Developer's Guide

1.5

Mobius Forensic Toolkit

Contents

1 Introduction

Every available open source forensic tools tries to solve a very specific problem under the investigation scope, and some of them are very successful in doing that. Unfortunately, most of them lacks integration and their development are made harder because of the absence of common code, and therefore of code reuse. Their outputs are not standardized, and most of them presents command line interface.

The Mobius Forensic Toolkit is a framework to develop forensic tools. It is written in C++ and 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 using the Mobius Forensic Toolkit API and on developing extensions for the Mobius Forensic Toolkit framework. Sample source code is presented when suitable. It is a work in progress and it is not intended to be a complete reference guide.

2

Mobius' Software Architecture

The Mobius Forensic Toolkit architecture is divided in two main parts:

1. The **Mobius Forensic Toolkit API**, which comprises the Python API, the C++ API and the Python wrapper to the C++ API. This API can be used both by stand-alone programs and by the Mobius Forensic Toolkit extensions;

2. The **Mobius Forensic Toolkit extensions**, independent programs written in Python that run on their own code sandbox, and interact only through the global mediator object `gdata.mediator`. The Mobius Forensic Toolkit application is a collection of both forensic and auxiliary extensions.

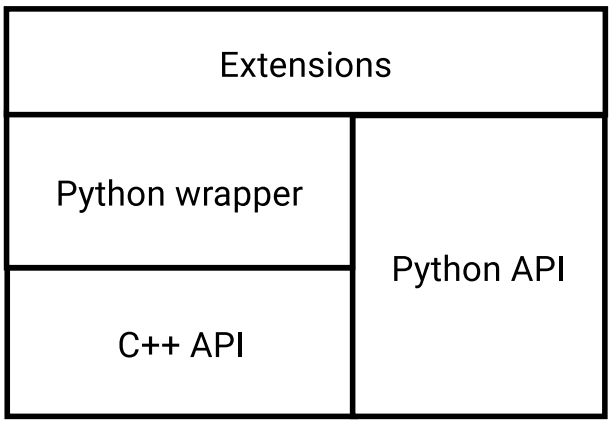

Figure 2.1: Extensions are written in Python, using both the Python API and the C++ API through the Python wrapper layer.

2.1 Extensions' communication

Any extension written to the Mobius Forensic Toolkit has a `gdata.mediator` object. This object is an instance of the global mediator and is used in two different ways:

1. Into the bulletin board pattern, where one extension **advertises** a service that might be **called** by any extension, including the advertiser extension itself;
2. 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.

2.1.1 advertise/call

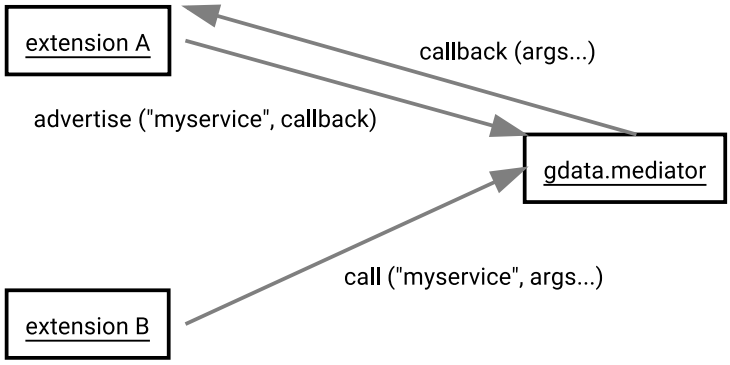

Figure 2.2: **advertise/call**: a) extension A advertises service "myservice"; b) extension B calls service "myservice'; c) `gdata.mediator` calls service callback function, passing back the function return value to extension B.

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

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

Figure 2.3: **advertise/call** extension A code

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

Figure 2.4: **advertise/call** extension B code

2.1.2 connect/emit

The code shown in figure 2.9 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.

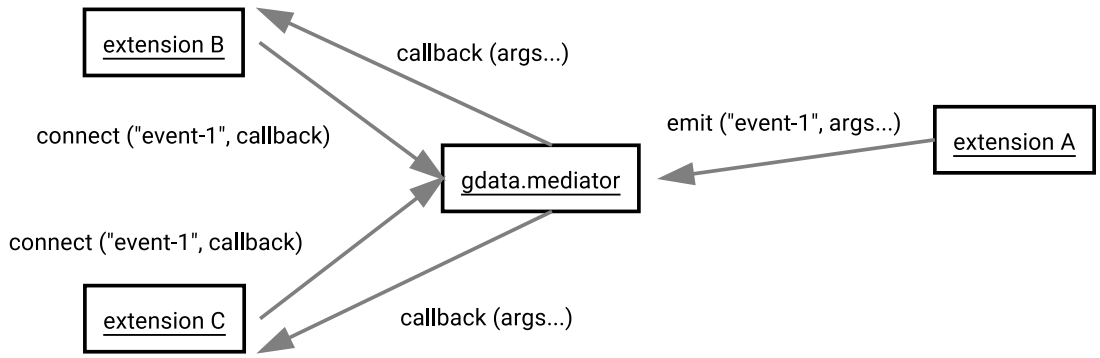

Figure 2.5: **connect/emit**: a) extensions B and C **connect** to event "event-1", passing callback functions to be called; b) extension A **emits** an event "event-1" along with args; c) extensions B and C callbacks are called by the `gdata.mediator` object.

```
def callback (case):
  print "extension B->event-1 on case '%s'" % case.name

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

Figure 2.6: **connect/emit**: extension B code

```
def callback (case):
  print "extension C->event-1 on case '%s'" % case.name

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

Figure 2.7: **connect/emit**: extension C code

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

Figure 2.8: **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 2.9: **connect/emit**: `date-code` extension code

3

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 3.1 will be opened.

3.1 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 3.2).

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

3.2 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 3.3).

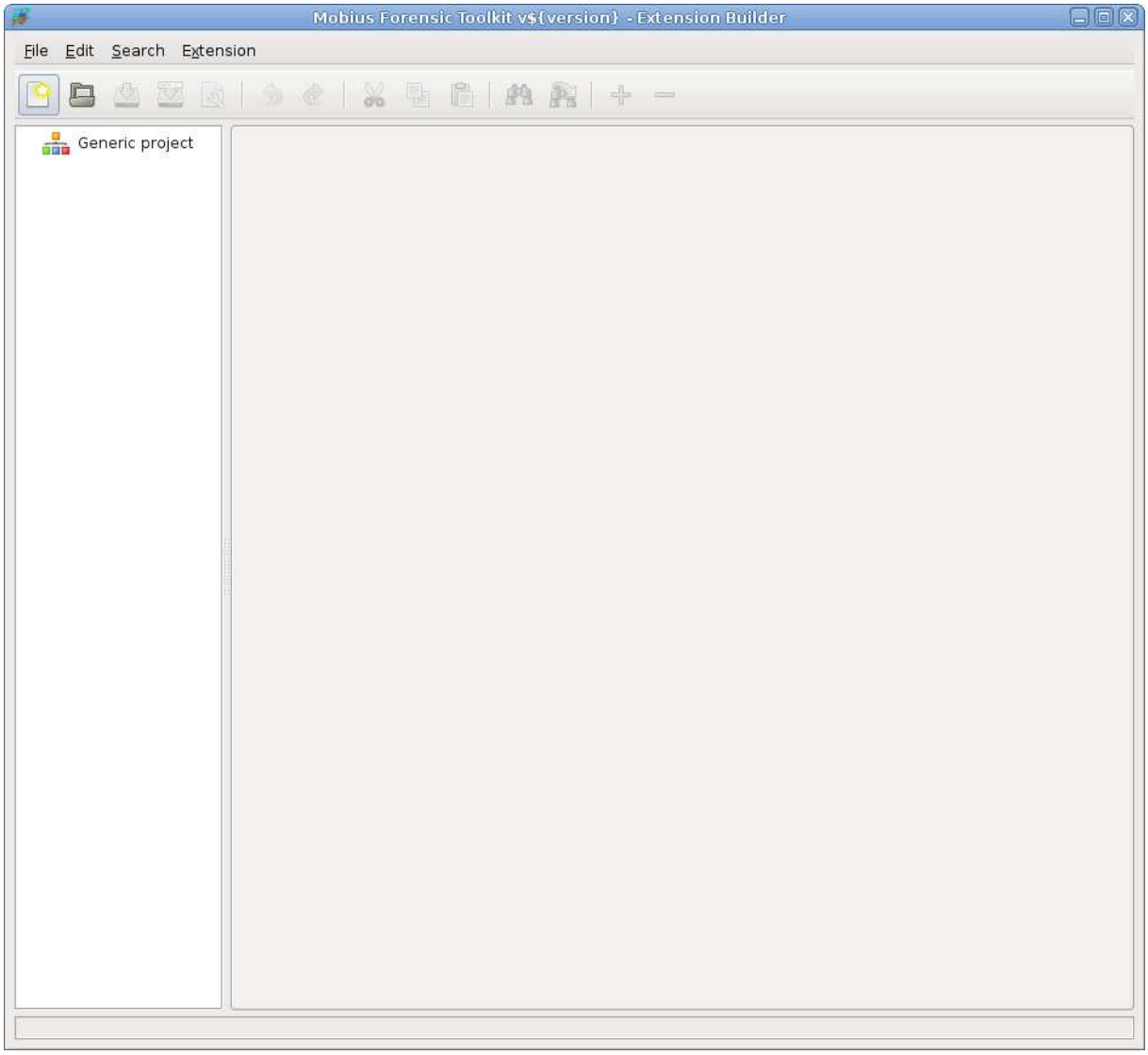

Figure 3.1: Extension Builder running

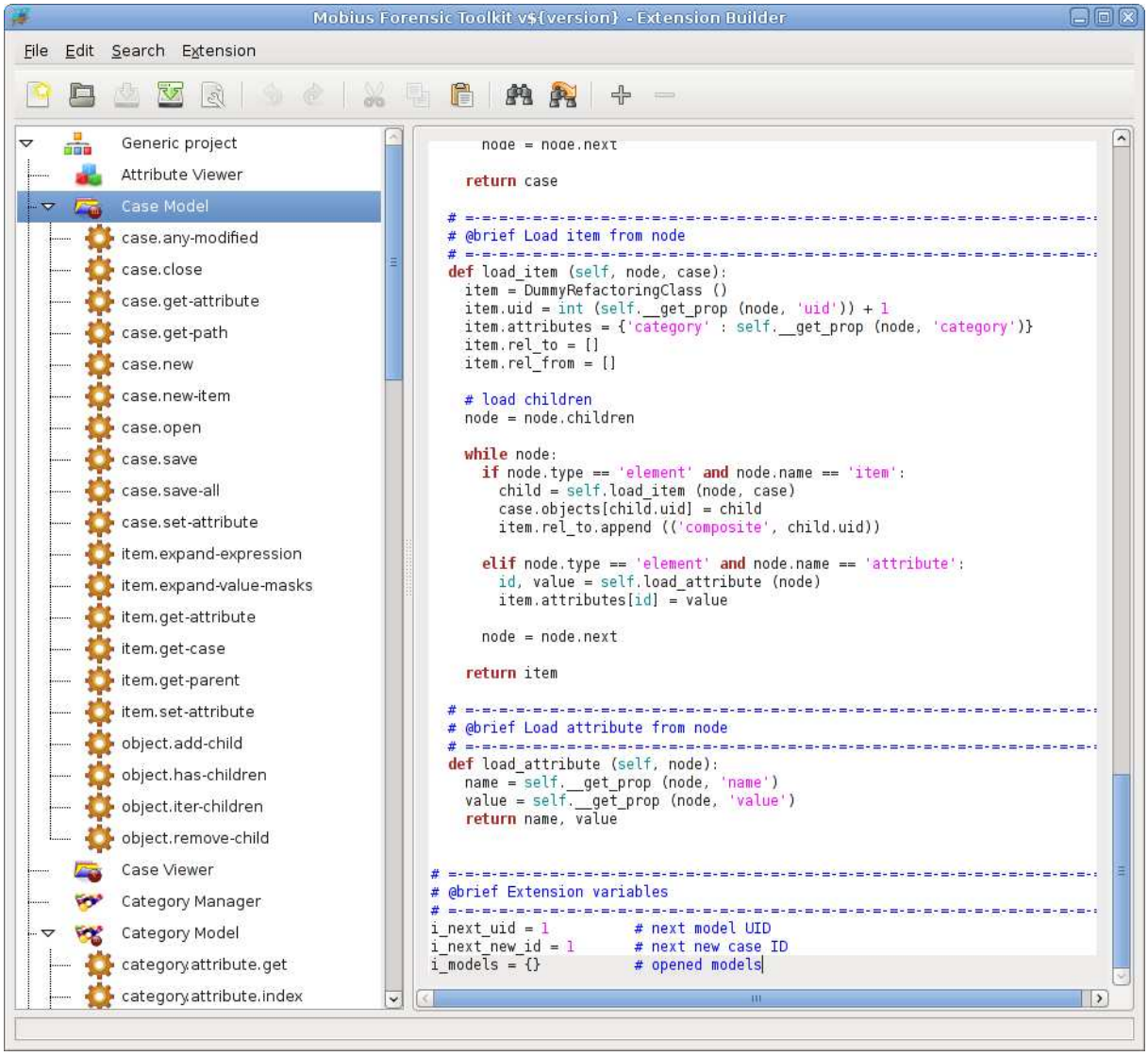

Figure 3.2: Extension Builder showing extensions

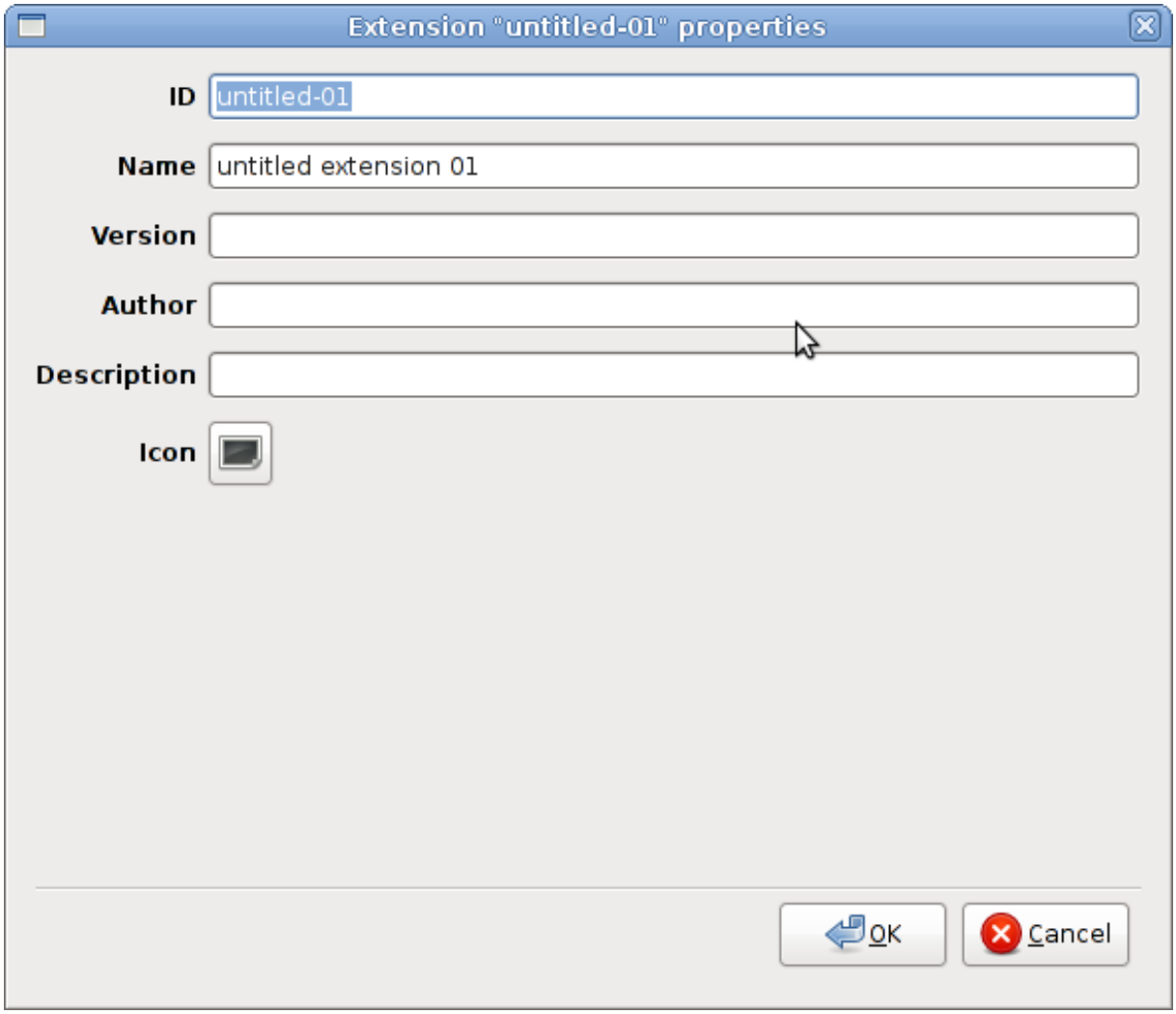

Figure 3.3: Extension Builder properties dialog

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).

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

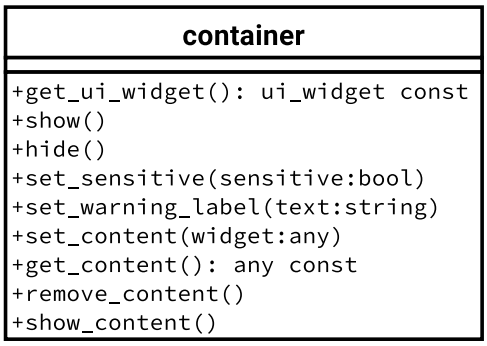

Figure 5.2: the **container** widget's interface

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