

A computer setup featuring a large monitor displaying a blue screen, a black keyboard, and a black mouse. A small external device is visible behind the monitor. The entire setup is positioned behind a white horizontal band that contains the title.

Developer's Guide

0.5.21

Mobius Forensic Toolkit

©2008,2009,2010,2011,2012,2013,2014 Eduardo Aguiar

Contents

1	Introduction	1
2	Developing extensions	3
2.1	Opening an extension	3
2.2	Creating a new extension	3
3	Mobius' Software Architecture	7
3.1	advertise/call	7
3.2	connect/emit	8
4	Datasources	11
4.1	services available	12
5	Widgets	13
5.1	container	13

1

Introduction

Nowadays, open source forensic tools are domain specific. Each tool tries to solve a problem of the investigation scope, and some do it very well. Unfortunately, they lack integration and their development is made harder because of the absence of common code, and therefore of code reuse. Their outputs are not standardized, and most of them uses command line interface.

The Mobius Forensic Toolkit is a framework to develop forensic tools. It is written in Python, using PYGTK and PyCairo. It is very extensible through specialized programs called **extensions**, and these programs share services, program environment and have access to a unified case model.

This guide is focused on developing extensions for the Mobius Forensic Toolkit framework. Sample codes are presented when suitable. It is a work in progress and does not intend to be a complete reference guide.

2

Developing extensions

The Mobius Forensic Toolkit is implemented using extensions. Each extension is a separated program that runs on its own independent namespace. The Extension Builder is an extension that was specifically made to edit extensions. It is a complete IDE that handles the underlying extensions and services structure, with code editing capabilities.

To start Extension Builder, click on **tools**→**Extension Builder** menu option. A window like the one shown in figure 2.1 will be opened.

2.1 Opening an extension

After you have started Extension Builder, click on **Open** menu option or on the corresponding icon in the toolbar, to open an extension.

Mobius Forensic Toolkit distribution files (**.tar.gz**, **.tar.bz2**, or **.zip**) have a directory named **extensions** where you can find all extensions that are distributed inside those packages. Feel free to open those extensions, and even to create new ones based upon their source codes. In this example, we have selected all extensions from **extensions** directory (figure 2.2).

To use an extension you have modified, you must install it using Mobius main window **tools** option.

2.2 Creating a new extension

As told before, you can open an existing extension, modify its source codes and save it as a new extension. But you can also start with a fresh new one. Click on **New** menu option or on the corresponding icon at toolbar, to create an extension.

Change your extension properties using **properties** option, and it will open up a dialog (figure 2.3).

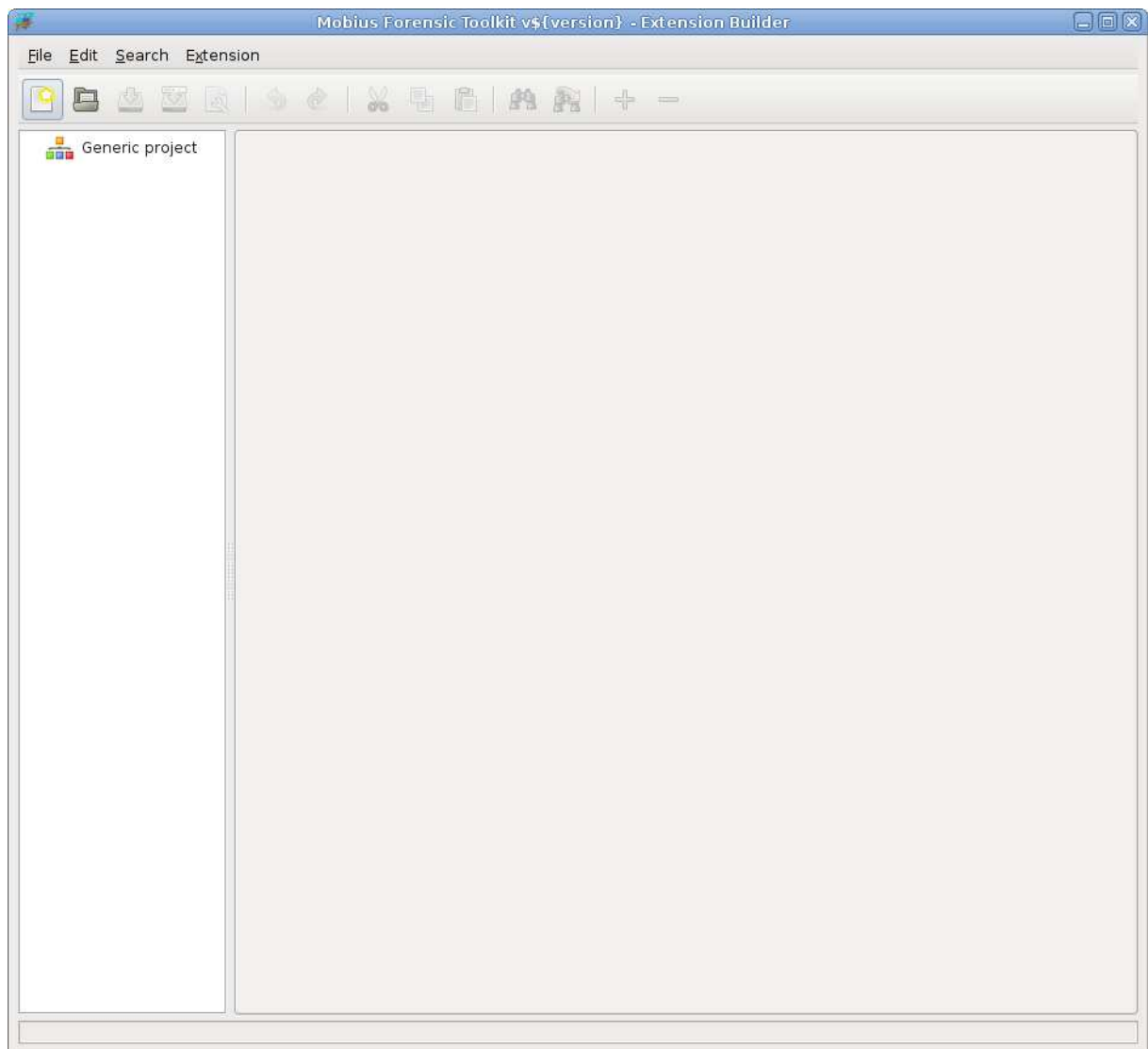


Figure 2.1: Extension Builder running

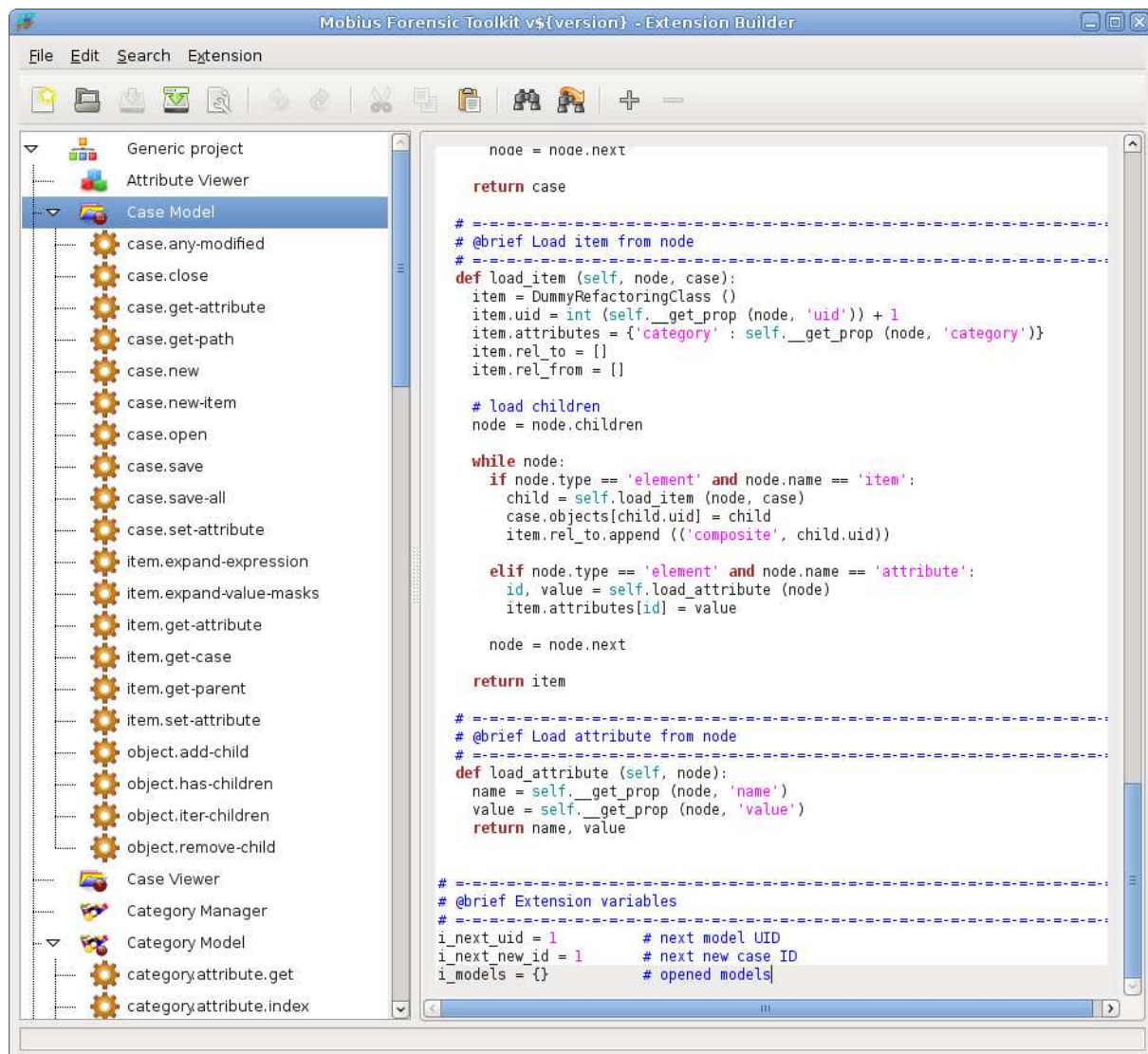


Figure 2.2: Extension Builder showing extensions

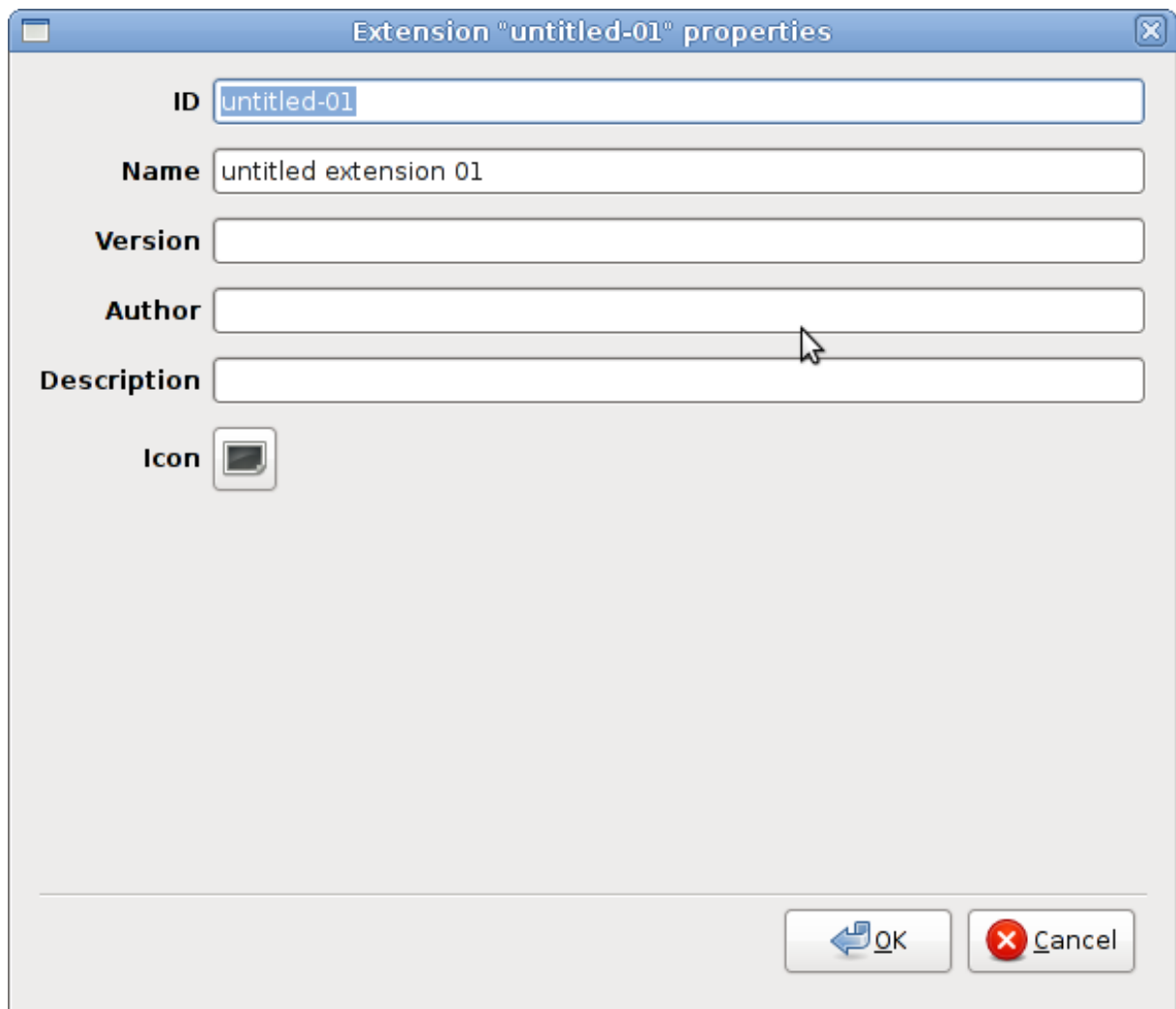


Figure 2.3: Extension Builder properties dialog

3

Mobius' Software Architecture

The Mobius framework is solely based on extensions, independent programs that run on their own sand-box, but that can interact through the global object `gdata.mediator`. This object is the only one that is shared among extension instances and is used as a mediator in two different ways: a) into the bulletin board pattern, where one extension **advertises** a service that might be called by any extension, including the advertiser extension; b) as a event broadcaster, where one extension **emits** an event and the extensions that are **connected** to this event receive a signal. The following sections show both kind of communication among extensions.

3.1 advertise/call

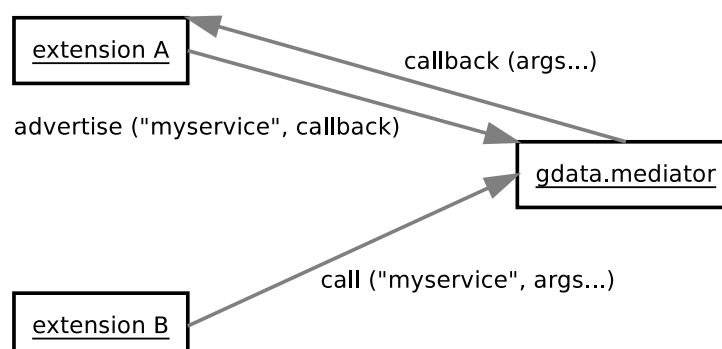


Figure 3.1: **advertise/call**: a) extension A advertises service “myservice”; b) extension B calls service “myservice”; c) `gdata.mediator` calls service callback function, passing back the function return value to extension B.

```
def callback_function (s):
    return "hello_" + s

gdata.mediator.advertise ("myservice", callback_function)
```

Figure 3.2: **advertise/call** extension A code

```
value = gdata.mediator.call ("myservice", "user")
print value      # shows "hello user"
```

Figure 3.3: **advertise/call** extension B code

3.2 connect/emit

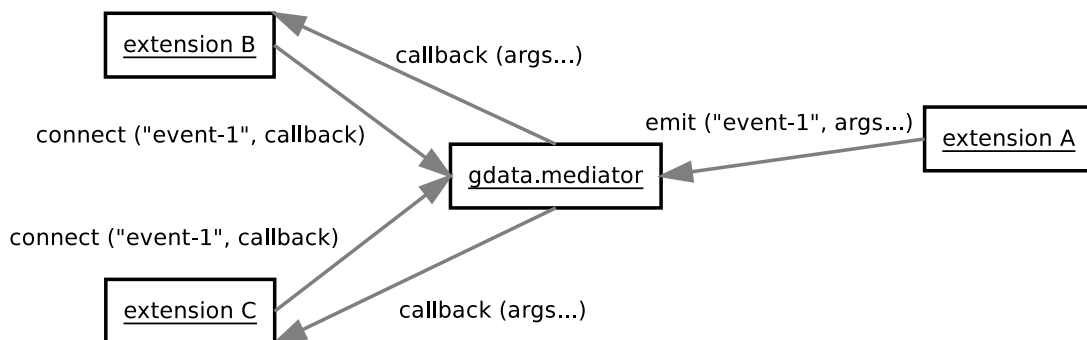


Figure 3.4: **connect/emit**: a) extensions B and C **connect** to event “event-1”, passing callback functions to be called; b) extension A **emits** an event “event-1” along with args; c) extensions B and C callbacks are called by the `gdata.mediator` object.

```
def callback (case):
    print "extension_B->event-1_on_case_" + case.name

gdata.mediator.connect ("event-1", callback)
```

Figure 3.5: **connect/emit**: extension B code

```
def callback (case):
    print "extension_C->event-1_on_case_" + case.name

gdata.mediator.connect ("event-1", callback)
```

Figure 3.6: **connect/emit**: extension C code

The code shown in figure 3.8 is the generated code of extension `date-code`. It connects to the event `object.attribute-modified` which is triggered every time an object’s attribute is modified.

```
...
if some_condition:
    gdata.mediator.emit ('event-1', case)
```

Figure 3.7: **connect/emit**: extension A code

```
def callback (obj, attr_id, old_value, value):
    if attr_id == 'manufacturing_date' and 4 <= len (value) <= 5:
        Y = int (value[0:2])
        W = int (value[2:-1])
        D = int (value[-1:])

        # fiscal year begins at first saturday of July
        d = datetime.date (Y + 1999, 7, 1)
        if d.weekday () < 6:
            days_to_saturday = 5 - d.weekday ()
        else:
            days_to_saturday = 6
        d += datetime.timedelta (days=days_to_saturday)

        # Add date code's weeks and days
        d += datetime.timedelta (weeks=W - 1, days=D - 1)

        obj.manufacturing_date = d.isoformat ()

gdata.mediator.connect ('object.attribute-modified', callback)
```

Figure 3.8: **connect/emit**: date-code extension code

4

Datasources

Datasources are objects that handle access to data. Each case item has an attribute `datasource` that can be assigned by the user and contains information on how to retrieve the data. The figure 4.1 illustrates an example on how to use the datasources:

```
datasource = item.datasource

# check if datasource is available
is_available = gdata.mediator.call ('datasource.is-available', datasource)
print 'datasource_is_available:', is_available

# retrieve metadata
metadata = gdata.mediator.call ('datasource.retrieve-metadata', datasource)

for attr_id, attr_name, attr_value in metadata:
    print attr_id, attr_name, attr_value

# read some bytes...
reader = gdata.mediator.call ('datasource.get-reader', datasource)
if reader:
    reader.open ()
    data = reader.read (512)
    reader.close ()

# get datasource path, when available
path = gdata.mediator.call ('datasource.get-path', datasource)
```

Figure 4.1: using datasources services

4.1 services available

- `datasource.get-metadata` returns a list of tuples containing the attribute ID, attribute name and attribute value of metadata.

```
metadata = gdata.mediator.call ('datasource.retrieve-metadata', datasource)

for attr_id, attr_name, attr_value in metadata:
    print attr_id, attr_name, attr_value
```

- `datasource.get-path` returns the datasource's path, when available. The idea behind this service is to allow third party tools to have access to the datasources. Note that extensions should not use this feature, because not every type of datasource has a local path (e.g. remote datasources).

```
path = gdata.mediator.call ('datasource.get-path', datasource)
print 'local_path:', path
```

- `datasource.get-reader` returns a reader object, when available, to read data from datasource.

```
reader = gdata.mediator.call ('datasource.get-reader', datasource)
if reader:
    reader.open ()
    data = reader.read (512)
    reader.close ()
```

- `datasource.is-available` returns True/False whether the datasource is available for reading, e.g. whether the physical device is attached and ready.

```
is_available = gdata.mediator.call ('datasource.is-available', datasource)
print 'datasource_is_available:', is_available
```


5

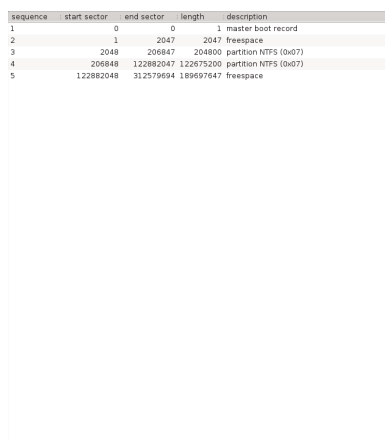
Widgets

Widgets are visual components that can be used in extensions. They are encoded to be independent of specific UI libraries. To construct a widget, call the `ui.new-widget` service, passing a widget class name, as follow:

```
widget = gdata.mediator.call ('ui.new-widget', 'tableview')
```

5.1 container

The container is a widget that contains another widget. It also has a warning label that can be used to show warning messages, calling the `set_warning_label` method (figure 5.1).



sequence	start sector	end sector	length	description
1	0	0	1	master boot record
2	1	2047	2047	freespace
3	2048	206847	204800	partition NTFS (0x07)
4	206848	122882047	122875200	partition NTFS (0x07)
5	122882048	312579694	189697647	freespace



Figure 5.1: the same container showing its content (left) and showing a warning label (right).

container
<pre>+get_ui_widget(): ui_widget const +show() +hide() +set_sensitive(sensitive:bool) +set_warning_label(text:string) +set_content(widget:any) +get_content(): any const +remove_content() +show_content()</pre>

Figure 5.2: the **container** widget's interface

The methods of the container widget are:

- `show ()` — show widget
- `hide ()` — hide widget
- `set_sensitive (flag)` — set whether widget is sensitive to user actions.
- `set_warning_label (text)` — set warning label
- `set_content (widget)` — set widget's content
- `get_content ()` — get widget's content
- `remove_content ()` — remove widget's content
- `show_content ()` — show widget's content instead of the warning label