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 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 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:

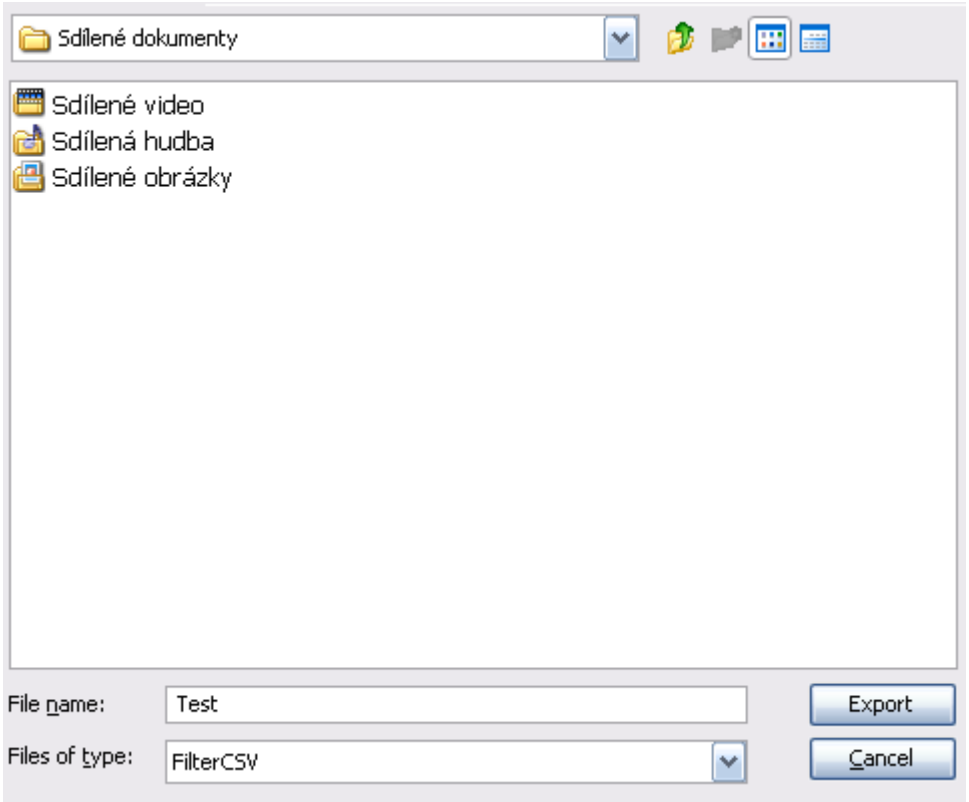
```
DefaultDirector.run()
  try
    build.header()
    ...
    for i = 0 to rows - 1 do
      ...
    build.footer()
  catch Exception E {
    setChanged()
    notifyObservers( E )
  }
```

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.

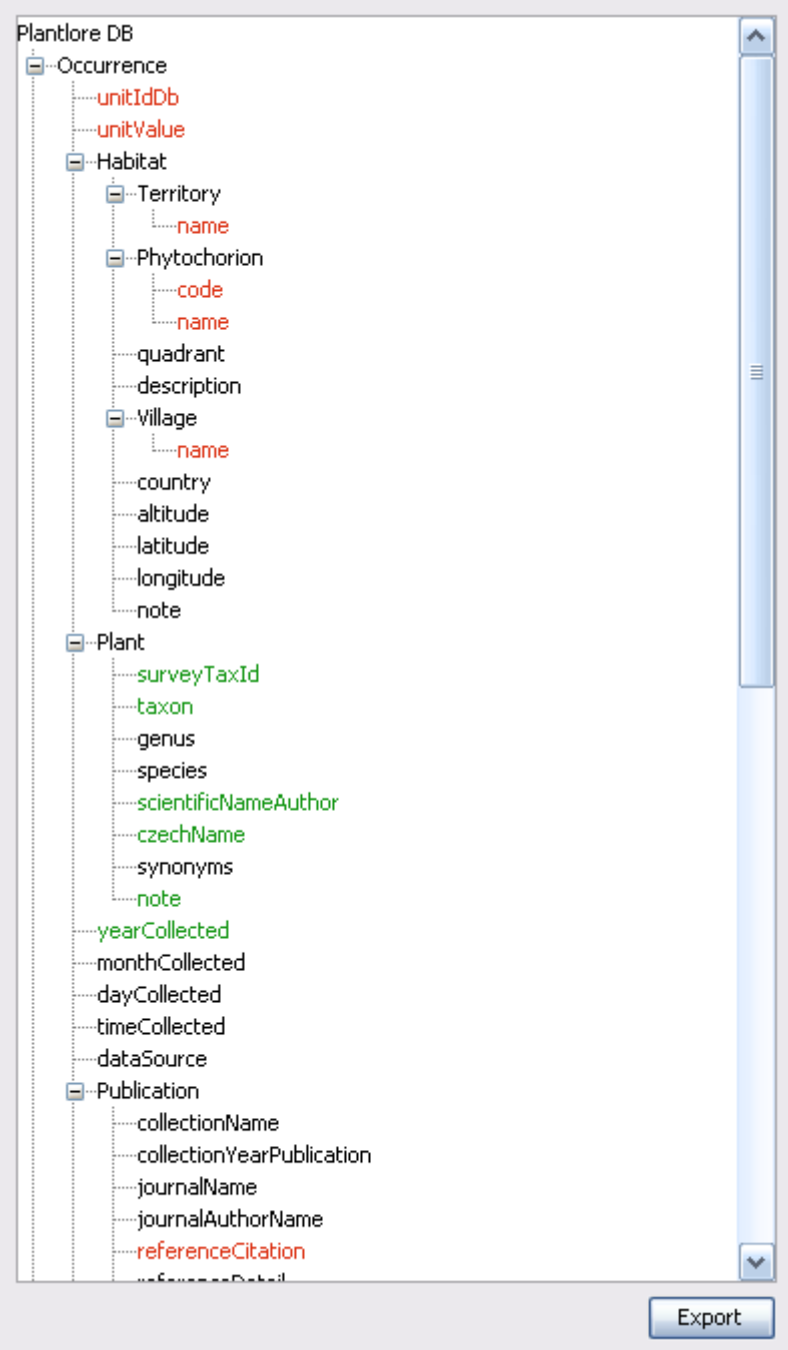
View A



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**.

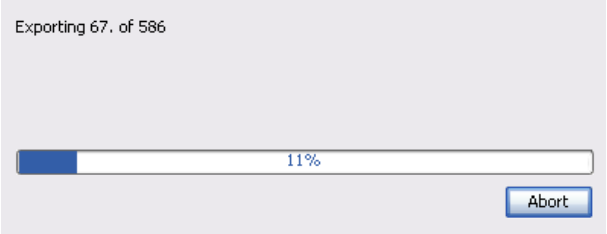
In the picture the **filename** "Test" was chosen and the **filter** is a CSVFilter which means that a CSVBuilder must be used. The default extension of the CSVFilter is ".txt" - the filename the **filter** will suggest is "Test.txt".

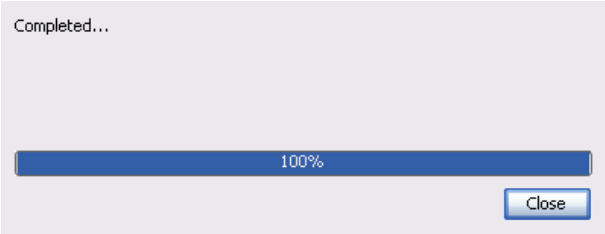
View B (optional)



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

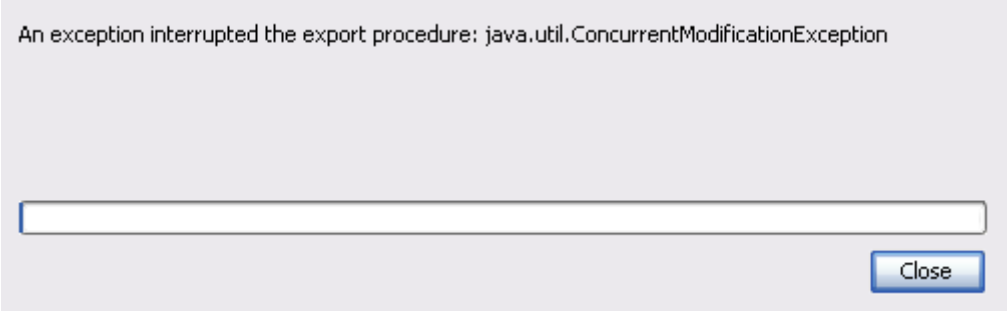




There is a progress dialog that monitor the state of the export. This dialog is a separate window and closing this window won't abort the export.

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.

How to Stuff the Data into the Database - The Import

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

Import is very similar to Export. However, there are several things that have to be considered, since the import actually modifies the database.

As you will see, Import resembles Export in many ways.

The Import described here concerns the Occurrence data only! This is where Import and Export differ - Import cannot be as versatile as Export. The DefaultDirector must ensure the integrity of the database: some tables cannot be modified during the import of occurrence data.

Import from Inside

This section contains information about how the Import is designed. It should provide you with sufficient information to be able to write your own **Parser**.

DBLayer (net.sf.plantlore.middleware.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 Import 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 import, is the timestamp. 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, 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,

Table	***Columns***
	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.

The legend - Tables:

- **bold** = the ***root table*** (the most important table, the core of each occurrence record),
- red = an ***immutable table***. The concept of immutable (read-only) tables was introduced to ensure the consistency of the database. Immutable tables cannot be changed by a common User. This mechanism shall prevent the User from inserting a possibly corrupted data. Take the list of plants for instance - the User is prevented from inserting a record that refers to a plant that doesn't exist.

  The only way to change an immutable table is via a special import.
- *italic* = one-to-many relationship. One Occurrence record can contain several AuthorOccurrences. This table links Occurrences with Authors and determines the kind of the relationship (like *collected*, *revised*).
- blue = another special table. The only table marked blue is Habitat. The Occurrence-Habitat relationship is *usually* one-to-one. There is only one case the Habitat is shared by several records - it is after the Occurrence is added via the Add dialog.

  The relationship should remain one-to-one forever.

The most important functions of **Record**:

- 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.
- void setValue(String column, Object value)

  Set a value of the specified column. This can simplify modifications of the record.

Parser (net.sf.plantlore.client.imports.Parser)

An interface. **Parser** is (surprisingly) the opposite of a **Builder**. While the Builder obtained some occurence data and created the output, the Parser is supposed to ***read the occurrence data from the file and return the whole record*** (as a tree of linked holder objects).

The Parser is a little bit tricky, though. Let's have look a its methods:

- void hasNextRecord()

  Returns true if there is Occurrence data in the file that have not been processed yet.

- Action fetchNextRecord()

  Fetches the next occurrence data from the file. This means: ***load the whole occurrence record from the file, create the holder objects, and fill them with values obtained from the file***.

  This method doesn't return the Occurrence data - you shall ask for it in the `nextPart()` method. This method ***returns the intended action*** the record shall undergo. There are several actions the Parser can suggest:

  - UNKNOWN - take the default action (usually Insert),
  - INSERT - insert the record into the database,
  - UPDATE - update the record in the database,
  - DELETE - delete the record from the database.

  It is the DefaultDirector that may use this information.

  So, fetchNextRecord parses and creates the whole record, and stores it in its cache. The DefaultDirector asks the Parser for certain part of the record later.

- Record nextPart(Class table)

  Gets a ***part of the whole record corresponing to the requested table***. The `fetchRecord()` method loaded another whole record from the database. To obtain parts of the record call this method, for instance:

```
(0)  parser.fetchNextRecord()
(1)  Occurrence occ = parser.nextPart(Occurrence.class)
(2)  while( parser.hasNextPart( AuthorOccurrence.class ) ) do
(3)    AuthorOccurrence ao = parser.nextPart( Au....class )
```

  (1) shall return the "core" of the record - the occurrence. The ***record should contain all its subrecords*** (foreign keys) already, ie. *Occurrence* should contain its *Habitat*, *Metadata*, *Publication* and *Plant*.

  (2) asks the parser whether there are some AuthorOccurrences associated with the Occurrence record and (3) obtains them from the Parser.

  If there is no such part of the table, **null** shall be returned.

Let's see how Parser is supposed to work on a simple example. Consider this file:

```
<Record>
  <Occurrence yearCollected="2002"  createdWhen="1.9.2002" ...>
     <Habitat country="CZE" ...>
       <Village name="Trebic"/>
       ...
     </Habitat>
     <Plant taxon="retra temperea" ... />
     ...
  </Occurrence>

  <AuthorOccurrence role="collect" />
    <Author wholeName="EK" ... />
  </AuthorOccurrence>

  <AuthorOccurrence role="revise" />
    <Author wholeName="LO" ... />
  </AuthorOccurrence>
</Record>
```

That file contains one record (the list has been shortened on purpose). The whole record is supposed to be parsed and stored into holder objects, when the `fetchNextRecord()` method is called.

```
Plant p = parser.nextPart( Plant.class )
```

The `nextPart()` method return the specified part of the record - the Plant in this case.

The whole mechanism has to be this complicated because of the one-to-many relationship of Occurrences and AuthorOccurrences. As you can see from the example above, there may be several Authors related to a Occurence record (via the AuthorOccurrences table), which is why the `nextPart(AuthorOccurrence.class)` has to be called repeatedly to obtain them all.

There's yet another method:

- Action intendedFor()

  Returns the operation the last record (obtained via the `nextPart()` method) is supposed to undergo. This method is meant for a more sophisticated work with some tables. If you are unsure what your implementation shall return, UNKNOWN will be the best choice.

As you can see, the Parser is a class where the versatility is not lost yet. The ConcreteParser can load complex occurrence records as well as records belonging to a single table.

This interface is meant to work for a simple import too. Simple import works only with one table (*Plants* for example) and is described later.

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

The **DefaultDirector** uses only a few other things. Its purpose is to ***fetch records from the file*** (using the **Parser**) and ***store them in the database***. It is clear what a DefaultDirector will require:

- The **database layer** to work with the database.
- The **Parser** to obtain parts of the occurrence data.
- The curretly logged **User** to ensure the integrity of the database.

Here's how the DefaultDirector works (the basic idea):

```
DBLayer db, Parser parser, User user

run()
  while parser.hasNextRecord() do
    intention = parser.fetchNextRecord()
    occurrence = parser.nextPart( Occurrence.class )

    process( occurrence, intention )

    while parser.hasNextPart( AuthorOccurrence.class ) do
      ao = parser.nextPart( AuthorOccurrence.class )
      intention = parser.intendedFor()

      process( ao, intention )
    done
  done
```

The algorithm above should give you a clear idea what DefaultDirector does. If you wish to write your own ConcreteParser, the algorithm above should suffice.

The whole DefaultDirector is much more complicated. First, there is a validity check. The record must have all its not-null parts set. Second, the imported record may already be in the database and it must be processed with great care. Third, some parts of the record (the immutable tables at least) must be shared and those matching parts must be found in the database. Fourth, modification of some records requires ***User's intervention***.

Let's describe the whole DefaultDirector once again, more thoroughly this time. As you will see, the Director is rather comlicated and some parts of the algorithm shall be discussed later.

```
Parser parser, DBLayer db
```

```
run()
  while parser.hasNextRecord() do
    intention = parser.fetchNextRecord()
    occ = parser.nextPart( Occurrence.class )

    if !occ.areAllNNSet() then continue

    occInDB = db.select( "occurrence with the same unique ID" )

    if occInDB != null && occInDB isNewerThan occ then
      decision = askUser( "The record in the database is newer
                           than the record in the file.
                           Do you want to replace it anyway?" )
      if decision != REPLACE_ANYWAY then continue

    if occInDB == null then
      switch intention
      case DELETE:
        break
      default:
        insert( occ )
    else
      switch intention
      case DELETE:
        delete( occInDB )
        if !isShared( occInDB.getHabitat() ) then
          delete( occInDB.getHabitat() )
        break
      default:
        update( occInDB, occ )

    *Process Authors associated with this Occurrence*

  done
```

The occurrence record obtained from the Parser must past several test before it is processed. The kind of the operation is based on the original **intention** and the state of the corresponding record in the database database. The unique identifier was introduced in order to distinguish the brand new occurrence data from the occurrence records that are already in the database.

There are several methods whose task will be described below.

- Action askUser()

  This method cause the DefaultDirector to ***stop and wait for the User's decision***. The DefaultDirector is supposed to run in a separate thread, which means this thread should wait for the notification from another thread. There is no such method as askUser() in the real code, because the mechanism is a little bit more complicated. See the section describing the threads and User's decisions. That part discusses the matter more thoroughly.

- void delete(Record r)

  Delete the specified record. This method ***lazy-deletes the deletable record***, ie. the record is marked as deleted. This is why the **Deletable** interface is used with some records - it denotes records that can be lazy-deleted.

- void insert(Record r)

  Insert the record into the database. This method is recursive and works like this:

```
Record insert(Record rec)
  if Record.belongsToAnImmutableTable( rec ) then
    counterpart = findMatchInDB( rec )
    if counterpart == null then throw ImportException
    else return counterpart
```

```
  else
    for each foreign key K of the record rec do
      rec.setValue( K, insert( rec.getValue(K) ) )
    done

    if !rec instanceof Habitat then
      counterpart = findMatchInDB( rec )

    if counterpart != null then
      return counterpart
    else
      db.executeInsert( rec )
```

If the record belongs to an immutable table, the corresponding counterpart is found. If no such counterpart exists in the database, it is an error - the User cannot modify an immutable table, ie. cannot insert new record into it.

If the record belongs to another type of table, all potential **subrecords** are inserted in the same manner (recursion). The record is then *"repaired"* using records that have been returned by insert(). If no corresponing counterpart exists, the record is inserted. If a suitable counterpart is already in the database, it is used; with one exception: the *Habitat*. You can interpret the act of finding the counterpart as ***reusing the records in the database***. However, the *Habitat* shall never be shared (due to the one-to-one relatioship) and thus it shall never be reused.

Reusing the records in the database is a very important feature of Import. The most illustrative example is the table of Authors. It is not an immutable table, yet you will most probably want to find an existing Author (counterpart) whenever possible. It would be silly to insert a new Author with the same properties for each record.

- void update(Record original, Record replacement)

Well, update is the most sophisticated method of them all. The task is to **replace** the **original** with values from the **replacement record**. There is one potential problem: if a record is shared by several other records, changing this record will affect the other associated records as well! Here the User's intervention is required once again. Let's describe how update works (when reading this, bear in mind that the **original record is a part of the record that is already in the database**)

```
Record update(Record original, Record replacement)
A if Record.belongsToAnImmutableTable( original ) then
    if doPropertiesMatch( original, replacement ) then
      return original
    counterpart = findMatchInDB( replacemet )
    if counterpart == null then throw ImportException
    else return counterpart

B else if original doesn't have foreign keys then
    if doPropertiesMatch( original, replacement ) then
      return original
    counterpart = findMatchInDB( replacement )
    if counterpart != null then
      return counterpart

    decision = UPDATE
    if isShared( original ) then
      decision = askUser( "The record " + original + " is shared
                          by several other records. Changing it
                          may affect them as well? Do you want
                          to make the change for all records or
                          create a new record?" )
    if decision == UPDATE then
      original.replaceWith( replacement )
      db.executeUpdate( original )
```

```
      return original
    else
      db.executeInsert( replacement )
      return replacement
C else
    for each foreign key FK except AuthorOccurrence.Occurrence
             of the record original do
      subOriginal = original.getValue( FK )
      subReplacement = replacement.getValue( FK )
      suggestion = update( subOriginal, subReplacement )

      if suggestion != subOriginal then
        original.setValue( FK, suggestion )
        dirty = true
    done

    if !doPropertiesMatch( original, replacement ) then
      original.replacePropertiesWith( replacement )
      dirty = true

    if !dirty then return original

    if original instanceof Occurrence then
      db.executeUpdate( original )

    else if !isShared( original ) then
      db.executeUpdate( original )

    else if original instanceof Habitat then
      db.executeInsert( original )

    else
      decision = askUser( "The record " + original + " is shared
                          by several other records. Changing it
                          may affect them as well? Do you want
                          to make the change for all records or
                          create a new record?" )
      if decision == UPDATE then
        db.executeUpdate( original )
      else
        db.executeInsert( original )

    return original
```

It is clear that this method is rather complicated. There are three sub-cases (**A**, **B**, and **C**).

Case **A** is quite simple and its behaviour resembles that of the insert().

Case **B** concerns tables like *Authors* or *Publications*. Once again, ***update tries to reuse the existing records whenever possible***. You shall notice that some records may be shared and the User is asked to make his decision about the record.

Case **C** is the most sophisticated one. This one concerns the *Occurrence*, *Habitat* and the *AuthorOccurrence* table. First, every **subrecord** is updated in the same manner. There is an extra feature here: if the original record contains this foreign key: `AuthorOccurrence.Occurrence`, it is skipped. It shall prevent this procedure from updating the Occurrence table, which has been already updated. Second, properties are replaced as well (if needed). Finally, if the record needs to be updated in the table, it is updated. There are several special cases, like: the ***Occurrence*** can be **updated only**, the **shared *Habitat*** will **always be inserted**, in other cases the User must decide.

One part of the whole algorithm is still missing - processing the associated authors. This part is

displayed here:

```
*Process Authors associated with this Occurrence*

  while parser.hasNextPart( AuthorOccurrence.class ) do
    ao = parser.nextPart( Aut....class )
    intention = (intention == DELETE) ?
                    DELETE : parser.intendedFor()

    if ao.getAuthor() == null then continue
    ao.setOccurrence( occInDB )

    counterpart = findMatchInDB( ao.getAuthor() )
    if counterpart == null && intention == DELETE then
      continue
    if counterpart == null then
      authorInDB = db.executeInsert( ao.getAuthor() )
    ao.setAuthor( authorInDB )

    counterpart = findMatchInDB( ao )

    if counterpart == null then
      switch intention
      case DELETE:
        break
      default:
        db.executeInsert( ao )
    else
      switch intention
      case DELETE:
        delete( counterpart )
        break
      default:
        break
  done
```

The last part is rather sophisticated. Let's review, what it does. Every AuthorOccurrence must be processed. Since the occurrence record already is in the database, we shall use it in the AuthorOccurrence. What's with the intention: if the *"master plan"* was to delete the whole occurrence record, the AuthorOccurrences will be deleted as well, regardless what's intented with them. If the Author is not in the database and the intention was to delete the AuthorOccurrence, we don't have to do a thing, because if the author is not in the database, the AuthorOccurrence refering to this Author cannot be in the database either.

Otherwise, the author must be inserted into the database. The AuthorOccurrence is a subject of another record reuse.

DefaultDirector - Threads and Waiting for the User's Decision

The DefaultDirector is supposed to run in a separate thread. Sometimes the execution must be suspended, because the User's intervention is required. The section about the **DefaultDirector** shall give you an idea why such decisions are needed. The DefaultDirector contains two methods, that allow the process to be stopped when the decision is needed and restarted after the decision has been made:

- Action expectDecision(Record about)

  The code will help to explain - we will have to be more Java-based this time:

```
class ObserverNotifier extends Thread {
  void run() { setChanged(); notifyObservers(problematicRecord) }
}

ObserverNotifier observerNotifier = new ObserverNotifier();
```

```
synchronized Action expectDecision(Record about) {
  lastDecision = Action.UNKNOWN;
  problematicRecord = about;
  observerNotifier.start();  // notify observers
  while( lastDecision == Action.UNKNOWN )
      ►
     try { wait(); } catch (InterruptedException e) {}
  return lastDecision;
}
```

- void makeDecision(Action decision)

```
synchronized void makeDecision(Action decision) {
  lastDecision = decision;
  notify();
}
```

Both codes must be discussed simultaniously. The most important parts are designated red.

The `run()` method of the **DefaultDirector** is executed in a separate thread. This thread may call `expectDecision()`. This method is the substitute for the `askUser()` in the algorithm above. There are two things worth noticing:

1. That method `expectDecision()` **must be synchronized**.
2. and `observerNotifier()` **must run in its own thread**.

Why must `expectDecision()` be synchronized? So as to ensure, that the decision doesn't come before we can notice. If `expectDecision()` was interrupted at ► and the decision was made and notify() called, we would **wait** for **notification** that would never come. Which is why this method must be synchronised - to assure we will go to sleep (wait) **before** the notification arrives.

Why must `observerNotifier()` run in its own thread? In order to avoid the deadlock. Suppose we call `notifyObservers()` from `expectDecision()`, that is synchronized. `notifyObservers()` simply calls `update()` of all registered Observers. If the update method wants to call `makeDecision()`, **which must also be synchronized, so as not to send notification before the expectDecision goes to sleep**, it would be impossible, because the lock is already owned by `expectDecision()`.

It may look like an unnecessary precaution, but this way we can ensure that both methods will be transparent for their callers.

Dealing with Exceptions during Import

Dealing with exceptions during Import **follows the same pattern** like Export. Please refer to that section to get the idea how exceptions are handled.

Other Issues

Other issues that are yet to be resolved:

- transactions - if the import of some part of the record fails, the whole record should be removed from the database,
- user rights - some records cannot be modified by user's that did not create them; although the database layer does the necessary check (because of security reasons), the import should do it as well in order to prevent possible traffic (if left unchecked)

Import from Outside

In this section we will concentrate on the usage of the whole framework to get our data

into the database. This section heavily resembles the Export from Outside section as both ImportMng and ExportMng share the same principles.

ImportMng (net.sf.plantlore.client.imports.ImportMng)

The ImportMng is responsible for ***creating all participants, starting the Import in the separate thread, performing the final cleanup, and providing the interface allowing the User to resolve conflicts and abort the import***.

There are few things the ImportMng requires:

- The **database layer** that will mediate the connection with the database.
- The **User**. The User that is currently logged in. It will be used to determine the access rights to processed records.
- The **Name of the file** where the data is stored.

To start the ImportMng, call the start() method, which works like this:

```
DBLayer db, User user, String filename

start()
  file = new File( filename )
  reader = new FileReader( file )

  switch ??
   "XML": parser = new XMLParser( reader )
   ...

  parser.initialize()

  director = new DefaultDirector( db, parser, user )

  T = new thread executing director.run()

  Cleanup = new thread executing {
    T.join()
    reader.close()
    parser.cleanup()
  }
```

The ImportMng creates the **DefaultDirector** and appropriate **ConcreteParser**. The import is started in a separate thread ***T***. The thread called ***Cleanup*** waits for thread ***T*** to end and after it is finishes, it performs the final cleanup.

There is a method that supports the **decision making** (User's intervention):

- void makeDecision(Action decision)

  Delegates the decision to the DefaultDirector.

- void setAskAboutTime(boolean arg)

  Set true if the DefaultDirector shall remember the **last decision** about replacing of records in the database that are newer than those from the file.

- void setAskAboutInsert(boolean arg)

  Set true if the DefaultDirector shall remember the last decision about replacing a shared record with a brand new one.

In order to **monitor** the process and **abort** it, use the following methods: `isAborted()`, `abort()`, `isImportInProgress()`, `getNumberOfImported()`, `getNumberOfRejected()`.

Problematic Records

So that the User can decide, which record should be kept in the database or whether a shared

record shall be updated to all records that share it, he must be presented with the whole record. The problematic record(s) can be obtained from the DefaultDirector, but the ImportMng provides a better solution: a transformation of that record(s) into a table model. The Observer can simply ask the ImportMng to create the appropriate TableModel and simply display it. Here is the interface:

- TableModel getProcessedRecords()

  Return the table model containing **all** properties (including properties of their foreign keys) of both records - the record from the database and the record from the file - so that the User can see the difference between them.

- TableModel getProblematicRecord()

  Return the table model containg **all** properties (including properties of its foreign keys) of the shared record so that the User can decide, whether he wants to replace it or modify for everyone.

Aborting Import

Aborting is accomplished via the same mechanism as in Export. Refer to the **Aborting Export** for further details.

Import from the User's View

---similar, the only new thing will be the DecisionView.java--- LATER