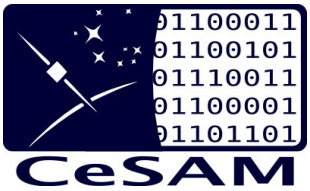

ETC-42 : Universal Exposure Time Calculator

Plugin Framework Manual

Laboratoire d'Astrophysique de Marseille
Centre donneeS Astrophysique de Marseille

Authors:
N. Apostolakos,
P.Y. Chabaud,
C. Surace

January 2012

Contents

1 Introduction

The ETC-42 exposure time calculator has been designed to be as generic and flexible as possible, so it can be used for as many instruments as possible. As a generic tool though, it can only accept generic input parameters and it produces as output only some common ETC results. To support instruments which require some specific treatment of the input and to allow the ETC-42 users to produce extra outputs (or even outputs depending on multiple simulation executions), the **plugin framework** is provided. This document describes this framework in detail and it serves as a reference for the plugin developers.

2 Required skills

The plugin framework is designed to make the extension of the ETC-42 as easy as possible as far as it concerns the plugin developers, without requiring any modification of the ETC-42 core. There are though some basic skills a developer should have, to be able to implement successfully ETC-42 plugins.

2.1 Java programming language

ETC-42 itself is written in Java, so the plugins also must be written in the same language. This means that the knowledge of the Java language is a requirement for being able to write a ETC-42 plugin. A good place for learning the Java language is the online tutorial provided by Oracle, which can be found on the following page:
`http://download.oracle.com/javase/tutorial/`

If a developer wants to implement the logic of the plugin in a different language, this is still possible, as long as the part of the plugin which communicates with the ETC-42 core is implemented in Java. For the communication between this part of the plugin and the code implemented with the different language there are two options:

- To build a native library and use the JNI (Java Native Interface) for directly calling its methods from Java
  `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/jni.html`
- To build an executable program and call it from Java using the Runtime.getRuntime().exec() method, using files for the input/output exchange

It is though strongly recommended to build the plugins entirely in Java.

2.2 Swing framework

In most of the cases the ETC-42 plugins will provide some kind of GUI for communicating with the user. The ETC-42 GUI is build using the Swing framework and all plugins are required to do the same. In fact, all the plugins are forced to extend the `JFrame` class. A good tutorial for Swing can be found on this link:

`http://download.oracle.com/javase/tutorial/ui/index.html`

To ease the plugin development, it is highly recommended to use a Swing GUI builder for all the non trivial plugin GUIs. All the popular IDEs provide such builders.

For development of plugins without any GUI window please see the section 5.4.

3 Framework Design

The ETC-42 plugin framework has been designed to separate as much as possible the implementation of the plugins from the core of the ETC-42. The figure 1 shows this design.

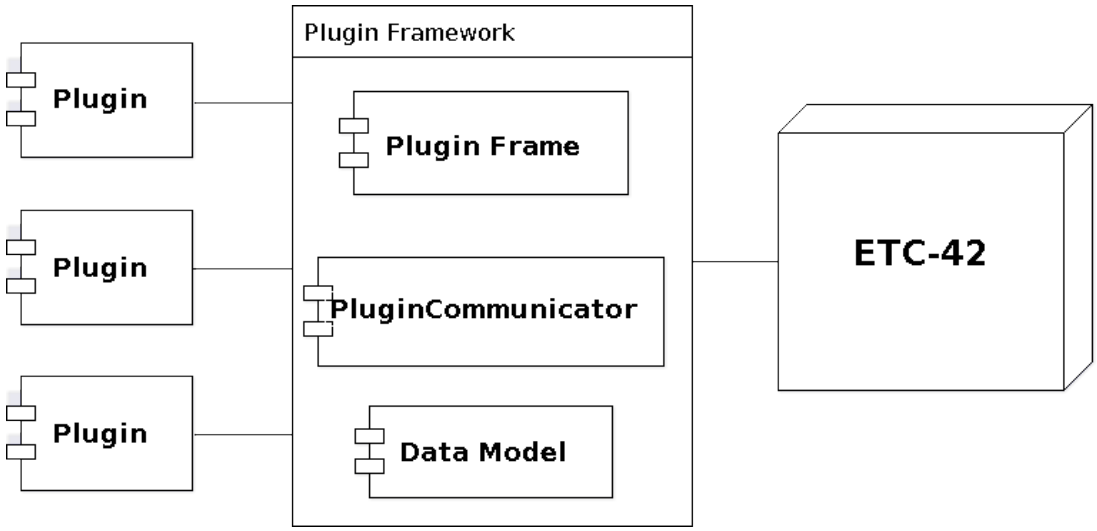

Figure 1: Plugin Framework Design

As it is visible on the figure, the plugin framework has three main parts. The **PluginFrame** (described in section 5) is the class which all the plugins must extend. The **PluginCommunicator** (described in section 6) is the way of the plugin to communicate with the core of the ETC-42 and to retrieve or set some configuration, to execute the simulation or to show some results. Finally, the **DataModel** (described in section 7) is the representation of the ETC-42 configuration.

4 Development Prerequisites

To develop ETC-42 plugins the only prerequisite is the jar file of the ETC-42. This file can be downloaded from the following link:
`http://projets.oamp.fr/projects/etc/wiki/Downloads`

This jar contains all the required classes. The plugin developers should set it as a dependency to their ETC-42 plugin projects. Note that during execution, ETC-42 already has access to these classes, so when the developers pack the plugins in a jar file, they should **not** include the contents of this jar (or the jar itself), to create smaller jar files.

5 PluginFrame

The **PluginFrame** class is the class which all the plugins must extend. Itself extends the **JFrame** class, so all the plugins have at least one GUI window (see section 5.4 for how to implement non GUI plugins). This will be the window the ETC-42 will show when the plugin will be lunched.

5.1 Initialization

The implementations of the **PluginFrame** class must have a **public default constructor** (a constructor without arguments). This will be the constructor the ETC-42 will use to create the plugin instance. The use of any other constructors by the plugin framework is not possible, but their existence is not forbidden (as they might be useful for testing purposes or for standalone execution of the plugin).

Note that during the execution of the constructor the plugin does **not** have access to the communicator yet. During this time only initialization which is not depended from communication with the ETC-42 core is possible. The initialization depended on the core of the ETC-42 should happen in the method **initialize (PluginCommunicator communicator)**, which must be implemented by all the plugins. This method will be called after the execution of the constructor and before the main plugin frame is shown.

5.2 Menu Determination

Each **PluginFrame** must be annotated with the **@Plugin** annotation. This annotation has two parameters, the **menuPath** and the **tooltip**. The **menuPath** is a mandatory parameter and it defines the menu path from where the plugin will be executed. This parameter accepts a list of strings representing names of sub-menus. The last string on the list is the name of the plugin. Note that the ETC-42 has a menu called **Plugins** and the path defined with the annotation is always under this menu. It is not possible to place plugins under a different menu. The **tooltip** parameter is optional and it defines a string which will be presented to the user when he will hover the mouse above the menu item of the plugin.

For example, to implement a plugin which will be available under the menu "**Plugins -> My Plugins -> Run the plugin...**", the following class should be created:

```
@Plugin(menuPath={"My Plugins", "Run the plugin..."}, tooltip="My first plugin")
public class MyPlugin extends PluginFrame {
    public MyPlugin() {
        // Initialization of the frame goes here
    }
    @Override
    protected void initialize(PluginCommunicator communicator) {
        // ETC-42 core related initialization goes here
    }
    ...
```

This class will have as result the menu on the figure 2:

Figure 2: Plugin Menu Example

The plugin developers should always chose reasonable plugin names to avoid name conflicts as much as possible. The ETC-42 will handle name conflicts by using an increasing counter as a postfix, but this makes the plugin selection unclear (because the correct plugin has to be determined by reading its tooltip). Of course same names for sub-menus can be used to group plugins together.

5.3 Logic Implementation

Except of the restrictions mentioned above, the ETC-42 plugin developers are free to implement the plugin logic as they like. For the communication between the plugin frame and the core of the ETC-42 they can use the **PluginCommunicator** class (described on section 6), which can be retrieved with the method **getCommunicator()** of the **PluginFrame** class.

The most common way implementing a plugin is to include in the GUI a button, which, when pressed, executes the logic of the plugin, and several input fields (like text fields, etc) to receive the users input.

5.4 Non GUI plugins

As mentioned on the sections above, all the ETC-42 plugins are forced to have at least one GUI window. There is though the need of having plugins which do not need user input, so the step of showing a window and pressing a button is redundant (and even maybe annoying). In this section is described how to implement this kind of plugins. Note though that interactive user input will not be available for such plugins (command line plugins are not supported).

The non GUI plugins still need to extend the **PluginFrame** class, like all the ETC-42 plugins, so they will actually have a GUI window, but this will never be visible. The ETC-42 plugin framework shows the plugin windows by using the **setVisible(boolean b)** method. A plugin which does not need to have a visible window should override this method to do nothing. For example:

```
    @Override
2   public void setVisible(boolean b) {
        // Do nothing here so the window will not be visible
4   }
```

Plugins which override this method will not be able to receive interactive user input, so all the logic of the plugin needs to be implemented in the **initialize (communicator)** method.

6 PluginCommunicator

The **PluginCommunicator** class is the way of communication between the plugins and the core of the ETC-42. The plugins can retrieve an instance of this class to use, by using the method **getCommunicator()** of the **PluginFrame** class. Here are described all the methods the communicator provides and how to use them to implement the plugin logic.

6.1 getCurrentSession()

The method **getCurrentSession** should be used to retrieve the current configuration of the ETC-42. The current is the one that the user has currently selected on the ETC-42 GUI and is the one which will be used if a simulation will be lunched.

The method returns an instance of the **Session** class. This object can be used to retrieve the necessary information for populating any GUI elements of the plugin or for performing the plugin calculations. For a detailed description of how to use the **Session** class and the Data Model in general please see section 7.

Note that the returned object is **mutable** (its state can be modified). The modifications are possible via the different setter methods of the **Instrument**, **Site**, **Source** and **ObsParam** objects. Any **modifications** done by the plugin **will be reflected** to the core ETC-42. This is the way of the plugin to modify the ETC-42 configuration.

As a simple example, imagine a plugin operating for spectrographs and it uses as source an emission line. The plugin might want to set the wavelength range for which the simulation will run according the wavelength and the FWHM of the emission line, to limit the amount of calculations. This can be done with the following code:

```
// Retrieve the session from the communicator
Session session = getCommunicator().getCurrentSession();
// Read the emission lines wavelength and FWHM
Source source = session.getSource();
double wavelength = source.getEmissionLineWavelength();
double fwhm = source.getEmissionLineFwhm();
// Set the wavelength range of the calculation to 2*FWHM
Instrument instrument = session.getInstrument();
instrument.setRangeMin(wavelength - fwhm);
instrument.setRangeMax(wavelength + fwhm);
```

6.2 sessionModified()

Even though when a plugin changes a parameter in the session the data model in the ETC-42 core is also updated, the GUI windows are **not** notified automatically for the new values. This

design was chosen mainly for performance reasons, both for the main ETC-42 window and the plugins themselves.

The way to update the values shown on the main ETC-42 window is to call the method **sessionModified()** of the **PluginController**. The plugins should call this method before disposing their windows if they make modifications on the **Session** (otherwise the ETC-42 main GUI will show old, invalid values), but they can also use it at any moment during their execution. This, with combination that the plugin frames are not modal (they are not locking the ETC-42 main frame), allows the development of plugins which can be used in parallel with the main ETC-42 window.

6.3 getAvailableDatasetList()

As it will described in the Data Model section (7), there are two different kinds of parameters in the session, single values (like doubles, integers, etc) and datasets (representing a set of data). Setting the single values is straight forward, as the related setter methods accept as parameters Java primitive types.

Setting the datasets though is not as easy. The setters for the dataset parameters accept as parameter a **DatasetInfo** object, which must describe an already existing dataset (see section 6.5 for how to create new datasets). If a plugin wants to let the user change a dataset parameter, it must present to him the possible values to select from. The **PluginCommunicator** provides the method **getAvailableDatasetList()** for exactly this reason.

This method gets two arguments, the dataset **type** and the **namespace** for which the available datasets will be returned. The **namespace** must be the name of the component (instrument, site, etc) for which the dataset list will be retrieved and the **type** is an enumeration defining the specific parameter of the component (for details see section 7). The returned value of the method is a list of **DatsetInfo**, representing all the available datasets for this parameter and for the specified component.

For example, if a plugin wants to retrieve all the available datasets for the total transmission of the instrument, the following code can be used:

```
// Retrieve the instrument from the communicator
Instrument instrument = getCommunicator().getCurrentSession().getInstrument();
// Get the name of the instrument
String namespace = instrument.getInfo().getName();
// Retrieve the list of available datasets
List<DatasetInfo> availableTransmissions =
        getCommunicator().getAvailableDatasetList(Type.TRANSMISSION, namespace);
```

6.4 getDataset()

For performance reasons the data model retrieved with the **getCurrentSession()** method does not contain any data of the dataset parameters. The getter methods of these parameters return objects of type **DatsetInfo**, which contain only the necessary information for retrieving the data when necessary.

The retrieval of the data can be done by using the **getDataset()** method. The plugins can use this method when they need the data for performing some calculation. This method accepts three parameters, the dataset **type**, the **DatsetInfo** describing it and the **option** in the case of a multi-dataset (it should be **null** for single datasets). The method returns an object of type **Datset**. The data can be retrieved from this object with the method **getData()**, which returns an ordered map containing all the data. The **Datset** object contain also other useful information, like the units in which the data are expressed.

As an example, to retrieve the data of the total transmission of the current instrument, the following code can be used:

```
// Retrieve the instrument from the communicator
Instrument instrument = getCommunicator().getCurrentSession().getInstrument();
// Get the dataset info for the transmission
DatasetInfo info = instrument.getTransmission();
// Retrieve the data
Dataset transm = getCommunicator().getDataset(Type.TRANSMISSION, info, null);
Map<Double, Double> data = transm.getData();
```

6.5 createDataset()

When a plugin wants to create a new dataset, the **createDataset()** method can be used. This method gets the following parameters:

- **type**: The type of the dataset to be created
- **info**: A **DatasetInfo** object containing the name and the description of the dataset to be created, as well as the namespace for which it will be available. If the namespace is null, then the dataset will be globally available.
- **xUnit**: The unit of the X axis
- **yUnit**: The unit of the Y axis
- **data**: A map containing the data of the dataset

Note that if there is already a dataset in the database with the same name-namespace pair, the method will fail throwing **NameExistsException**. After the method creates the new dataset, it also sets it as selected for the current session, so the plugins don't need to do this extra step. Extra care needs to be given when setting the axis units. If the units of the dataset are wrong, the ETC-42 will not perform the SNR calculation.

For example, the following code will create a new dataset for the instrument transmission, which will define a constant transmission of 0.8 for the wavelengths between 5000 and 10000 angstrom, and will be available only for the current instrument:

```
// Get the name of the instrument to use as namespace
Instrument instrument = getCommunicator().getCurrentSession().getInstrument();
String namespace = instrument.getInfo().getName();
// Create the dataset info and the units
DatasetInfo info = new DatasetInfo("0.8 transmission", namespace,
                                   "A constant transmission");
String xUnit = "\u00C5"; // Angstrom
String yUnit = null; // Transmission is unitless
// Generate the data. Note that TreeMap is used for having an ordered map
Map<Double,Double> data = new TreeMap<Double, Double>();
data.put(5000., 0.8);
data.put(7500., 0.8);
data.put(10000., 0.8);
// Create the dataset
getCommunicator().createDataset(Type.TRANSMISSION, info, xUnit, yUnit, data);
```

6.6 createMultiDataset()

The **createMultiDataset()** method is very similar with the **createDataset()**, but it is used for creating multi-datasets. A mutli-dataset contains multiple sets of data, called its options. Each option has a key, which is a string, and it is used from the ETC-42 core to select the correct data during the simulation.

The only difference of the **createMultiDataset()** method from the **createDataset()** is that the given data are is a map of type **Map<String,Map<Double,Double>>**, which means it is a map of pairs option-data.

6.7 runSimulation()

The ETC-42 plugins are not limited only on modifying the current session. They can also execute the ETC simulation as many times as they please, receiving the calculation results for each execution. This way a script can produce outputs which require multiple executions of the simulation (for example a plot of SNR values for different exposure times).

The method to execute the simulation is the **runSimulation()**. This method is **synchronous**, which means that it will **not** return until the simulation is finished. The returned object is of type **CalculationResults** and it contains all the available results of the simulation (final, intermediate and debug). Please see the section 8 for more details about the results.

Note that when a plugin executes a simulation the ETC-42 core window will **not** show any results automatically. This is done so plugins can run multiple simulations before producing their output.

6.8 showResults()

The last method of the **PluginCommunicator** allows the plugins to use the ETC-42 results panel for showing results to the user. This method accepts an object of type **CalculationResults**. The plugins can pass as argument either the object returned from the **runSimmulation()** method, or even create a new **CalculationResults** which will contain only their results. See the section 8 for more information about the **CalculationResults** class and how to use it.

7 Data Model

The data model of the ETC-42 consists of three main components. These are the **instrument**, which contains information about the telescope and the attached instrumentation, the **site**, which contains information about the location where the telescope is located, the **source**, which contains information about the astronomical target and the **observing parameters**, which contain the configuration of the specific observation (like exposure time, etc). The **Session** object returned by the **getCurrentSession()** method is a container for an instance of each of the components.

A detailed description of all the parameters of the four components exists in the document [1]. For this reason this information will not be duplicated here. Please refer to this document for more details.

8 Calculation Results

The plugins have access not only on the final results of the ETC-42 simulation, but also to all the intermediate calculated values. Because of that the number of the available results is big, so a dedicated framework has been designed for representing the results of calculations. The plugin developers need to know this framework both for using the results of a simulation and for presenting extra results to the user via the ETC-42 results panel.

8.1 Result Levels

All the results of the ETC-42 simulations are divided in four levels. The different parts of the ETC-42 can use the level information for filtering out results. The different levels are represented with the **CalculationResults.Level** enumeration and the possible values are:

- **FINAL**
  The results which are considered as the final output of the simulation
- **INTERMEDIATE IMPORTANqT**
  Results of intermediate calculations for which the users might be interested
- **INTERMEDIATE UNIMPORTANT**
  Results of intermediate calculations for which the users most probably will not be interested

- **DEBUG**
  The configuration for which the simulation run, available mainly for debugging reasons

A full list of the available results for each category can be found in the document [1]. The algorithms used for producing every result can be found in the document [2].

8.2 Result Types

The ETC-42 results framework provides four main types of results. All the result types inherit from the abstract class **CalculationResults.Result**, which just defines that all results should have a specific name (for the available result names please refer the document [1]). The different result types are the following.

8.2.1 String results

These are results which consist only from a string. The class which represents this type of results is the **CalculationResults.StringResult**.

8.2.2 Single value results

These are results which consist from a single value and the unit in which it is expressed. There are two types of single valued results, the **CalculationResults.LongValueResult** and the **CalculationResults.DoubleValueResult**. One of them accepts as values only long integer values and the other one accepts floating point of double precision values.

8.2.3 Dataset results

These are results which consist from a set of data. It contains the units of the two axis and a map containing the pairs of data. The available dataset results are the **CalculationResults.LongDatasetResult**, in which both axis are expressed as long integer values, the **CalculationResults.DoubleDatasetResult**, in which both axis are expressed as floating point of double precision values and the **CalculationResults.LongDoubleDatasetResult**, in which the X axis (the keys of the map) is expressed as long integer and the Y axis (the values of the map) is expressed as floating poing of double precision.

8.2.4 Image results

These are results which consist of a single, two dimensional image. The class representing this type of results is the **CalculationResults.ImageResult**. This class contains an object of type **Image**, which provides methods for retrieving the size of the image and the value of specific pixels.

8.2.5 Image Set result

These are results which keep a map with number keys and values **Image** instances. If, for example, a calculated image varies with over the wavelength, the resulted data cube is shown with this type of result.

8.3 Retrieving the results

After executing a simulation and obtaining an instance of **CalculationResults**, there are two ways to access the results in it. The first is the method **getResultByName(name)**, which gets as argument the name of the result to retrieve. Note that for this method it does not matter the level of the result. For a detailed description of the available result names please refer the document [1]. The second way is to use the method **getResults(level)**, which returns a map with all the results of a specific level. The keys of the map are the result names.

Note that both methods return the type **CalculationResults.Result**. It is the responsibility of the plugin developer to cast the result to the correct type.

8.4 Adding new results

If a plugin wants to use the ETC-42 main window to show some results, it needs to add them in a **CalculationResults** object (it can use the instance retrieved after executing the simulation or create a new one).

To add string, image or single value results there is only one way. The constructors of the related classes must be used to create the result instances and then the **addResult(result, level)** method can be used. This method can also be used for dataset and image set results.

For datasets and image sets though there is a second method, which can be used if the total number of data (or images) is not known and the data (or images) are entered one at the time. This can be done with the methods **addResult(name, xValue, yValue, xUnit, yUnit, level)** and **addResult(name, key, image, keyUnit, level)**. These methods will create the result with the given name if it doesn't exist, or they will add the given data or image in it if it exists.

9 Plugin Deployment

The deployment of the ETC-42 plugins is fairly easy. The plugin developer needs just to create a jar file containing the implementation of his plugin. Note that a single jar file can contain more than one plugins and the plugins can be in any package. The ETC-42 will discover automatically all the available plugins and there is no need for any configuration files.

It must be reminded here that the plugin jar file does **not** need to contain any of the classes from the ETC-42 framework. These are already available during the execution of the ETC-42. Including all these files will have as result unnecessarily big plugin jar files. For this reason double check how the building method you use (IDE, ant, maven, etc) behaves.

There are two ways to deploy a plugin jar file to the ETC-42. The first one is to copy it manually in the **plugins** directory under the ETC-42 home directory. If not specified with the **−Detc.home** parameter during execution (please see the ETC-42 user manual for more details), the home directory is named **.ETC** and it is under the users home directory. After copying the

plugin jars in the **plugins** directory, the plugins can be reloaded either by restarting ETC-42 or by selecting the **Plugins -> Reload plugins** menu (figure 3). This will result with the new plugins being available under the **Plugins** menu.

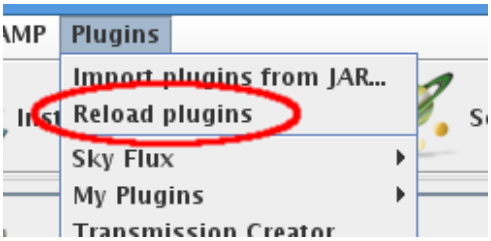

Figure 3: Reload plugins menu

The second way to deploy a plugin is to use the **Plugins -> Import plugins from JAR** menu (figure 4). This menu will open a window for selecting the JAR file containing the plugins and it will handle its deployment without any further actions of the user.

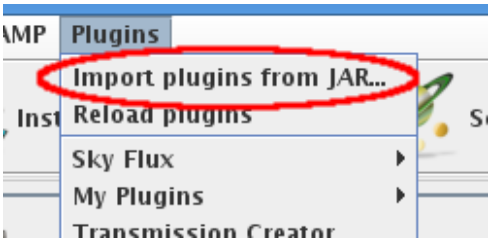

Figure 4: Import plugins from JAR menu

In the case a plugin makes use of some external libraries (it has dependencies in one or more other jar files), the library jar files must also be copied in the **plugins** directory. After copying the files it is necessary to reload the plugins, either by restarting ETC-42 or by selecting the **Plugins -> Reload plugins** menu (this needs to be done also when the second way of deployment is used). ETC-42 plugin framework will load all the jar files in the **plugins** directory and will make available the classes in them to all the plugins.

An extra note must be given here for the way **Eclipse** exports a project as a jar file. **Eclipse** (to solve the problem of including all the dependencies in one jar without unpacking them) is creating an executable jar file which uses a customized class loader having the ability to load classes from jars included inside the main jar. That means you **cannot** use the produced jar as a library for other projects without including the jar files of all its dependencies separately, even if everything works when this jar is executed. It is highly recommended to don't use this way for packaging your plugin classes, as it might lead to confusion and also creates unnecessarily big jar files. If though you really want to use this way, do not forget to also copy in the **plugins** directory all the plugin dependencies.

References

[1] ETC-42 : Dictionary
http://projets.oamp.fr/projects/etc/wiki/Documentation

[2] ETC-42 : Calculation Procedure
http://projets.oamp.fr/projects/etc/wiki/Documentation