Developer's Guide

0.5.17

Mobius Forensic Toolkit

Contents

1 Introduction

Nowadays, open source forensic tools are domain specific. Each tool tries to grab a little 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 use command line interface.

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 through extensions. Each extension is a separated program that runs in 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).

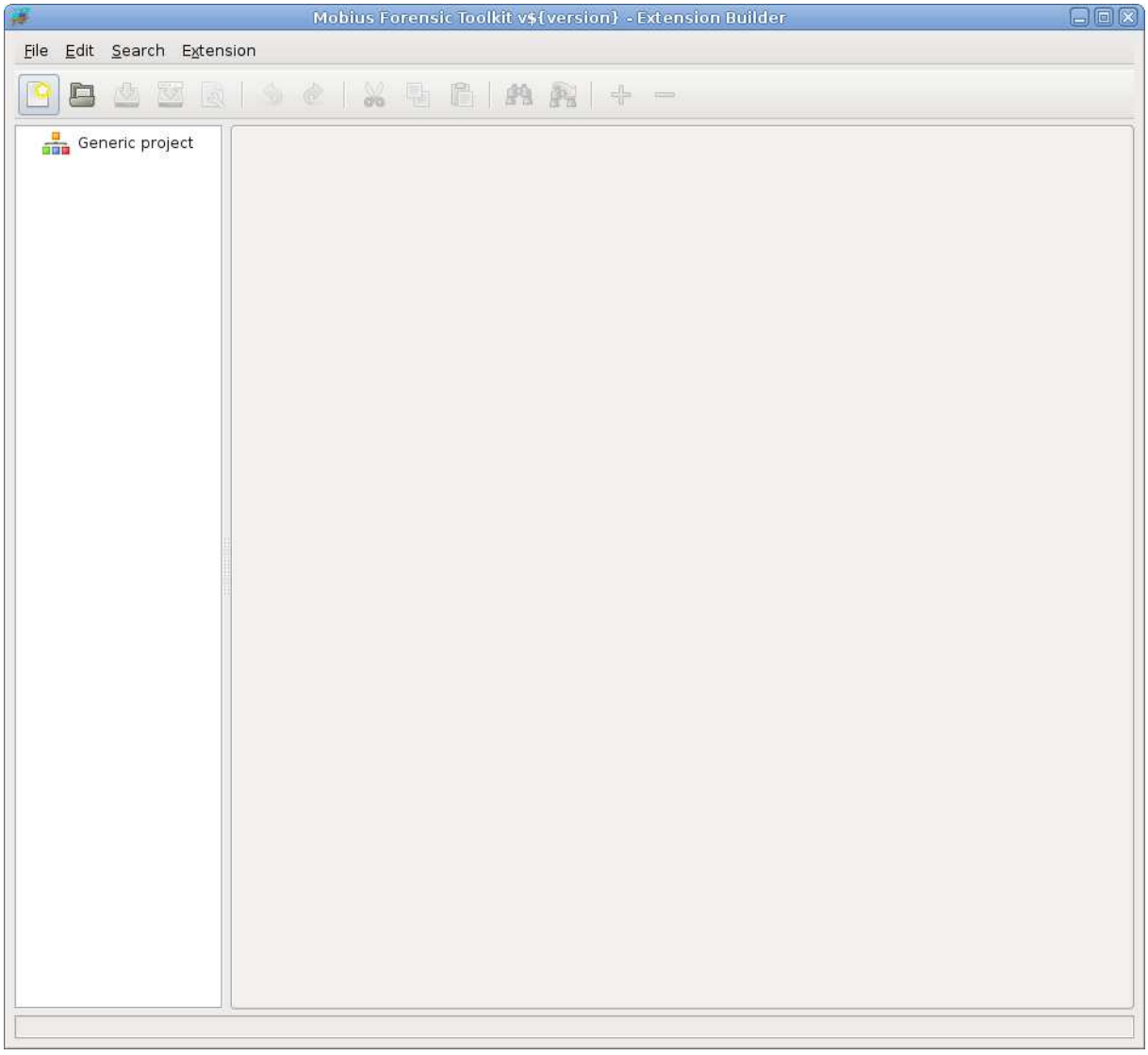

Figure 2.1: Extension Builder running

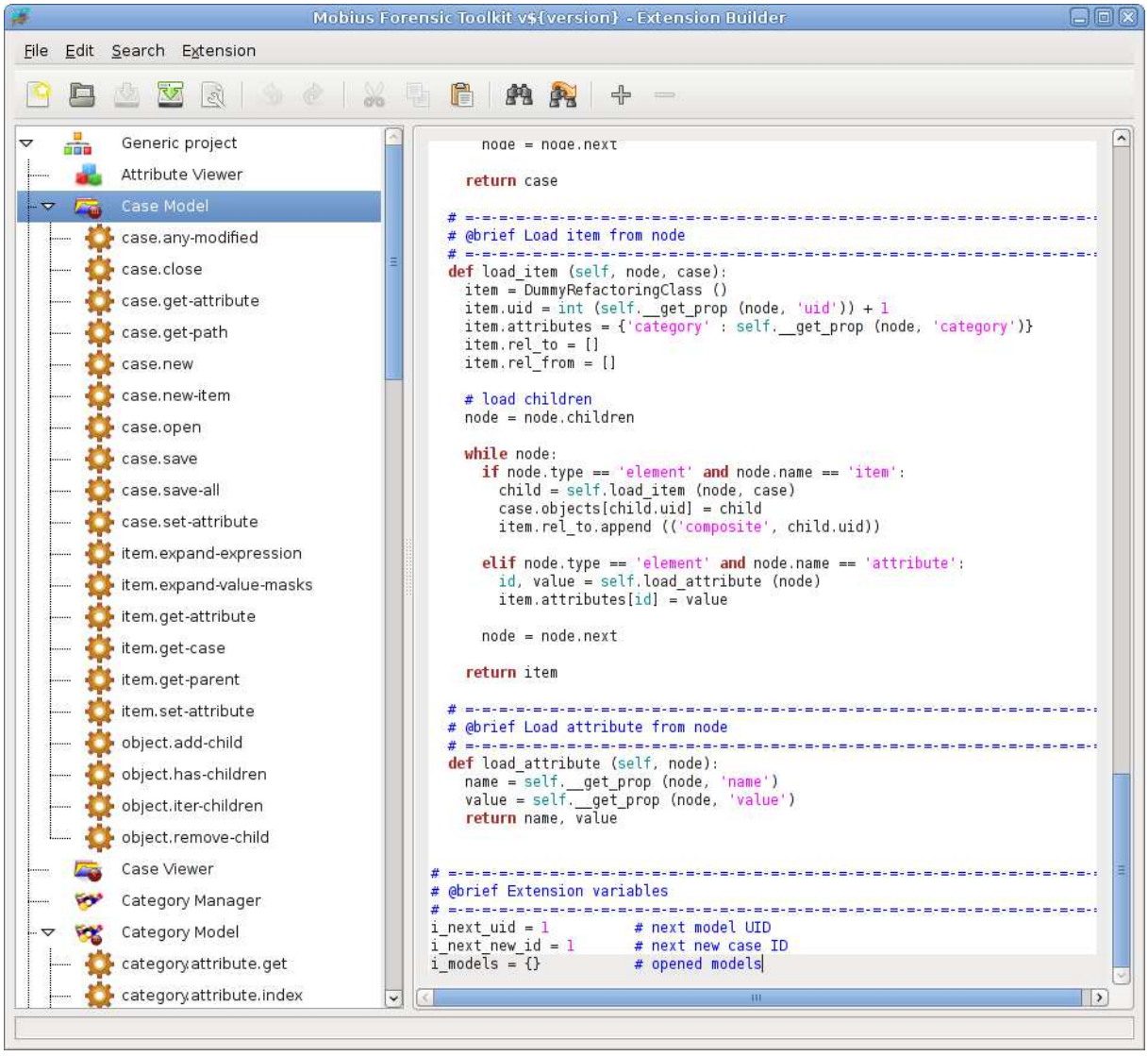

Figure 2.2: Extension Builder showing extensions

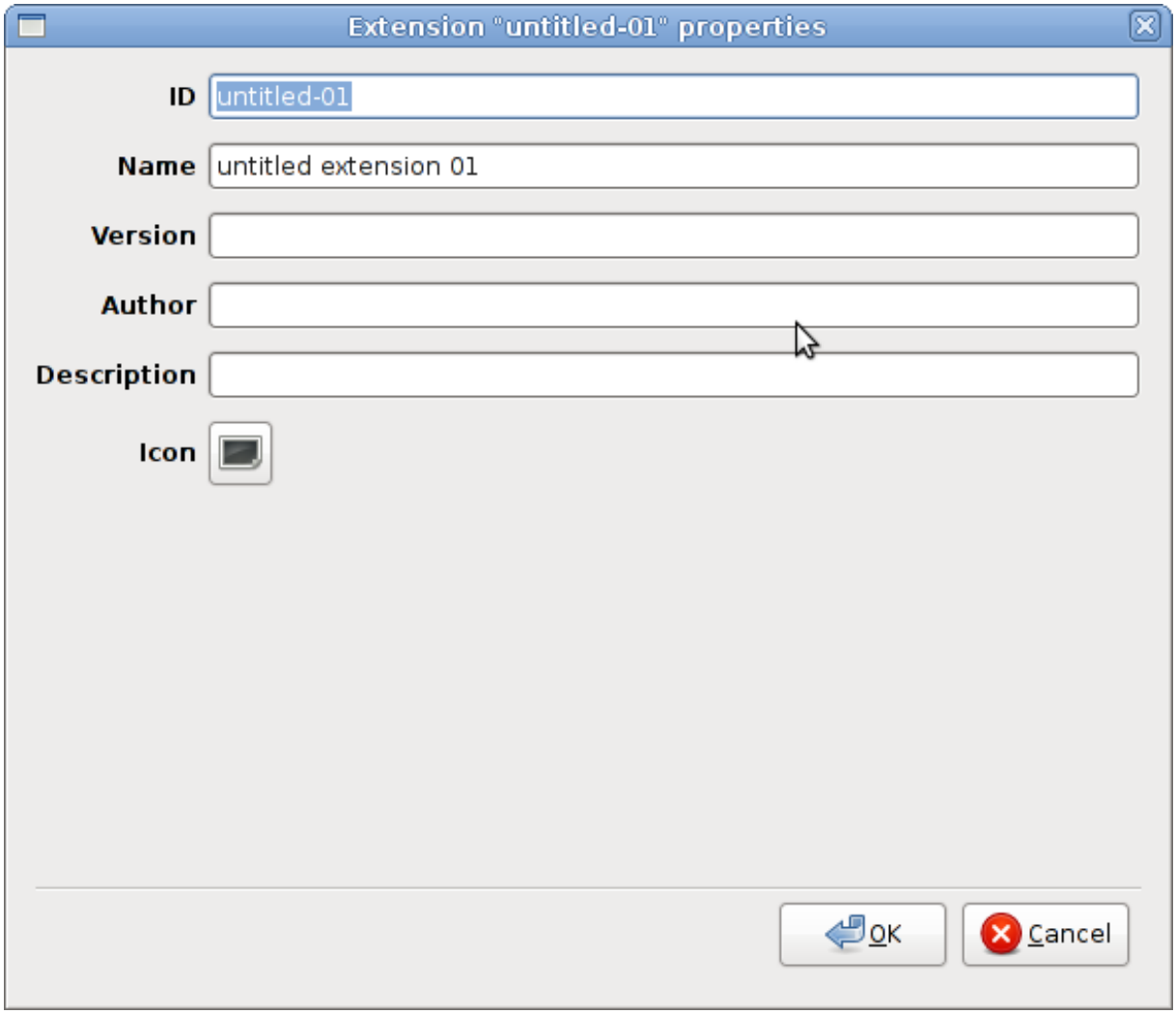

Figure 2.3: Extension Builder properties dialog

3

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 3 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 3.1: using datasources services

3.1 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
```