Import & Export

How Is It Done?

The Introduction

Import and Export are very similar in the way how the framework is designed. The list of responsibilities is strictly separated and well defined for every part of the structure. Technically, it follows the famous Design Pattern - **Builder**.

In the next sections we will study the Import and Export framework more thoroughly - the common properties, implementation details, and extra cases.

How to Spit the Data from the Database - The Export

The Export framework will be described in detail here. We will start with the list of all classes that are involved when exporting data takes place and what their responsibilities are. Later, we shall discuss how the whole framework is linked to the whole.

The task of the export procedure is to get specified parts of the selected data from the database and store it into a file.

Export from the Inside

This section contains information about how the Export procedure works (how is it actually done). It should provide you with sufficient information to be able to write your own **Builder**.

DBLayer

The database layer is the basic part. There is a document by *Tomáš Kovařík* describing this matter more thoroughly. You shall find it in the documentation, too.

Record (net.sf.plantlore.common.record.Record)

The **Record** is a common ancestor of all holder objects. Every holder object corresponds with a certain table in the database. The most important thing is, that Export only cares about several tables that are directly connected with the Occurrence data. These tables are called **The Basic Tables**. The basic tables are: ***Author***, ***AuthorOccurrence***, ***Habitat***, ***Metadata***, ***Occurrence***, ***Phytochorion***, ***Plant***, ***Publication***, ***Territory***, ***Village***. Some of these tables contain references (foreign keys) to other tables, such as *User*. Foreign keys to those tables are unimportant and neither Export nor Import makes any changes to them. It is the DBLayer that shall set them - because of the security reasons; only the database layer (possibly running on the server) can assure, that those foreign keys are set appropriately.

Another part of the records, that is ignored by the export, is the timestamp of some records. Again, it is the database layer that should set those columns appropriately.

The following table shows important columns of each table:

Table	***Columns***
Occurrence	UnitIdDB, UnitValue, ***Habitat***, ***Plant***, YearCollected, MonthCollected,

Table	***Columns***
	DayCollected, TimeCollected, IsoDateTimeBegin, DataSource, ***Publication***, Herbarium, ***Metadata***, Note
Habitat	***Territory***, ***Phytochorion***, ***NearestVillage***, Quadrant, Description, Country, Altitude, Latitude, Longitude, Note, Deleted
Phytochorion	Code, Name
Territory	Name
Village	Name
Plant	Taxon, Genus, Species, ScientificNameAuthor, CzechName, Synonyms, Note, SurveyTaxId
Metadata	TechnicalContactAddress, TechnicalContactEmail, TechnicalContactName, ContentContactAddress, ContentContactEmail, ContentContactName, DatasetDetails, DatasetTitle, SourceId, SourceInstitutionId, OwnerOrganizationAbbrev, BiotopeText, RecordBasis
Publication	CollectionName, CollectionYearPublication, JournalName, JournalAuthorName, ReferenceCitation, ReferenceDetail, Url, Note, Deleted
Author	WholeName, Organization, Role, Address, PhoneNumber, Email, Url, Note, Deleted
AuthorOccurence	***Author***, ***Occurrence***, Role, Note, Deleted

For a further description of the meaning of all columns see the Database Model Documentation by *Lada Oberreiterová.*

The legend - columns:

- ***bold italic*** = a ***foreign key*** (a reference to another table),
- plain = a ***property*** (a column that contains some value, like String or Integer),
- green = this column is NOT-NULL (it always contains a value); note that foreign keys are very often not-null, but it is not a rule: the Occurrence.Publication is a foreign key, yet it may not be specified.

Note that some tables may contain more columns - if they are not listed in the table above, they are not important for the Export procedure.

The most important functions of this object:

- Object getValue(String column)

  Returns a value in the specified column of a record. The programmer doesn't have to decide which getter he must call in order to obtain the value of a column - it is this function that does it for him.

Selection (net.sf.plantlore.common.Selection)

This class ***holds the list of selected records***. It doesn't specify what kind of records it holds. The Selection is typically created by the invoker of the Export procedure - there are several methods that support creating the selection: `add()`, `remove()`, `invert()`, `none()`, `all()`. For a detailed description see the JavaDoc.

The Selection means - in the database terminology - the *restriction of rows*.

The most important method for the Export framework is the

- boolen contains(Record record);

  This method give true, if the record is part of the selection.

Template (net.sf.plantlore.client.export.Template)

The **Template** is very similar to the Selection. It ***stores information about the selected columns***. The User can specify which columns are important for him and the framework should export only those columns.

The Template is - in the database terminology - known as the *projection of columns*.

There are several methods that support creating of the Template: `set()`, `unset()`, `setEverything()`, `unsetEverything()`, `setAllProperties()`.

Builder (net.sf.plantlore.client.export.Builder)

An interface. The interface contains several methods that every ConcreteBuilder must implement. ConcreteBuilder is responsible for ***building the output from a given record (in the form of a tree of holder objects)***.

The ConcreteBuilder constructs a concrete output in a particular file-format.

The Builder requires:

- The **Template** that describes which columns should be sent to the output.

The most important functions are:

- void startRecord()

  Informs the Builder that a new record will be sent to the output. Some records may comprise more parts - for instance an Occurrence data may *contain* several AuthorOccurrences.

- void finishRecord()

  Informs the Builder that all parts of the record were handed over. There will be no more parts of this record.

- void part(Record r)

  This method receives a part of the whole record - such as Occurrence and AuthorOccurrences. It is the task of this method to ***traverse the given record*** - because it may span over several tables - ***and build the output appropriately***. Note that this method should reflect the decision of the User and ***use the Template to decide which columns should be exported and which should not***.

  Consider the following example:

  Someone (the DefaultDirector, see later) calls `part(occ)`; where `occ` is a record containing some occurrence data. This method must perform this action:

```
Template template

part( Record R )

  for each property P of the record R do
    if template.isSet( P ) then output( R, P )

  for each foreign key K of the record R do
    part( (Record)R.getValue( K ) )
```

  First, it must send all desired columns (properties) to the output. Second, it must traverse the record and do the same with its subrecords.

  Here is an example. Let's suppose the User wanted to export Habitats and is interested in the following columns: *Habitat.Description*, *Habitat.Country*, *Territory.Name*, *Village.Name*.

  The Builder receives an instance of **Habitat.class**. The output must be called four times for each Habitat:

```
output( Habitat, Description )
output( Habitat, Country )
output( Territory, Name )
output( Village, Name )
```

Because the Builder is an interface and this action should perform every Builder, an **AbstractBuilder** were created to simplify this task. The AbstractBuilder ***provides the particular implementation of the part() method*** listed above. The only thing when writing your own Builder is to subclass the AbstractBuilder and implement its method

- abstract void output(Class table, String column, Object value);

  The method receives three parametres that should suffice to create the output. The first is the "name" of the table, the second is the name of the column, and the third is the particular value contained in that column.

  For instance: `output( Author.class, Author.WHOLENAME, "Wang Jinrey" );`

It is recommended to override other methods of the AbstractBuilder as well (`startRecord()`, `finishRecord()`, `header()`, `footer()`).

DefaultDirector (net.sf.plantlore.client.export.DefaultDirector)

The **DefaultDirector** uses several entities. Its purpose is to ***fetch specified records from the database and*** for those that are selected ***call the Builder's method part()***.

Now, it should be clear what the DefaultDirector needs:

- The **database layer** to obtain the specified records.
- The **Selection** to decide whether that record is selected or not.
- The **Builder** that will construct the output.
- The **resultset identificator**. The resultset identificator identifies the result of a Select Query. The DefaultDirector will iterate over this resultset to obtain records that should be sent to the Builder.

Here's how the DefaultDirector works:

```
Builder build, DBLayer db, Selection selection, Integer result

run()
  build.header()
  rows = db.getNumRows( result )
  for i = 0 to rows - 1 do
    Record record = db.more( result, i, i ) // fetch the i-th record
    if !selection.contains( record ) continue

    build.startRecord()
    build.part( record )

    if record instanceof Occurrence then
      for each AuthorOccurrence AO associated with record do
        build.part( AO )

    build.finishRecord()
  done
  build.footer()
```

As you can see, the DefaultDirector is very versatile. ***It doesn't care what kind of records it processes***. Except, of course, the so called Occurrence data. If the record's instance of the Occurrence, the User surely wanted to process the associated AuthorOccurrences as well. The AuthorOccurrences should **in fact** be part of the Occurrence table, it's just that the one-to-many relationship cannot be modelled any other way.

Look at the difference: The User wanted to export Habitats. Everything conecerning the

Habitat - such as Phytochorion, Territory, Village - can be reached directly via the foreign keys.

With the Occurrences the situation is different. The Occurrence record doesn't contain the reference to the AuthorOccurrence table which is why the AuthorOccurrences associated with this Occurrence must be loaded separately. This is the only one-to-many relationship the export must deal with.

The AuthorOccurrence table merely provides the link from the Occurrence to its Authors and their role when it comes to this Occurrence.

See the Database Model Documentation to see the detailed description of the tables.

Note that the DefaultDirector expects the **Builder** - it doesn't create it!

Export from the Outside

In this section we will concentrate on how to use the whole framework to get our data exported. Ie. how to use the framework without the knowledge of how it actually works.

ExportMng (net.sf.plantlore.client.export.ExportMng)

The ExportMng simplifies the usage of the whole export procedure. There are several ways to set the Export Manager's properties.

The Export manager ***creates all necessary participants, starts and controls the export, and performs the final cleanup***.

There are two things that have to be supplied to the Export manager:

- The **database layer** that will carry out the ExportMng's requests.
- Either the **Select Query** or the **resultset identifier**.

  If the Select Query is supplied, the Export Manager executes it and - during the final cleanup - disposes of it. It is **higly recommended** to use a Select Query instead of the resultset identificator. The Export procedure runs in a separate thread and there is no telling how long it will take to export all the selected records. If the Export Mng was supplied with the result set identifier, the caller would have to ensure the Select Query is not closed during the export procedure. This may be difficult which is why it is recommended to let the Export Mng execute the Select Query by itself.

There are several things that can be supplied optionally:

- The **Selection** that stores the marked records of the result. If the Selection is not specified, the Export Mng assumes that every row is selected.
- The **Template** that stores the important columns. If the Template is not specified, the Export Mng assumes that every column of the record is selected.

Before the Export can begin, following things must be specified as well:

- The **XFilter**. The XFilter stores a list of file extension that are related to the specified file format. ***The XFilter*** in fact ***specifies which ConcreteBuilder (file format) should be used*** to construct the output.
- The **Filename**. This is the filename as the User typed it. The XFilter has a method that can produce a correct filename (with the correct extension).

Here's how the Export Manager works:

```
DBLayer db, SelectQuery query, Template template,
Selection selection, XFilter filter, String filename

start()
  result = db.executeQuery( query )
  file = new File( filter.suggestName( filename ) )
```

```
writer = new FileWriter( file )

switch( filter.getDescription() )
  "CSV": builder = new CSVBuilder( template, writer )
  "XML": builder = new XMLBuilder( template, writer )
  ...

director = new DefaultDirector( db, result, builder, selection )

T = new thread executing director.run()

Cleanup = new thread executing {
  T.join() // wait untill T finishes
  writer.close()
  db.closeQuery( query )
}
```

The Export Manager creates the **DefaultDirector** and the appropriate **ConcreteBuilder** and starts the export in another thread ***T***. It also creates a thread that is responsible for performing the final cleanup after the thread ***T*** finishes.

The Export Manager provides several methods that can be used to **monitor** the progress of the export or **abort** it: `abort()`, `isAborted()`, `isExportInProgress()`, `getNumberOfResults()`, `getNumberOfExported()`.

Aborting Export

Aborting export is very simple. The **DefaultDirector** in its `run()` method contains a test whether the export was aborted.

The `DefaultDirector.run()` works like this:

```
for i = 0 to rows - 1 do
  if aborted then break
...
```

Setting `aborted` to **true** will cause the loop to stop, write the footer, and terminate the thread ***T***. After the thread is terminated, the thread ***Cleanup*** (started by the Export Manager) performs the final cleanup.

Dealing with Exceptions during Export

The most difficult thing was the Exception handling. Part of the export runs in a separate thread and it would be next to impossible to catch those exceptions. Which is why the following mechanism was introduced:

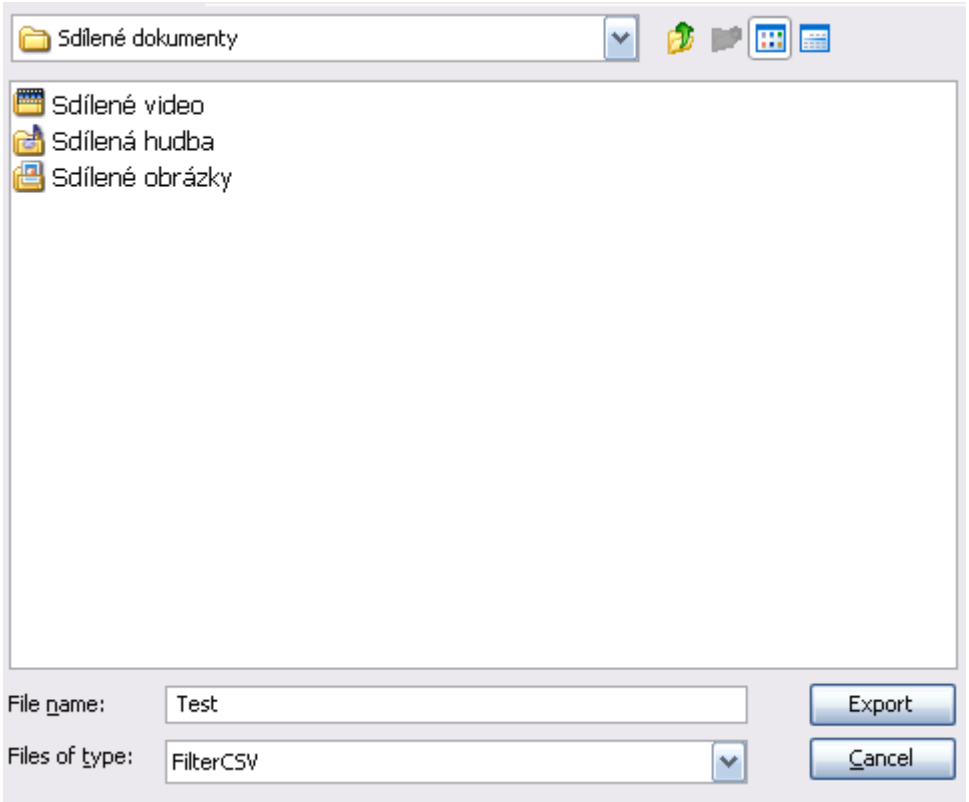
```
DefaultDirector.run()
  try
    build.header()
    ...
    for i = 0 to rows - 1 do
      ...
    build.footer()
  catch Exception E {
    setChanged()
    notifyObservers( E )
  }
```

Every exception is caught and sent to the Observers. The `run()` method ends, but the observers may present that exception (error) to the User informing him what went wrong. Because it is the Export Manager who creates the DefaultDirector, it also mediates the notification of all observers - the Observer simply observers the Export Manager which in turn observes the DefaultDirector. If the notification is needed, the Export Manager passes it through.

Export from the User's View

This section describes briefly the whole export procedure from the view of the User. It makes clear where some participants come into existence. See the **ExportMng** for further details.

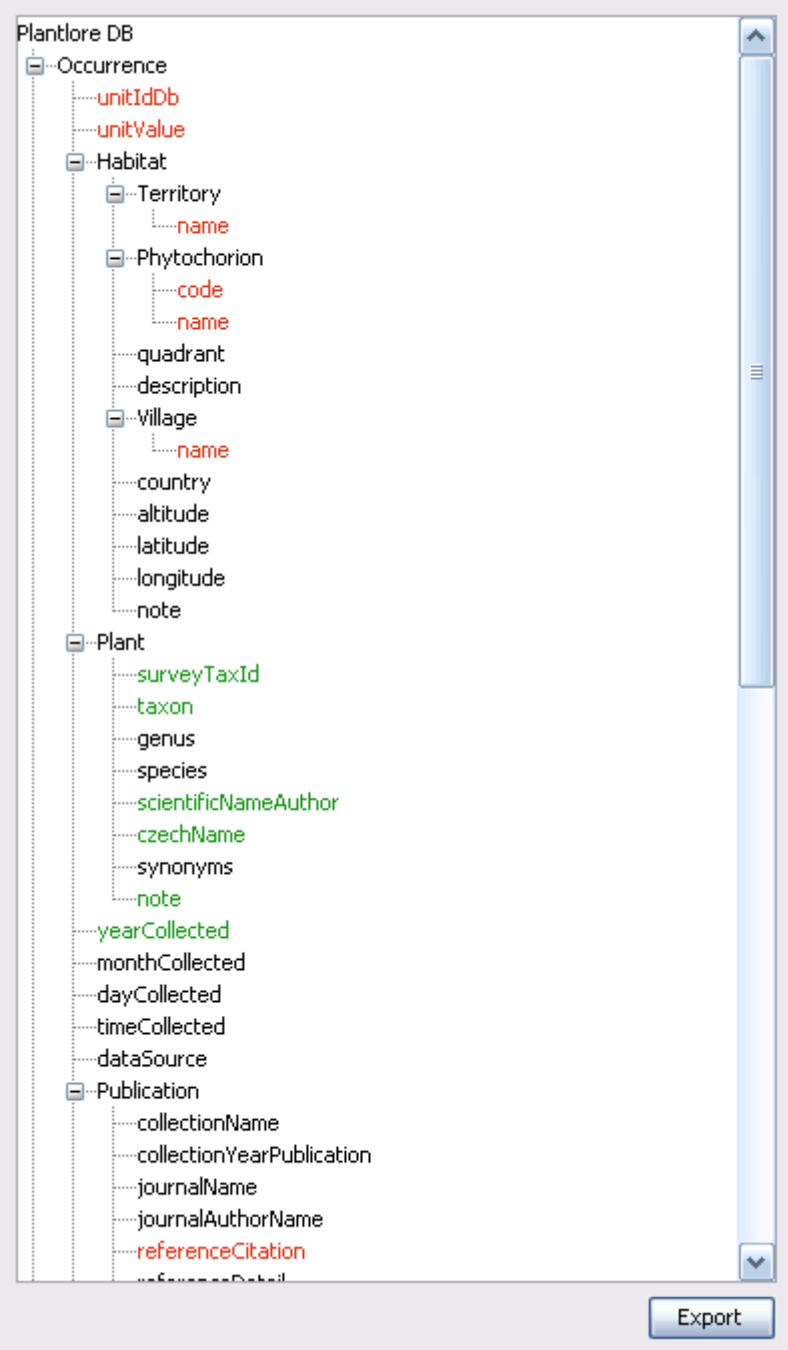
View A



First, the User is presented with a "Save-File" dialog. He ***specifies a filename and a filter***. The **filename**'s extension may not be specified, it will be created accordingly to the **filter**.

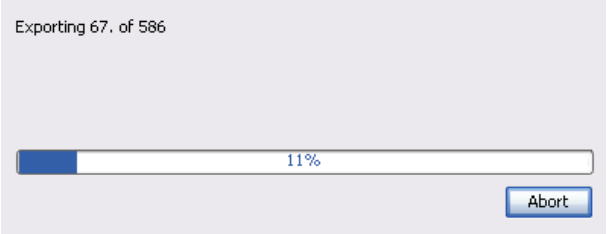
In the picture the **filename** "Test" was chosen and the **filter** is a CSVFilter which means that a CSVBuilder must be used. The default extension of the CSVFilter is ".txt" - the filename the **filter** will suggest is "Test.txt".

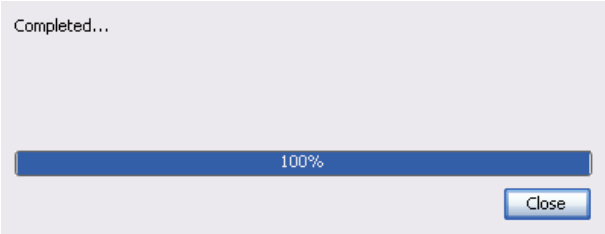
View B (optional)



Some formats - like CSV or XML - allow the User to specify which columns he wants to export. There is a special modification of the JTree called **XTree** that presents the User with the list of available columns. Not-null values are marked in red. Selected columns are displayed in green. The ***XTree can produce a Template*** that stores the list of selected columns.

Progress

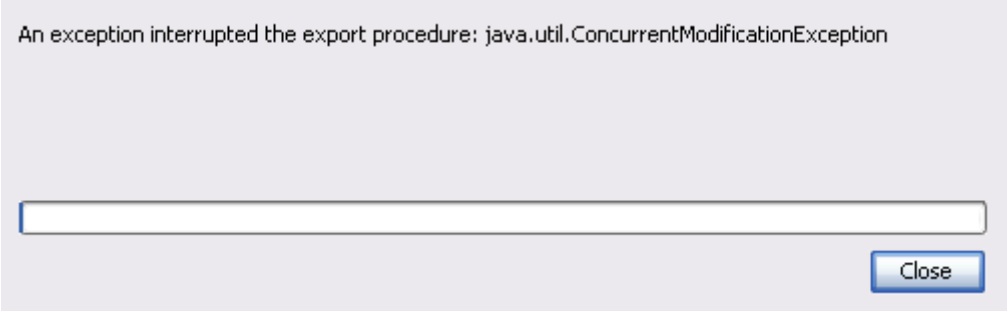




There is a progress dialog that monitor the state of the export. This dialog is a separate window and closing this window won't abort the export.

Clicking the ***Abort button will abort the export*** immediately.

Errors



As you can see the Progress Dialog also shows exceptions that occurr during the export procedure.

Conclusion

We have described the whole Export procedure from several points of view. It should be clear what all participants are responsible for, when they are created, who creates whom, and what is the source of some events.