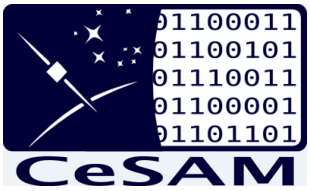

ETC-42 : Universal Exposure Time Calculator

HOW TO
add a new parameter to the data model

Laboratoire d'Astrophysique de Marseille
Centre donneeS Astrophysique de Marseille

Authors:
N. Apostolakos

November 2012

Contents

1 Introduction **2**

2 Data model changes **2**

2.1 Update the related interface 2

2.2 Dataset parameter specific changes 3

3 Database changes **3**

3.1 CreateTables.sql script 3

3.2 Update the schema version 4

3.3 DbUpdater patch 4

4 JPA changes **5**

4.1 Entity class 5

4.2 Static metamodel class 8

4.3 Entity controller 8

4.4 SessionManagerImpl class 9

4.5 DatasetInfoEntityController class (dataset parameters only) 9

5 XML Beans updates **10**

5.1 XSD file 10

5.2 Build auto generated classes 10

5.3 ComponentWithDatasets interface 10

5.4 ComponentWithDatasetsImpl class 10

5.5 DataImporterImpl class (dataset parameters only) 12

5.6 DataExporterImpl class 12

6 Epilogue **13**

1 Introduction

This is a technical document of the ETC project and is written as a help for developers who want to apply code modifications. It describes in detail the necessary modifications for successfully adding a new parameter in any of the components (*Instrument*, *Site*, *Source* or *ObsParam*). The actions described in this document should be done when there is the need for storing some new parameter given by the user.

2 Data model changes

2.1 Update the related interface

The first step is to update the related interface in the **ETC-DataModel** project. These interfaces are in the package `org.cnrs.lam.dis.etc.datamodel` and they are the classes `Instrument.java`, `Site.java`, `Source.java` and `ObsParam.java`.

If the new parameter is a **primitive data type** (int, long, float, etc) then the only thing need to be done is the declaration of public getter and setter methods. For example if the new parameter is a double with name slitLength, the following lines need to be added at the interface:

```
1 public double getSlitLength();
  public void setSlitLength(double slitLength);
```

In the case the parameter is a **dataset** there is need again only for declaration of public getter and setter methods. For example if the new parameter is the total transmission, the following lines need to be added at the interface:

```
  public DatasetInfo getTransmission();
2 public void setTransmission(DatasetInfo transmission);
```

If the parameter should be represented with an enumeration, then the enumeration needs also to be defined in the interface. For example if the new parameter is the psfType and it can have one of the options auto, profile and adaptive optics, the following lines need to be added at the interface:

```
  public enum PsfType {
2     AUTO, PROFILE, AO
  }
4
  public PsfType getPsfType();
6 public void setPsfType(PsfType type);
```

Finally, if the parameter has a unit, a getter method for it needs also to be declared. For example, the slitLength of the first example is expressed in some unit, so the following line needs to be added:

```
public String getSlitLengthUnit();
```

Note that the returned type has to be **String** and the convention used by ETC for naming the getter is **get*ParameterName*Unit**.

2.2 Dataset parameter specific changes

When the added parameter is a dataset the following extra steps that need to be also performed. First, a new dataset type need to be added in the`Type` enumeration of the `Dataset.java` class (in the `datamodel` package). For example, if the dataset `transmission` was added in the `Instrument` component, the following line needs to be added:

```
public enum Type {
    ...
    TRANSMISSION(ComponentType.INSTRUMENT, false),
```

Note that the first parameter is the component in which the dataset was added and that the second parameter is a flag showing if the dataset is a multi-dataset or not, where true means multi-dataset and false single dataset (most of the times this will be false).

Second, a humanly readable translation of the new type must be added in the `messages.properties` file, in the directory `src/main/resources/org/cnrs/lam/dis/etc/datamodel`. Note that the exact name of the enumeration defined above must be used as the translation code:

```
TRANSMISSION=Instrument transmission
```

Finally, a humanly readable translation for the axes labels (used when the dataset is plotted by using the GUI) need to be added in the `messages.properties` file of the project ETC-Controller, in the directory `src/main/resources/org/cnrs/lam/dis/etc/controller`. For the transmission dataset the following lines need to be added:

```
TRANSMISSION_PLOT_X_DESC=Wavelength
TRANSMISSION_PLOT_Y_DESC=Total instrument transmission
```

3 Database changes

3.1 CreateTables.sql script

The **ETC-Persistence** project contains an SQL script for generating all the database schema from scratch (in the `src\main\sql` directory). This script needs to be updated to add a new column for the new parameter in the related table (and a column for the unit, if necessary). The ETC convention for the column name is to use capital letters with underscores (_) between the different words.

For the correct conversion of the data between the java and the database the following mapping of data types must be followed for the column type:

Java Type	Database Type
boolean	SMALLINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
String	VARCHAR(50)
enumeration	VARCHAR(255) NOT NULL
Dataset	BIGINT

For example, for adding the slitLength (double with unit), the psfType (enumeration) and the transmission (dataset) described earlier, the following lines need to be added at the creation of the INSTRUMENT table:

```
SLIT_LENGTH DOUBLE,
SLIT_LENGTH_UNIT VARCHAR(50),
PSF_TYPE VARCHAR(255) NOT NULL,
TRANSMISSION BIGINT,
```

3.2 Update the schema version

Because the ETC database is installed locally in every machine where the system is running, there is the need for versioning the database schema. This is done with the use of the table **ETC_INFO** which contains the field **DB_VERSION**. This field must **always** be increased **BY 1** every time the schema is modified, as it is used by the system to apply patches to older databases.

The version number must be increased in two places. The first one is the `CreateTables.sql` script (mentioned above). The line which initialize the values of the table **ETC_INFO** needs to be modified with the new version. For example, if the previous version was 3, the line must be modified as:

```
INSERT INTO ETC_INFO VALUES (4);
```

The second place where the database schema version needs to be changed is in the file `DbUpdater.java` of the package `org.cnrs.lam.dis.etc.controller` (project **ETC-Controller**). This class has the field `DB_VERSION` which keeps the current database schema version. For example, if the previous version was 3, the class `DbUpdater.java` should be modified as:

```
private static final int DB_VERSION = 4;
```

3.3 DbUpdater patch

The method of the ETC for updating the already installed databases is implemented by the class `DbUpdater` mentioned above. The way this class works is by applying patches to the

database for updating version by version, until the latest version. For implementing a new patch a new method needs to be added in the class. This method must be **static**, it must get one argument of type `java.sql.Statement` and it must be named like **updateFromVersion**<***OldVersion***>**To**<***NewVersion***>. Here it must be noted that the old version and the new version must differ only by one. There is no case where a patch might increase the version number by 2 or more (a version change that does not require a patch doesn't make sense anyways).

The update method must implement all the table modifications, by using the statement provided. The method is going to be called from inside a transaction, so transaction control is not necessary, but the method should throw an `SQLException` at the case of any failure. The given statement is handled by the caller and it should not be closed. If the patch need to use a prepared statement, it can use the database connection of the parameter statement like this:

```
statement.getConnection().prepareStatement(...);
```

In the case the new parameter is an enumeration (and the new column is not allowing null values), a proper default value must also be given. This value **must** be the name of the enumeration constant. The same should be done also for units, as the current version of ETC does not allow their modification.

For example, for updating the database for the slitLength, the transmission and the psfType as mentioned above, assuming the previous version was 3, the following method needs to be added in the `DbUpdater` class:

```
private static void updateFromVersion3To4(Statement statement)
                                                throws SQLException {
    // Make the table modifications
    statement.executeUpdate("ALTER TABLE INSTRUMENT ADD SLIT_LENGTH DOUBLE");
    statement.executeUpdate("ALTER TABLE INSTRUMENT ADD " +
                            "SLIT_LENGTH_UNIT VARCHAR(50) DEFAULT 'arcsec'");
    statement.executeUpdate("ALTER TABLE INSTRUMENT ADD TRANSMISSION BIGINT");
    statement.executeUpdate("ALTER TABLE INSTRUMENT ADD " +
                            "PSF_TYPE VARCHAR(255) NOT NULL " +
                            "DEFAULT 'AUTO'");
}
```

4 JPA changes

4.1 Entity class

For the correct communication between the Java world and the database word, the entity class of the related component must be updated. These classes are in the project **ETC-Persistence** and the package `org.cnrs.lam.dis.etc.persistence.database.entities`. They are the classes ending with the keyword **Entity** (for example `InstrumentEntity`). There are two necessary modifications in the entity files.

First, a new field need to be created to map to the new parameter. The type of the field must be according the modifications done to the component interface (first step of this document). The field must also have the following JPA annotations:

`@Column(name="<COLUMN_NAME>")`	For all the parameters except of datasets
`@Basic(optional=false)`	For columns which are not null (like enumerations)
`@Enumerated(EnumType.STRING)`	For enumerations
`@JoinColumn(name="<COLUMN_NAME>", referencedColumnName="ID")`	For datasets
`@ManyToOne`	For datasets

For example, for the slitLength, the psfType and the filter transmission used earlier, the following fields should be added in the `InstrumentEntity` class (note that the filter transmission is of type `DatasetInfoEntity` and it refers to another entity class):

```
@Column(name = "SLIT_LENGTH")
private double slitLength;

@Column(name = "SLIT_LENGTH_UNIT")
private String slitLengthUnit;

@Basic(optional = false)
@Column(name = "PSF_TYPE")
@Enumerated(EnumType.STRING)
private PsfType psfType;

@JoinColumn(name = "TRANSMISSION", referencedColumnName = "ID")
@ManyToOne
private DatasetInfoEntity transmissionEntity;
```

Second, getter and setter methods need to be added for accessing the new parameters. The setter methods must take care to update the `modified` flag when the parameters value is changing. For example:

```
@Override
public double getSlitLength() {
    return slitLength;
}

@Override
public void setSlitLength(double slitLength) {
    if (this.slitLength != slitLength)
        modified = true;
    this.slitLength = slitLength;
}

@Override
public String getSlitLengthUnit() {
    return slitLengthUnit;
}
```

```
18 public void setSlitLengthUnit(String slitLengthUnit) {
     if(this.slitLengthUnit != null && !this.slitLengthUnit.equals(slitLengthUnit))
20       modified = true;
    this.slitLengthUnit = slitLengthUnit;
22 }

24 @Override
   public PsfType getPsfType() {
26     return psfType;
   }
28
   @Override
30 public void setPsfType(PsfType psfType) {
     if (this.psfType != psfType)
32       modified = true;
     this.psfType = psfType;
34 }
```

In the case of a dataset parameter, the implementation of the getter and setter methods of the interface is a little bit more complicated, as there is a variable of `DatasetInfoEntity` type and not directly a `DatasetInfo`. Because of that, there is need for getter and setter methods for the entity variable and the interface methods should use them to retrieve and set the entity (updating also correctly the `modified` flag). For example:

```
   public DatasetInfoEntity getTransmissionEntity() {
2    return transmissionEntity;
   }
4
   public void setTransmissionEntity(DatasetInfoEntity transmissionEntity) {
6    this.transmissionEntity = transmissionEntity;
   }
8
   @Override
10 public DatasetInfo getTransmission() {
     if (getTransmissionEntity() == null) }
12       return null;
     }
14   return new DatasetInfo(getTransmissionEntity().getName(),
                            getTransmissionEntity().getNamespace(),
16                          getTransmissionEntity().getDescription());
   }
18
   @Override
20 public void setTransmission(DatasetInfo transmission) {
     // Check if the new transmission is the same with the old one or not
22   if ((this.transmissionEntity == null && transmission != null) ||
           (this.transmissionEntity != null && transmission == null) ||
24         (this.transmissionEntity != null && transmission != null &&
           !(this.transmissionEntity.getName().equals(transmission.getName())) &&
26         (this.transmissionEntity.getNamespace() == null
               ? transmission.getNamespace() == null
28             : this.transmissionEntity.getNamespace()
```

```
                        .equals(transmission.getNamespace()))
            ))))
            modified = true;
        if (transmission == null) {
            setTransmissionEntity(null);
            return;
        }
        DatasetInfoEntityJpaController controller
            = new DatasetInfoEntityJpaController();
        setTransmissionEntity(controller
            .findDatasetEntity(Type.TRANSMISSION, transmission));
}
```

4.2 Static metamodel class

The ETC uses static metamodel classes to allow use of strongly typed criteria. These classes are in the package `org.cnrs.lam.dis.etc.persistence.database.entities` of the project **ETC-Persistence** and they are the classes ending with the keyword **Entity_** (for example `InstrumentEntity_`). These classes are used to map the entity fields to specific types and a new mapping needs to be added for the new parameter. The mapping is done by adding a `SingularAttribute` field in the class, which uses generics for doing the mapping.

For example, for the slitLength, the transmission and the psfType, the following fields need to be added in the `InstrumentEntity_` class:

```
public static volatile SingularAttribute<InstrumentEntity,Double> slitLength;
public static volatile SingularAttribute<InstrumentEntity,String> slitLengthUnit;
public static volatile SingularAttribute<InstrumentEntity,PsfType> psfType;
public static volatile SingularAttribute<InstrumentEntity,DatasetInfoEntity>
                                                        transmissionEntity;
```

4.3 Entity controller

Each entity has a corresponding controller class, which provides operations related with the entity. These classes are in the package `org.cnrs.lam.dis.etc.persistence.database.entities` of the project **ETC-Persistence** and they are the classes ending with the keyword **EntityJpaController** (for example `InstrumentEntityJpaController`). This controller contains a method for creating new entity objects, named `createNewUnpersisted<Component>Entity` which needs to be updated to set default values for the new parameters. If a new parameter is an enumeration (or any other not null field) or it has a unit this step is necessary. In any other case it can be omitted.

For continuing the example with the psfType and the slitLength, the following lines need to be added in the `createNewUnpersistedInstrumentEntity` method of the `InstrumentEntityJpaController` class:

```
instrument.setSlitLengthUnit("arcsec");
instrument.setPsfType(PsfType.AUTO);
```

Note that a default is given also for the units, as currently there is no GUI support for setting the units of the parameters.

In the case of a dataset parameter the method `find<Components>UsingDataset` (where `<Components>` is replaced with Instruments, Sites, etc accordingly) must also be modified, to add in the `cb.or` method the new dataset as parameter. For the transmission example the following line should be added:

```
cb.equal(from.get(InstrumentEntity_.transmissionEntity), datasetInfoEntity)
```

4.4 SessionManagerImpl class

Finally, the `SessionManagerImpl` class needs to be updated to copy over the new parameters when the SaveAs option is executed. This class is in the project **ETC-Persistence** and the package `org.cnrs.lam.dis.etc.persistence.database`, and the related methods are the ones named `save<Component>As`.

For our example the following lines need to be added in the method `saveInstrumentAs`:

```
// Copy the slit width and its unit
newInstrument.setSlitWidth(instrument.getSlitWidth());
newInstrument.setSlitWidthUnit(instrument.getSlitWidthUnit());
// Copy the PSF type
newInstrument.setPsfType(instrument.getPsfType());
// Copy the transmission (change the namespace if necessary)
if (instrument.getTransmission() == null
        || instrument.getTransmission().getNamespace() == null) {
    newInstrument.setTransmission(instrument.getTransmission());
} else {
    DatasetInfo datasetInfo = instrument.getTransmission();
    if (!newInfo.getName().equals(datasetInfo.getNamespace())) {
        datasetInfo = copyDatasetInNamespace(Type.TRANSMISSION
            , instrument.getTransmission(), newInfo.getName());
    }
    newInstrument.setTransmission(datasetInfo);
}
```

4.5 DatasetInfoEntityController class (dataset parameters only)

In the case a dataset parameter is added the method `find<Component>DatasetsInNamespace` (where `<Component>` is replaced with Instrument, Site, etc) of the class `DatasetInfoEntityController` needs to be modified, to search also for the new type of dataset.

For the transmission example, the following line should be added in the `cb.or` parameters:

```
1 cb.equal(from.get(DatasetInfoEntity_.type), Type.TRANSMISSION)
```

5 XML Beans updates

5.1 XSD file

The ETC uses XML files for importing and exporting components and more precisely the XML-Beans framework. The XSD definitions of the XML are in the directory `src\main\xsd` of the **ETC-DataImportExport** project. These files need to be updated for the new parameters.

For the slitLength, transmission and psfType example, the following lines need to be added in the `Instrument.xsd` file:

```
1 <xs:element name="slitLength" type="DoubleUnit"/>
  <xs:element name="psfType" type="xs:string"/>
3 <xs:element name="transmission" type="Graph"/>
```

5.2 Build auto generated classes

The XMLBeans auto generated classes are generated automatically when the **ETC-DataImportExport** project is compiled. It is though recommended at this step to generate them, so if an IDE tool is used (like eclipse, netbeans, etc) the autocomplete will work correctly for the next steps. The generation of the classes can be done by trying to build the ETC. The build will fail, as there are more code changes need to be done, but the XMLBeans classes will be generated.

5.3 ComponentWithDatasets interface

Only for the case a new parameter is a dataset, a new method must be added in the interface `<Component>WithDatasets` of the pachage `org.cnrs.lam.dis.etc.dataimportexport`, to allow retrieval of the dataset. For the example of the transmission the following method should be added:

```
1 public Dataset getTransmissionDataset();
```

5.4 ComponentWithDatasetsImpl class

The **ETC-DataImportExport** project defines an ETC component implementation, which is used as a wrapper above the XMLBeans classes. These classes are in the package `org.cnrs.lam.dis.etc.datai` and they are ending with the keyword **WithDatasetsImpl** (for example `InstrumentWithDatasetsImpl`). As these classes are used only when importing a component, the getter method of the parameter must be implemented to delegate the call to the XMLBean and the setter method must be

implemented to throw an `UnsupportedOperationException`. Note that in the case of a dataset parameter an extra method needs to be implemented, to return the data (and not only the dataset info).

For example, for the slitLength, the transmission and the psfType, the following methods need to be added in the `InstrumentWithDatasetsImpl` class:

```
@Override
public double getSlitLength() {
    return instrumentXml.getSlitLength().getValue();
}

@Override
public void setSlitLength(double slitLength) {
    throw new UnsupportedOperationException("Read only version of Instrument");
}

@Override
public String getSlitLengthUnit() {
    return instrumentXml.getSlitLength().getUnit();
}

@Override
public PsfType getPsfType() {
    return PsfType.valueOf(instrumentXml.getPsfType());
}

@Override
public void setPsfType(PsfType type) {
    throw new UnsupportedOperationException("Read only version of Instrument");
}

@Override
public Dataset getTransmissionDataset() {
    Graph graph = instrumentXml.getTransmission();
    return Helper.convertGraphToDataset(graph, componentInfo.getName()
                                        , Dataset.Type.TRANSMISSION);
}

@Override
public DatasetInfo getTransmission() {
    if (instrumentXml.getTransmission() == null) {
        return null;
    }
    return new DatasetInfo(instrumentXml.getTransmission().getName()
                           , componentInfo.getName(), null);
}

@Override
public void setTransmission(DatasetInfo transmission) {
    throw new UnsupportedOperationException("Read only version of Instrument");
}
```

5.5 DataImporterImpl class (dataset parameters only)

If the new parameter is a dataset, the `DataImporterImpl` class needs to be modified. More precisely the method `importDatasetFromComponentFile` need to be modified to have a new case in the switch statement, for example:

```
switch (type) {
    ...
    case TRANSMISSION:
        dataset = getDatasetFromInstrument(file, type, info);
        break;
    ...
}
```

and also the method `getDatasetFrom<Component>` need to be modified to return the correct dataset, for example:

```
case TRANSMISSION:
    return instrument.getTransmissionDataset();
```

5.6 DataExporterImpl class

Finally, the class `DataExporterImpl` of the package `org.cnrs.lam.dis.etc.dataimportexport` need to be updated to export correctly the new parameter. This class contains methods with names `export<Component>InFile` which are creating a new XMLBeans object and they set all the parameters.

For example, for the slitLength, the transmission and the psfType, the following lines need to be added in the `exportInstrumentInFile` method of the `DataExporterImpl` class:

```
// Set the slit length value and unit
DoubleUnit slitLength = instrumentXml.addNewSlitLength();
slitLength.setValue(instrument.getSlitLength());
slitLength.setUnit(instrument.getSlitLengthUnit());

// Set the psf type
instrumentXml.setPsfType(instrument.getPsfType().name());

// Set the transmission
if (instrument.getTransmission() != null) {
    Graph transmission = instrumentXml.addNewTransmission();
    populateGraph(transmission, provider.getDataset(Dataset.Type.TRANSMISSION
                                                    , instrument.getTransmission()));
}
```

Note that for the enumerations the parameter passed is the result of the `name` method. This is for the case the `toString` method is overridden.

6 Epilogue

After applying all the above changes the ETC will be handling correctly the new parameter. The next steps will be to modify the **ETC-Calculator** project for implementing the new logic of the SNR calculation related with the new parameter and to implement the GUI modifications necessary in the **ETC-UI** project for presenting the new parameter to the user and let him modify its value. These steps are out of the scope of this document and are covered in other dedicated documents.

Here is must be noted that after the modifications mentioned in this document the command line version of the ETC (when it runs with the parameter `-Detc.uiMode=cl`) will be directly usable with the new parameter, as it uses reflection and it will automatically detect the new parameter. This mode can be used for developing the calculator modifications before implementing the GUI changes.