Features

Listing the most important **features** of the DiaShell:

```
# - backup audio, or data, to usb or hdd.
# - Compiles apps and chain_compiles even more apps.
# - Type news to open some webpages for news stuff.
# - Vim "support", which means you use vim to edit stuff.
# - Trigger certain events.
# - On startup of this shell, you can define some actions which
# need to be done, i.e "rdia rsong" to play random songs. More
# about this, please see the section here called "startup".
# - ISO Creation, CD and DVD burning.
# - ri-like browsing behaviour.
# - Open your favourite files by specifing something like "blxorg"
# - Open a local page in your editor by issuing "rf keyword"
# - InBuilt help file.
# use "songs" to view your local songs and songs 44-48 to play song 44 up to including song 48
```

It really has a lot more features, it is best to look at the page here - have a look around.

It should include all what is necessary - and if not, then I will add this information here.

Design Statements and Philosophy

@aliases should contain ALL aliases.

Every string data that the shell gathers - before it is sent to the user - will be stored in a variable. This way you can tap into this variable and make further use of it, for whatever reason.

@this_file points to the last assigned or otherwise used file. Only this, and only one variable may refer to the last used file.

History Support

The DiaShell will log all commands that you enter (this can be turned off by setting **enable_history_logging** to false in the config file; you can do this at runtime if you do **disable logging** or **no logging**).

To access this information of your history, you can do:

```
h
history?
```

This will then display the history.

You can also use a position of the history at, i.e. place 5, like this:

```
history 5
```

This will substitute the history command of position 5. In other words, this allows you to use history you already did input at an earlier time.

Please note - we wont append to the history log if the last command entered is the same as the last entry in the history log (keeping duplicates is not so useful).

Sitemap

You can empty/clear the history any given time by doing:

```
empty history
```

You can also disable logging of the history by doing:

```
disable history
```

And of course you can disable history support on startup of the DiaShell as well:

```
rda --nohistory # or one of the aliases towards --nohistory
```

We log all commands into the file **last_commands.yml**.

Gzip

You can compress a string using gzip/zlib:

```
gzip bla
```

However right now I can not think for a good usage scenario for this ...

Generate Part

To generate a new page, issue:

```
generate BIOS.cgi
```

This will simply generate a new .cgi file.

Feedback Part

To get feedback about every Xorg component, do this:

```
gfeedback all_of_xorg
```

Note that this will also generate a .html page.
As usual, using "*gfeedback help*" or "*gfeedback ?*" will give you some help about this method.

To feedback the version of a program, do this:

```
programversion cups
```

To feedback the available programs, do:

```
aprograms
```

You can also get feedback about the documentation of Ruby.

```
docu? Array#sort
```

The above would use Ri to feedback information about Ruby.

This syntax here also works:

```
docu? Array.sort
```

And you can also grab the documentation for .rb files:

```
docu? foobar.rb
```

Startup

The DiaShell will allow some things to be done at startup time.

Please keep in mind that there are aliases possible for your convenience. For instance, the option nocolours and nocolors are synonymous. Also, you can use either - or --, for instance --nocolours will be the same as -nocolours.

You can decide what you prefer on your own.

Now here the list of currently supported Commandline-Options:

--version	**Display the version number of this shell.**
--nocolours	**disable colours.**
--run_method	**run a specific method, then exit.**
--norc	**do not load the configuration file on startup.**
--extract	**simply extract something, with the aid of the DiamondShell, such as: rda extract all.**
--help	**show the various help options, then exit**
--silent-startup	**we will be rather silent on startup**

You can also invoke a specific method from the DiaShell from the commandline.

Example:
```
rdia --runmethod run_pkg_config
```

You can also get tab completion on the commandline.

For instance, type this:

```
rda --<TAB>
```

This will complete to --help

This will also work if you wish to tab-complete on the current files in the working directory of your shell.
So if you have a file called foo.txt, this will work:
```
rda fo
```

Call any method

You can call any method by its name from within the DiaShell.

The **general syntax** for this procedure is:

```
cmethod name
```

Examples:

```
cmethod foobar # would call the method foobar, if it exists.
cmethod clear_history
cmethod preformed_action?
cmethod history?
cmethod upload_screenshots?
```

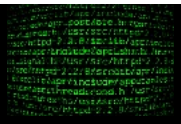
method and cmethod works the same, method is just an alias to cmethod.

To list all available methods (which you could then call with **cmethod**), do this:
```
all_methods
```

Also note that you can get feedback about a method since August 2015 if you pass the name:

```
method? yel
method? clear_history
```

Compile



If you want to get a list of all programs which can be compiled, do this:

compile help

You can compile i.e. nano by doing:

compile nano

Weekday

You can find out the weekday of any date like so:

wday 13.11.2011
This day is a Sunday.

wday today
This day is a Sunday.

wday today?
This day is a Saturday.

Prompt

You can modify the prompt by doing something like this here:
prompt foo

Some options exist in regards to the prompt. For a listing of these, query via help, such as this:
prompt help

The Disco Prompt may be fun to try out.

You can set a mixed prompt, meaning that directories will be colourized as default, but the / between them will have another colour.

Let's look at a screenshot for this:

```
/Depot/Audio/
```

And now the code for this:
setprompt mixed

If you wish to clear the prompt again, do this:
prompt nil

Triggering Events

You can trigger (and register) certain events. For instance.
in 5 sekunden trigger wecker
Would register this action to be run in 5 seconds from the time of entering the input..

A german variant of this works as well:
in 3 sekunden starte wecker

Aside from triggering events, we also have some event-like options that can be used to configure the DiamondShell.

These can be stored in the configuration file **events.yml**
For instance, the event :on_remove_directory will invoke a cleanup script, whenever a directory is removed under the /Programs hierarchy (or more accurately, whatever value was set to PROGRAMS_DIR)

Information about Programs

You can get information about programs like this:

pinfo libassuan

You can also use purl, which gives you the url of a program:

purl wine

And to also open this URL, do this:

purl wine open

This will also set the variable **@last_url**.

In fact, anything that stores a URL should also keep this information in @last_url. At any time you can retrieve this URL by issuing something like:

open_url
openurl

Note that certain other commands also set this url. A rule of thumb is, that whenever you notice an URL output, then this variable will also be set, and you can then simply use openurl on it easily.
Also note that two commands exist here "url" and "purl". The current way is that if you use "url", and we do not find an URL for it, we will automatically invoke "purl". I added it because gaining information was more convenient to me if I only have to type once.

Wecker and Time

You can set a Wecker, like when you want to wake up at a specific time.

This wecker formula is quite flexible right now:

wecker 6 std 8 minuten
wecker 855 minuten
wecker 2 tage 8 stunden

Time-related queries also work nicely.

For instance, if you wish to find out when the next Weekday is, try:

next sonntag
next sunday

Serve local pages

You can serve local .cgi pages, just do this:

serve foobar.cgi
serve PC.cgi
serve /Users/x/DATA/SCIENCE/THINKING_METHODS.cgi
serve http://localhost/GAMES/DSA/DSA_COMPUTERSPIELE.cgi

By default this will output the html code onto your console. Additionally, it will save the result into a .html page, within the directory of the DiamondShell log directory.

Notification Format and Notifications

notify.rb

Right now the format is rather primitive.

One can use this:

At 06.07.2008 15:00:34 send "hey your kitchen burns"
In 20 seconds send "hey your kitchen is really burning!"

- If you omit the time, we will assume **12:00:00** as the default.
- If you use "in 1000 seconds" then we will calculate the proper time automatically.

Open

The command **open** allows you to open files.

Some keywords are used for it, for example:

```
open last
```

Would open the last file. This may be useful when you have opened a local page via some shortcut and want to refer to it. In this case, "last" refers to that file.

This works also for programs and their project URL, for example try this:

```
pinfo php open
```

This would give you some information about php, and open the webpage in a browser.

Popular open-related actions include the following:

```
open wikipedia
open pdf
open html
```

Note that you can also open URLs - in this case, we will fire up the browser to load it.

You can open the last assigned file by doing either of the following:

```
open LAST_FILE
open ?
```

Note that we will try to glob towards an existing file if we did not find a valid file to open otherwise.

Create something with the DiaShell

To **create a webpage**:

```
createw location
createw $IMG/FUN/ANIMALS
createw $IMG/NJOY/
```

You can also give a file and split it up. Example:

```
createw TANRIEL 3
```

Would split this file into 3 .html pages.

You can also create simple files. In UNIX Environments, you commonly use "touch file_name" to create a file.

This works when you use quotes too, for instance:

```
touch "foo bar"
```

Will create a file called "foo bar", including the space.

You can also create multiple files like so:

```
touch {temp1,temp2,temp3,temp4}
```

The DiaShell also tries to be "smart" when it notices you want to create a file. For instance, when you try to create a .rb file, we automatically insert a program skeleton into that .rb file.

This behaviour can be adapted and changed by you.

Disable

Currently if you want to generate certain files, such as **foo.rb**, we create a default layout for that.

This is done by the **class CreateFileSkeleton**.

This is usually useful, but can sometimes be annoying. In that case, you can simply disable the autogeneration via:

```
disable autogeneration
```

To find out whether we will autogenerate files/content for files, do this:

```
autogeneration?
```

To show all available actions for disable, do:

```
disable HELP
```

Burning

The DiaShell currently relies on i.e. libburn or cdrskin and growisofs to burn CD and DVD iso files.

You can simulate a "test" run issuing:

```
burn test
```

The program to burn CDs and DVDs can be set in the configuration file.

You can get its current value by doing:

```
dvd_burn?
```

Alternatively, you can set the run_simulation variable to true like so:

```
run_simulation = true
```

Creating ISO files

You can create .iso files with the DiaShell easily.

One way is to call the following method:

```
create_interactive_iso
```

You can also directly create an iso by doing this:

```
make_iso /path/to/directory/here
make_iso /tmp
make_iso /Depot/Games
make_iso /Users/x/DATA
```

System Information

To get System Information do this:

```
system?
```

This will use the following class:

```
SystemChecker.new
```

Input

A few words should be given about user input to the Diamond Shell.
These may be useful to understand certain design decisions.

- Everything that comes after the first encountered **#** character will be regarded as a comment.
For instance:

```
exit # we will exit here. What comes after the # is ignored.
```

The above command would assume that the user wanted to exit the shell, and the dia-shell will ignore everything that comes after the # (and the # character will also be ignored)

This is basically the same behaviour you can find in other shells as well.

The character **;** is treated as an **instruction_separator**, unless it is found as part of an URL. For example:

```
cd $J; ll
```

Is assumed to be two different commands: first a change_directory instruction, then a listing_content of a directory instruction. This is consistent with how other shells act.

Recipe Creation

To create a recipe interactively (i.e. write up information about it in a step-by-step fashion), run this command:

```
interactive_recipe
```

Superuser

To change the superusername (on UNIX like systems), do this:

```
become new_name
become god # now the superusername would be called "god"
```

You can also show which users are available no a system.

To do this, do:

```
users?
```

And you can find a specific User ID, such as User ID 11:

```
user 11
```

All

You can use a special keyword "all" as alias towards *.

That way you can apply a common operation on several file objects, for instance:

```
tovilla; bl 4/all*
```

This would open all files within the directory at position 4 in the bluefish editor (or whatever else is your defined editor under @config.editor).

> In general, "all" is considered a shortcut for "*", unless a file exists with that name in question.

Program URLs

To get the URL of a program, do this:

```
programurl wine
programurl ruby
programurl php
```

Colours

The DiaShell will use colours if you specified that you want to use colours. This can be used inside the DiaShell through the method called **use_colours?**.

This option can be customized within the config file and it can also be changed at run-time (see the coming section how to do that).

If you do not want to use colours, set the variable **use_colours** to false. This can also be done by issuing

```
disable colours
```

in your **running shell**.

To enable the colours, logically, you can do **enable colours** again.

Currently, directories and files will have different colours, as defined in the config file:

colours.yml
cat $DIA/YAML/colours.yml # Lets display the content.

To get a listing of which colours are used in the shell, do:

colours?

This will display the colours in use too, for instance:

```
And the colours for use in the DiaShell are:

    {"colour_for_end"=>"white",
     "colour_for_warnings"=>"redb",
     "colour_for_normal"=>"whitebonblack",
     "colour_for_fancy_content"=>"darkgreen",
     "colour_for_directories"=>"brown",
     "colour_for_arguments"=>"red",
     "colour_for_important_content"=>"teal",
     "colour_for_files"=>"lightblue",
     "colour_for_symlinks"=>"lightgreen"}
```

You can then set a new colour live, by doing something like:

colour_for_directories = red

From this point on, all directories will be displayed in red.

For fun and giggles, you can also randomly switch the colours, just to try out how things will look.

For this, do:

switch colours
rotate colours
switch
rotate
scolours

ASCII

You can use ascii art too:

to_ascii some_image.jpg
to_ascii 1
to_ascii 2
to_ascii all

Just try these.

In case we do not find any match, we will try to use a glob(), before we fail finally.

IRC

You can use IRC from the DiaShell.

To do this, do:

irc channel_name_here
irc ruby-lang
irc ruby

Exiting the shell

To exit the shell do one of these:

q
exit

Passwords

To clear the password of your user account, do:

nopasswd

This will simply delete the password.

To show all available passwords, you may also do:

passwords?

Fetching all .ogg/mp3 files in a given dir

all ogg
all mp3

Of course you can apply other criterias as well, such as "all php" or something.

The result will be stored in **@file_listing** which you can manipulate lateron.
To retrieve it, simply type it in the shell.

@file_listing

Since Sep 2011 the following also works:

@_

Since Jan 2012 we allow to shuffle the dataset stored in **@file_listing** by doing:

shuffle

Downloading

You can download a package like so:

download http://ftp.gnome.org/pub/gnome/sources/pygtksourceview/2.3/pygtksourceview-2.3.0.tar.bz2

If you want to download **to** a specific directory, use the "**to**" keyword like this:

download http://ftp.gnome.org/pub/GNOME/sources/gtk+/3.15/gtk+-3.15.4.tar.xz to /Depot/j
download http://ftp.gnome.org/pub/GNOME/sources/gtkmm/3.17/gtkmm-3.17.50.tar.xz to /Depot/j

Note that this can work with special targets/variables as well for your conenvience. For example:

url = http://www.calno.com/evilvte/evilvte-0.3.1.tar.gz
download url to 1

would download to the location saved at slot 1, which defaults to ENV['MY_SRC'], or expanded, would become /Users/x/SRC on my system.
url in this case is a variable which was asigned to some url before.

Whenever we download something, we will also set the variable @last_downloaded.

You can then use this variable to manipulate the file in question.
For example, to download, and then extract the file, you can use this:

download ftp://ftp.astron.com/pub/file/file-5.03.tar.gz to /Depot/j
extract last_downloaded

Also note that you can download Youtube videos by just passing in the URL of it.
The DiaShell will automatically try to download and also extract the audio file into a .mp3 file.

Displaying File Content

Use cat or get_file_content to display the file content of a file .

For example, if the file is called **foobar.txt**, do this:

cat foobar.txt

You can specify a range:

cat file 10-20

This would display only line 10 up to 20.

This works only on files, i.e. not on directories.

Also note that this slicing for files should work in general, and can even be extended. Look at playing video for an example.

You can also read a header, via:

```
header foo
header 5
```

How many lines will we display that way? By default we will show the first 25 lines, but you can change this setting, by doing something like:

```
header? 10
```

This setting can be saved, by issuing **save**.

Note that if the file is a .rb file, we will try to use the Coderay gem to parse the result.

As of August 2014, we will also allow you to use the last file that was assigned when you use ?:

```
cat ?
```

show-source

You can show the source of a method of the DiamondShell by doing:

```
show-source run
```

undo

You can undo certain actions. Right now only file-move actions can be undone. For instance, let us assume that you have a file called "test" in the directory /tmp, and that you moved this file to the directory /home, like so:

```
mv /tmp/test /home
```

Now you can do this command:

```
undo
```

And the file will be moved back to its old place.

This is not extremely useful yet, but I will probably add code to allow a more general purpose undo action at a later time.

Move

You can move files and directories to new locations.

This can be done like so:

```
mv foo /tmp
```

The above command would move the file foo to the directory /**tmp**. This is similar to how bash or zsh acts.

Note that you can also move multiple files.

The syntax for this would be:

```
mv temp{1,2,3,4} $JJJ
```

This would try to move these directories or files to whatever value is stored in $JJJ (which, on my system, points to the directory /Depot/jjj).

Screenshot

You can use `scrot` to make a screenshot. A common alias is:

```
shot
```

The first argument is an optional name for the .png file:

```
shot my_desktop
```

The second argument is an optional delay (in seconds)

```
shot my_desktop 5 # 5 seconds delay
```

Note that as of April 2014, we will automatically upload that screenshot to a remote host if the environmental variable IS_ROEBE is set to 1.
The code for screenshots is stored in the file **screenshot.rb**.

Ruby keywords

To get a list of all ruby keywords, do this:

```
keywords?
```

Saving

You can save to a file, for example:

```
save to $J/foo
```

This would save whatever is inside **@usher.string** into this file. (By default it will append, though this behaviour may change lateron.)

Merging files

Merging ogg files can be done like so:

```
merge x.ogg file1.ogg file2.ogg file3.ogg
```

You can omit arguments too as in:

```
vobmerge # will pickup all .vob files in the directory
```

Ftp

Now let's talk a bit about the FTP functionality in the DiaShell.

You can upload a whole project by issuing this:

```
ftp_upload project dia
```

Of course that task must have been registered somewhere first.

To show all available tasks, do either of these:

```
show available tasks
stasks
```

Birthdays

I display upcoming Birthdays.

Currently this is set to 5 days in advance.
It will report in ways like:

```
We found these Birthday candidates:
    Wolfgang Edlinger "Edl": 23.01.2012
```

To query this, you can do this:

```
bday?
```

If you wish to look n days into the future, you can pass it as an argument:

```
bday? 30
bday? 120
```

Aliases

All aliases are kept in the instance variable **@aliases**.

You can display this variable any time by issuing either of the following two commands:

```
@aliases
aliases?
```

To get the content of an alias, either type this:

```
alias name_of_alias
```

alias pwr

Positional arguments are also possible (if an alias with that number does not exist, that is):

alias 567

Or simply type the following to get all these aliases:

@aliases

You can of course also unset an alias. I recommend you use this syntax:

unalias name_of_alias

unalias pwr

To set an alias, you can do something like this:

alias foo=55

This would set the key **foo** to the value of 55.

As of January 2014 I can download my remote aliases by issuing:

download_aliases

The Clipboard buffer

The DiaShell uses a **Clipboard buffer**.

This file is stored at **CLASSES/clipboard.rb**.

You can get the current value it has via:

buffer?

b?

If you want to assign it to a specific string, do start this with the token:

{

Yup, this is the start command for the buffer. I thought that using {} as delimiter to enter into the Buffer directly would be nice, so that is why I added that featre.

Also note that since December 2011 we will additionally try to use xclip when using the buffer. This way, you should be able to manipulate the buffer on your own.

You can clear the buffer any time by doing:

clear buffer

Since **March 2012** we also use a file called **buffer.yml**. If this file exists on startup, then we will load it into the current buffer.

Note that we also use the Xorg buffer on Linux.

This Xorg buffer understands some special commands such as:

buffer generate ruby

This will generate ruby code.

You can store this buffer to a file, as in the following way:

save buffer to $J/jtj.rb

Index

By default, DiaShell assigns every entry in a given directory a number, the so-called *index number*.

This number is displayed on the left side usually, and it may be used by you as a shortcut to access a specific file as well.

For example, if you have two files in a given directory, called "long_name_of_a_file_1" and "long_name_of_a_file_2" you can use numbers to work with them instead, like so:

cat 1

cat 2

The following small screenshot shows this - the *index number* can be seen in green colour:

```
-rw-r--r-- root   (1)     0  B 2015.03.13 11:51:36  /Depot/jjj/long_name_of_a_file_1
-rw-r--r-- root   (2)     0  B 2015.03.13 11:51:39  /Depot/jjj/long_name_of_a_file_2
drwxr-xr-x root   (3)     4 KB 2015.03.13 11:51:42  /Depot/jjj/test_directory/
```

cp 2 $J # This would copy the file at second position to whatever value is contained in $J.
On my system, this defaults to the directory **/Depot/j**.

You can turn off showing the index in your config file (look at the configuration option **display_index**):
disable display_index

Query .pc Packages

You can query packages installed on your system by appending a **?** to the name, for example:

libpng?
mono?
xtables?

This will not always work, but it will probably make your life a bit easier sometimes.

Appending to a file

As it is the case with other shells, you can use >> to save something to a file:
foobar >> this_new_file
This would save foobar in the file `this_new_file`

As a reminder, in Bash it works a bit differently, in that the stream is redirected. In DiaShell, we use a simplified approach.
Also note that you can use macros here, such as the **@buffer** macro.
For instance, if you do:
@buffer >> foo
Then we will store the content of the buffer into the file **foo**.
Note that you can also use > which will create a new file and then add the content to that file.

Copy

You can copy a complete directory. By default, we will copy to the current working directory if no argument was given.
You can test this by doing something like:
cp to $JJ

You can also copy whatever is stored inside **@file_listing** to another target. The following example shows this, by first entering to your directory containing all audio songs, then fetching all .ogg files, then copying them to whatever is held in the environment variable J:

cd $SONGS
all ogg
cp to $J
cp here to # this willc opy the content of @_ variable to the current working directory
cp to here # Same as above
cp all to $JJ # this copies the content of this directory to the environment variable $JJ
cp all to here # Copy to the current directory

More about the environment can be read here.

Please also note that the variable @target_directory can be set to a specific directory like so:
target /Depot/Song
The effect is that this variable can now be used to copy files or directories to this target. So after you have done the above, you can then do this:
cp /Home/OldSongs/
And it will, **in this situation**, copy it to this directory. If this was not set, a notification will alert the user that the target directory is missing in this action.

Of course, if you manually give another target, then we will use this instead.

If for some reason you want to copy all images under a specific directory to a new target, use this:

```
cp_images
```

Note that this will copy to the directory you are in.

Useful to have is that you can also assign all current files in a directory, and then copy these to somewhere else even at a later time. This works like so:

```
assign all
cp to here
cp to $J # you can use absolute targets too, of course.
```

Environment

To retrieve the value of an environment variable, do something like this:

```
e $AUDIO
```

Note that if you input just $AUDIO then we will cd into that variable if it is a directory.

The key lies in using $ to denote the variable in question, and e is used to echo the content.

If the variable was not found or defined, we will display a little helpful note to the user.

Example:

```
e $A_VARIABLE_THAT_REALLY_DOES_NOT_EXIST
```

Also note that if you do something like:

```
e $L*
```

Then we will try to feedback all environment variables starting with L.

To feedback all known environment variables, do this:

```
environment?
```

Help

The helpfile section can be invoked via:

```
help
```

The colouring of the helpfile section is done via the class DiamondHeader.

Additionally, we can display useful help information before we startup.

We do this via:

```
rda --help
```

Below is a (partial) screenshot from how the Help Menu looked in April 2014:

```
cd_burn                 # to burn a cd
clock                   # to display a simple ASCII Clock
compile                 # to compile a single program [Example: compile php]. Use compile ? to get help
contacts                # to display some contacts
create_large_file       # <FILE_NAME> to create a large file (Current size defaults to 200 MB)
create_zip              # to create a .zip archive
cure                    # to repair your system
defaults                # to revert to shell-defaults again
default_action          # to do your preformed action
description             # <name> to get the description of a program from cookbook.yml
downcase                # to downcase all files in the current directory.
download                # <name> to download files (just like wget)
dvd_burn                # to burn a dvd
english                 # to ask an english word
ex                      # it to exit the DiaShell
extract                 # <file> to extract a package, no matter its format
fetch_form              # to fetch the first HTML formular
find                    # <something>, as in [find all pdf]
ftp                     # to ftp upload something
755                     # 555, 644 and so forth to chmod your files (change permission and access level)
generate_feedback       # to feedback help about programs, ie: gfeedback games
gtk                     # to enter the GTK shell (which provides gtk apps)
generate_random_image   # <name> to generate a random image with some text
grabc                   # to grab a colour (delay: 1 second)
include_program?        # <name> to query whether a program exists in our "database"
install_grub            # to install grub
install_nvidia          # to install nvidia
irb                     # to start irb
irc_log                 # to append to QUOTES file (IRC Quotes)
keyboard                # to set your keyboard, i.e. to another layout
keywords?               # to show the ruby keywords
make_iso                # to generate an .iso file
mount_cd                # to mount a CDROM
new_tab                 # to open a new tab in konsole
```

As of September 2014, I have added documentation for some commands. This documentation is stored in the **class DocumentedHelpOptions**.

You can get them via a leading **?** e. g.:

?touch
?grabc
?symlink

Gist

You can use Gist from the DiaShell, via the gist gem.

Example:

gist foobar.rb

This would upload the foobar.rb file to gist github.

Modes

There are several modes in the DiaShell, even a hidden mode - the play mode.

If the play mode is active, we will try to play all video files in the currenty directory.

You can set a specific mode by doing something like this:

mode = editor

You can query the currently active mode like so:

mode?

This would output "edit" in the above example.

Mode-specific code can be found in the file **mode.rb**.

Viewing a Map

You can view a map (Stadtplan):

```
stadtplan landstrasse 22
```

Right now this defaults to Vienna, my hometown.

If anyone requires this for another town, well. Write to me so that I can add it.

Show something

You can show files by doing something like:

```
ll
```

By default this will not show hidden files. There are at least two ways to show hidden files though, since obviously this is an important component of working with a filesystem.

The first way is to explicitely tell the shell once that you want to see hidden files. You can do this by doing:

```
show hidden
```

From this point onwards, you will always see hidden files.

The other way is to use a switch, such as:

```
ll --almost-all
```

This is akin to **Bash**.

You can show the content of a file via **cat**, as in:

```
cat foo.rb
```

If the file **foo.rb** does exist, this command will show you the content of that file. If coderay is installed, and if you enabled colours, we will use Coderay on the terminal output to also display colours.

You can disable coderay output by doing:

```
disable Coderay
disable coderay # this also works
```

You can sort the output by doing something like:

```
sort_by size
```

Obviously, the above command will sort by size.

Removing something - removing files or directories

You can remove directories i.e.:

```
remove foobar/
```

protected_directories.yml

You can specify directories which should **never** be removed, within the file **protected_directories.yml**. I recommend to add the directory / to that yaml file.

You can also **add new directories** to this list:

```
protect /some/directory
protect /tmp
```

Please also note that you can specify a number as well, e. g. the following would remove the first entry in a directory:

```
remove 1
```

Syntactic sugar also exists. To remove all directories in a given directory, you can also do:

```
remove all directories
```

Note that german words also work as aliases to these actions.

You can also remove files that contain a " " character. For instance, consider a file called foo bar.txt.

In order to remove such a file, try this:

```
rm "foo bar.txt"
```

Reading content

The most obvious way for reading content is by reading a file.

To do this one would do something like:

```
read_file /location/of/file.txt
read_file /Users/x/DATA/STD.bfproject
rfile /etc/hosts
```

Some aliases exist for this, i.e. "readfile" would work also, as would "cat".

The configuration option "padding" is also used to pad the file to the left.

You can obtain the current value of padding by doing:

```
padding?
```

And you can set it to a new value by doing something like this here:

```
padding = 15
```

As of January 2014, the padding of input is done when we read in a file. This may change at a later time again.

If you do not like any padding at all, you can either set padding to 0, or do this:

```
disable padding
```

Downcasing

To **downcase all files in a given directory** (might be useful if one wants to downcase VIDEO1.VOB files for example), simply do one of this:

```
downcase all
downcase everything
downcase *
```

Also, if there is a directory called FOO and you type "cd FOo", the shell will attempt to correct this to FOO automagically.

Upcasing

You can upcase directory and file names by doing something like this:

```
upcase all directories
```

This would upcase all directories in the current directory.

Enter directory

If you want to enter a directory, you just type the name of it, or you prefix via "cd", such as in "cd /tmp".

The Diamond shell will attempt to enter the upcased version of said diretory as well if it exists, ie if you enter "foobar" dia-shell will also try to enter a dir FOOBAR. (This will happen silently - if a directory called FOOBAR does not exist, we will not generate any error message at all. This feature exists only due to convenience.)

Also note that whenever you change the directory we will register the new directory in a yaml file, so that we can remember which directory was the last one you worked in. This information is stored in the file **last_directory.yml** and you can also query this via last_directory?.

You can use numbers instead of names to enter directories.

Number 1 will correspond to the first directory, number 2 will correspond to the second directory and so on.
Example:

```
cd 3 # would enter the third directory
```

This works with english words as well. For instance, the two following examples mean the same - enter the first directory.

Both examples work:

```
cd 1
cd first
```

You can - by convenience - also enter an archive. We will assume that you want to extract it.
Example:

```
cd rpm* # where rpm* points to the archive rpm-1.0.0.tar.bz2
```

Also note that the aliases wd1 wd2 etc.. are being treated as instruction to "jump to" specific directories. More about this at the section Jumpers.

Note that every time you change directory or copy something, we set a variable called $LAST_DIR. You can use this variable, like so:

```
cp * /System/Fonts
cd $LAST_DIR
cd $LD
```

You can change directory to the last created directory by issuing:

```
cd !#:1
```

Also note that if you use cd(), and pass a number to it, we will assume that you wish to enter only directories or archives from the current directory.

Jumper

Via jumpers you can quickly change to another directory.

Expressed in another way, jumpers are " Bookmarks for Terminals ".

On startup we will load the variouos yaml files that constitute our configuration, which should also include all of our default jumpers, in other words making our jumpers "persistent".

You can add or remove entries to this list of jumpers easily, as the following example will show - *how to add this directory as a jumper-directory*:

```
jumper # will set a jumper to the current directory
```

Lateron, you can jump to this directory.

To see the list of available jumper positions, do this:

```
jumpers?
```

To jumper specifically to one of them, you can give a number, as in:

```
jumpto 3
```

Also note that you can use an alternate syntax as well, via cd:

```
cd ~3
```

or like this:

```
wd3
```

This will be treated as "jump to position 3".
It is thus a shorter way to jump to a specific directory, and "wd" will be treated as alias.

```
wd1
```

Would mean "jump to first jumper directory", wd2 would mean "jump to second jumper directory", and so forth.

Note that you can **remove** a jumper again like so:

remove jumper 6
remove_jumper 6
rmjumper 6

This would remove the jumper at position 6.
I prefer the second variant, **remove_jumper**. It seems to be a useful mnemonic.

You can also integrate different jumpers at run-time. Every file called jumpers.yml will be treated as a jumper directory.

You can add new directories with a jumper.yml file to the configuration file. If anyone wants to extend this further, let me know - personally I am quite happy with how it currently behaves. I do not use jumper-directories much myself.

Extracting

To extract something, you simply do:

extract *tar.bz2

This should work with **.lzma** Files as well.

Additionally you can specify a location, i.e.:

extract gcc-4.3.1.tar.bz2 /tmp
extract gcc-4.3.1.tar.bz2 to /tmp
extract 1 to $MY_TEMP

Both instructions will work and extract the content of the tarball to the specified directory.

To extract the audio from a Flash .flv file, do this:

eaud # or extract_audio

You can also extract all files matching a certain criteria, by doing something like this here:

extract all *bz2

You can also extract all archives given in a directory.

For example:

extract all archives

would extract all .tar.bz2 .zip .gz and so forth...

Please note that there is a configuration option that will allow you to automatically enter such an extracted directory. This configuration option is called **enter_extracted_directory**. Set it o true if you want to automatically enter an extracted tarball.

You can use this from the commandline as well, without properly starting the DiaShell.
Example:

rda extract all dontstartup

This would extract all archives in the current directory.

Record

You can record stuff with the DiaShell. Right now this is limited to *streamtuner* (Audio Streams from the internet), i.e. do this:

```
record streamripper
```

You may have to install/compile streamripper first.

Require

You can require something in the Diashell via:

```
require "sinatra"
require 'thor'
```

If it works, we will be silent. If it fails, we will report that error.

This allows you to use the DiaShell a little bit like IRB.

Glob

In the event that we do not find a certain input, we will try a glob event.

First, we try with trailing * but if that fails we will also try with leading *.

Dictionary

If you have some dictionary files in YAML format, you can have autocomplete support for them.

For this, the use_dictionary.yml file must have a value of true.

This can be disabled, e. g. via:

```
disable askeng
```

Autocompletion

Autocompletion means that if you press the TAB key, you will be able to choose from a selection.

Before explaining this further, try it. Start your diamond shell and do this:

```
extract|TAB|
```

The |**TAB**| notation here means "**press the tab key here**".

You will get a selection between **extract_audio** and **extract_video** and possibly **extract_it** as well.

Press v and then TAB again. You will have selected video.

The file that handles autocompletion is readline.rb, which you can find in the base/ subdirectory.

Note note that we allow you to autocomplete to the list of available instance variables.

For instance:

```
@al
```

Would list which instance variables with a name starting with @al exist.

Additionally, all entries found in $PATH will also be completed.

As of August 2014, we will also add support for some completions of the Ruby stdlib.

Since as of September 2015, we will tab-complete on leading **@configuration.** entries.

Editing 

To find out which editor is your current main editor, do this:

```
editor?
```

You can switch to another editor by doing this:

```
seted bluefish
```

This setting is persistent, and will be stored in tje file **editor.yml**.

Also note that since September 2011, if you try to open a directory, we will now open the content of the directory. (Before September 2011, we used to display an *error message* instead.)

You can also open based on the class name, if the file can be found.

For instance:

```
bl BirthdayNotifications:
bl BirthdayNotifications
```

The above will try to find a file called birthday_notifications.rb.
If it can be found then we will open it in the editor.

Note that as of 31.07.2015 you can also open a file based on the name of a method defined, if this method is part of the Diamond Shell Project.

Example:

```
bl show_completable_commands
```

This would open the file show.rb because the above method is defined in that file.

Path 

The Diamond shell will use the $PATH environment variable, if it exists.

All binaries known there will be used for both the completion, and as known-executables. This is similar to how other shells handle the $PATH variable.

Audio 

The DiaShell simplifies dealing with audio data in quite some ways. Please read this section carefully if you wish to manipulate audio data with the DiaShell in any way.

You can remove a comment in an OGG File (**.ogg**) by issuing this:

```
remove_comment your_ogg_file.ogg
```

If you want to play several songs and use a range specification, do this:

```
77-99
```

This would then play the songs from position 77 to position 99 in the current directory.

Also note that since Sep 2011 you can issue the following command, and we will play the song found in /Depot/Audio

play song 33

Also note that when you play audio files, this is stored in an array.
To find out the last song played, do this:

last_song?

Since Nov 2011 we will try to glob whenever we try to play a multimedia file.

You can convert Audio Formats to one another. For instance, to convert a .mp3 file to a .wav file, you can do this:

```
assign foo.mp3
convert_to wav # Convert to .wav format.
```

You can use the DiaShell to elegantly *slice .mp3* files.

Currently we rely on *ffmpeg* to do these conversions. Now, how can you **slice** a .mp3 file with the DiaShell?

Say you have a mp3 file, and **file?** also reports this.

Specific Example - we first assign a .mp3 file as the main file to work with:

```
assign Strack_MI1Theme_U2.mp3
file? # => "Strack_MI1Theme_U2.mp3"
```

Now you can then specify the start position:

cut_start 15

And then the end position

cut_end 45

in Seconds.

To start the actual cutting,  do:

cut

To feedback the positions, do:
cut?

This makes use of class CutAudio, which you may be able to use standalone (thus, without the DiamondShell as well).

You can also play all audio files in a given directory.

In order to do this, try this here:

play all audio

This is almost the same as play * except that we will know which files are audio files and which ones are not.

You can also add or remove entries to the favourite audio file listing.
To do this, do:

```
add_fav foo.mp3
rem_fav foo.mp3
```

To show your favourite audios, do:

show favourite audio

show_favourite_audio

Either of these two will work fine.

Conversions

You can use the DiamondShell to convert certain formats into one to the other.

The following section shows how this is done.

To convert 55 celsius to fahrenheit, do this:

convert 55 celsius

You can also output a list of Celsius to fahrenheit table (convert into a table):

convert 10-100 celsius

If you want to convert audio tracks  , just use this:

convert *mp3 towav

This would convert all mp3 files to .wav files.
to_wav would also work.

To compare mp3 into ogg, do:

convert *mp3 toogg

Please also note that the DiaShell can try to automatically convert numbers to file names or directory names.

This setting can be changed in the config file at:
autoconvert_numbers: true/false

Variables explained

@_ The variable @_ is a throwaway variable. It can be used for pretty much any data. I often tend to assign all files in a directory to this variable.

Assignments in the DiaShell

The DiaShell allows you to assign to a main file.

In order to do so, you can assign to the main file like this:

asssign foo.txt

Now, in several command actions, like copying, you can omit typing the name.

copy to here

The above command would copy the main file to the current location, aka Dir.pwd (current working directory).

We will do this automatically for at the very least these actions:

copy a file
create a new file
when we play an audio file
when we open a file in an editor
whenever we stat a file
whenever we match to a local URL
whenever we open a file in the browser

whenever we append to a file
whenever we use jp2a
whenever we play a video file
whenever we trigger an action
whenever we create an iso
whenever we read a file

Collapse

This means to display the full path to a given file or directory.

If collapse is **true**, then we will NOT display the full path to a given file or directory.

You can change this behaviour just as any of the other configuration values.

Video

You can play Videos or songs, like so:

```
tosongs; mpl 3-5,33
```

The above would enter the song directory - tosongs is my alias - and then mplayer (mpl is an alias for mplayer) would try to play file 3, 4, 5, and also file 33.

The ; is a separator in this context. You can chain command instructions separated by ; that way.

Note that you can search on IMDB via the shell too:

```
imdb dou fo sin # this would search for the movie called "dou fo sin".
```

Note that if you invoke "play" without any arguments, if we had assigned a file prior to this, we will play this file.

Installation

In the future this section is supposed to allow you to install a **Linux distribution**. For now though, we will focus on installing several packages in one go, i.e. compiling them.

To do this, try this command:

```
install default
```

This will compile all programs which are listed in the file
/Users/x/DATA/PC/OS/LINUX/YAML/installation_procedere.yml

The file that will be used is *$RUBY_SYS/profile_install.rb*.

You may have to use your own installation_procedere.yml file. I will make it available for RBT.

Run

You can "run" a file, and we attempt to run this file with the default application that was registered.

For example, if you do:

```
run foo.pdf
```

Then we would open this file.

This is a bit similar to the command open, but run specifically also works when we have files like .rb files, .py files and so on. It is a bit like an instruction to **execute that file**.

Stat

Stat is used similar to the **UNIX command stat**.

```
stat some_file
```

Additionally we include the info from BeautifulUrl so whatever you added to that class will work with the stat command too, no matter where that file is.

You can also apply stat on the @_ token, like:

```
stat @_
```

Additionally, you can do this:

```
stat {1,2}
stat 1,2
```

The above commands would stat the position 1 and 2.

Note that leading file: entries will be removed from the given input.

PDFs 

If you want to find all PDFs, do this:

```
find all pdf
```

There is also a configuration option called **@open_downloaded_pdf_files_at_once**.

If this configuration option is true, then we will open PDF files the moment we downloaded them.

You can also generate a new Manual for the DiamondShell by issuing:

```
manual
create manual
```

This will create a .pdf file.

Also note that you can, thanks to prawn, create a .pdf file out of a .txt file (any text file). We support this in the Diamond Shell as well.

To do this, issue:

```
to_pdf foo.txt
```

Here we assume that foo.txt exists and can be used ot create a foo.pdf file.

Find - finding something 

We can find something in the DiaShell, by doing something like:

```
dfind threads.rb
```

This will find threads.rb on your HDD and also set it to the @last_file used, which you can query via **file?**.

Repeat last action

You can repeat the last action by doing:

```
~
+
-
++
--
```

Positional modifiers work too, to some extent. +30 would mean "play the song at position +30 in relative terms to the last song played".

DiamondShell Parameters

You can get all the parameters of the methods in the DiamondShell.

In order to do this, do the following command from within a running instance of DiamondShell:

```
parameters?
```

Here is a partial screenshot showing the output of that command:

```
704 - `upload`: [:req, :i, :opt, :ftp_mode_to_use]
705 - `upload_screenshot`: [:opt, :i]
706 - `url`: [:req, :i]
707 - `use`: [:req, :i]
708 - `use_command_at_history_position`: [:req, :this_position]
709 - `use_ftp?`:
710 - `use_this_config`: [:opt, :i]
711 - `user_interaction_change_prompt`: [:opt, :ask_user]
712 - `userfind`: [:req, :i]
713 - `verbose_answer_does_this_file_exist?`: [:req, :i]
714 - `version`: [:opt, :of_what]
715 - `watch`: [:req, :what]
716 - `wav_to_midi`: [:req, :i]
717 - `wav_to_mp3`: [:opt, :i]
718 - `wav_to_ogg`: [:req, :i]
719 - `wecker`: [:opt, :wait_for_how_long]
720 - `which_platform?`:
721 - `which_windowmanager_is_running?`:
722 - `whoami`:
723 - `xorg_running?`:
```

Diamond Shell on Windows

The Diamond Shell works on Windows too, but some problems may exist there, as Windows is not the main OS for the Diamond Shell to be tested.

First, to check whether you are running on windows or not, run this check:

```
if OPERATING_SYSTEM == :windows
```

The DiaShell should of course also work on Windows. For this to work, some convenience methods were added.

```
dcomcnfg
```

Also try out this:

```
config windows
```

This should enable some windows-specific configuration settings.

Configuration

Important Configuration Files for the DiamondShell and how to handle them.

default_applications.yml This file will include your favourite applications for specific file types.

In here you can specify which program to use whenever you want to open a .pdf file, open a .png file and so on.

```
bl $RUBY_DIA/lib/diamond_shell/YAML/default_applications.yml
```

To show the current values set in that file, you can do this:

```
default_applications?
```

Note that you can reload the yaml config by doing:

```
reload configuration
```

This will also reload the aliases at run-time.

Another configuration option allows you to display shorter file names, i.e. if a file name is too long we will simply truncate it.

Every available configuration option can be found in the **main configuration directory**.

```
cd $RUBY_DIA/lib/diamond_shell/yaml/configuration
```

For local use I recommend you to only use the saving option. The following example illustrates this:

```
config.debug = true
```

This changes the configuration of the shell during runtime.

To make this change permanent, do this:

```
save
```

or

```
save config
```

Now, on next startup, we will load this saved config file.

You can also save individual components. For instance, you can save the debug value, or the padding value.

```
save debug
save padding
```

But this is no longer that useful. Since March 2014, I recommend you to simply to just "save" alone.

If you have set shorten_directory_names to true, then a leading * will indicate that the name was shortened.
You can toggle this at run-time too, and untoggle again, as in:

```
shorten
unshorten # or "longen"
```

Note that **shorten** will also get rid of the ModTime information.
You can also deliberately disable the display of the modification time. In order to do so, do:

```
no_mod_time
```

And to undo, do:

```
show_mod_time
```

You can modify every configuration option directly like so:

```
config.use_colours = false
```

To feedback this option, do this:

```
config.use_colours
```

You can also append a **?** if you want to:

```
config.use_colours?
```

Rather than use **config.**, some aliases exist, such as @configuration, @config and so on. Use whatever suits you best, although **config.** is probably the shortest and simplest notation for accessing the DiaShell Configuration Settings.

Of course, if you want to disable the colours, you could also use this:

```
disable colours
```

You can reload your original configuration by issuing this:

```
reload config
```

This will also reload the colours, so you can use it to change the currently running shell.

The option **run_simulation** is used to run in a simulation mode - currently this means that we do not actually burn a CD or DVD, no. It means we will feedback which command we would use.

Also note that when you are on windows, you could do something like:

```
config windows
```

To conveniently switch to a windows-based configuration.

If you look at the configuration, it is a simple key => value mapping, a hash. You can modify values of this hash in a fairly simple manner:

```
editor = bluefish
```

This would set your editor to bluefish.

You can change specific parts of the config by using aliases. For instance, this will work to change the location of Firefox:

```
firefox.location = /Programs/Firefox/Current/bin
```

Since *March 2012* we are using a specialized class called **Configuration** to handle all configuration aspects of the Diamond Shell.

You can also query the config directlry by doing something like this:

```
config.padding?
config.use_irc?
```

An alternative syntax also exists such as:

```
Diamond.configuration.padding = 5
```

You can designate another Encoding format to use via the configuration option preferred_encoding.
I tend to use ISO, you may use UTF there instead.
This encoding format will be used several times during a DiamondShell session.

Conventions for the Diamond Shell

- All * are meant to be a shortcut for "all files". Or more precisely, all files, directories and symlinks.
- @this_file and @_. Not sure about the differences. I think @this_file is used to denote the main file right now, whereas @_ is more like a **throwaway variable**.

Links

- C Programming Language
- Ruby Build Tools
- Master Pipe