs.**

- **WobMapRequest**: This event is generated prior to `WobMap` when some treatment has to be done before mapping the wob. In particular, client top wobs will send this event the first time they are mapped to allow placement computation or interaction with the user. Icons may also receive this event (but this should be done by YOUR configuration).

- **WobIconifyRequest from_user** and

- **WobDeiconifyRequest from_user**: These events are received when requests are made to iconify or de-iconify a client. The `from_user` parameter indicates whether the request was made by the user (by clicking for example) or by the program(Control-z under Emacs).

- **WobRefresh**: This event is received when a wob must redraw its window (after a change in what is drawn, or when the X window is exposed).

- **WobWithdrawn**: This event is generated when a client requests to unmap its window. As a consequence, Gwml must stop decorating the corresponding window.

- **WobPropertyChange property**: This event is generated when a client modifies a property on the client X window. The changes in the property values

- **WobSetInputFocus**: This event is received by a wob in the decoration of a client when the focus should be set to the client.

- **WobDeleteWindow**: This event is received by a wob in the decoration of a client when a request to delete the window (using the DELETE_WINDOW protocol) has been received.
- **WobInstallColormap force**: This event is received by a wob when its colormap should be installed (or by a wob in the decoration of a client whose colormap should be installed).
- **WobMessage message**: This event can be used to extend the current set of wob events. The message is a string, which should be recognized by some wobs involved in a particular protocol.

The client structure

The structure of a client descriptor is:

```
type client_desc =
  {
    c_window : window;
    c_set_focus : bool;
    mutable c_colormap : colormap;
    mutable c_new_window : bool;
    c_geometry : rect;
    mutable c_name : string;
    mutable c_icon_name : string;
    c_machine : string;
    c_class : string * string;
    c_size_hints : wm_size_hints;
    mutable c_wm_hints : wm_hints;
    c_transient_for : window;
    mutable c_colormap_windows : window list;
    c_icon_size : wm_icon_size list;
    mutable c_wm_state : wmState;
    mutable c_wm_icon : window;
    mutable c_delete_window: bool;
    mutable c_take_focus: bool;
    mutable c_decorated: bool;
    mutable c_protocols: atom list;
  }
```

`c_window` is the X window of the client. `c_colormap` is the colormap needed by the client (or noColormap), `c_geometry` is the geometry required by the client. `c_name`, `c_icon_name`, `c_machine`, `c_class`, `c_size_hints`, `c_wm_hints`, `c_transient_for`, `c_protocols` and `c_colormap_windows` are the so-called properties of the client. `c_set_focus` indiquates whether the client requires assistance of the window-manager to take the focus. `c_delete_window` indiquates whether the client participate in the DELETE_WINDOW protocol, and

the `c_take_focus` if the client participate in the TAKE_FOCUS protocol. Finally, `c_decoated` indicates whether the client has already been decorated, and `c_new_window` is the client is a new window, or if it was already on screen when Gwml was started.

The screen structure

The structure of a root window descriptor is:

```
type screen_desc = {
    s_clients : (window, client_desc * wob) Hashtbl.t;
    s_scr : Xtypes.screen;
    s_colors : (string, pixel) Hashtbl.t;
    s_pixmaps : (string, int*int*pixmap*pixmap) Hashtbl.t;
    s_fonts : (string, font * queryFontRep) Hashtbl.t;
    s_scheduler : Eloop.display;
    mutable s_last_cmap : colormap;
    mutable s_cmap_wob : wob;
  }
```

`s_clients` associates X windows (for client window and top windows) to client wobs. `s_scheduler` is used to query Eloop. `s_last_cmap` indiquates which colormap has been installed by Gwml, and `s_cmap_wob` which wob is associated with that colormap.

The decorations

Each client is decorated by four wobs: the left wob, the right wob, the title wob and the bottom wob. These wobs are not created directly by the user. Instead, the user call the function `Client.decorate` which will create the top wob (the wob containing the client and its decorations), and then the other decoration wobs, from the descriptions given by the client.

The following structure is used to describe the decoration wobs:

```
type wob_desc = {
    wob_first_hook : hook;
    wob_last_hook : hook;
    mutable wob_hooks : hook list;
  }
```

This structure only describes the hooks which will be used in the wobs. Such descriptions are predefined for some useful wobs, such as pixmaps (use `Pixmap.make`), labels with string (use `Label.make`), bars containing other wobs (use `Bar.make`) and empty wobs (use `Null.make`).

Parameters

The following values are computed from parameters and shell variables:

```
val mapall : bool ref
val load_path : string list ref
val dpyname : string ref
val args : string list ref (* the anonymous args *)
val retry : bool ref
val no_gwmlrc : bool ref
val load_files : string list ref
val batch_mode : bool ref
```

Display values

```
val screens : wob array ref
val display : Xtypes.display
val x_res : string Xrm.t
```

`display` is the display to be used in X requests. `screens` is the array containing the screens controlled by Gwml. `x_res` is the ressource database, containing the user ressources and the system ressources.

Configuration hooks

This hooks are used to configure Gwml:

```
val screen_opening_hooks : (wob -> unit) list ref
val top_opening_hooks : (wob -> unit) list ref
```

The first ones are executed with each screen wob, when Gwml is started. The second ones are executed each time a new top wob is created.

Useful functions

```
val font_make : wob -> string -> Xtypes.font * Xtypes.queryFontRep
val color_make : wob -> string -> Xtypes.pixel
val pixmap_make : wob -> string ->
        Xtypes.size * Xtypes.size * Xtypes.pixmap * Xtypes.pixmap
```

`font_make` converts a font name into an X font and its properties (as returned by X.queryFont). `color_make` converts a color (either a name, or an RGB value started with # (such as #80ff00)) into an X pixel. Finally, `pixmap_make` converts a pixmap filename into a pixmap data (width, height, pixmap, mask).

Useful atoms

These are useful atoms when dealing with ICCCM compliance:

```
val xa_wm_protocols : Xtypes.atom
val xa_wm_colormap_windows : Xtypes.atom
val xa_wm_state : Xtypes.atom
```

```
val xa_wm_take_focus : Xtypes.atom
val xa_wm_delete_window : Xtypes.atom
val xa_wm_change_state : Xtypes.atom
```

4.1.4 Wob

The Wob module defines operations on Wobs:

```
val send : Gwml.wob -> Gwml.wob_event -> unit
```

`Wob.send` sends a wob event to a particular wob.

```
val info : unit -> Gwml.wob_info
val set_grabs : Gwml.wob -> Gwml.grab list -> unit
val grabs : Gwml.wob -> Gwml.grab list
val set_background : Gwml.wob -> string -> unit
```

These functions deals with generic parameters of wobs. They are not currently used, but they could replace direct X requests in the future.

```
val make : Gwml.wob -> Gwml.wob_desc -> Gwml.wob
```

`Wob.create` creates a new wob from a parent wob and a description. This function should never be called to create top wobs. Instead, it can be used in nested decoration wobs (see the Bar module).

```
val setenv : Gwml.wob -> 'a Env.var -> 'a -> unit
val getenv : Gwml.wob -> 'a Env.var -> 'a
val sgetenv : Gwml.wob -> 'a Env.var -> 'a -> 'a
```

These functions deals with wob local envirronments. `sgetenv` is the same as `setenv`, but the user must provide a default value in case the variable is not defined in the envirronment.

```
val mask : Xtypes.eventMask list
val xevents : Gwml.wob -> Xtypes.xevent -> unit
```

These are the default wob mask for X event and the default handler for these events. It has the following behavior:

- When the mouse enters the wob, the wob receives a `WobEnter` event.
- When the mouse leaves the wob, the wob receives a `WobLeave` event.
- When a button is pressed in the wob, the wob receives a `WobButtonPress` event.
- When a button is released in the wob, the wob receives a `WobButtonRelease` event.
- When a key is pressed in the wob, the wob receives a `WobKeyPress` event.
- When the X window of the wob should be redrawn, the wob receives a `WobRefresh` event.

4.1.5 Eval

The Eval module is responsible for loading and evaluation of bytecode modules.

```
val load : string -> unit
```

`load` loads the bytecode file associated with a module name (`Eval.load "Gwmlrc"` loads the file "gwmlrc.cmo"). If the loaded file depends on other files to be loaded, it will try to load the other files and their interfaces before.

4.1.6 Main

The main module is the final module of Gwml. Since this module does not terminate, it is not available from configuration modules.

4.2 Predefined Wobs

4.2.1 Top

The Top modules defines the functions used to handle top wobs:

```
type top_desc = { mutable borderwidth: int; mutable borderpixel: string }
val default : unit -> top_desc
val make :
  Gwml.wob ->
  top_desc ->
  Gwml.hook list ->
  Gwml.wob_desc ->
  Gwml.wob_desc option ->
  Gwml.wob_desc option ->
  Gwml.wob_desc option ->
  Gwml.wob_desc option -> Gwml.wob
```

`Top.make screen_wob top_desc hooks center left right title bottom` creates a new top wob, with one internal wob (its center) and four decoration wobs. The top wob is not immmediately mapped. To map it, you need to send it a `WobMap` event. Before, you can modify its position by modifying its `w.w_geometry`.

```
val resize : Gwml.wob -> int -> int -> unit
val resize_top : Gwml.wob -> int -> int -> unit
```

Resizing a wob is more complicated than resizing a simple wob. Indeed, some internal wobs may require special care on their size. Thus, the top wob resizing has two phase: in the first phase, a `WobGetSize` is sent to each internal wob. In the second phase, the sizes requested by internal wobs are gathered, and a `WobResize` is sent to each internal wob to take its new size into account.

`Top.resize w width height` is used to resize the center wob of a top wob. `Top.resize_top` is used to resize the top wob itself.

4.2.2 Client

The Client module defines the function used to create the top wob and the decorations associated with a new client:

```
val decorate :
   Gwml.wob ->
   Top.top_desc ->
   Gwml.hook list ->
   Gwml.client_desc ->
   Gwml.wob_desc option ->
   Gwml.wob_desc option ->
   Gwml.wob_desc option ->
   Gwml.wob_desc option -> Gwml.wob
```

`Client.decorate screen_wob top_hooks top_desc client_desc left right title bottom` creates a top wob for the client with the decorations as described by `left, right, title` and `bottom`. The top wob contains a special wob in its center, the client wob.

The client wob defines a particular behavior when it receives special events (mainly for ICCCM compliance):

- `WobResize`: Resize the client to the new geometry.
- `WobCreate`: Add the top X window and the client X window to the screen `s_clients` table. Reparent the client window in the top wob. Map the client in its top window (the client X window is never unmapped).
- `WobDestroy`: Remove the client X window and its top X window from the screen `s_clients` table.
- `WobMap`: Change the ICCCM WM_STATE to NormalState.
- `WobUnmap`: Change the ICCCM WM_STATE to IconicState.
- `WobWithdrawn`: Reparent the client window to the root window, unmap the client X window, change the ICCCM WM_STATE to WithdrawnState, and send a `WobDestroy` to its top window to remove all decorations for the client.
- `WobSetInputFocus`: If the client requested help of the window-manager for setting the focus (WM_HINTS.input), explicitely set the focus to the client window. Else, sends a XClientMessage to the client for it to take the focus (ICCCM compliance).
- `WobDeleteWindow`: if the client supports the DELETE_WINDOW protocol, sends a XClientMessage to the client for it to delete its window.
- `WobInstallColormap`: try to install the colormap needed by the client, according to its colormap hints.

- `WobPropertyChange WM_COLORMAP_WINDOWS`: install the new client colormap if its previous colormap was installed.

4.2.3 Screen

The Screen modules defines the screen wobs. However, it does not contain functions directly usable by the user.

4.2.4 Null

The Null module defines a wob which doesn't contain anything. It can be used to fill a rectangle between other wobs.

```
val make : Gwml.hook list -> Gwml.wob_desc
```

4.2.5 Label

The Label module defines a wob which displays a string in its window.

```
type label_desc =
  { mutable string: string;
    mutable font: string;
    mutable background: string;
    mutable foreground: string;
    mutable min_width: int;
    mutable min_height: int }
val default : string -> label_desc
val make : label_desc -> Gwml.hook list -> Gwml.wob_desc
```

4.2.6 Pixmap

The Pixmap module defines a wob which displays a pixmap in its window. `Pixmap.make` defines the description which can be used in a client decoration.

```
type pixmap_desc =
 { pixmap: Xtypes.pixmap;
   mask: Xtypes.pixmap;
   min_width: int;
   min_height: int;
   bitmap: bool;
   mutable foreground: string;
   mutable background: string }
  val default : Gwml.wob -> string -> pixmap_desc

  val make : pixmap_desc -> Gwml.hook list -> Gwml.wob_desc
```

4.2.7 Bar

The Bar module defines a wob which contains other wobs. In the future, other functions to add or remove wobs in the bar will be provided.

```
type sens = | Vertical | Horizontal
and bar_desc =
  { mutable min_width: int;
    mutable min_height: int;
    mutable wobs: Gwml.wob array;
    sens: sens;
    mutable sizes: Xtypes.rect array;
    mutable background: string;
    mutable foreground: string }
val default : sens -> bar_desc

val make : bar_desc -> Gwml.wob_desc array -> Gwml.hook list -> Gwml.wob_desc
```

4.3 Gwml Standard Config

4.3.1 User

The User module defines a function to interactively move or resize a client window.

```
type move_resize = | Move | Resize

val user_move_resize_client :
  Gwml.wob -> move_resize -> int -> int -> Xtypes.button option -> unit
```

4.3.2 Virtual

The Virtual module defines a virtually infinite screen. `Virtual.toggle_move` on a wob makes the wob always stay on the current screen, or stay in a local screen. `Virtual.move screen_wob dx dy` move the screen from `dx` and `dy`. `Virtual.goto wob` moves the screen to the local screen of the wob.

```
val toggle_move : Gwml.wob -> unit
val update : Gwml.wob -> unit
val move : Gwml.wob -> int -> int -> unit
val start : Gwml.wob -> info
val omit_move : Gwml.wob -> bool -> unit
val omit_draw : Gwml.wob -> bool -> unit
val goto : Gwml.wob -> unit
```

4.3.3 Stdconfig

The Stdconfig module defines the default config for Gwml. It is provided more as an example for development of new configurations. Interesting are the `icon_make` function (to create an icon for a client), the `popup_menu` function (to pop-up a menu defined by an array), the `c_hook` and `s_hook` functions (hooks for client top wob and screen wobs), and the `simple_window` and `no_deco` functions which create the decoration for clients.