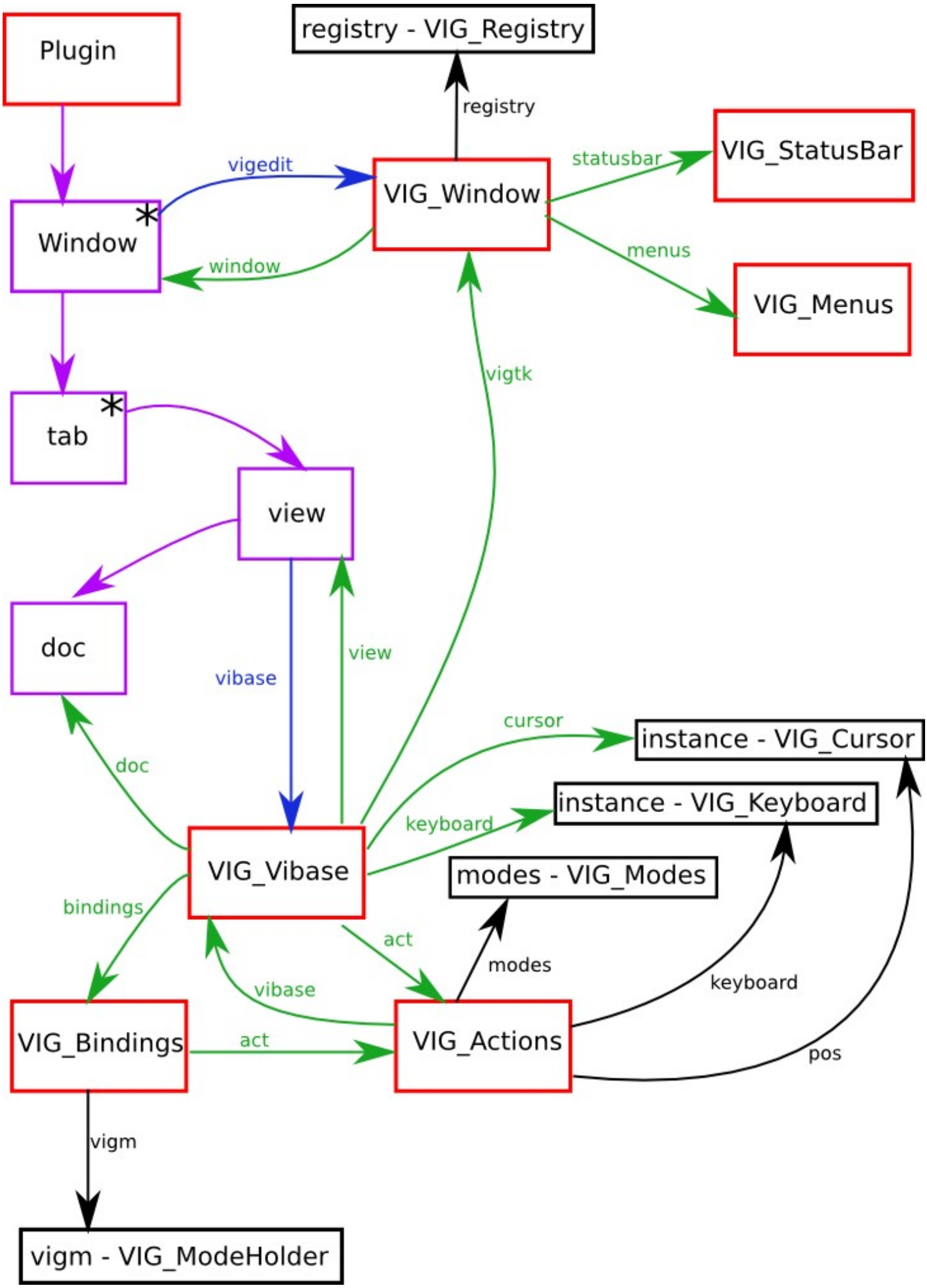
How it works


Plugin
registry - VIG_Registry
registry
vigedit
VIG_Window
statusbar
VIG_StatusBar
Window
window
menus
VIG_Menus
vigtk
tab
view
doc
vibase
view
cursor
instance - VIG_Cursor
doc
keyboard
instance - VIG_Keyboard
VIG_Vibase
modes - VIG_Modes
bindings
act
modes
keyboard
vibase
VIG_Bindings
act
VIG_Actions
pos
vigm
vigm - VIG_ModeHolder

- Purple blocks are things already existing in gedit
- Black blocks are only instantiated once over the lifetime of gedit
- Red blocks are instantiated for every object it is attached to
- Blue lines indicate the use of the set_data function
- A star in the corner says there are many of these per object it is attached to

So essentially, gedit will have one or more windows. For each window it will have one or more tabs. For each tab it will have a view, and for each view, a document.

Vigedit defines 12 main classes that it uses to make gedit into a modal editor

- VIG_Vibase
- VIG_Window
- VIG_Statusbar
- VIG_Menus
- VIG_Cursor
- VIG_Keyboard
- VIG_Registry
- VIG_Bindings
- VIG_Modes]
- VIG_Actions
- VIG_ModeHolder
- VIG_ModeBase

The folder structure of the plugin is laid out as such :

- ViGedit/
  - __init__.py - Plugin class
  - cursor.py - VIG_Cursor and 'instance' (an instance of VIG_Cursor)
  - keyboard.py - VIG_Keyboard and 'instance' (an instance of VIG_Keyboard)
  - options.py – some options (currently just configuring verbosity)
  - static.py - VIG_Modes, 'modes' (instance of VIG_Modes) and 'ignored_keys' (list of keys to ignore)
  - vi.py - VIG_Vibase
  - vigtk.py - VIG_Menus, VIG_Statusbar and VIG_Window
  - bindings/
    - ___init__.py - VIG_Registry, VIG_ModeHolder, VIG_Bindings, 'registry' (instance of VIG_Registry) and 'vigm' (instance of VIG_ModeHolder)
    - base.py - VIG_ModeBase
    - A file for every mode - The available modes are defined in VIG_Modes
  - actions/
    - __init__.py - VIG_Actions and 'act' (instance of VIG_Actions only to be used by the modes when registering their bindings)
    - A file for every module of actions - The available modules are defined in VIG_Actions.MODULES

Currently there are 8 action modules :

- blocks.py – Code to select/change blocks (i.e. Everything between a { and a })
- ex.py – Rules for expression mode

- fileOperations.py – Operations that can be done on the file
- insert.py – Functions to insert in particular places
- lines.py – Functions focusing on lines of text
- others.py – Random collection
- text.py – Functions focusing on the text
- trace.py – Printing module (allows coloured text to the terminal and control on verbosity)

And at the moment, there are 15 different modes. (I am yet to create documentation on these). All the modes inherit VIG_ModeBase, which is found in bindings/base.py.

VIG_Window

VIG_Window will attatch and detach an instance of VIG_Vibase to each gedit view as they are added and removed. It also holds a reference to the registry, which is used by vibase to determine what bindings are available, and used by all the modes to register their bindings.

VIG_Vibase

This will set up event handlers for when the mouse button has been released and for when a key has been pressed.

When it realises the mouse button has been released and it is in either selection or command modes, it will determine if their is a selection of text. If there is, it will enter selection mode, otherwise it will enter command mode.

When it realises a key has been pressed, it will firstly trace this event, determine what message to set in the statusbar and set it. Determine if any modes have defined any rules to use on this event and execute any.
It will then determine if esc has been pressed, if it has, it will enter command mode.
Otherwise it will see if it's in selection mode and a directional key has been pressed, in which case it'll enter command mode.
Otherwise still if the event is an ignored key, it won't proceed any further.

If it makes it this far it will start looking to see if any bindings exist for this event. Look at the processKey() function in VIG_Vibase for more info on that.

Outside vibase, objects can register rules to be used when processing key presses, and extra messages to be displayed on the statusbar.

To register a rule, the setRule() function is used. It accepts a number expressing the number of keypresses this rule will last for, and a function that accepts two parameters, which will be called for each event.

To register extra messages, the setExtraStatus() function can be used. It also accepts a lifetime and function.

Something to note here is that many rules can be registered at any time, but only one extra message can be set at any one time.

VIG_Actions

This is the heart of the plugin. From an instance of this (that belongs to a VIG_Vibase object) can be used to access everything else. An instance of this is passed to everything to establish context (as in which view is active)

It uses a __getattr__ function so you can access many things using dot notation. As well as been able to access action modules by name, the following list is what can also be accessed through an instance of VIG_Actions.

- ex – an instance of exManager which exists in actions/ex.py
- fileops – the fileOperations.py module found in the actions folder

- static – the static.py module found in root folder
- modes – static.modes
- keyboard – the VIG_Keyboard instance found in keyboard.py
- pos – the VIG_Cursor instance found in cursor.py
- gtk – the gtk module
- gdk – gtk.gdk
- getmenu – A lambda function that returns a lambda function that can be used by modes when registering bindings (when there are no VIG_Window instances, and hence no instance of VIG_Menus

The following are also available, but only if VIG_Actions is instantiated with an instance of VIG_Vibase (so they aren't available to the modes when registering bindings)

- vibase – the active tab's VIG_Vibase instance
- doc – vibase.doc
- bindings – vibase.bindings
- mode – the current mode
- vigtk – the current window's VI_Window instance
- menus – the current window's VI_Menus instance

VIG_Modes

This is used to define what modes are avalable. It is also used to get strings and descriptions representing each mode.

I've defined a special __getattr__ function such that you can get a string representation for each mode by accessing the mode name using dot notation. It will also raise an exception if you try to access a mode that doesn't exist. To demonstrate with an example :

```
print act.modes.ex
#will print "ex"

print act.modes['ex']
#will print "Expression Mode"
```

This cannot be changed at runtime.

VIG_Bindings

This is used to gain access to the functions defined by each mode, change the current mode and also determine what the current mode is.

To change the mode, I have defined a special __setattr__ function. To describe with an example, to change to command mode, all you must do is

```
act.bindings.mode = act.modes.command
```

If a mode accepts an option when it is introduced, then you can set act.bindings.mode to a tuple (or even a dictionary) and it will use * and ** magic to convert the tuple/dict into positional/named arguments in the desired mode's introduce() function.

Reading act.bindings.mode will return the current mode (vibase.get_data("mode"))

And act.bindings.vigm will get the instance of VIG_ModeHolder so you can access functions defined by each mode.

VIG_Registry

This holds all the bindings set by the modes.

Each bindings is set using the register function, which accepts three positional parameters and 8 other named parameters. The three positional arguements are mode, function and keycode. Note that the each VIG_ModeBase has a register function that handles the first positional parameter for you. Function is a callable object that is called if the specified binding is activated; and keycode is the key that must be pressed to activate this binding.

The other arguements are

- final – A boolean that specifies whether vibase.number and vibase.numlines should be reset after this binding is activated
- repeat – A boolean that specifies whether this function is allowed to be repeated (i.e. Whether pressing a number before this binding will have an effect)
- after – A string representing the name of the mode to introduce after this binding has been activated.
- pos – A boolean that specifies whether the position of the cursor should be the same after the binding has been activated as it was before.
- stack – A string representing what must be in vibase.stack to activate this binding.
- IgnoreStack – When vibase is looking for a binding, it will first use the stack in the keycombo. If it doesn't find a binding, it will look again, but without the contents of the stack. If it then finds a binding, it won't use it unless ignoreStack is set to True.
- control – A boolean specifiying whether control has to be pressed for this binding to be activated.
- meta – A boolean specifying whether alt has to be pressed for this binding to be activated.

VIG_ModeHolder

This holds an instance of all the modes (which modes are instantiated is defined by VIG_Modes).

I've done this so I only have to import them in one place.

VIG_ModeBase

This is the base class for all the modes. It defines nine functions.

- __init__ – When a mode is instantiated, two arguments are passed in, registry and mode. Mode is the name of the mode, and registry is an instance of VIG_Registry. __init__ creates three instance variables, self.registry, self.mode and self.fr. Self.fr is a dictionary of "final" = True and "repeat" = True which is used for convenience in a binding. Finally, __init__ will call the setup function.

- Setup – This function is intended to be overwritten such that it can be used as the point to register bindings.

- Introduce – This function is called whenver the current mode is set to this mode. It will then call intro(), trace() and status(). It is not intended to be overwritten.

- Intro – This method is used as a point to do anything that needs to be done when the mode is introduced. By default it empties vibase.stack, sets vibase.select to False and makes sure no text in the current document is selected.

- Trace – This function is used to trace any information to the terminal.

- Status – This returns a string which is then used as the text in the statusbar.

- Handle – This function is called when this is the current mode, a key is pressed, and no binding is found.

- Nop – A function that does nothing. This allows a binding to be made that does nothing but

introduce a new mode. (the after option in the binding doesn't work if the function is set to None)

- reg – A convenience function that calls the registry's register function with the current mode's name as the first argument.

VIG_Cursor

This provides functions to be used for moving the cursor.

VIG_Keyboard

This provides functions to be used for emitting events and checking events for use of modifier buttons (i.e. Ctrl, alt and shift).

VIG_Statusbar

This is used to provide an interface to the statusbar. Thus far it only provides an "update" function that is used to set the text in the statusbar.

VIG_Menus

This is used to gain access to predefined menu items. For example, vibase.menus['save'].activate() will activate the 'save' item in the file menu.

Timeline

When the plugin is instantiated, it imports VIG_Window from vigtk.py

In vigtk.py, registry gets imported from bindings/__init__.py

Inside bindings/__init__.py, act is imported from actions/__init__.py

inside actions/__init__.py an instance of VIG_Actions is created and all the actions modules are imported and held in that instance.

Back in bindings/__init__.py, an instance of VIG_Registry (registry) and VIG_ModeHolder (vigm) are created.

Then all the modes are imported and stored in a dictionary and vigm.start() is called.

Vigm.start() will then instantiate each mode, passing in registry and the name of the mode (which means all the bindings are registered in registry).

This finally leads back to VIG_Window which will then start creating instances of VIG_Vibase, which will the start looking for keypresses and go from there.