Contributing to When

Release 0.9

Francesco Garosi

September 21, 2016

Contents

1 Contributing **3**
- 1.1 Some Notes about the Code . . . 3
- 1.2 Dependencies . . . 4
- 1.3 How Can I Help? . . . 4

2 Localization **7**
- 2.1 Template Generation . . . 7
- 2.2 Create and Update Translations . . . 8
- 2.3 Create the Object File . . . 8
- 2.4 Translation Hints . . . 8

3 The When Wizard **11**
- 3.1 Plugin Rationale . . . 12
- 3.2 Reserved Attributes . . . 12
- 3.3 Task Plugins . . . 14
- 3.4 Condition Plugins . . . 16
- 3.5 Plugin Packaging and Installation . . . 19
- 3.6 Write a Simple Plugin . . . 19
- 3.7 How to Choose a Suitable Name . . . 33
- 3.8 Parametric Item Definition Files . . . 33

4 Packaging **37**
- 4.1 Requirements for Packaging . . . 37
- 4.2 Package Creation: LSB Packages . . . 38
- 4.3 Package Creation: the Old Way . . . 40

5 Test Suite **41**

6 DBus Remote API **43**
- 6.1 Interface . . . 43
- 6.2 General Use Methods . . . 44
- 6.3 Reserved Methods . . . 45

7 History **47**
- 7.1 Version 0.9.12 (beta) . . . 47
- 7.2 Version 0.9.11 (beta) . . . 47
- 7.3 Version 0.9.10 (beta) . . . 47
- 7.4 Version 0.9.9 (beta) . . . 47
- 7.5 Version 0.9.8 (beta) . . . 47

7.6 Version 0.9.7 (beta) . . . 48
7.7 Version 0.9.6 (beta) . . . 48
7.8 Version 0.9.5 (beta) . . . 48
7.9 Version 0.9.4 (beta) . . . 48
7.10 Version 0.9.3 (beta) . . . 48
7.11 Version 0.9.2 (beta) . . . 48
7.12 Version 0.9.1 (beta) . . . 49
7.13 Version 0.7.0 (beta) . . . 49
7.14 Version 0.6.0 (beta) . . . 49
7.15 Version 0.5.0 (beta) . . . 50
7.16 Version 0.3.0 (beta) . . . 50
7.17 Version 0.2.0 (beta) . . . 50
7.18 Version 0.1.1 (beta) . . . 51
7.19 Version 0.1.0 (beta) . . . 51

8 License (BSD) 53

Contents:

CHAPTER 1

Contributing

When is open source software, this means that contributions are welcome. To contribute with code, please consider following the minimal recommendations that usually apply to the software published on GitHub:

1. Fork the When repository and pull it to your working directory
2. Create a branch for the new feature or fix: `git checkout -b new-branch`
3. Edit the code and commit: `git commit -am "Add this feature to When"`
4. Push your changes to the new branch: `git push origin new-branch`
5. Compare and submit a Pull Request.

A more general discussion about contribution can be found here. Otherwise, just submit an issue to notify a bug, a mistake or something that could just have been implemented better. Just consider that the applet is intended to be and remain small in terms of code and features, so that it can stay in the background of an user session without disturbing it too much.

1.1 Some Notes about the Code

The applet is in fact a small utility, and I thought it would have even less features. It grew a little just because some of the features could be added almost for free, so the "*Why Not?*" part of the development process has been quite consistent for a while. The first usable version of the applet has been developed in about two weeks, most of which spent learning how to use *PyGObject* and friends, and not on a full time basis: by the 5th day I had to freeze the features and focus on the ones I wrote down. So, being small and mostly not reusable, the single-source option seemed the most obvious, also to keep the package as self-contained as possible. However, the way the applet starts and defines its own system-wide and user directories allows the development of modules that can be imported without cluttering and polluting the system: the `APP_DATA_FOLDER` variable defines a dedicated directory for the application where modules can be installed, and normally it points to `<install-base>/when-command/share` or `/usr/[local/]share/when-command` or something similar and well identifiable anyway.

The code tries to follow the usual guidelines for Python 3.x, and takes inspiration from other Gnome applets that sit in the indicator tray. I tried to reduce the comments to the very least, and let the code speak for itself. Some of the conventions here are the following:

- system wide constants are spelled in all uppercase (as usual in *C/C++*)
- variables tend to be all lowercase, both globals and locals
- class names start with an uppercase letter and are in camelcase
- global instances of classes are lowercase, like simple variables
- private members start, as usual, with an underscore

- function names are all lowercase with underscores
- transitional (or debug) functions start with underscores
- the core classes implement their own loggers, borrowing from the global one
- user interaction strings (and log messages) use double quotes
- program internal strings use single quotes
- statements tend to be split in lines of at most 80 characters, apart from log messages
- log messages mostly sport a prefix to determine what part generated them
- log messages containing the `NTBS` strings are *never to be seen*.

All user interaction strings (except log messages) are surrounded by the usual `_(...)` formula used in software that implements the `gettext` functions: this allows to translate **When** without intervention on the code itself, in the usual fashion for Linux applications.

1.2 Dependencies

Being an applet oriented mostly towards users of recent Ubuntu editions, it is developed in *Python 3.x* and uses the latest supported edition of *PyGObject* at the time. It shouldn't rely on other packages than `python3-gi` and `python3-pyinotify` on the Python side. [1] The *Glade* user interface designer is almost mandatory to edit the dialog boxes.

To implement the "*Idle Session*" based condition (i.e. the one that lets a task run when the session has been idle for a while), however, the external `xprintidle` command may be used that is not installed by default on Ubuntu. As of version *0.9.6-beta.1* it is no more strictly necessary, however the dependency is kept (also in packages) because it could be used as a fallback method for detecting idle time when `libXss` cannot be linked, and because it causes an implicit dependency from `libXss` itself. Normally if `libXss` is installed and **When** is run from a source code based installation, the installation of `xprintidle` can be safely skipped.

1.3 How Can I Help?

There are several ways to help. Even though **When** has been developed for a while now and has a quite thorough Test Suite, the devil can still figure out a way to stay around in the details. Such details can be of several kinds: be it bugs, missing features or whatever else, they have to be addressed. So, here is what can be done to help development of **When**.

1.3.1 Review the code

A lot can be done by reviewing the code. The single Python file is quite big now, and although I tried to keep it as tidy as possible, it still might contain places where it's hard to read. For example, in a quite recent beta tag (*v0.9.7-beta.3* to be exact) I finally removed a completely useless operation that was dangling there probably from the very beginning. Issues with the code may include:

- useless or redundant parts
- things that could be done in a more optimized way – as long as it does not introduce inelegance or obscurity
- excessive logging

[1] In fact the other packages that could possibly require installation are the ones mentioned in the chapter devoted to the applet install process. No *-dev* packages should be needed because **When** is entirely developed in the Python language.

- incorrect language in output, UI and comments
- inconsistent identifiers (variable and function names, mostly), or even names that are just inexplicative.

Interventions in this field can be done either using the *issue* mechanism or just by telling me otherwise what's supposedly wrong with the code and suggesting possible amendments.

1.3.2 Review the Documentation

This is important. At this point it should be clear that I'm not a native English speaker or writer. And, even worse, I don't have anyone at hand who can proofread these documents. So, if there is any volunteer out there who would like to help in this field, feel free to throw an *issue* in the related repository:

- User Documentation
- Contribution Guide

Also, within the User Documentation, a part that can be improved is the Tutorial, where probably more complete examples could be provided. If you use **When** and have an example that is particularly useful, feel free to submit it.

1.3.3 Other Linux Distributions

In my opinion **When** is now ready to better support other Linux distros that use *GLib* and/or *Gnome*. The *fallback* mechanism used to implement stock events should allow easy creation of fallback entries where DBus functionality specific to *Ubuntu* is not available. *Packaging* too is a quite important in supporting other distributions.

1.3.4 Improve the Test Suite

I tried to make the Test Suite as thorough as possible, and in fact it covers almost all features (with the exception of most UI/UX parts, because they are quite hard to test). Obviously there should still be something left out. The completeness of tests gives more chances that the final product is actually working, and makes the task of porting **When** to other Linux distributions much easier and obviously quicker. Improvements are of course possible by adding more tests (as long as they are not redundant), but also by making the existing components of the suite itself easier to read and to modify. Another aspect where the Test Suite can be improved, is the compatibility with continuous integration (CI) tools: I'd like to have **When** tested in Travis CI some day, but for the moment I am not able to figure out how to get there.

1.3.5 Localization

When has support for *localization*, described in this guide. Also, this guide provides some hints on how to localize the applet in your language. Feel free to provide a translation if you want, you will be credited for the contribution.

1.3.6 Packaging

Packaging is another place where things could be better. Firstly, the *Ubuntu* package (LSB version) installs and blends acceptably with the distribution, but there are still some rough edges to smooth off, such as the translations that should be separated from the rest of the program. Also, packages specific to other distributions that might be supported should be created.

1.3.7 Add-ons

With release *v0.9.7-beta.3* **When** has gained an almost full DBus API, documented later in this guide, which can be used to interact with a running instance of the applet. Using this interface and its provided methods, the applet can be configured using an external application. I am on the way to providing a more streamlined interface (I'm calling it the **When Wizard**) for users that would like the complexity of the *"raw"* applet hidden, and a wizard-like tool with a library of *conditions* and *tasks* ready for general use. [2]

[2] As soon as I publish an early release of this application, there will also be a dedicated section for it in this developer guide.

CHAPTER 2

Localization

Starting with version *0.9.1-beta.2* **When** supports the standard localization paradigm for Linux software, via `gettext` and its companion functions. This means that all translation work can be done with the usual tools available on Linux, that is:

- `xgettext` (for the Python source) and `intltool-extract` (for the *Glade* UI files)
- `msginit`, `msgmerge` and `msgfmt`

This should allow for easier translation of the software. In fact I provide the Italian localization (it's the easiest one for me): help is obviously welcome and really appreciated for other ones.

I can provide some simple instructions for volunteers that would like to help translate **When** in other languages: I've already seen some activity in this sense, and very quickly after the first public announcement. I'm really glad of it, because it helps **When** become more complete and usable.

Think of the following instructions more as a *recipe* than as an official method to carry the translation tasks.

2.1 Template Generation

Note: Normally, to translate the applet, a translator only needs access to the most recent *message template* (which is `po/messages.pot`); however these instructions also try to show how to generate such template in case some text in the source has changed, for example while fixing a bug.

Basically the necessary tools are:

- `intltool-extract` to retrieve text from the UI files
- `xgettext` to extract text from the main applet source.

When in the source tree base, the following commands can be used to generate the template without cluttering the rest of the source tree:

```
$ mkdir .temp
$ for x in share/when-command/*.glade ; do
>   intltool-extract --type=gettext/glade $x
>   mv -f $x.h .temp
> done
$ xgettext -k_ -kN_ -o po/messages.pot -D share/when-command -D .temp -f po/translate.list
```

After template generation, which is stored in `po/messages.pot`, the `.temp` directory can be safely deleted. If `po/messages.pot` already exists and is up to date, this step can be skipped.

2.2 Create and Update Translations

To create a translation, one should be in a localized environment:

```
$ cd po
$ export LANG=it_IT
$ msginit --locale=it_IT --input=po/messages.pot --output=po/it.po
```

where `it_IT` is used as an example and should be changed for other locales. For all `po/*.po` files (in this case `it.po` is created), the following command can be used to create an updated file without losing existing work:

```
$ msgmerge -U po/it.po po/messages.pot
```

where `it.po` should be changed according to locale to translate. The generated or updated `.po` file has to be modified by adding or updating the translation, and there are at least two options for it:

- use a standard text editor (the applet source and string set is small enough to allow it)
- use a dedicated tool like poedit.

After editing the *portable object*, it must be compiled and moved to the appropriate directory for proper installation, as shown below.

2.3 Create the Object File

When the `.po` file has been edited appropriately, the following commands create a compiled localization file in a subtree of `share/locale` that is ready for packaging and distribution:

```
$ mkdir -p share/locale/it/LC_MESSAGES
$ msgfmt po/it.po -o share/locale/it/LC_MESSAGES/when-command.mo
```

Also here, `it.po` and the `/it/` part in the folder have to be changed according to the translated locale. In my opinion such command-line based tools should be preferred over other utilities to create the compiled object file, in order to avoid to save files in the wrong places or to possibly pollute a package generated from the repository clone. However, for the editing phase in *Step 2* any tool can be used. If `poedit` is chosen and launched from the base directory of the source tree, it should automatically recognize `po` as the directory containing translation files: open the one that you would like to edit and you will be presented with a window that allows per-string based translation. [1]

2.4 Translation Hints

I have tried to be as consistent as possible when writing UI text and command line output in English. Most of the times I tried to follow these basic directions:

- I preferred US English over British (although I tend to prefer to speak British)
- text in dialog box labels follows (or at least should follow, I surely have left something out) title case
- text in command line output is never capitalized, apart from the preamble and notes for the `--help` switch output, and the applet name in the `--version` output.

These guidelines should also help to recognize where a string belongs when translating a newly created `xx.po` file: basically, all (or almost all) sentences that begin with a lower case letter are used in console output, and strings that begin with a capital letter are in almost all cases in the graphical UI. However a translator is strongly advised to give

[1] Consider that `poedit` would not show new or untranslated strings by default.

When a try, and explore its English interface (both UI and console, by testing the CLI switches using the `--verbose` modifier) to be sure of what he is translating. Also, the following command should be issued

```
$ when-command --help
```

to locate text that belongs to brief command help. Please note that some words in the help text for the `-h` switch cannot be modified: they are directly handled by the Python interpreter. Some more detailed instructions follow:

1. help text for switches should remain *below 55 characters*
2. letters inside brackets in help text should not be changed
3. console output strings should remain *below 60 characters*, and consider that `%s` placeholders in some cases might be replaced by quite long strings (like 20 characters or so)
4. strings in ALL CAPS, numbers and mathematical symbols *must NOT be translated*
5. labels in dialog boxes should remain as short as possible, possibly around the same size as the English counterpart
6. labels that are *above* or *aside* text entries (especially the time specifications that appear in the *Condition Dialog Box* for time based conditions and the *DBus parameter* specification strings like *Value #* and *Sub #*) should *not* be longer than the English counterpart: use abbreviations if necessary
7. most of the times, entries in drop down combo boxes (such as condition types) *can* be somewhat longer than the English counterpart
8. keep dialog box names short
9. *button* labels *must* follow commonly used translations every time it is possible: for example, the *Reload* button is present in many applications and the most common translation should be preferred
10. menu entries that have common counterparts (such as *About...*, *Settings...* and *Quit*) should be translated accordingly
11. button labels should not force the growth of a button: use a different translation if necessary, or an abbreviation if there is no other option
12. column titles should not be much longer than the English counterparts, use abbreviations if necessary unless the related column is part of a small set (like two or three columns)
13. *title case* is definitely *not* mandatory: the most comfortable and pleasant casing style should be used for each language
14. try to use only special characters normally available in the default ASCII code page for the destination language, such as diacritics: if possible avoid other symbols and non-printable characters.

Note: There is one point where the translation might become difficult: the `"showing %s box of currently running instance"` *msgid*. Here `%s` is replaced with a machine-determined nickname for a dialog box. For the *About Dialog Box* the message would be `"showing about box of currently running instance"` and the word `about` cannot be translated. Feel free to use quotes to enclose the nickname in a translation, if you find it necessary.

A personal hint, that I followed when translating from English to Italian, is that when a term in one's own language is either obsolete, or unusual, or just "funny" in the context, it has not to be necessarily preferred over a colloquially used English counterpart. For example, the word *Desktop* is commonly used in Italian to refer to a graphical session desktop: I would never translate it to *Scrivania* – which is the exact translation – in an application like **When**, because it would sound strange to the least.

CHAPTER 3

The When Wizard

The **When Wizard** aims at becoming, possibly, the main interface to **When** for those users who just want to instruct their workstations to perform simpler tasks on a rich subset of the available conditions, or for those system administrators who want to provide standardized sets of tasks and events or conditions that may trigger such tasks. The **When Wizard** is available (in *early development* stage) at its own repository.

The **When Wizard** has been designed for extensibility: it is completely based on plugins that are loaded by a small application core. Some plugins are provided by default (I call them *stock plugins*), others may be developed and easily added to the application.

> **Warning:** The **When Wizard** is still in its early development stages, and this means at least that its API is still subject to changes. Such changes are expected to be drastically reduced before it reachs a *beta* status, but for now things could change abruptly, even on a convenience basis.

Plugins are the parts, in the application, that actually *define* items in **When**, while the surrounding application only provides the wizard interface and the steps that communicate with **When** to create such items. Communication with **When** is made possible mainly via the *DBus Remote API* later discussed in this manual.

There are two types of plugins that can be developed:

- *task* plugins, and
- *condition* plugins

even though in the latter case there are subtypes, one of which has to be chosen when developing a *condition* plugin. *Task* plugins are probably the ones that would gain more attention, because there are virtually infinite tasks that can be defined to ease the user's life. But also some types of conditions can be added as shortcuts to more complex ones:

- conditions that check *system commands*, and
- conditions that react to *user defined events*, for which the user event definition could be provided as an *Item Definition File*; in this case the plugin developer should make the *Item Definition File* available along with the plugin package (see below) and document that the file has to be imported through the appropriate **When Wizard Manager** page for the plugin to function.

The following paragraphs will illustrate briefly how a plugin for the **When Wizard** can be implemented. After the general discussion about **When Wizard** plugin development, an example is provided on how to develop a plugin from scratch and how to distribute it.

3.1 Plugin Rationale

The idea behind the plugin-based structure is that it's almost impossible to implement (or try to implement) all available reactions to all available events, and probably also just the features that most users would like to see are quite difficult to find and to blend with a possibly closed or monolithic environment. Thus the plugin system: also basic functionality is implemented as plugins in the **When Wizard**, and the system will dynamically look for plugins both in common areas and in the user home, making it easier to install functionality on a per-user basis.

Plugins are essentially *Python 3.x* modules with a structure, and should obey some simple rules:

1. each plugin module *must* export a class named `Plugin` derived from either `TaskPlugin` or from one of the base `...ConditionPlugin` classes
2. the derived class must initialize some class parameters through the base class constructor: many of these parameters are there for classification and documentation purposes only, but they end up being useful for a correct representation of the plugin functionality in the UI of the **When Wizard**
3. plugins may or may not allow for configuration in the wizard interface, but if they do they have to export their configuration pane through a well-defined interface
4. plugins should document what they do after configuration (if any) in a way that summarizes their purpose after configuration.

In fact all of this can lead to a plugin that does everything it has to do just via its constructor. In many cases the plugin will sport a configuration pane and signal functions for controls in the configuration pane, that update the inner variables and construct the values required by the underlying task or condition. Templates are provided for base plugins, because the plugin structures tend to be very similar to each other, so that the coding effort can be reduced to the bare minimum.

Note: To facilitate development of plugins, a recommended method is provided that allows easier testing and packaging, and that consists in the creation of a specific directory where *all* plugin files are located: the plugin source file, its auxiliary (executable) scripts, the custom icon [1] file, other graphic files if needed, and the configuration pane resource file – whose extension must be either `.ui` or `.glade`. To test the plugin in the **When Wizard** in such an environment, it is sufficient to define an environment variable, `WHEN_WIZARD_DEVPLUGIN`, to point to the absolute path where the plugin files are found. In this way both the wizard and the manager will look for the plugin and its related files in that directory before trying the actual plugin directories. Also, when this variable is correctly defined and the **When Wizard** applications are invoked from the command line, in case of an error while loading a plugin the application prints a Python stack trace to the console instead of just skipping the failing plugin. Also, for task plugins to correctly work, the main **When** applet *must* be started in a shell where the environment variable is appropriately set to the plugin development directory.

3.2 Reserved Attributes

Instead of building a complex suite of private members and getters/setters for base class properties, the quick approach has been chosen to directly expose some values to the derived classes through member variables. There are thus two types of attributes with special meanings – which doesn't mean that they shouldn't be accessed or changed: in some cases they *must* be updated – that is:

- special member variables, and
- special methods.

[1] It is not necessary to provide a custom icon: one of the stock ones can be used too and it is rather encouraged, as this would keep the style consistent. In case of need, the custom icon must be a 24x24 pixel PNG with transparency, possibly in a flat colored style.

Most special member variables are defined at initialization time, with the appropriate base constructor parameter:

Variable/Parameter	Description
category	the plugin category, must be one of the values defined in `PLUGIN_CONST` (usually explicitly imported from the `plugin` module): it's available only for *task* plugins as a constructor parameter [2]
basename	the base name of the plugin, should correspond to the base name of the plugin file
name	a descriptive name for the plugin, to be kept short
description	a short description of the plugin
author	the name of the plugin author
copyright	the usual copyright string, with year and so on
icon	the name of the icon: should correspond to the base name of a *PNG* file without extension either in the application resource directory or in the user resource directory
help_string	a sufficiently long help string: it will appear in the wizard box to document what the plugin does; it should not exceed about 250 characters, and all newlines are converted to spaces.
version	a possibly sortable version string

The values set here are available for reading withinthe plugin class in case of need – for example, to derive the base name of another file, such as an icon or resource file.

There are other reserved variable names: `unique_id`, `module_basename`, `module_path`, `stock`, `plugin_type`, `summary_description`, `forward_allowed`, `scripts`, `resources` and `graphics`. Some are used internally, but the following ones should be assigned or modified in the derived class to change the behavior of the plugin and to allow the plugin to be correctly installed or removed:

- `summary_description` must be given an explanatory value that will be shown in the summary page of the wizard; it can be modified while the plugin is being configured and can contain values of the configuration parameters
- `forward_allowed` should be set to `False` in the derived plugin constructor if the default values for its parameters (that is, the ones that will be first shown in the configuration pane) *must* be modified before the wizard can step forward; if it's set to `False`, then the `allow_forward()` method shown below must be used to enable the *Next* button
- `scripts` can contain the list of script files (basenames only) that are used by the plugin: such scripts must be executable and available in the plugin development directory; the recommended way to update this variable (and the next two) is via `self.scripts.append('filename.ext')`
- `resources` can contain the list of resource files (basenames only) used by the plugin: normally it only contains the `.glade` (or `.ui`) file that defines the configuration pane, if needed; these files too must be available in the plugin development directory
- `graphics` must contain the list of graphic files (basenames only) that are used by the plugin, including the plugin icon file (whose basename without extension is specified in the base constructor call) if a custom icon is used; same as above for where the graphic files must be located.

> **Warning:** The **When Wizard** installer does not check whether or not a plugin file name or the names of its auxiliary files are already taken: if so, a newly installed plugin may overwrite other installed plugins, although never the ones that come with the application, or parts of them. It is advisable to use very specific names for plugins, and that the auxiliary files have the same name (except for the extension) at least as a prefix.

All plugins have these methods:

[2] For condition plugins the category is automatically set depending on the type of condition plugin the actual plugin is derived from. However it can be changed after invoking the base class constructor if the automatic setting does not fit the nature of the plugin.

Method	Description
get_dialog(name)	returns a dialog builder object from a file that has the base name (without extension: supported extensions are `.ui` and `.glade`) as the provided parameter
get_image(name)	returns a *pixbuf* loaded from a file whose base name is the provided parameter; icons are looked for in two paths: the user resource path and the application resource path, so that a non-stock plugin can also use one of the icons that come with the application
get_script(filename)	returns the full path to an executable script if it is needed by the plugin either to execute an action or to test a condition; the filename should be the base name only, including any extension (like `.py` or `.sh`)
allow_forward()	if called without arguments (or with `True` as argument) it causes the wizard button to become *sensitive*: it has to be called when the configuration pane controls contain acceptable data; if a `False` parameter is provided, the wizard button will become *not sensitive*
get_pane()	if the plugin has a configuration pane, this method *must* be overridden and return a reference to the outmost container object in the plugin pane dialog structure
data_store(data)	store persistent data related to the plugin itself (that is, common to all instances of the plugin): data should be a simple value, or a list (or tuple) of simple values or even a dictionary thereof, however this method is not meant to store complex data such as class instances
data_retrieve()	return data previously saved using `data_store()`
file_storage([sub])	return the full path to a directory where persistent files can be created: what to save in this directory is left to the plugin author and can be either plugin or instance data; if `sub` is provided it is used as the last subdirectory and can be used to exchange data between instances of different plugins, however is less safe than the parameterless version (which uses the plugin base name); in case of an error `None` is returned instead of a valid path
register_action()	this method can optionally be overridden if there is any code that should be run upon registration of the associated action, that is when a condition is set to trigger a consequence, and it is the last chance to setup the associated task (command line) or condition parameters: should return `True` on success
remove_action()	this method can optionally be overriddeen if there is some cleanup that has to be done *before* the instance is removed and should return `True` on success.

There are also other reserved method names common to all pugins: `to_dict`, `from_dict`, `to_item_dict`, `to_itemdef_dict`, `to_itemdef`, `desc_string_gui`, `desc_string_console`, `data_store`, `data_retrieve`, `set_forward_button`, and `get_config`. These names should not be overridden in plugin implementations as overriding them would cause the plugin not to work properly.

3.3 Task Plugins

Task plugins should just provide a *command line* that will be run whenever the associated condition occurs. The easiest case is when the command is fixed and no configuration is needed: in such a case the constructor will define the command and no other code is needed. For example, the command to lock a session is

```
$ dm-tool lock
```

with no configurable options. This means that a plugin whose task is to lock the running session will only configure the `command_line` member variable of the task plugin to be `dm-tool lock`.

The variables that can be set in a task plugin to modify its behavior are the following:

Variable	Description
command_line	the command that will be executed by the task in its entirety, including parameters: it will be executed in a shell, so it can also be the path to a script
process_wait	determine whether or not the calling process should wait for the called process to end; for simple tasks it is safe to skip this and let the process be left alone as soon as it is started

In case a task plugin should be configured, the `get_pane()` method must be overridden to return a reference to the outmost container of the configuration pane, and dialog signal handling functions must be defined to retrieve configuration values from the pane just as if it were a standard *Gtk* dialog box.

Task plugins also give the possibility to set one and only one of the following variables:

Variable	Description
success_status	if the status code of the called process has to be checked for a specific success value; must be an integer and defaults to `0`
failure_status	if the status code of the called process has to be checked for a specific failure value; must be an integer
success_stdout	a string that, if corresponding to process output (written to *stdout*), will let the process execution be considered a success; modifiers specified below can change the way the correspondance is checked
failure_stdout	a string that, if corresponding to process output (written to *stdout*), will let the process execution be considered a failure; same as above for modifiers
success_stderr	string that, if corresponding to process output (written to *stderr*), will let the process execution be considered a success; same as above for modifiers
failure_stderr	a string that, if corresponding to process output (written to *stderr*), will let the process execution be considered a failure; same as above for modifiers

and these are the modifiers for string *stdout/stderr* variables:

Variable	Description
match_exact_output	if the specified string should match from start to end, if `False` the correspondance will be found when the given string is contained in the output
match_case_sensitive	if `True` the comparison is case sensitive
match_regexp	if `True` the given string is considered a regular expression and matched against the process output

These attributes are all booleans, and default to `False`: output will be searched for a substring with no distinction between uppercase and lowercase. Values for the modifier variables can be set independently on all of them: for example if `match_exact_output` is set to `True` and `match_regexp` too, the provided regular expression will be checked at the beginning of the process output, if `match_exact_output` is `False` **When** will just try to find a match for the regular expression in the output.

The base class for this type of plugin is `TaskPlugin`: at the beginning of a plugin there must always be the following statement

```
from plugin import TaskPlugin, PLUGIN_CONST
```

in order to derive the `Plugin` class. [3] The above mentioned `category` base constructor parameter can be given one of the following values:

[3] Note that the provided plugin development templates also import the `plugin_name` module function, so that it is possible to automatically derive the plugin *base name* from the file name itself instead of having to specify it. The same yields for both task and condition definition plugins.

Constant	Related plugins
PLUGIN_CONST.CATEGORY_TASK_APPS	For plugins that concern applications, such as starting or killing a program or system utility
PLUGIN_CONST.CATEGORY_TASK_SETTINGS	When the plugin manages session, desktop or system settings
PLUGIN_CONST.CATEGORY_TASK_POWER	For power-management related plugins
PLUGIN_CONST.CATEGORY_TASK_SESSION	For session management related plugins, like session lock, unlock or logout
PLUGIN_CONST.CATEGORY_TASK_FILEOPS	This has to be used for plugins that perform file operation, such as backups or synchronizations
PLUGIN_CONST.CATEGORY_TASK_MISC	All other task plugins belong here

These values should be assigned carefully, because the user will be able to choose a plugin only after category has been selected.

3.4 Condition Plugins

There are several types of condition plugins: for each type the appropriate base class must be used. In the same way as for task plugins, the base class be imported in the plugin code:

```
from plugin import <SpecificConditionPlugin>, PLUGIN_CONST
```

where `<SpecificConditionPlugin>` must be replaced with one of the names specified below. The plugin category is determined by the condition plugin type, but in case the developed plugin belongs to a different category, its value can be assigned one of the following constants:

Constant	Related plugins
PLUGIN_CONST.CATEGORY_COND_TIME	Category for plugins that define conditions concerning time: *time*, *idle time*, and *interval* based conditions normally belong to this category
PLUGIN_CONST.CATEGORY_COND_NETWORK	Category for plugins that define conditions related to network activity
PLUGIN_CONST.CATEGORY_COND_POWER	Category for plugins that define conditions related to power management
PLUGIN_CONST.CATEGORY_COND_EVENT	Category for plugins that define conditions related to *events*, both stock and user defined
PLUGIN_CONST.CATEGORY_COND_MISC	All other condition plugins belong here

The `category` member variable can be reassigned *after* the base class constructor has been called – otherwise the new category is overwritten.

Just like task plugins, condition plugins must offer a `get_pane()` method that returns a reference to the outermost container object in case they need any configuration.

There are some *flags* (in the form of attributes, as usual) that can be set to either `True` or `False` to change how the generated condition check will behave:

Variable	Description
sequential	if there is a task list instead of a single associated task the tasks in the list are run sequentially; since the application only provides conditions associated with single tasks this flag can be left alone; set to `True` by default
repeat	if `True` checks will persist after first successful one
suspended	if `True` then checks for the associated condition are suspended on condition registration
break_on_failure	when a sequence of tasks is given, break after the first failed task; normally it is ignored, and defaults to `False`
break_on_success	when a sequence of tasks is given, break after the first successful task; normally it is ignored, and defaults to `False`

Other attributes, methods and other member data may be present in subclasses that can be derived from, as specifically described below.

3.4.1 Interval Based Condition Plugins

Such plugins must provide the length of an interval in minutes, in the `interval` member variable. A simple plugin of this kind is already provided by the application and derivatives are unlikely to be actually useful.

The base class for this type of plugin is `IntervalConditionPlugin`.

3.4.2 Time Based Condition Plugins

Plugins of this type must define a time specification dictionary in the `timespec` member variable: the dictionary values are integers, with the following keys (as strings):

- `'year'`
- `'month'`
- `'day'`
- `'hour'`
- `'minute'`
- `'weekday'`

The `'weekday'` key, if used, allows for week-based repetition. A value of `0` is for monday, `6` is for sunday. It should not be used in conjunction with other date specifications. Values that must not be checked can just be skipped: for a condition that must occur at quarter past any hour of the day, just

```
self.timespec['minute'] = 15
```

should be set in the plugin. Instead of providing a single plugin of this type with all possible settings, several plugins with more specific scope can be a better option to give the users an easier way to choose what kind of time based condition they need.

The base class for this type of plugin is `TimeConditionPlugin`.

3.4.3 Idle Time Based Condition Plugins

In this type of plugin the `idlemins` member variable must contain the time in minutes that the session has to be idle before the condition occurs; since a simple plugin of this kind is already provided, this one is unlikely to be derived.

The base class for this type of plugin is `IdleConditionPlugin`.

3.4.4 File Change Based Condition Plugins

In these a path containing a file or directory to be watched must be provided using the `watched_path` string member variable. Stock plugins, one for files and another one for directories, are already available.

The base class for this type of plugin is `FileChangeConditionPlugin`.

3.4.5 Stock Event Based Condition Plugins

These plugins provide the counterpart of the *Event Based Conditions* in the **When** applet, and only occur when stock events occur. They must hold the event name in the `event` member variable, and are unlikely to need any form of configuration. However plugins for stock events are provided by the application, the only exception being possibly command line driven events, which are virtually useless in the **When Wizard** context.

The base class for this type of plugin is `EventConditionPlugin`.

3.4.6 User-Defined Event Based Condition Plugins

Plugins of this kind must store the name of the user-defined event (as known by **When**, thus the name that has been possibly given to the event in an *Item Definition File*) in the `event_name` member variable. These can be very useful to create condition that occur on events that are not handled by **When** by default, and the possibilities are virtually endless.

Because the corresponding conditions occur when the related *DBus* signal is fired, in most cases the related plugins will need no configuration pane.

The base class for this type of plugin is `UserEventConditionPlugin`.

3.4.7 Command Based Condition Plugins

Command based conditions are probably the ones that will benefit most from the implementation of specific plugins: almost every check can be done using system commands, possibly combined into scripts, and many types of event can be discovered or triggered in this way.

Such conditions are possibly where **When** can show the highest flexibility, but are also the ones that require a certain knowledge of Linux, of the shell and the system commands, and that might require some programming skills. The ability to include scripts with the plugin and the possibility to modify the command line using data gathered through the pane-based configuration gives the possibility to check for whatever actual status of the system – from the availability of files or devices to the connection status or the existence of resources online, just to mention a few.

Plugins of this type must store the actual command line in the `command_line` member variable, and depending on the command result the related event will either occur or not.

Just like in *Task Plugins* there are attributes to check command outcome: since there is no concept of success or failure in conditions, but just either occurrence or not, the attributes only specify what to expect.

Variable	Description
expected_status	the status that the called process should return to consider the underlying condition to occur; it must be integer and by default it is set to `0`
expected_stdout	string to find a correspondence for in the *standard output*
expected_stderr	string to find a correspondence for in the *standard error*

Here too modifiers are available, as for *Task Plugins*, and have the same identifiers and specifications:

Variable	Description
match_exact_output	if the specified string should match from start to end, if False the correspondance will be found when the given string is contained in the output
match_case_sensitive	if True the comparison is case sensitive
match_regexp	if True the given string is considered a regular expression and matched against the process output

Same as above, the modifiers are all set to False by default.

The base class for this type of plugin is CommandConditionPlugin.

3.5 Plugin Packaging and Installation

The **When Wizard** suite contains a simple utility to package plugins for installation. It can be invoked as follows:

```
$ when-wizard plugin-package <directory_name>
```

where <directory_name> is the name of the directory where the plugin is being developed. The utility is very basic, and just creates an archive with a name of the form plugin-basename.1433e3da13d9f700.wwpz: the middle part is just some hexadecimal blurb to make the name as unique as possible, and the package can be safely renamed after creation, apart from the .wwpz extension. The packaged plugin can be installed from the command line by issuing

```
$ when-wizard plugin-install [/path/to/]plugin_archive_file.wwpz
```

where [/path/to/]plugin_archive_file.wwpz is the file name of a packaged plugin, possibly including the path if needed.

3.6 Write a Simple Plugin

This section illustrates how to write a simple plugin for the **When Wizard**. First a command-based condition plugin is created that needs no configuration as it only does a fixed thing. Then the plugin will be expanded in order to be configurable and thus expose a configuration pane that will be shown in the wizard interface.

3.6.1 Step 1: Preparation

Preparation is quite easy: a directory for the plugin is needed as well as some source files to start from. These files can be found in the *share* directory where **When Wizard** is installed: assuming that the application is installed canonically in /usr/bin, the directory where the development templates are is /usr/share/when-wizard/templates/. For a condition plugin based on command execution, the template code is in the file called template-cond-command.py. Thus, assuming that the plugin will be called *Fire This*:

```
~$ mkdir firethis
~$ cd firethis
~/firethis$ cp /usr/share/when-wizard/templates/template-cond-command.py .
~/firethis$ mv template-cond-command.py firethis.py
```

And this is all for preparation. There is still a lot to do, though.

3.6.2 Step 2. Change the Plugin Code

This is what the template code looks like:

```
# file: share/when-wizard/templates/template-cond-command-plugin.py
# -*- coding: utf-8 -*-
#
# Template for a command based condition plugin
# Copyright (c) 2015-2016 Francesco Garosi
# Released under the BSD License (see LICENSE file)


import locale
from plugin import CommandConditionPlugin, PLUGIN_CONST, plugin_name

# Gtk might be needed: uncomment if this is the case
# from gi.repository import Gtk


# setup localization for both plugin text and configuration pane
# locale.setlocale(locale.LC_ALL, locale.getlocale())
# locale.bindtextdomain(APP_NAME, APP_LOCALE_FOLDER)
# locale.textdomain(APP_NAME)
# _ = locale.gettext

# if localization is supported, uncomment the lines above, configure
# them as appropriate, and remove this replacement function
def _(x):
    return x


HELP = _("""\
This is a template for a generic command condition plugin: it can be expanded
suitably to the needs of the plugin. A command line based condition plugin
must provide the full command line to be executed for the condition to be
verified: if the command is successful (zero-status) the condition is true.
""")


# class for a plugin: the derived class name should always be Plugin
class Plugin(CommandConditionPlugin):

    def __init__(self):
        CommandConditionPlugin.__init__(
            self,
            basename=plugin_name(__file__),
            name=_("Template"),
            description=_("Explain here what it does"),
            author="John Smith",
            copyright="Copyright (c) 2016",
            icon='puzzle',
            help_string=HELP,
            version="0.1.0",
        )
        # to repeat checks after first success uncomment the following line
        # self.repeat = True

        # the icon resource is only needed if the plugin uses a custom icon
        # self.graphics.append('plugin_icon.png')

        # the items below might be not needed and can be deleted if the
        # plugin does not have a configuration panel
```

```
        self.resources.append('template-plugin_generic.glade')
        self.builder = self.get_dialog('template-plugin_generic')
        self.plugin_panel = None
        self.forward_allowed = False         # forward not enabled by default

        # define this only if the plugin provides one or more scripts
        # self.scripts.append('needed_script.sh')

        # mandatory or anyway structural variables and object values follow:
        self.command_line = None             # full command line to run
        self.summary_description = None      # must be set for all plugins

        # this variable is defined here only for demonstrational purposes
        self.value = None

    def get_pane(self):
        if self.plugin_panel is None:
            o = self.builder.get_object
            self.plugin_panel = o('viewPlugin')
            self.builder.connect_signals(self)
        return self.plugin_panel

    # all following methods are optional

    def click_btnDo(self, obj):
        o = self.builder.get_object
        o('txtEntry').set_text("Some text")

    def change_entry(self, obj):
        o = self.builder.get_object
        self.value = o('txtEntry').get_text()
        if self.value:
            self.summary_description = _(
                "Something will be done with %s") % self.value
            self.allow_forward(True)
        else:
            self.summary_description = None
            self.allow_forward(False)


# end.
```

There is a lot of code that is not needed, because the plugin will display no configuration pane and will not use custom resources, not even graphics. However, since further development is planned, it might be better just to comment out at least part of the code that is not needed for now, especially the configuration pane related functions. As no scripts will be used, the two lines about scripts will be removed, as well as localization lines and the commented out import of the *Gtk* library. Here is the result:

```
# file: share/when-wizard/templates/template-cond-command-plugin.py
# -*- coding: utf-8 -*-
#
# Template for a command based condition plugin
# Copyright (c) 2015-2016 Francesco Garosi
# Released under the BSD License (see LICENSE file)


from plugin import CommandConditionPlugin, PLUGIN_CONST, plugin_name
```

```
# if localization is supported, uncomment the lines above, configure
# them as appropriate, and remove this replacement function
def _(x):
    return x


HELP = _("""\
This is a template for a generic command condition plugin: it can be expanded
suitably to the needs of the plugin. A command line based condition plugin
must provide the full command line to be executed for the condition to be
verified: if the command is successful (zero-status) the condition is true.
""")


# class for a plugin: the derived class name should always be Plugin
class Plugin(CommandConditionPlugin):

    def __init__(self):
        CommandConditionPlugin.__init__(
            self,
            basename=plugin_name(__file__),
            name=_("Template"),
            description=_("Explain here what it does"),
            author="John Smith",
            copyright="Copyright (c) 2016",
            icon='puzzle',
            help_string=HELP,
            version="0.1.0",
        )
        # to repeat checks after first success uncomment the following line
        # self.repeat = True

        # the icon resource is only needed if the plugin uses a custom icon
        # self.graphics.append('plugin_icon.png')

        # the items below might be not needed and can be deleted if the
        # plugin does not have a configuration panel
        # self.resources.append('template-plugin_generic.glade')
        # self.builder = self.get_dialog('template-plugin_generic')
        # self.plugin_panel = None
        # self.forward_allowed = False        # forward not enabled by default

        # mandatory or anyway structural variables and object values follow:
        self.command_line = None              # full command line to run
        self.summary_description = None      # must be set for all plugins

        # this variable is defined here only for demonstrational purposes
        # self.value = None

    # def get_pane(self):
    #     if self.plugin_panel is None:
    #         o = self.builder.get_object
    #         self.plugin_panel = o('viewPlugin')
    #         self.builder.connect_signals(self)
    #     return self.plugin_panel

    # all following methods are optional
```

```
    # def click_btnDo(self, obj):
    #     o = self.builder.get_object
    #     o('txtEntry').set_text("Some text")

    # def change_entry(self, obj):
    #     o = self.builder.get_object
    #     self.value = o('txtEntry').get_text()
    #     if self.value:
    #         self.summary_description = _(
    #             "Something will be done with %s") % self.value
    #         self.allow_forward(True)
    #     else:
    #         self.summary_description = None
    #         self.allow_forward(False)


# end.
```

which looks definitely simpler. Some paperwork is needed for the plugin to work, so the "anagraphic" details have to be defined. This is done via the invocation of the base constructor:

```
def __init__(self):
    CommandConditionPlugin.__init__(
        self,
        basename=plugin_name(__file__),
        name=_("Fire This"),
        description=_("Expect a file called 'fire.this' in the home directory"),
        author="Francesco Garosi",
        copyright="Copyright (c) 2016",
        icon='file',
        help_string=HELP,
        version="1.0.0",
    )
```

The `icon` parameter has been changed to `file` because in the stock icons directory (all of which are kindly provided by icons8 under the Good Boy License) [4] there is a `file.png` icon, which is more suitable than the *puzzle* default icon. However it is still not the best option for this plugin, and it may change in further development. Also, the long help string has to be changed into something helpful, like

```
HELP = _("""\
This is a sample command based condition plugin: it will only fire when it
finds a file called ~/fire.this (that is, created in the home directory
with this specific name but regardless of the contents).
""")
```

Next, the only needed features are:

- a command line
- some text that would explain what the plugin will do in the summary pane.

The second one is not strictly needed: if skipped, it defaults to the plugin description. However it is better to give more detailed information especially if it can contain references on how the plugin has been possibly configured. Such information can be given as in the `summary_description` attribute in string form.

To test if there is a file called *fire.this* in the home directory, the following command is more than sufficient:

[4] Needless to say that I love *icons8*.

```
test -f ~/fire.this
```

and it is exactly what the `command_line` attribute will contain.

```
self.command_line = "test -f ~/fire.this"
self.summary_description = "On creation of a 'fire.this' file in the home directory"
```

Note that summary_description should be quite short too, for it should fit in a short text line. The plugin source code now looks like the following (where commented out lines are omitted for clarity):

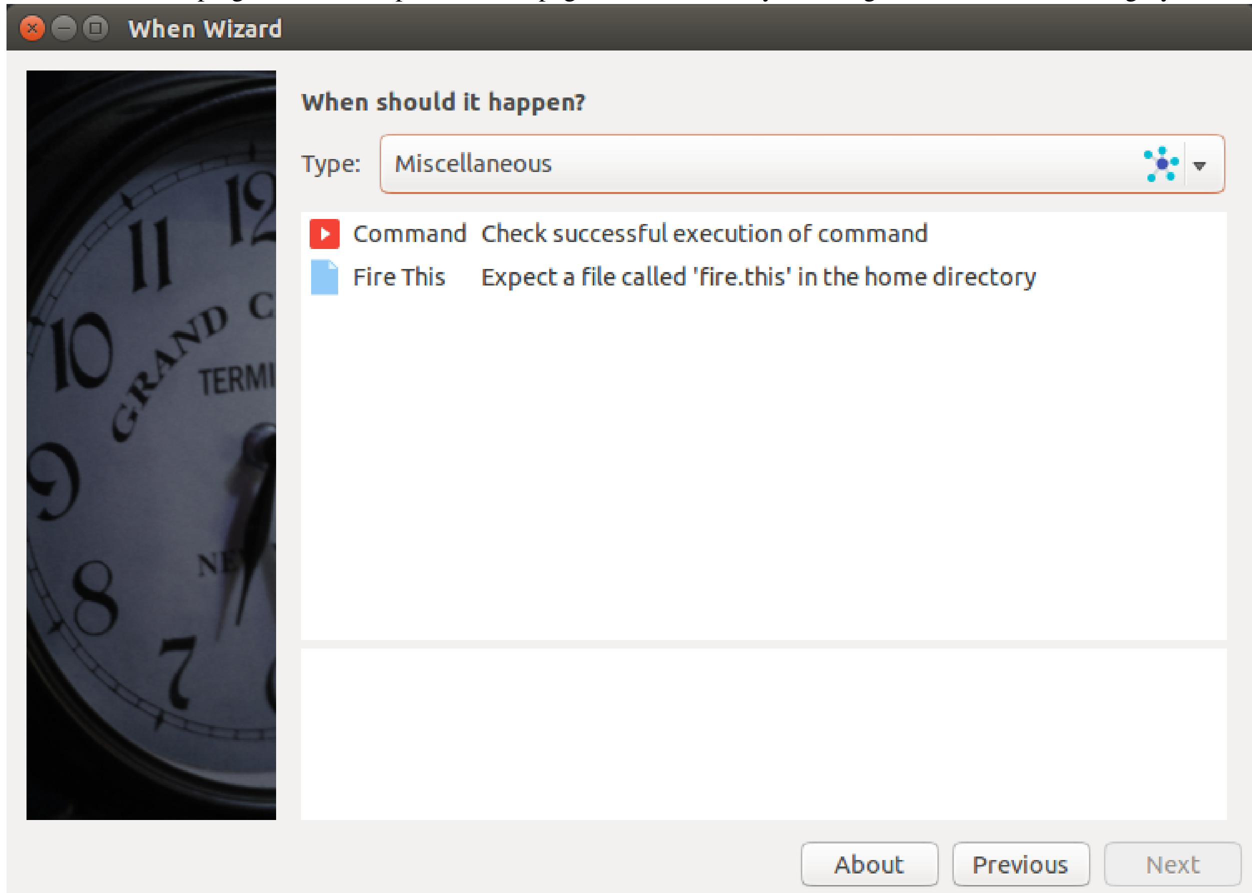
```
# file: firethis.py
# -*- coding: utf-8 -*-
#
# A very basic command-based condition plugin
# Copyright (c) 2015-2016 Francesco Garosi
# Released under the BSD License (see LICENSE file)


from plugin import CommandConditionPlugin, PLUGIN_CONST, plugin_name


# if localization is supported, uncomment the lines above configure
# them as appropriate, and remove this replacement function
def _(x):
    return x


HELP = _("""\
This is a sample command based condition plugin: it will only fire when it
finds a file called ~/fire.this (that is, created in the home directory
with this specific name but regardless of the contents).
""")


# class for a plugin: the derived class name should always be Plugin
class Plugin(CommandConditionPlugin):

    def __init__(self):
        CommandConditionPlugin.__init__(
            self,
            basename=plugin_name(__file__),
            name=_("Fire This"),
            description=_("Expect a file called 'fire.this' in the home directory"),
            author="Francesco Garosi",
            copyright="Copyright (c) 2016",
            icon='file',
            help_string=HELP,
            version="1.0.0",
        )

        # mandatory or anyway structural variables and object values follow:
        self.command_line = "test -f ~/fire.this"
        self.summary_description = "On creation of a 'fire.this' file in the home directory"


# end.
```

and is actually a *working* plugin, that does exactly what it says. To prove it it can be tested in place: assuming it is being developed in the `firethis` subdirectory of the home directory, and assuming that the **When Wizard** launcher

is in the PATH variable, as said above a single environment variable definition is needed:

```
~$ export WHEN_WIZARD_DEVPLUGIN="$HOME/firethis"
~$ when-wizard start-wizard
```

and the condition plugin will show up in the third page of the wizard, by selecting the *Miscellaneous* category.



3.6.3 Step 3: Allow Plugin Configuration

The plugin could be made more generic, by letting the user choose the name of the file to watch for. For the purposes of this example things are kept as easy as possible and no file or directory chooser dialog is used, but nothing forbids to use such utilities, and in fact many stock plugins do. Of course the configuration pane can be built from scratch using *Python* code, but in this case a resource file will be used, and edited with the Glade Interface Designer. The template directory contains a simple resource file, template-plugin_generic.glade, that can work as a starting point. From within the plugin development directory:

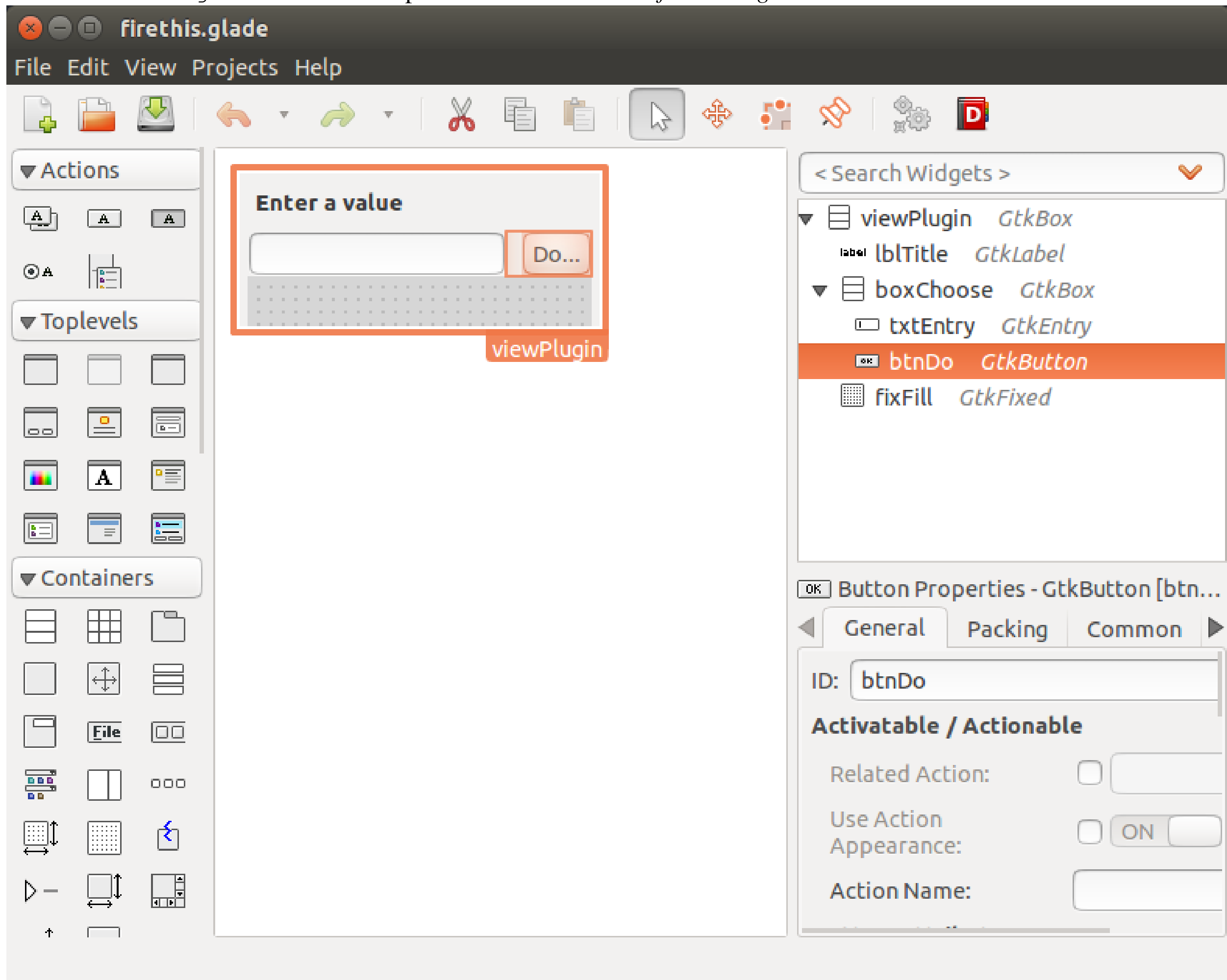
```
~/firethis$ cp /usr/share/when-wizard/templates/template-plugin_generic.glade .
~/firethis$ mv template-plugin_generic.glade firethis.glade
```

Also, since the icon is not very convincing, and assuming that a suitable 24x24 pixel PNG has been stolen from the icons8 web site (please, be kind to them, I think I'm abusing their patience) and is in ~/Downloads, the following step will help give the plugin a nicer icon: [5]

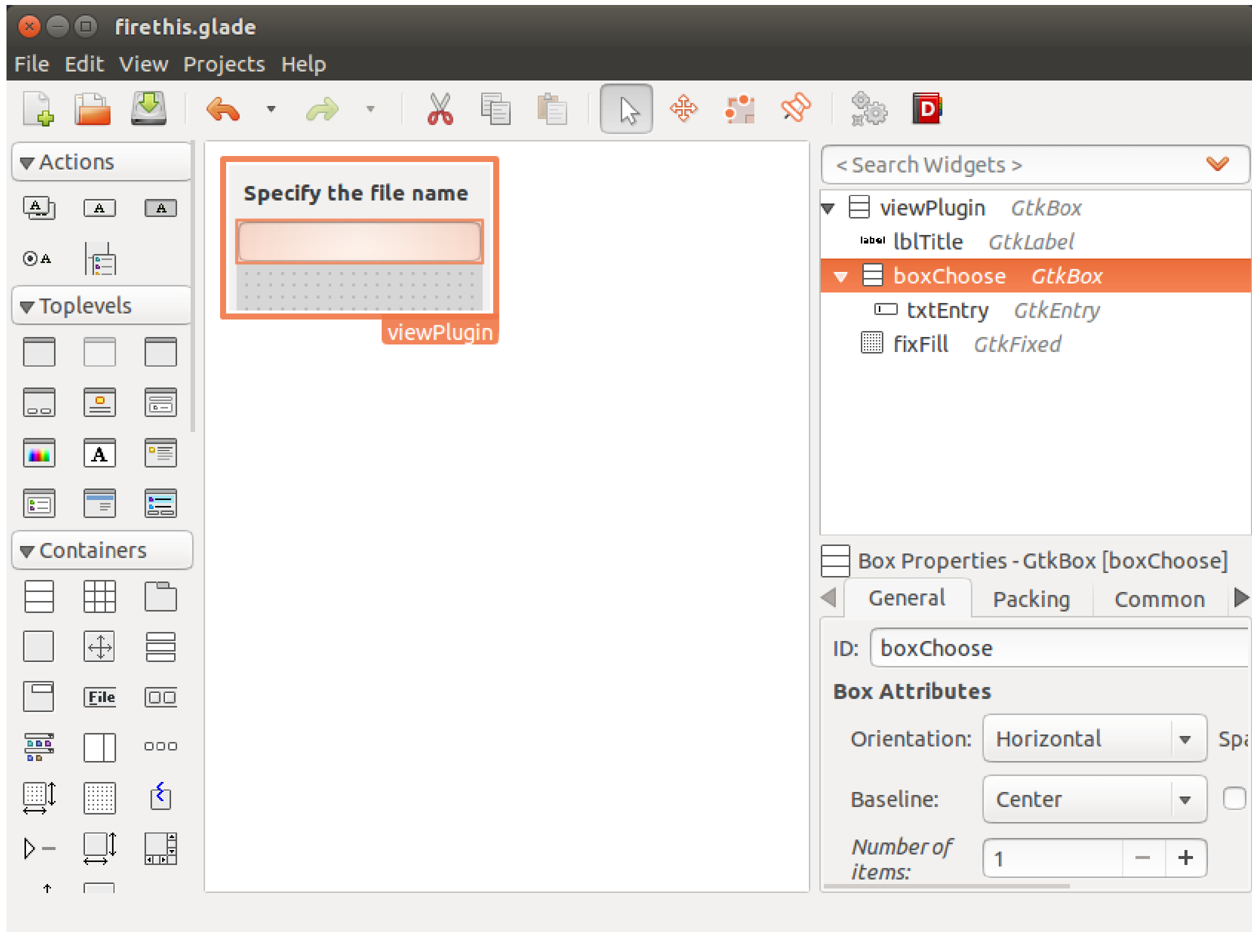
```
~/firethis$ mv ~/Downloads/Fire\ Element-24.png firethis.png
```

[5] I chose the *Fire Element* icon, and their site offers the possibility to download an already resized icon in a custom size.

The `firethis.glade` file can be opened in the *Glade Interface Designer*:



but the *Do* button is not needed, and the entry field should fit the entire width of the pane. Thus, after getting rid of the button, the size of the *boxChoose* box can be reduced to 1:



and the label text can be turned into something more explicative. As for the control names, they can be modified at pleasure, as long as they are correctly referred to in the code.

The *txtEntry* field already has a handler for the *changed* event, that points to a function called `change_entry`, thus it has to be edited in the plugin code. The commented out one can be used in this case:

```
def change_entry(self, obj):
    o = self.builder.get_object
    filename = o('txtEntry').get_text()
    if filename:
        self.summary_description = _(
            "On creation of a '%s' file in the home directory") % filename
        self.command_line = "test -f ~/'%s'" % filename
        self.allow_forward(True)
    else:
        self.summary_description = None
        self.command_line = None
        self.allow_forward(False)
```

The `allow_forward(bool)` function is used to tell the wizard that the *Forward* button can be enabled (on `True`) or disabled (on `False`). The reference to the `value` variable can be removed in the constructor because a local variable has been used to create the command line, and the code that helps build the pane should be restored. Also, the plugin must be instructed to consider resource files for automatic installation. The following code goes in the constructor, after the call to the base class constructor.

```
# the append steps inform the plugin installer of the resource files
self.graphics.append('firethis.png')
self.resources.append('firethis.glade')

# here the pane is prepared in the same way as a dialog box, but
# it is not initialized: the initialization is deferred to the first
# attempt to retrieve the pane
self.builder = self.get_dialog('firethis')
self.plugin_panel = None
self.forward_allowed = True

# the default command line is almost the same as before
self.command_line = "test -f ~/'fire.this'"
self.summary_description = \
    "On creation of a 'fire.this' file in the home directory"
```

Note the `forward_allowed` attribute set to `True`: this authorizes the wizard container to keep the *Forward* button enabled as soon as the pane shows up. This is intentional, because the text entry is initialized with the default file name in the pane initialization step below.

The last thing to restore is the `get_pane` function, otherwise the plugin will still have no configuration possibility. The pane initialization step will be performed here instead of overburdening the constructor:

```
def get_pane(self):
    if self.plugin_panel is None:
        o = self.builder.get_object
        self.plugin_panel = o('viewPlugin')
        self.builder.connect_signals(self)
        o('txtEntry').set_text('fire.this')
    return self.plugin_panel
```

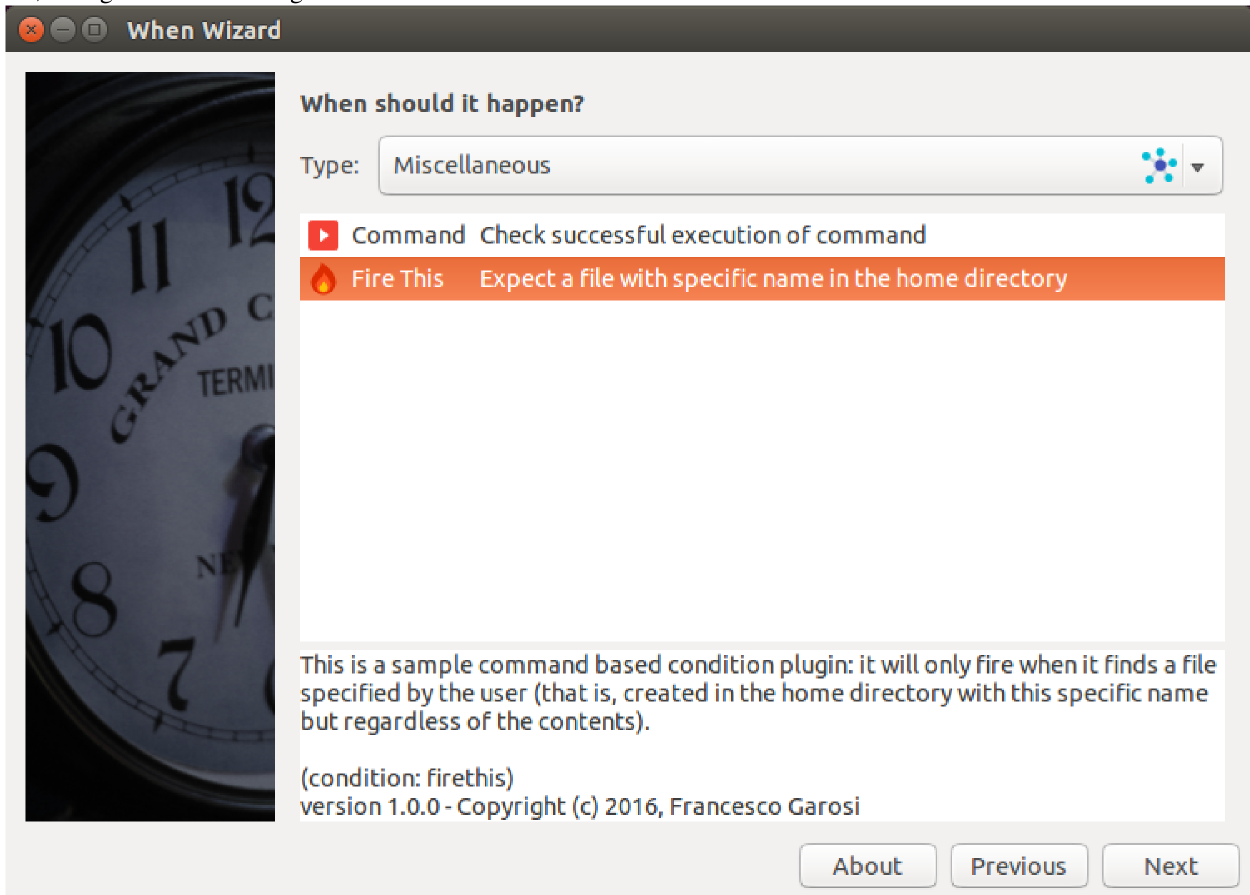
The default value of the text entry is set only in the initialization step so that when the user navigates back and forth between pages it will not be reset to the default value. The complete plugin file is the following:

```
# file: firethis.py
# -*- coding: utf-8 -*-
#
# A very basic command-based condition plugin
# Copyright (c) 2015-2016 Francesco Garosi
# Released under the BSD License (see LICENSE file)


from plugin import CommandConditionPlugin, PLUGIN_CONST, plugin_name


# if localization is supported, uncomment the lines above, configure
# them as appropriate, and remove this replacement function
def _(x):
    return x


HELP = _("""\
This is a sample command based condition plugin: it will only fire when it
finds a file specified by the user (that is, created in the home directory
with this specific name but regardless of the contents).
""")


# class for a plugin: the derived class name should always be Plugin
```

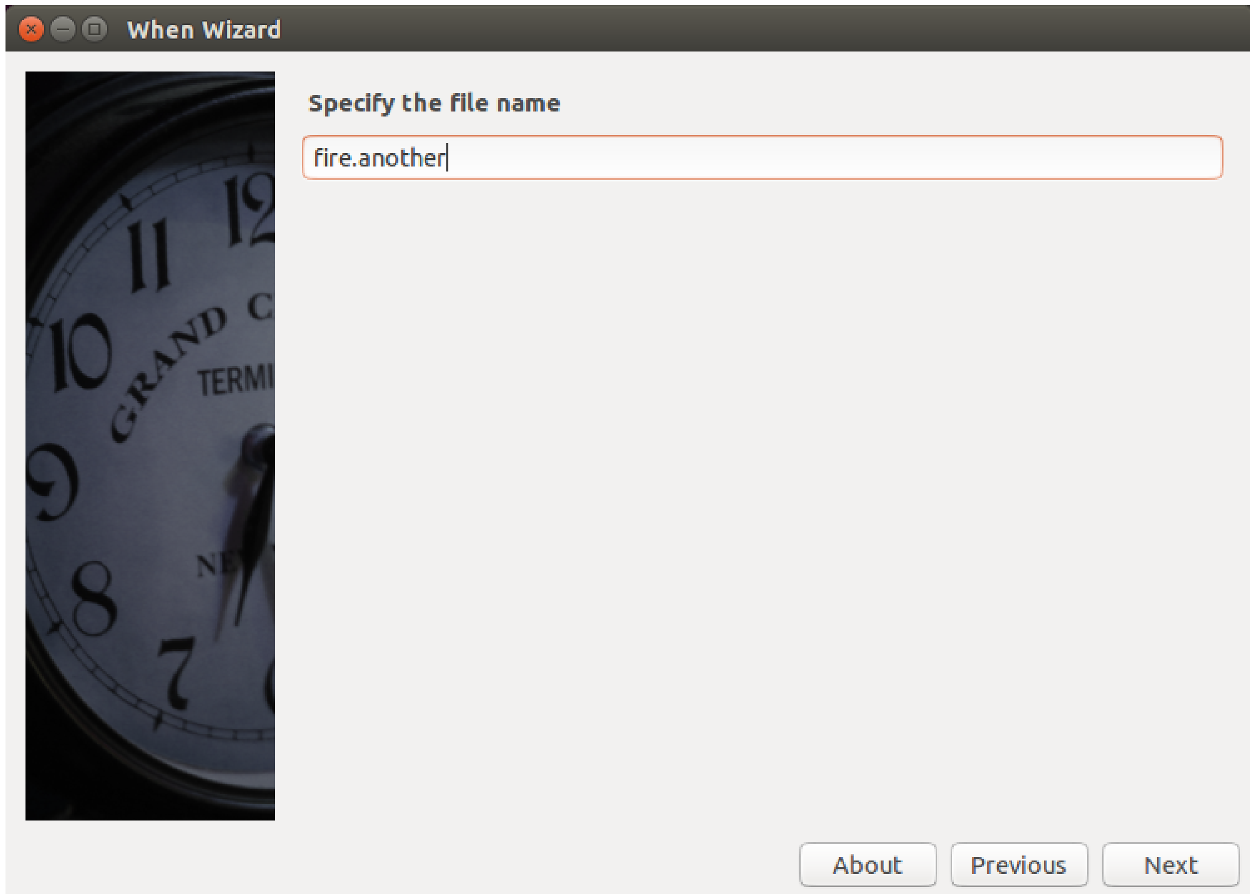
```
class Plugin(CommandConditionPlugin):

    def __init__(self):
        CommandConditionPlugin.__init__(
            self,
            basename=plugin_name(__file__),
            name=_("Fire This"),
            description=_(
                "Expect a file with specific name in the home directory"),
            author="Francesco Garosi",
            copyright="Copyright (c) 2016",
            icon='firethis',
            help_string=HELP,
            version="1.0.0",
        )
        # the append steps inform the plugin installer of the resource files
        self.graphics.append('firethis.png')
        self.resources.append('firethis.glade')

        # here the pane is prepared in the same way as a dialog box, but
        # it is not initialized: the initialization is deferred to the first
        # attempt to retrieve the pane
        self.builder = self.get_dialog('firethis')
        self.plugin_panel = None
        self.forward_allowed = True

        # the default command line is almost the same as before
        self.command_line = "test -f ~/'fire.this'"
        self.summary_description = \
            "On creation of a 'fire.this' file in the home directory"

    def get_pane(self):
        if self.plugin_panel is None:
            o = self.builder.get_object
            self.plugin_panel = o('viewPlugin')
            self.builder.connect_signals(self)
            o('txtEntry').set_text('fire.this')
        return self.plugin_panel

    def change_entry(self, obj):
        o = self.builder.get_object
        filename = o('txtEntry').get_text()
        if filename:
            self.summary_description = _(
                "On creation of a '%s' file in the home directory") % filename
            self.command_line = "test -f ~/'%s'" % filename
            self.allow_forward(True)
        else:
            self.summary_description = None
            self.command_line = None
            self.allow_forward(False)


# end.
```

Note that the `description` parameter for the base constructor has been modified to better describe the plugin, and the icon name has been changed to `'firethis'` which is the base name of the custom icon. The `HELP` text above was also slightly modified to reflect the behavior. Calling the wizard with the “development” environment variable

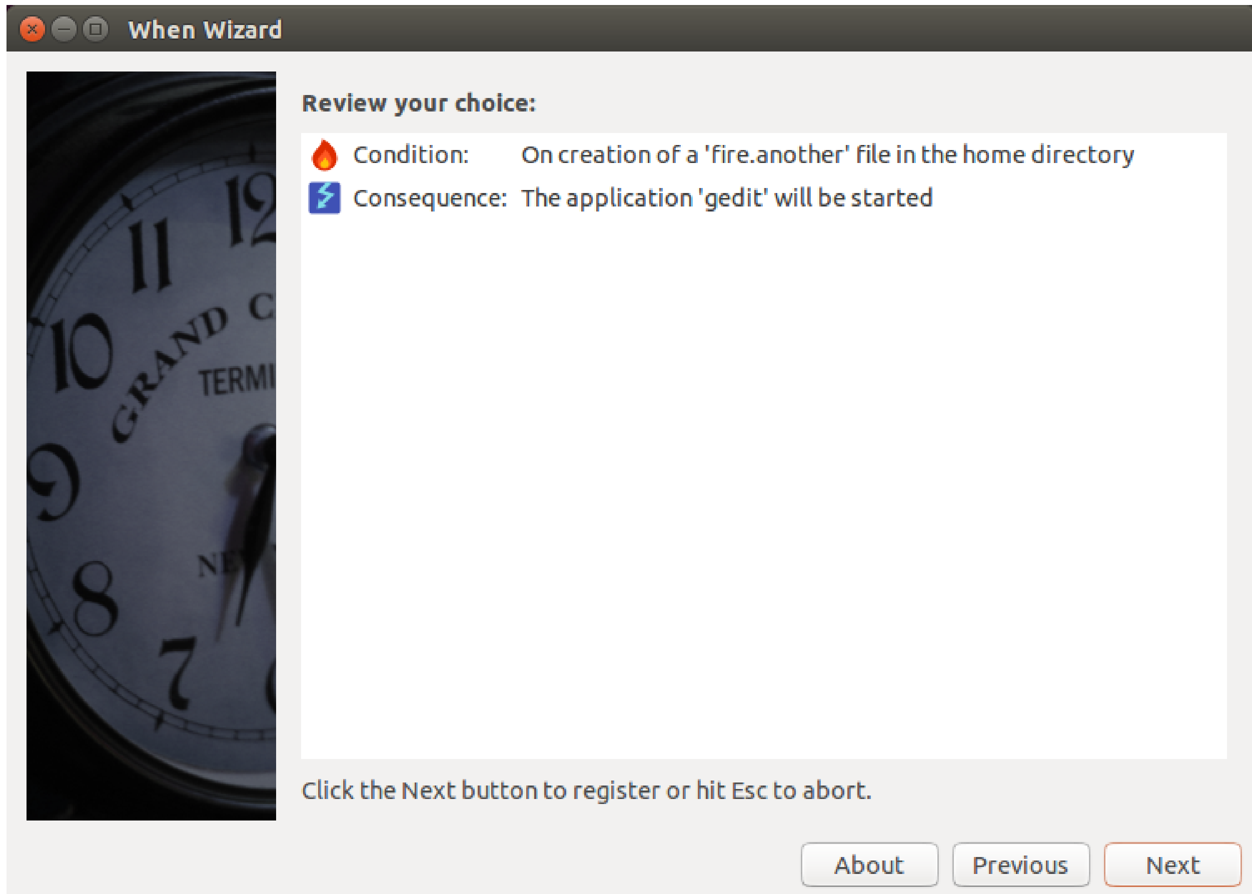
set, now gives the following choice for *Miscellaneous* conditions:



which gives the possibility to modify the default value:



and such possible modification is reflected in the summary and confirmation page of the **When Wizard**:



More complex and complicated plugins can be created using this simple pattern and starting from the appropriate template. The steps followed for this plugin are very similar for *task* plugins too, with the aforementioned exceptions. The complete sample plugin code can be downloaded here as well as the pane resource file and the icon.

3.6.4 Step 4: Packaging

To make distribution of plugins easier, a convenient packaging utility has been included in the **When Wizard** suite, as mentioned above. To create a package for the `firethis` plugin, it is sufficient to issue the following commands in a terminal window:

```
~/firethis$ cd ..
~$ when-wizard plugin-package firethis
```

This will create a file with a name like `firethis.14346484091d5400.wwpz` (the string between the two dots will be different) which will be recognized by the installation page of the **When Wizard Manager** application. The plugin can be installed and it will be usable in the **When Wizard** without having to set the development environment variable.

Note: A plugin package is nothing special: it just consists of a flat *zip* file containing all the files declared in the plugin constructor section, plus the plugin code file itself, with a *.wwpz* extension. This approach was chosen in order to allow, for instance, to download the *zip* file for a GitHub repository and install it as a plugin directly: the extra files are simply ignored and skipped during installation. However, as the graphical installation utility will not recognize *.zip* as a suitable extension, either the downloaded file is renamed or the console utility is used, as in `when-wizard plugin-install firethis-master.zip` for a hypothetical repository of the `firethis` plugin used in the

examples.

3.7 How to Choose a Suitable Name

Plugins are installed in a flat fashion in the user home: there are three directories in `~/.local/share/when-command/when-wizard` for plugin code, resources and scripts. If two plugins share the same *base name*, the most recently installed plugin overwrites the former. Same occurs for other files that the plugin provides, so it's advisable to:

- choose a base name that describes the plugin behavior as precisely as possible, with no concerns for the length: this will reduce the chances of a conflict
- prefix resource, graphic, and script file names with the base name of the plugin itself.

Since the **When** item names are constructed using the base name of the plugin itself, it comes as a consequence that such base names must obey the naming rules for **When** items, that is they can only consist of letters, digits, dashes and underscores. A plugin base name could start with a dash or an underscore, but it's advisable to choose a letter anyway. **When** will simply refuse to use items with non compliant names.

3.8 Parametric Item Definition Files

Another way to provide an user with complex actions that wouldn't be easy to set up is through *item definition files*. As per the **When** manual, **When Wizard** chapter, the user can easily specify an IDF to import using the **When Wizard Manager**, which saves her or him from the command line. If the IDF is provided with the `.widf` file extension, it can also be selected through a convenient file chooser dialog box.

Unfortunately IDFs are not easy to modify: if configuration for a certain item combination is needed, dealing with a text file might lead to mistakes that cause **When** to refuse the file, or even worse to monitor the wrong things. That is where the **When Wizard** suite comes to help, thanks to the possibility of specifying *parameters* within the file itself. If the manager application encounters a parametric IDF during import, it shows a dialog box to the user containing all the entries that correspond to parameters that can be configured. Each entry is pre-filled with a default value that the IDF developer has provided, and for each value there is the possibility to add a validity check, so that invalid values will not be accepted in the first place.

Parameters are specified in the *item definition file* with special lines that have the following form:

```
@param_name Description:t[ype]:default[:validity_check]
```

where `param_name` is an identifier starting with a letter and containing only letters, digits and underscores, in a case sensitive fashion. The `Description` field is what will appear in the label for the entry field in the configuration dialog box: it can contain spaces. `type` is one of `string`, `integer`, `real`, `choice`, `file`, and `directory`, or any abbreviation thereof. `default` is obviously the default value (mandatory) and the optional validity check depends on the entry type. Possible validity checks are:

- a regular expression for `string` entries
- a `min:max` (separated by a colon) for numeric entries
- a comma separated list of strings for `choice` entries: in fact in this case it's almost mandatory to provide the list because choices are shown in a drop-down combo box, which would only contain the default value if no list is specified.

Entries for files and directories can not be checked, however the interface will provide appropriate file chooser dialog boxes to help the user. Apart from the parameter name, which is separated from the rest of the line by blank characters,

the definition line is composed by fields separated by colons. To include a colon in the default value, it has to be prefixed with a backslash. A backslash too has to be prefixed by a backslash to be shown.

The resulting dialog box will show parameters to be configured *in the same order* as they appear in the parametric IDF, so if there is a consequential rationale for parameter order it has to be reflected in the definition file.

Parameters must appear within the regular lines of the IDF in their full form, that is `@param_name` – an *at* sign followed by the identifier.

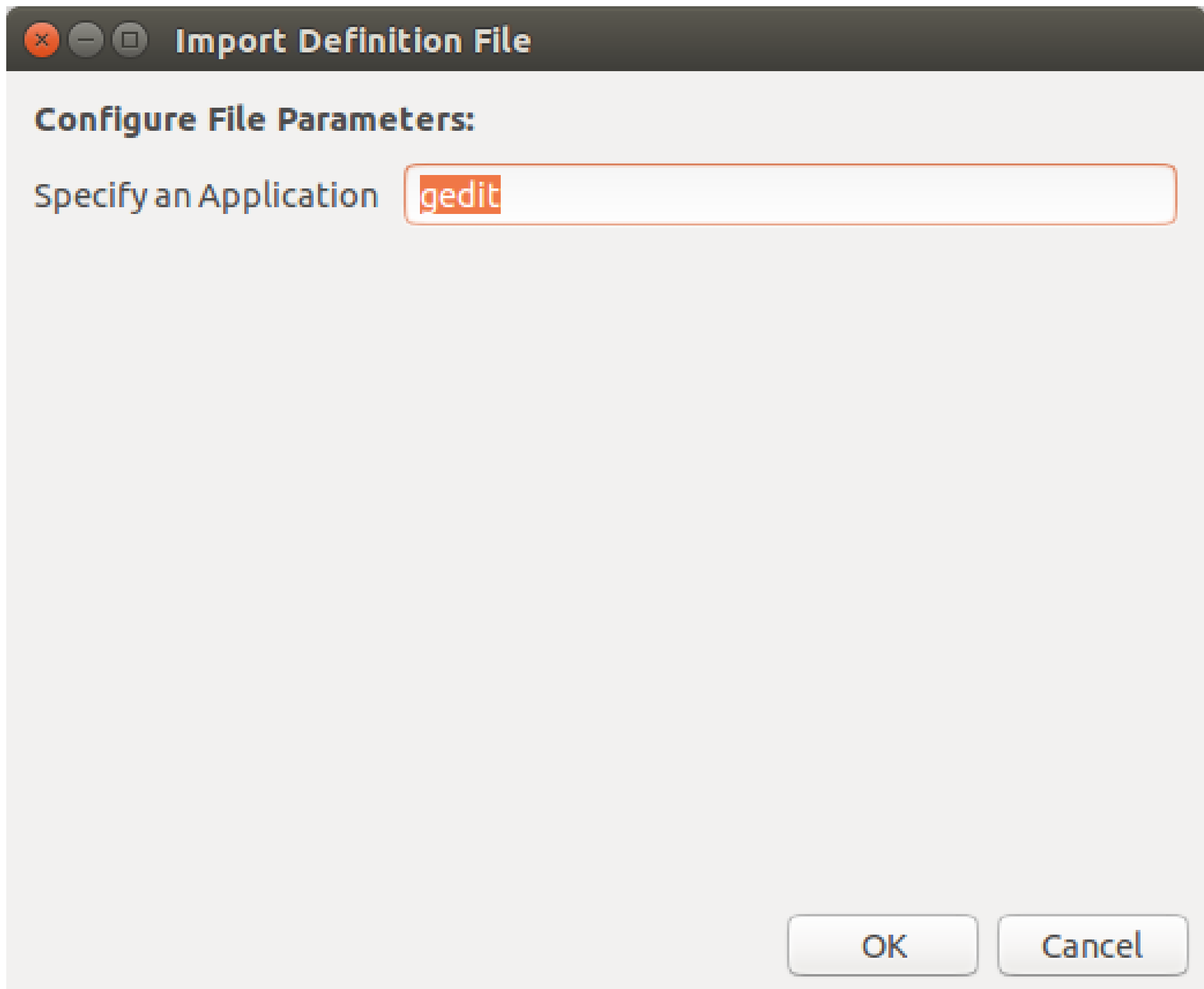
Parameters are replaced *textually* in the *item definition file*: even if they are substrings of a longer identifier their occurrences will be substituted. However, if a parameter is a prefix for another, the manager application will take care to avoid that the shorter one is confused with the longer one. There are chances that, if a parameter *occurrence* is accidentally entered by the user, it can be replaced if a parameter with a matching name is part of the IDF's parameter set. Parameters can be thought of as *macros*, to some extent.

To make things clearer, a simple example is hereby provided.

```
# Test that a certain application has been started started

[AppsChanged]
type: signal_handler
bus: session
bus name: org.ayatana.bamf
object path: /org/ayatana/bamf/matcher
interface: org.ayatana.bamf.matcher
signal: RunningApplicationsChanged
parameters:
  0, contains, /usr/share/applications/@app.desktop

[ShowBadge_AppsChanged]
type: task
command: notify-send -i info "Apps Changed" "The application '@app' has been started."
check for: nothing

[Check_AppsChanged]
type: condition
based on: user_event
event name: AppsChanged
task names: ShowBadge_AppsChanged


# Parameter
@app Specify an Application:string:gedit:[a-zA-Z0-9_-]+$

# end.
```

This example uses the *BAMF* daemon to verify that a certain application has been started in the graphical environment. It's by far and away an over simplification, as not all the `.desktop` files reside in the `/usr/share/applications` directory, but it demonstrates how to use a parameter in an *item definition file*. Apart from the parameter line and the occurrences of the `@app` token, it is a normal IDF with a task, a signal handler and a condition depending on that handler. It simply displays a badge whenever an application whose desktop file is recognized as matching with `@app.desktop` is started.

If an user tries to import it in the **When Wizard Manager**, the following dialog box is shown:



where the user can enter an appropriate application name that should comply with the specified regular expression. As stated above, the *description* is used to prefix the text entry that is available to the user, and the text entry itself comes with the provided default value of `gedit`. If the user accepts the default, *gedit* will be monitored and a badge will be shown each time it is started.

CHAPTER 4

Packaging

In order to build a package that is compatible with the Linux FHS and LSB, many changes have been introduced in the directory structure of the **When** source tree. The most significant changes are:

- a new position for the main applet script
- a slightly different hierarchy in the `share` directory, with the introduction of
  - a folder for standard icons in different sizes, under `share/icons/hicolor/<size>/apps/when-command`
  - a `share/doc/when-command` folder which contains documentation that is installed with the applet (`README.md`, `LICENSE` and `copyright`)
  - the `share/when-command` folder containing all the resources and the main applet script (`when-command.py`)
- the files needed by the standard Python setup script, as well as the setup script itself (namely `setup.py`, `setup.cfg`, `stdeb.cfg` and `MANIFEST.in`), have been added to the project
- a stub file that will serve as the main entry point to start `when-command` instead of invoking the main applet script directly
- other files required by the utilities used to build the Debian package.

Other changes involve the code itself: parts of the script has been modified in order to allow better recognition of the *LSB-based* installation (the one that expects the entry point to be installed in `/usr/bin` and data files in `/usr/share`), even though the possibility has been kept to build a package that installs **When** in `/opt` as it has been usual until now. From now on the preferred installation mode will be the *LSB-based* one, the `/opt` based package is supported on a "best effort" basis for whoever would want to keep **When** separated from the Linux installation.

Unfortunately the new directory setup could require some more effort to allow for local installations (e.g. in the user's home directory), although I'll try to do my best to make this process as easy as possible.

4.1 Requirements for Packaging

When uses Python 3.x `setuptools` (package `python3-setuptools`, it is possibly already installed on the system) to create the source distribution used bu the packaging system. Most information about how to package an application has been retrieved in Packaging and Distributing Projects, in Introduction to Debian Packaging and Python libraries/application packaging, as well as in the setuptools documentation. Especially, the `stdeb` for Python 3.x has been used: this package is not provided by the official repository in *Ubuntu 14.04*, so a *pip* installation may be required:

```
$ pip3 install --user stdeb
```

Also, to build a `.deb` package, the standard `debhelper`, `build-essential` and `fakeroot` packages and tasks are needed. I also installed `python-all`, `python3-all`, `python-all-dev`, `python3-all-dev` and `python-stdeb` (which is available, but it is for Python 2.x and quite old), but they might be unnecessary.

4.2 Package Creation: LSB Packages

As far as I'm concerned, this step can be considered black magic. I expected packaging to be a relatively simple thing to do, something more similar to stuffing files into a tarball and then adding some metadata to the archive to allow for the installation tools to figure out how things have to be done. Apparently there is much more than that, especially when it comes to Python applications. And when the main entry point of such a Python application contains a dash, things get worse: none of the standard installation methods that use the `setup.py` script seems to be suitable. That is why, for instance, the `when-command.py` script is considered a data file in the whole process, whereas a stub script named `when-command` (with no extension) is marked as *script*: we will not use the `entry_points` setup keyword, because we don't absolutely want `setup.py` to generate the stub script for us, since the *supposed-to-be-library* file contains a dash and could be not imported in an easy way.

However, here are the steps I perform to build a `.deb` package.

4.2.1 The Easy Way with `setup.py`

After unpacking the source tree, the following commands can be used to easily build the `.deb` package:

```
$ cd <when-source-tree>
$ python3 setup.py --command-packages=stdeb.command bdist_deb
```

The `python3 setup.py ... bdist_deb` actually builds a `.deb` file in the `deb_dist` directory: this package is suitable to install **When**. The same `deb_dist` directory also contains a source package, in the form of a `.dsc` file, `.orig.tar.gz` and `.debian.tar.gz` archives, and `.changes` files. However the `.dsc` and `changes` files are not signed: to upload the package to a *PPA*, for instance, they need to be signed using `debsign`. [1]

4.2.2 Using the Packaging Utilities Directly

First a source distribution has to be created: the `setup.py` script comes handy in this case too because it can do this job automatically using the `sdist` command. After the source tree has been unpacked or cloned, the following operations will create a proper source distribution of **When** and move it to the top of the source tree:

```
$ cd <when-source-tree>
$ python3 setup.py sdist
$ mv dist/when-command-<version_identifier>.tar.gz .
```

where `<version-identifier>` is the suffix of the newly created archive in the `dist` subdirectory. Then use the `py2dsc` tool to create the structure suitable for packaging :

```
$ py2dsc -m "$DEBFULLNAME <$DEBEMAIL>" when-command-<version_identifier>.tar.gz
$ cd deb_dist/when-command-<version_identifier>
```

The guide in Python libraries/application packaging suggests then to edit some files in the `debian` subdirectory, namely `control` and `rules`. The files should read as follows:

control:

[1] Just `gpg --clearsign` is not sufficient because file checksums change in the process.

```
Source: when-command
Maintainer: Francesco Garosi (AlmostEarthling) <franz.g@no-spam-please.infinito.it>
Section: misc
Priority: optional
Build-Depends: python3-setuptools, python3, debhelper (>= 7.4.3)
Standards-Version: 3.9.5
X-Python3-Version: >= 3.4

Package: when-command
Architecture: all
Depends: ${misc:Depends}, ${python3:Depends}, python-support (>= 0.90.0), python3-gi, xprintidle, gi
Description: When Gnome Scheduler
 When is a configurable user task scheduler, designed with Ubuntu
 in mind. It interacts with the user through a GUI, where the user
 can define tasks and conditions, as well as relationships of
 causality that bind conditions to tasks.
```

rules:

```
#!/usr/bin/make -f

%:
	dh $@ --with python3

override_dh_auto_clean:
	python3 setup.py clean -a
	find . -name \*.pyc -exec rm {} \;

override_dh_auto_build:
#	python3 setup.py build --force

override_dh_auto_install:
	python3 setup.py install --force --root=debian/when-command --install-layout=deb --install-lib=

override_dh_python3:
	dh_python3 --shebang=/usr/bin/python3
```

Since we use a stub file, no `links` specification is actually necessary. This in fact differs from the advices given in the aforementioned guide: instead of specifying the target directory for *scripts* as `/usr/share/when-command` (same as the main script) in the package creation `rules`, we let the package install the stub in `/usr/bin` directly and don't rely on symbolic links. The package creation procedure is slightly simplified in this way, and provides a tidier setup. Also, the comment in the `override_dh_auto_build` rule is intentional, and better explained in the guide.

To build the package the standard Debian utilities can be used in the following way:

```
$ cd <source-directory>
$ pkgdir=deb_dist/when-command-<version_identifier>
$ cp $pkgdir/share/doc/when-command/copyright $pkgdir/debian
$ cd deb_dist/when-command-<version_identifier>
$ debuild
```

The package is in the `deb_dist` directory. After entering the source directory, the first two lines just synchronize the `copyright` file from the unpacked source tree to the `debian` "service" directory just to avoid some of the complaints that `lintian` shows during the build process, while the last two lines are the commands that actually build the Debian package.

This process also creates a source package in the same form as above, with the exception that the `.dsc` and `.changes` files should be already signed after the process if the environment is correctly configured. In fact, to

build the package, the DEBFULLNAME and DEBEMAIL environment variables are required, and must match the name and e-mail address provided when the *GPG key* used to sign packages has been generated: see the Ubuntu Packaging Guide for details.

At a small price in terms of complexity, this method has one main advantage over the "easy" one as it allows some more control on packaging by allowing to review and edit all the package control files before creation.

4.3 Package Creation: the Old Way

As suggested above, a way to build the old /opt based package is still available. I use a script that moves all files in the former locations, removes extra and unused files and scripts, and then builds a .deb that can be used to install the applet in /opt/when-command. This file can be found in a GitHub gist, together with the control_template file that it needs to build the package. It has to be copied to a suitable build directory together with control_template, made executable using chmod a+x makepkg.sh, modified in the variables at the top of the file and launched.

CHAPTER 5

Test Suite

As of version *0.7.0-beta.1* **When** has an automated test suite. The test suite does not come packaged with the applet, since it wouldn't be useful to install the test scripts on the user machine: instead, it's stored in its dedicated repository, see the specific `README.md` file for more details.

Whenever a new feature is added, that affects the *background* part of **When** (i.e. the loop that checks conditions and possibly runs tasks), specific tests should be added using the test suite "tools", that is:

- the configurable *items* export file
- the *ad hoc* configuration file
- the test functions in `run.sh`.

It has to be noted that, at least for now, the test suite is only concerned about *function* and not *performance*: since **When** is a rather lazy applet, performance in terms of speed is not a top requirement.

CHAPTER 6

DBus Remote API

Starting with version *v0.9.7-beta.3* (not corresponding to a packaged release though) **When** has gained a *remote API* through DBus. This API can be used to control various aspects of a running instance of **When**, so that it can even be almost completely managed by an external application. Operations available through the remote API cover:

- managing all types of items: *tasks*, *conditions* and *signal handlers*
- managing the configuration and configuration file
- pausing or resetting a running instance
- retrieving current history

and more. This interface has been created to allow development of a companion application, the upcoming **When Wizard**, that will provide a different and easier way to manage **When** letting it only perform as a mere engine.

When exposes methods that are somehow *reserved* for itself (mainly the ones that allow communication between the command line utility and a running applet instance), *and* also methods that are available for external control. This section only briefly documents the former ones, while trying to be more extensive with the latter type.

6.1 Interface

Details on how to estabilish DBus a connection to the **When** applet follow:

Item	Value
Bus	*session*
Application ID	`it.jks.WhenCommand`
Unique Bus ID	`it.jks.WhenCommand.BusService`
Object Path	`/it/jks/WhenCommand/BusService`
Interface	`it.jks.WhenCommand.BusService`

These values can be used to build a *proxy* to the **When** interface, see the DBus documentation for more details. Specifically, to build a proxy in *Python* the following model can be used:

```
import dbus

bus = dbus.SessionBus()
proxy = bus.get_object('it.jks.WhenCommand.BusService',
                       '/it/jks/WhenCommand/BusService')
```

It will give access to the whole API in the form `proxy.call(p1[, p2 ...])` where `call` is the name of an API method (see below) and `pN` are the parameters expected by the method.

6.2 General Use Methods

The following methods have been designed aiming at interoperability, thus they are useful for the purposes explained above.

Method	Description
`AddItemByDefinition(dic, save)`	add an item to the collection of items managed by **When**, and optionally save the collection where the item belongs. The item must be provided in dictionary form, as specified below, in the `dic` parameter, while the `save` argument is a boolean indicating whether or not to save the collection; returns *True* on success, *False* ortherwise [1]
`AddItemsBatch(item_data)`	add multiple items using a string in the *item definition file* format (see the user guide for detailed information); the argument should follow the format exactly, with newlines, indents and so on; returns *True* on success, *False* ortherwise
`GetConfig(section, entry)`	return a value (enclosed in a *variant*) from the running configuration, which in turn most likely reflects the same value in the configuration file; `section` and `entry` are strings [2]
`GetHistoryEntries()`	return the list of entries in thehistory of the running instance: the entries are returned as a list of strings, corresponding each to a line of an exported history file – except for headers – that is a semicolon separated list of values
`GetItemDefinition(item_spec)`	given an *item specification* [3] as argument, return the definition of the corresponding item as a mapping (dictionary) if it exists [1]
`GetItemNames(item_type)`	return a list of item names possibly corresponding to the specified type of item if the `item_type` parameter is one of `'tasks'`, `'conditions'` and `'sighandlers'` (or an abbreviation thereof), or all items if the empty string is passed
`IsSuspendedCondition(cond_name)`	tell whether or not the condition whose name is provided as argument is suspended
`Pause(pause)`	set the paused state to the one provided in the *boolean* `pause` argument: paused if *True*, resumed if *False*
`Paused()`	return the current paused state as a self-explanatory *boolean* value
`ReloadConfig()`	reconfigure the applet from static data
`RestartConditions()`	reset the internal flag of conditions that avoids to repeat checks if one test had already been successful, thus reenabling *non-recurring* conditions
`RemoveItem(item_spec)`	given an *item specification* [3] remove the corresponding item; returns *True* on success, *False* orthewise
`Reset(clear_history)`	reset the applet and reload data, similar to a *restart* but without running startup and shutdown actions; if the `clear_history` parameter is set to *True* also clear the current task history
`SaveItems(item_type)`	save all items of the type provided in `item_type` (see `GetItemNames` above on how to specify it), all items are saved when providing the empty string
`SetConfig(sec, ent, v, reload)`	set the configuration entry `ent` at section `sec` in the running applet to the value specified in `v` (which must be provided as a *variant*); if the *boolean* argument `reload` is set to *True* the configuration is reloaded after the operation; returns *True* on success and *False* on failure
`SuspendCondition(cond_name, s)`	if `s` is *True* the condition is suspended, if *False* it is resumed

Item definition dictionaries returned by `GetItemDefinition` and handled by `AddItemByDefinition` are

[1] the DBus documentation explains how to access *DBus dictionaries*; in this particular case the keys are *strings* and values must be enclosed in *variant* objects.

[2] sometimes **When** expects data to be enclosed in a *variant* container: there are several methods to achieve this, including the use of the `GLib.Variant` (`GLib::Variant`) constructor.

[3] a string consisting of the tipe of item (or an abbreviation thereof) in *tasks*, *conditions*, and *sighandlers*, following by a colon and the unique name of the item itself. For example, if there is a task named `SomeTask`, then `task:SomeTask` is a correct item specification (where *task* is actually an abbreviation of the more general *tasks*).

implemented using strings as keys and variants as values.

6.3 Reserved Methods

Methods that should be avoided generally are: `ExportHistory` that exports history entries to a file given its name, `KillInstance` and `QuitInstance` (especiatlly the former) that causes the applet to exit, `RunCLIBasedCondition` that is only used to force a *command-line based* condition to occur and `ShowDialog` to fire up a dialog box. These methods are only used with `when-command` as a controlling utility, and are pretty useless in external applications.

CHAPTER 7

History

7.1 Version 0.9.12 (beta)

- Reset condition tests via menu, command line or wakeup events
- Inspect DBus interfaces for supported signals
- Bug Fixes

7.2 Version 0.9.11 (beta)

- Support external storage events on Xenial

7.3 Version 0.9.10 (beta)

- Add Remote API to suspend conditions
- Bug Fixes

7.4 Version 0.9.9 (beta)

- Remote DBus Interface fixes
- More flexible and robust item conversion functions
- Future Ubuntu version compatibility
- Bug fixes

7.5 Version 0.9.8 (beta)

- Full DBus Interface
- Fix window z-order problems on XUbuntu
- Minor fixes

7.6 Version 0.9.7 (beta)

- Suppress Task History box in Minimalistic Mode
- Ellipsize History Box columns
- Better DBus interface
- Make most configuration changes immediately active
- Code cleanup
- Bug fixes

7.7 Version 0.9.6 (beta)

- Avoid use of *xprintidle* when possible
- Better Idle Time Condition dialog

7.8 Version 0.9.5 (beta)

- Modular stock event management
- More efficient removable storage device detection
- Support for more Linux distributions

7.9 Version 0.9.4 (beta)

- Manage items from the command line
- Use human-readable files to define items
- Minimalistic Mode
- Bug fixes

7.10 Version 0.9.3 (beta)

- Battery status related events
- Minor fixes
- Documentation reorganization and relocation

7.11 Version 0.9.2 (beta)

- New directory structure following LSB FHS
- Standard Python setup script
- Debian and Ubuntu compatible package

7.11.1 Directory structure

The new directory structure is more compliant with the LSB FHS, in order to simplify the production of Debian and Ubuntu compatible packages. To keep things tidy, the main script has been moved to the `share/when-command` directory, and has to be linked by the installation utilities in `/usr/bin` under the name `when-command` (without the `.py` suffix). The *"old style"* `/opt` based installation is still possible: the script in https://gist.github.com/almostearthling/009fbbe27ea5ca921452 can be used to build the appropriate package.

7.12 Version 0.9.1 (beta)

- Support localization and translations using portable objects
- Italian localization provided (also as an example)
- Spanish localization (thanks @fitojb)
- Bug fixes

7.13 Version 0.7.0 (beta)

- Implement generic DBus signal handler and related toolbox
- Some conditions can be activated from the command line
- Conditions based on file and directory changes
- Environment variables with task and condition names
- Refactoring and code simplification
- Export task history to a text file
- Bug fixes

Warning: *Compatibility break*
This release breaks compatibility with previous version regarding the binary format of static data (conditions), as it introduces a new condition type for file notifications. The problem only affects *downgrading* from this to previous releases, upgrades are safe and all static data is correctly loaded. Unless file notification conditions are enabled *and* defined, a downgrade should be safe as well.

7.14 Version 0.6.0 (beta)

- Match regular expressions in command output for tasks and command based conditions
- Stop task sequence on task outcome
- Major bug fixes
- Refactoring for better integration with host environment

Warning: *Compatibility break*

This release breaks compatibility with previous version regarding the binary format of static data (tasks and conditions), as it introduces new parameters in both tasks and command based conditions. A dump and restore of static data is required for **When** to work correctly.

- Before upgrade: `/opt/when-command/when-command --export --shutdown`
- Upgrade: `sudo dpkg --install when-command-0.6.0-beta.1.deb` (or your preferred upgrade method)
- After upgrade: `/opt/when-command/when-command --import`

Then you can start the applet from *Dash* or at the next login. This should be done for all accounts that use *When* on the system.

7.15 Version 0.5.0 (beta)

- More consistent dialog boxes
- Task and condition naming rules
- Command line options for - configuration management - accessing dialog boxes - applet information - applet control
- Import and export static data across incompatible versions

Note: *About compatibility breaks*

This release introduces a way to save static data (tasks and conditions) in a portable format that is not subject to significant changes across versions: this should solve the concern about compatibility breaks when the core structures of the program are modified in an incompatible way.

7.16 Version 0.3.0 (beta)

- Perform shutdown tasks on logout, shutdown and reboot (Issue #8)
- Create autostart directory when not present (Issue #15)
- Keep pause state across sessions (configurable, default: on, Issue #11)

7.17 Version 0.2.0 (beta)

- Code refactoring and cleanup
- Some GTK warnings were addressed

Warning: *Compatibility break*

This release is not compatible with previous ones, both *Tasks* and *Conditions* must be redefined from scratch. Hopefully this will be the one and only compatibility break. To clean up tasks and conditions, run the following commands in a terminal window (on Ubuntu):

```
$ rm ~/.config/when-command/*.list
$ rm ~/.config/when-command/*.task
$ rm ~/.config/when-command/*.cond
```

This preserves at least global configuration.

7.18 Version 0.1.1 (beta)

- All known issues closed
- Dialog boxes jump to top level
- Exit codes are forced to integers

7.19 Version 0.1.0 (beta)

- First usable public beta release
- Tasks
- Conditions (time and interval based, command based, idle time, and event)
- History
- Pause/Resume
- Global settings
- Auto configuration at first use

CHAPTER 8

License (BSD)

Copyright (c) 2015-2016, Francesco Garosi

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of when-command nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.