PLANTLORE

Developer's Manual

Charles University
Faculty of Mathematics and Physics
Prague, Czech Republic

Supervisor: RNDr. Antonín Říha, CSc.

Lada Oberreiterová ladaob@seznam.cz

Jakub Kotowski fraktalek@gmx.net

Tomáš Kovařík tkovarik@gmail.com

Erik Kratochvíl

Table of Contents

Installation

Preface

This document describes installation of different parts of Plantlore system. For more information about Plantlore see Plantlore User manual or visit http://plantlore.berlios.de Some parts of this document concern only selected operating system environment. These sections are marked with appropriate OS name. Sections marked „Advanced“ are intended for advanced users such as system and database administrators.

Requirements

Since Plantlore is written in Java programming language it should be possible to run it on any system with Java Virtual Machine (given the hardware requirements are satisfied). This is not completely possible since it uses a PostgreSQL database management system as a data storage and therefore is restricted to systems running PostgreSQL version 8.1 or higher (see PostgreSQL documentation for the list of such systems)

Plantlore was successfully tested on Windows platform (Windows 2000 or higher), Linux and Mac OS X 10.4
Recommended hardware requirements for PC platform are:

- 1GHz processor
- 256 MB RAM
- 100 MB of disk space (including PostgreSQL database system)

Neccessary software requirements:

- Java Runtime Environment ver. 1.5.0 or higher. On Windows platform installer will detect suitable JRE and if it is not found it will ask you to download and install it.

Installation of Plantlore Client and Server

Installation is done using an installation file that can be obtained from the project homepage. On windows platform, installer is an executable file you can run by doubleclick. On Linux and Mac OS X you have to execute plantlore-installer.jar file either by doubleclick or by executing the following command on the command prompt: „java -jar plantlore-installer.jar“

Automated installer will guide you through the whole installation of Plantlore Client and Plantlore Server. These are the steps you will go through during the installation:

1. Select the language to use during the installation (either english or czech).
2. View basic Plantlore description and accept GNU General Public Licence.
3. Select installation directory. If the directory does not exist, it will be created for you.
4. Choose which components you would like to install. Installer shows description for each of the components.
5. WINDOWS: In case you are running Windows OS and have chosen to install PostgreSQL database system, you will be asked for the details required for PostgreSQL isntallation. See section „Installing PostgreSQL“ below.
6. In this step all the components will be copied to the installation directory
7. WINDOWS: Installer will offer you to create shortcuts in the Start menu and on the

desktop. This feature is only available on Windows platform.

8. Installation is finished successfuly, if you wish you can create automatic installation script (recommended for Advanced users only)

After the installation is finished

Once you successfuly finish installing plantlore (and optionally PostgreSQL database system) you can start Plantlore Client and Plantlore Server (using Start menu shortcuts or executing files from the installation directory manually). In case you want to work on your local machine and you hav PostgerSQL database system installed, you are expected to create new database using Plantlore Client. Further information on how to do this can be found in Plantlore interactive help or Plantlore User manual.

Installation of PostgreSQL database system

In case you want to use Plantlore on your local workstation you will have to install a database system. Currently only PostgreSQL is fully supported. If you will only connect to remote Plantlore servers you don't need to install PostgreSQL system.

Windows platform

On windows platform, PostgreSQL installation is a part of the automated installer. If you choose to install PostgreSQL package, installer will automatically install and confihure PostgreSQL on your system although some additional information will be required. Installer will ask you for these during the installation. You are expected to provide this information:

- Username and password for new OS user – PostgreSQL runs under a special unpriviliged operating system user account to ensure maximum system security. Installer will create this account for you but you have to provide valid login and password to be used. Make sure that the username is unique in your system, otherwise PostgreSQL installation will not be successfull.
- Username and password of the new database user – After PostgreSQL and Plantlore are installed, you will use this username and password to create your first Plantlore database. Make sure that the username contains only alphanumeric characters (A-Z, a-z, 0-9). Otherwise the PostgreSQL installation will fail.
- Port number – Number of the port where the PostgreSQL service will listen for connections. Default port number is preset for you. If you do not know what this means, leave the default value. PostgreSQL is by default set not to accept remote connections (which is OK to run Plantlore) therefore you will not compromise your system security by installing it.
- Create shortcuts for PostgreSQL – Installer can create Start menu shortcuts for PostgreSQL user and administration tools automatically. This option is recommended for advanced users.
- Location where PostgreSQL will be installed – This is a directory where PostgreSQL system will reside. Make sure you have enough space on the target filesystem. IMPORTANT NOTE: The target drive for PostgreSQL installation must use NTFS filesystem in order for PostgreSQL to work properly.

Linux and Mac OS X [ADVANCED]

On these operating systems PostgreSQL installation is not automatic. You will have to install PostgreSQL by yourself. See www.postgresql.org where you can download up to date version of PostgreSQL and read the installation instructions for your platform.

During installation you will have to create your first database user. Remember the login and password for this user since you will need it to create your first Plantlore database using Plantlore client. For further information on how to create new database see Plantlore user manual.

Building Plantlore from sources [ADVANCED]

Plantlore is an open source project released under GNU General public licence and therefore you have the rights and responsibilities granted by this licence agreement. Source files are distributed as a part of the installation program but if you are interested in Plantlore developement we encourage you to get the most up to date version of sources from the SVN repository hosted at http://developer.berlios.de

You can checkout your working copy by executing the following command:

```
$> svn checkout svn://svn.berlios.de/plantlore/trunk/
```

In case you would like SVN access write permissions contact any of the developers listed at the project homepage.

After you check out the latest sources from the repository you can build Plantlore by the following commands (assuming you have Apache Ant ver. 1.6 or higher) from the sources directory:

```
$> ant build
```

This command will build both Plantlore Client and Plantlore Server into the ./dist directory. If you'd like to see what other ant targets are available, run the following command:

```
&> ant -projecthelp
```

Installation of BioCASE provider software

Tasks and the Dispatcher

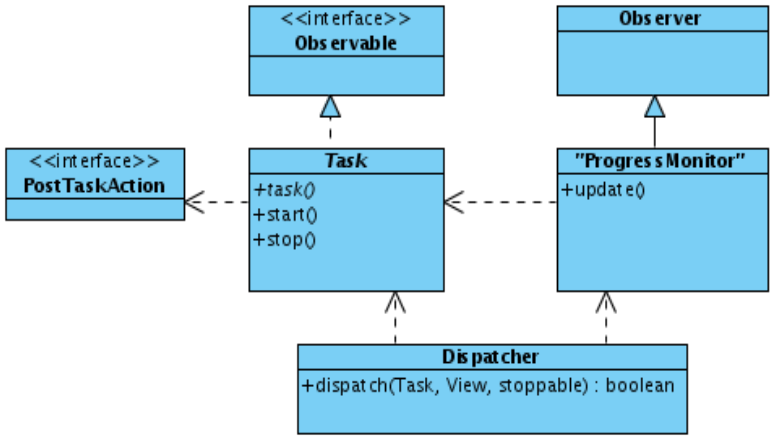
To solve the problem of a freezing Swing GUI when running long tasks Task class was introduced. Task is an abstract class that runs given code in a separate thread. To prevent problems when running multiple threads we introduced the Dispatcher class that actually starts (dispatches) each Task and takes care of Task synchronization.

The goal

Several goals lead us to the Task/Dispatcher solution. We need to present the user with a responsive GUI that not only appears not frozen (does repaint itself) but also keeps the user informed about ongoing computation whenever possible. Other important issue is safety. We need to prevent possible conflicts when running more tasks at the same time.

The solution

The solution has actually three parts. The Task and Dispatcher classes and some progress monitor classs. Task takes care of the computation, dispatcher assures that at any time there is only one Task running when Dispatcher is used and the progress monitor watches the Task's computation and informs the user about progress.



Task

Task builds upon the SwingWorker helper class and upon the Observer design pattern as implemented in standard Java. Task implements the Observable interface. It is the Task's user responsibility to inform the Observers about progress of the computation. The actual new thread creation and return value handling is delegated to the SwingWorker class.

Progress monitor

We call a progress monitor any class that extends the Observer class and watches the Task's computation. The progress monitor's responsibility is to observe the task and to inform the user about the progress apropriately. This is usually done by using some form of Swing's JProgressBar but doesn't have to be. Plantlore implements several progress monitor classes. The mostly used are the ProgressBarManager controlling the JProgressBar located in AppCoreView's status panel and the SimpleProgressBar2. SimpleProgressBar2 is used mainly for very long running Tasks and supports Task cancelation. It is used for example to inform the user about the progress of the export Task. In most other cases the ProgressBarManager is used.

Dispatcher

Dispatcher's responsibility is to make sure that at any time only one Task is running. Other classes have to ask Dispatcher to run the Task for them by calling the Dispatcher.dispatch(Task, View, Stoppable) method. According to the third argument of this method dispatcher chooses which progress monitor to use for that Task. If the task should be stoppable then it uses SimpleProgressBar2 otherwise it uses the ProgressBarManager. When called, the dispatcher checks whether other Task is already running and if yes it just returns, doing nothing. If no other Task is running it installs the proper progress monitor to the Task as an observer and starts the Task.

Incorporating it in the application

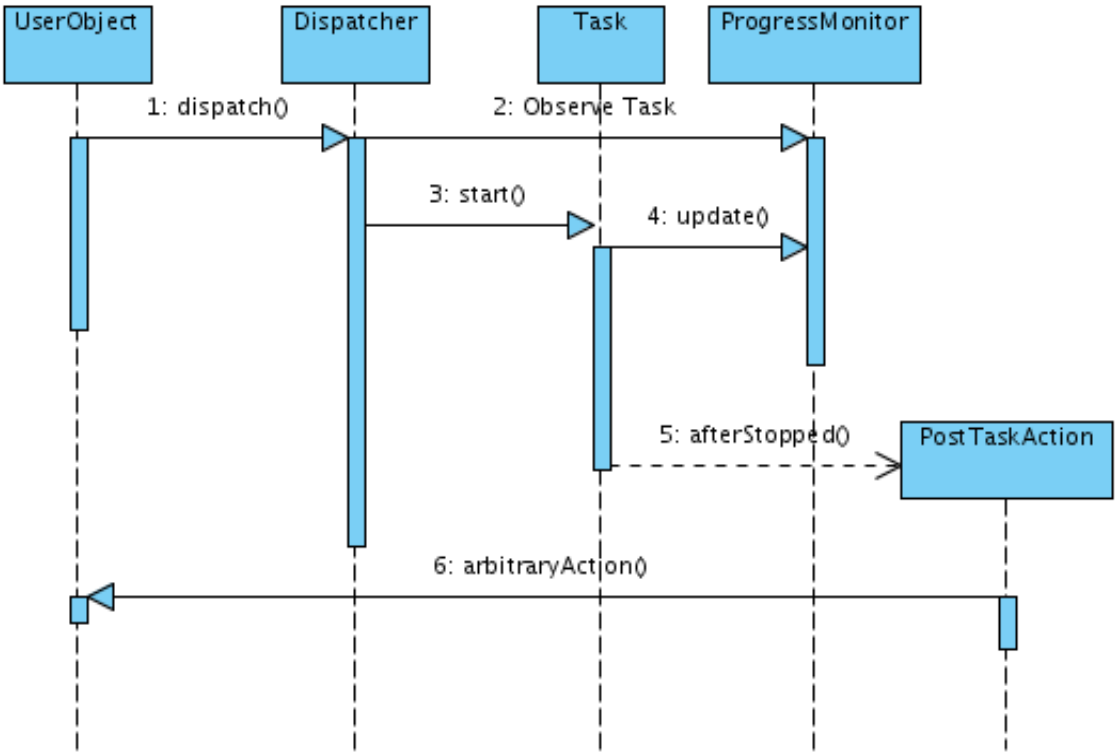
To make sure the Dispatcher is able to fulfill it's responsibility it is made a singleton. The callers have to call Dispatcher.getDispatcher() to get the actual object. The caller also usually needs to be informed about the end of the Task's computation and perform some action then. To this end a PostTaskAction interface has been introduced:

```
interface PostTaskAction {
      public void afterStopped(Object value)
}
```

The Task upon it's end calls the afterStopped() method of a supplied PostTaskAction and as the parameter it uses the Object the Task returned.

Handling errors

The Task's user in Plantlore typically has to use DBLayer which can at any time throw a DBLayerException or a RemoteException. The Task/Dispatcher/progress monitor trio is prepared for that. The Task catches each Exception thrown by the Task's user and supplies it to the Observing progress monitor. The progress monitor is typically running in the Swing's EventDispatchThread and therefore can inform the user about the problem and possibly offer a solution. Progress monitors call the DefaultExceptionHandler.handle(view, ex) to this end.



The Database Model

The database serves the aim of collecting, browsing and administrating data about plant occurrences.

While developing the database model, emphasis was placed on:

- Content completeness and mutual linkage of occurrence data
- Web client and database communication
- Data access security
- Establishment of fundamental rules for table and column names

Content completeness and mutual linkage of occurrence data

The database model was projected to support standards for access and exchange of Fundamentals biological data. Supported standards include ***Darwin Core 2, ABCD 1.20, and ABCD 2.06.***

DarwinCore 2

Darwin Core is a specification of data structure related to extraction and harmonization of fundamental data documenting the occurrence of organisms in time and space and also the occurrence of organisms in biological collections. You can find the documentation for Darwin Core v. 1.4 at http://darwincore.calacademy.org/Documentation/DarwinCore2Draft_v1-4_HTML .

ABCD Schema

ABCD Schema (Access to Biological Collections Data) is a complex, highly structured standard for access and exchange of fundamental biological data. It is compatible with several existing data standards. Among the groups participating on its development are TDWG (Taxonomic Databases Working Group) and CODATA (Committee on Data for Science and Technology). For more detailed information please visit http://bgbm3.bgbm.fu-berlin.de/TDWG/CODATA/default.htm.

ABCD version 1.2 and ABCD version 2.06 have been implemented in project BioCASE. You can find the documentation for both of the versions at http://www.bgbm.org/scripts/ASP/TDWG/frame.asp or at http://www.bgbm.org/biodivinf/Schema/.

Web client and database communication

The Biological Collection Access Service for Europe, BioCASE, is an international network of all kinds of biological collections. BioCASE provides for the spread of an integrated access to a distributed and diverse European database of collections and observations. It utilizes independent open-source software and open data standards and protocols.

The web client in project Plantlore is supported by the BioCASE Provider Software (BPS). The utilization of BPS automatically enables the participation in the international BioCASE project.

The configuration of BPS requires the definition of the database structure and the mapping of the database on various schemes (i.e. it is necessary to define the relationships between the attributes of our database and elements of individual standards). In present case, it was essential to map the database to DarwinCore 2, ABCD Schema 1.2 and ABCD Schema 2.06. Each of the schemes is composed of obligatory elements that must be mapped.

BPS has been configured to access the database server directly. The access to the data has been limited by means of roles and views. Therefore, only a subset of the existing data is accessible from the Internet.

Data Access Security

Client

- Data access has been limited by means of roles and user rights, stated in table tRight. When a user is not listed in table tUser or when the value of cDropWhen is not null, that user will not be permitted to access the data.

Web Client

- Data access has been limited by means of roles and views for the web client.

Fundamental rules for table and column names

- Table names begin with prefix t.
- Column names begin with prefix c.
- View names begin with prefix v.

Description of the database model tables

tOccurrences

This table is the primary table for an occurrence. It includes time data concerning individual occurrences, information about the source and herbarium. The table is linked with tables tHabitats, tPlants, tMetadata, tPublication, tUser and through table tAuthorOccurrences with table tAuthors.

- ***cID*** – primary key of the table (autoIncrement). Obligatory item.
- ***cUnitIdDb*** – unambiguous database identifier. It is used in connection with value cUnitValue during export and subsequent import to determine whether a given record is in the database. Obligatory item.
- ***cUnitValue*** – unique value of a record in a given database. It is used in connection with value cUnitIdDb during export and subsequent import to determine whether a given record is in the database. Obligatory item.
- ***cHabitatId*** – foreign key for table tHabitat where the information about the locality is stored. Obligatory item.
- ***cPlantId*** – foreign key for table tPlants where the information about the taxon is stored. Obligatory item.
- ***cYearCollected*** – year when the occurrence took place. Obligatory item.
- ***cMonthCollected*** – month when the occurrence took place. Obligatory item.
- ***cDayCollected*** – day when the occurrence took place. Obligatory item.
- ***cTimeCollected*** – time when the occurrence took place. Obligatory item.
- ***cIsoDateTimeBegin*** – created by putting together cYearCollected + cDayCollected + cTimeCollected. It was added from the ABCD 1.20 scheme and has been further employed when working with BioCASE.
- ***cDateSource*** – source of the occurrence (field occurrence, literature, or herbarium)
- ***cPublicationsId*** – foreign key to table tPublications with detailed publication information
- ***cHerbarium*** – code determining the herbarium
- ***cCreateWhen*** – date and time of entering the occurrence record into the database. Obligatory item.
- ***cCreateWho*** – foreign key to table tUser containing detailed information about the user who entered the record (occurrence) into the database. This item is important in the process of verification of the rights to access this record. Obligatory item.
- ***cUpdateWhen*** – date and time of the most recent change of the record. Obligatory item.

- ***cUpdateWho*** – foreign key to table tUser with detailed information about the user who carried out the last change of the record. Obligatory item.
- ***cNote*** – note accompanying the occurrence.
- ***cMetadataId*** – foreign key to table tMetadata. In case there exist more than one projects, each record will be linked with its project. Obligatory item.
- ***cDelete*** – contains information about the currency of a given record. A current record (cDelete == 0) is a record that is being displayed to the user. A non-current record (cDelete > 0) is a record deleted by the user and the deletion information about which is listed in the history tables. Obligatory item.
- ***cVersion*** – represents the version of the record. This column uses Hibernate to implement pessimistic column locking in tables for UPDATE operation. Obligatory item.

tAuthorsOccurrences

This table deals with M:N table reference between tables tAuthors and tOccurrences. It assigns the relationship *"author – occurrence"* the information about the property of this relationship (the author is the collector, the author identified the occurrence, the author revised the occurrence).

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cAuthorId*** – foreign key to table tAuthors. Obligatory item.
- ***cOccurrenceId*** – foreign key to table tOccurrences. Obligatory item.
- ***cRole*** – information about the relationship between the author and the occurrence (found, identified, revised).
- ***cNote*** – author's note about the occurrence. It can include e.g. the date and the result of the revision.
- ***cDelete*** – containns information about the currency of a given record. A current record (cDelete == 0) is a record that is being displayed to the user. A non-current record (cDelete > 0) is a record deleted by the user and the deletion information about which is listed in the history tables. Obligatory item.
- ***cVersion*** – represents the version of the record. This column uses Hibernate to implement pessimistic column locking in tables for UPDATE operation. Obligatory item.

tHabitats

This table contains the information about the locality. This information was not included in table tOccurrences as a single locality can include multiple occurrences.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cTerritoryId*** – foreign key to table tTerritories where the list of territories is stored. Obligatory item.
- ***cPhytochoriaId*** – foreign key to table tPhytochoria where the list of codes and names of phytogeografical territories is stored. Obligatory item.
- ***cQuadrant*** – quadrant on the territory division on the map.
- ***cDescription*** – detailed description of the place of the occurrence.
- ***cNearestVillageId*** – foreign key to table tVillages containing a list of villages and cities.

Obligatory item.

- ***cCountry*** – state of the occurrence.
- ***cAltitude*** – altitude of the occurrence.
- ***cLatitude*** – latitude of the occurrence.
- ***cLongitude*** – longitude of the occurrence.
- ***cNote*** – note about the territory.
- ***cCreateWho*** – foreign key to table tUser containing detailed information about the user who entered the record (occurrence) into the database. This item is important in the process of verification of the rights to access this record. Obligatory item.
- ***cDelete*** – containns information about the currency of a given record. A current record (cDelete == 0) is a record that is being displayed to the user. A non-current record (cDelete > 0) is a record deleted by the user and the deletion information about which is listed in the history tables. Obligatory item.
- ***cVersion*** – represents the version of the record. This column uses Hibernate to implement pessimistic column locking in tables for UPDATE operation. Obligatory item.

tVillage

This table contains the list of villages and cities.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cName*** – name of the village, city. Obligatory item.

tTerritories

This table contains the list of the territories.

- ***cId*** - primary key of the table (autoIncrement). Obligatory item.
- ***cName*** – name of the territory. Obligatory item.

tPhytochoria

This table contains the list of codes and names of the phytogeographical territories.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cCode*** – code of the phytogeographical territory. Obligatory item.
- ***cName*** – name of the phytogeographical territory. Obligatory item.

tPlants

This table contains the list of plants. Checklist Survey is employed to acquire the list of plants. Survey is a system for recording of entire biosphere (plants, fungi, animals), interconnected with GIS.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cSurveyTaxId*** = id_tax (the code under which Survey stores the taxon). Obligatory item.
- ***cTaxon*** – cSpecies + cScientificNameAuthor. Obligatory item.

- ***cGenus*** – name of the genus under which the plant is classified (e.g. *Abies*)
- ***cSpecies*** – plant species. Checklist Survey also marks this item as species (see Survey, e.g. *Abies alba*).
- ***cScientificNameAuthor*** – the author describing this. Checklist Survey marks this item as autor_sp.
- ***cCzechName*** – plant name in Czech. Checklist Survey marks this item as czech_sp.
- ***cSynonyms*** – Latin synonym for the plant. Checklist Survey marks this item as t_synonym.
- ***cNote*** – note about the plant (information about the person who added the plant to the checklist, or information about the file).

tPublications

This table contains information about the literature.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cCollectionName*** – name of the collection.
- ***cCollectionYearPublication*** – year when the collection was published.
- ***cJournalName*** – article name. Obligatory item.
- ***cJournalAuthorName*** – article author.
- ***cReferenceCitation*** – cCollectionName + cCollectionYearPublication + cJournalName + cJournalAuthorName
- ***cReferenceDetail*** – detailed information about the location (e.g. page number, table number).
- ***cUrl*** – web document.
- ***cNote*** – note about the publication.
- ***cCreateWho*** – foreign key to table tUser containing detailed information about the user who entered the record (occurrence) into the database. This item is important in the process of verification of the rights to access this record. Obligatory item.
- ***cDelete*** – containns information about the currency of a given record. A current record (cDelete == 0) is a record that is being displayed to the user. A non-current record (cDelete > 0) is a record deleted by the user and the deletion information about which is listed in the history tables. Obligatory item.
- ***cVersion*** – represents the version of the record. This column uses Hibernate to implement pessimistic column locking in tables for UPDATE operation. Obligatory item.

tAuthors

This table contains the list of authors of a given occurrence. An occurrence author is every person who found, identified or revised a plant.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cWholeName*** – name and surname of the occurrence author. Added due to further use in BioCASE. Obligatory item.
- ***cOrganization*** – organization whose member the author is.

- ***cPhoneNumber*** – occurrence author's phone number.
- ***cRole*** – information about the occurrence author's occupation (botanist or not, etc.).
- ***cAddress*** – occurrence author's contact address.
- ***cEmail*** – occurrence author's contact e-mail.
- ***cURL*** – occurrence author's homepage.
- ***cNote*** – note about the occurrence author.
- ***cCreateWho*** – foreign key to table tUser containing detailed information about the user who entered the record (occurrence) into the database. This item is important in the process of verification of the rights to access this record. Obligatory item.
- ***cDelete*** – containns information about the currency of a given record. A current record (cDelete == 0) is a record that is being displayed to the user. A non-current record (cDelete > 0) is a record deleted by the user and the deletion information about which is listed in the history tables. Obligatory item.
- ***cVersion*** – represents the version of the record. This column uses Hibernate to implement pessimistic column locking in tables for UPDATE operation. Obligatory item.

tMetadata

This table contains metadata necessary for work with standards Darwin Core 2, ABCS 1.20 and ABCD 2.06.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cTechnicalContactName*** – name of the person responsible for the technical part of the application. (Obligatory item of ABCD 2.06.) Obligatory item.
- ***cTechnicalContactEmail*** – contact e-mail of the person responsible for the technical part of the application.
- ***cTechnicalContactAddress*** – contact address of the person responsible for the technical part of the application.
- ***cContentContactName*** – name of the person responsible for the content part of the application. Obligatory item.
- ***cContentContactEmail*** – contact e-mail of the person responsible for the content part of the application.
- ***cContentContactAddress*** – contact address of the person responsible for the content part of the application.
- ***cDataSetTitle*** – short name of the botanical project. Obligatory item.
- ***cDataSetDetails*** – detailed description of the botanical project.
- ***cSourceInstitutionId*** – unequivocal identifier of the institution that owns the original data source (institution that carried out and recorded the occurrences). Obligatory item.
- ***cSourceId*** – name or code of the data source. Obligatory item.
- ***cOwnerOrganizationAbbrev*** –abbreviation of the organization.
- ***cDateCreate*** – date and time of project creation. Obligatory item.
- ***cDateModified*** – date and time of the most recent modification, entering, or deletion of

records in the project. Obligatory item.

- ***cRecordBasis*** – information about what the given records describe (preserved specimen, …)
- ***cBiotopeText*** – information about the biotope.
- ***cDelete*** – containns information about the currency of a given record. A current record (cDelete == 0) is a record that is being displayed to the user. A non-current record (cDelete > 0) is a record deleted by the user and the deletion information about which is listed in the history tables. Obligatory item.
- ***cVersion*** – represents the version of the record. This column uses Hibernate to implement pessimistic column locking in tables for UPDATE operation. Obligatory item.

tHistoryColumn

This table contains the pair table name and column name. Since there is a fixed number of tables and their columns in a database, there will be a maximum of x lines in this table, where x = sum of the columns in tables tOccurences, tAuthorsOccurrences, tMetadata, tHabitats, tVillages, tPhytochoria, tTerritories, tPublications, tMetadata, + 9 (number of tables for which history is being kept)

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cTableName*** – name of the table. Obligatory item.
- ***cColumnName*** – name of the column. If insert or delete operations were carried out, its value is "null".

tHistoryChange

This table contains information about occurrence modifications. Because there might be more changes at the same time and by the same user for a single record in the table, this table does not contain information about the old and new values.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cRecordId*** – unequivocal identifier of the record in the table where it was entered, edited, or deleted. Obligatory item.
- ***cOperation*** – information about the operation carried out – insert (1), edit (2), delete (3). Obligatory item.
- ***cWhen*** – date and time when the record was entered, edited, or deleted. Obligatory item.
- ***cWho*** – foreign key to table tUser where information about the user who entered, edited or deleted the record is stored. Obligatory item.
- ***cVersion*** – represents the version of the record. This column uses Hibernate to implement pessimistic column locking in tables for UPDATE operation. Obligatory item.

tHistory

This table contains the information about the old value and new value of the modified item.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cColumnId*** – foreign key to table tHistoryColumn. Obligatory item.

- ***cChangedId*** – foreign key to table tHistoryChange. Obligatory item.
- ***cOldValue*** – the value of the item prior to editing is stored here (it is only stored in case of editing). If it is an item containing a foreign key, the value of the object to which the foreign key points will be stored here.
- ***cNewValue*** - the value of the item after editing is stored here (it is only stored in case of editing). If it is an item containing a foreign key, the value of the object to which the foreign key points will be stored here.
- ***cOldRecordId*** – if, as a result of editing, the item containing a foreign key was modified, the previous value of this item (foreign key) will be stored here.
- ***cVersion*** – represents the version of the record. This column uses Hibernate to implement pessimistic column locking in tables for UPDATE operation. Obligatory item.

tUser

This table contains the information about users who either have or previously had access to the database.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cLogin*** – login for database user authentication, unique value. Obligatory item.
- ***cFirstName*** – first name of the user.
- ***cSurname*** – surname of the user.
- ***cWholeName*** – whole name of the user (name + surname). Utilized in work with BioCASE.
- ***cEmail*** – user's contact e-mail, to enable the administrator to contact the user.
- ***cAddress*** – user's contact address, to enable the administrator to contact the user.
- ***cCreateWhen*** – user's registration information. Obligatory item.
- ***cDropWhen*** – information about when the user's registration was cancelled. If the user is still registered, the value will be "null". After the registration is cancelled, the record is kept in the database for possible future retrieval of user's information.
- ***cRightId*** – foreign key to table tRight, containing information about user rights. Obligatory item.
- ***cNote*** – note about the user.
- ***cVersion*** – represents the version of the record. This column uses Hibernate to implement pessimistic column locking in tables for UPDATE operation. Obligatory item.

tRight

This table contains the information about the individual user rights.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cAdministrator*** – if the value is 1, the user has all the rights including the right to edit user accounts and create the database.
- ***cEditAll*** – if the value is 1, the user may edit all of the records regardless of who created them.
- ***cEditGroup*** – this string would contain a list of users who enabled the user to edit their

records… (consequently, if the user does not have the right to edit all the records, s/he can edit own records and records of the users from this list).

- ***cAdd*** – this item gives the right to enter occurrence data into the database.
- ***cVersion*** – represents the version of the record. This column uses Hibernate to implement pessimistic column locking in tables for UPDATE operation. Obligatory item.

tUnitIdDatabase

This table contains a unique identifier of the database. The identifier's function is to make it possible to trace which database the occurrence data originate from. The identifier is used in connection with the value of cUnitValue from table tOccurrences during export and subsequent import to determine whether a given record already is in the database.

- ***cId*** – primary key of the table (autoIncrement). Obligatory item.
- ***cUnitIdDb*** – unequivocal identifier of the database. Obligatory item.

Database views

It was necessary to create database views for communication between the web client and the database. These views guarantee the correctness of the occurrence data mapping onto standards, reading and subsequent displaying of the current data only. The following database views have been created:

- ***vOccurrences*** – the view contains only the current occurrences from table tOccurrences
- ***vHabitat*** – the view contains only the current occurrences from table tHabitat
- ***vMetadata*** – the view contains only the current occurrences from table tMetadata
- ***vPublications*** – the view contains only the current occurrences from table tPublications
- ***vAuthorCollected*** – the view links the occurrences with the authors that collected them
- ***vAuthorIdentified*** – the view links the occurrences with the authors that identified them
- ***vAuthorRevised*** – the view links the occurrences with the authors that revised them

A *current record* is a record that fulfills the condition (cDelete == 0). It is a record which is being displayed to the user.

A *non-current record* is a record that fulfills the condition (cDelete > 0). It is a record which was deleted by the user and the deletion information about which is listed in the history tables.

Database roles

Three main roles have been created in the database. Every user created in project Plantlore shall be assigned one of these roles:

- ***Plantlore_Role_Admin*** – the user with this role shall have all the rights for work with the database, for creating, editing and deleting of users, and for creation of databases. This role

is designated for the user with administrator rights.

- ***Plantlore_Role_User*** – the user shall have limited rights to access the database. The limitations apply for tables tUser, tRight, tHistoryColumn, which can only be read by this user. The user does not have the right to create new users or databases. This role is designated for a user without administrator rights.
- ***Plantlore_Role_www*** – the user shall have limited rights to access the database. The user may perform only operation select over views vOccurrence, vHabitat, vMetadata, vPublication, vAuthorCollected, vAuthorIdentified, vAuthorRevised and over tables tPlants, tPhytochoria, tTerritories, tVillages a tUser. This role is designated for a web client.

The Database Layer

This document describes the role of database layer in Plantlore and how this database layer is implemented and should be used in Plantlore. The purpose of this document is to provide detailed explanation of the implementation of different aspects of database management in Plantlore. Basic knowledge of the Hibernate ORM system is expected. Refer to hibernate documentation at www.hibernate.org if necessary.

Detailed description of classes and methods is not included in this document. Each section lists classes and methods you may want to consult should it be necessary.

Terminology

- *DBLayer* – is used for the database layer as a whole (includes several classes listed below)
- *Client* – In this document client is a user of DBLayer - classes using DBLayer interface.
- *User* – is a person using Plantlore
- *Plantlore user* – entity in Plantlore identified by a username with the given privileges
- *Database user* – User of the underlying database management system.

References

This document uses information included in different parts of the documentation. Refer to the following documents if necessary:

- Database model description
- Communication layer (RMI) and server documentation
- JavaDoc documentation for Plantlore sources

Implementation and source files

Database layer in Plantlore is implemented in several classes in the source tree.

- `net.sf.plantlore.middleware.DBLayer` – Interface defining API for working with database.
- `net.sf.plantlore.server.HibernateDBLayer` – Implementation of the DBLayer interface using Hibernate. Currently only this implementation is provided, however different implementations (e.g. for direct database access using JDBC) should be possible.
- `net.sf.plantlore.middleware.SelectQuery` – Interface defining API for buidling SELECT queries.
- `net.sf.plantlore.server.SelectQueryImplementation` – Implementation of SelectQuery interface using Hibernate criteria queries..
- `net.sf.plantlore.server.SubQueryImplementation` – Implementation of SelectQuery interface for executing subqueries (subselects). Reasons for this implementation are explained later in this document.

Role of the DBLayer

Separate database layer over the Hibernate ORM API and JDBC API is necessary for different reasons. Although Hibernate offers very simple and portable way to access data in relational database system, it is not suitable in some situations.

- *Remote Method Invocation* – In order to use Hibernate over the RMI we would have to use its classes and interfaces as remote objects which is not reasonable since we do not have complete control over the implementation. By using DBLayer which covers the Hibernate we can minimize the number of remote objects. used
- *Additional data processing before persisting* – In order to implement all the features we need to modify data before they are stored to persistent storage. For security reasons this cannot be done by the clients

This approach has several drawbacks as well. Since DBLayer has to cover Hibernate completely and offer clients a lot of different features we are copying a lot of Hiberanate API. Another problem is that Hibernate doesn't handle the fact that we are using RMI very well.

Initializing database connection

Initialization of database connection occurrs in the **initialize()** method. Connection parameters are partially loaded from the *hibernate.cfg.xml* resource file (database mapping documents) and partialy created dynamically from the supplied data (database to connect to, login and password). Initialization includes database user authentication and plantlore user authentication (against data in the *tUser* table).

In the initialization phase Hibernate *SessionFactory* is created which is then used for opening database sessions for executing SQL statements. Existence of this SessionFactory object determines whether the DBLayer is connected to a database and every method in the database layer working with database has to check the existence of SessionFactory before executing anything. Information about the logged user are stored as well and are used to check user privileges when executing database operations.

Classes and methods involved: HibernateDBLayer.initialize()

Transaction management support

Hibernate ORM makes the JDBC transaction support available via its interface which is sufficient for the short database transactions and clients do not need to care about this. In case we need to execute multiple SQL queries (such as importing multiple records to the database), we have to provide interface to open transaction, commit and rollback it so that clients have control over the transaction. This is also neccesary since we work remotely using RMI mechanism and therefore clients only have access to methods exported by DBLayer.

When the transaction is started clients can use special methods for executing SQL queries using this transaction. Only one such long running transaction is allowed at a time.

Application transactions and pessimistic row level locking

In many cases we have to deal with application transactions often including user input which might be problematic from the database point of view. One of the approaches known as optimistic locking is to lock database rows when they are requested for updating and unlocking them after the update. Firstly, this requires a lot of additional work by the database system and brings number of

problems. Secondly, the support for this on the database level is not universal (not all database systems implement SELECT FOR UPDATE statement) and Hibernate support is even worse.

Therefore plantlore uses pessimistic locking strategy, which on the other hand is offered and implemented by Hibernate and is very easy to use. This is implemented by record versioning where each record has its version stored in the database and updated on every UPDATE operation. When two concurrent updates occurr, it is easy to determine (using the version numbers) that another transaction already updated the data and that the second transacion has to rollback.

The downside of the pessimistic approach is that updates made in the second transaction are lost. Since we don't expect Plantlore to run in highlyconcurrent environment, using this method is feasible.

Classes and methods involved:

- beginTransaction(), commitTransaction(), rollbackTransaction()
- executeInsertInTransaction(), executeUpdateInTransaction(), executeDeleteInTransaction()
- createUser(), dropUser(), alterUser(),

All of these methods can be found in *HibernateDBLayer* class

Executing INSERT/UPDATE/DELETE statements

When executing database updates most of the work is done by Hibernate. DBLayer receives holder objects (which are object representation of database rows) with data to be inserted/updated and provides them to Hibernate layer for the actual operation. The role of the DBLayer in this case is to implement operations that cannot be done on the client side. These operations include:

- Modifying received data prior to persisting them
- Checking user priviliges
- Saving history of database modifications

Modifying data prior to persisting

In order to keep the database in a consistent state and protect it from unauthorized or malicios changes, DBLayer modifies received records. The following modifications are performed:

- *Storing the author of the record* – to be able to track the ownership of the records and
- *Storing the time of modifications* - so that we can order the changes and allow the "undo to specific date" operation for example.
- *Storing unique identifiers of records* – This modification is in place so that we prevent data corruption when records are replicated and merged together.

Checking user privileges

Although user privileges should be verified before calling DBLayer, in order to rule out any unwanted modifications of the database (by some modifying the client code) the privileges must be also verified on the level of DBLayer. Exact rules for table access are explained in the User documentation provided with Plantlore and their exact specification is not important for us.

Saving history of database modifications

Since the goal of Plantlore is to offer best possible data mangement to the user we implement saving history of database modifications. This allows users to undo unwanted changes to data and keep track of intentional changes.

From the nature of this feature, it is reasonable to implement it on a DBLayer level so that we have complete control over the database. Detailed rules for saving history records are explained in a separate document included in Plantlore documentation.

Only selected modifications for a selected types of records are saved to history. Therefore DBLayer also providesmethods which do not modify the history. Refer to JavaDoc documentation for details.

Deleting records

When client wishes to delete selected record from the database the delete itself is not executed (there are exceptions described later). Since we want to store history of the modifications, records are only marked as deleted (by setting "deleted" parameter of the record) and are not made available to the user. In case we want to undo some of the changes it is just a matter of reviving selected records.

The only situation when we have to physically delete records from the database is when the user requests to clean the database and remove all records marked as deleted .

Classes and methods involved:

- executeInsert(), executeUpdate(), executeDelete()
- executeInsertInHistory(), executeUpdateInHistory(), executeDeleteInHistory()
- executeInsertInTransaction(), executeUpdateInTransaction(), executeDeleteInTransaction()
- checkRights(), saveHistory(), completeRecord()
- conditionalDelete()

All of these methods can be found in *HibernateDBLayer* class

Executing SELECT queries

Selecting data from the database is a little bit more complicated. In order to be able to construct general SELECT queries without writing the SQL or HQL queries themselves, we decided to use so called criteria queries offered by Hibernate. We have created our own interface for constructing and executing these queries so that clients are completely shielded from the underlying Hibernate API. Therefore we suggest to consult Hibernate documentation concerning criteria queries since DBLayer API is very similar. However, since SELECT queries are the most common operation, the following paragraphs describe the creation and execution of SELECT query in more detail.

Constructing a query

Construction of a SELECT query is started by calling **createQuery()** method defined by the DBLayer interface. An argument of this method is a Class object representing one of the holder objects. This holder object is a primary record we want to select from the database. Of course records of other types can be associated and joined into the query.

createQuery() method returns an instance of *SelectQuery* object which is later used for query construction and finally for the execution of the query.

Implementing projections

Projections allow us to select columns we want to have in the result after query execution. When no projections are set, all the columns of the selected table(s) are returned.

Projections can be added to the query by means of **addProjection()** method defined in the *SelectQuery* interface. Parameters of this method are type of projection and name of the projected column. Names of the columns are available as constants stored in the respective holder objects.

Restrictions

Restrictions give us a way how to limit the selection of records to those satisfying certain constraints. In SQL, restrictions are expressed as the WHERE clause of the SELECT query.

Restrictions can be added to a query using **addRestriction()** method defined in the *SelectQuery* interface. This method takes up to 5 parameters:

1. *Type of restriction* – defines which SQL operator should be used. Available types are defined in `net.sf.plantlore.common.PlantloreConstants`. Please consult this file for more information.
2. *First property name* – name of the column used for operators which accept one or two columns as arguments.
3. *Second property name* – name of the column used for operators which accept two columns as arguments. If the restriction works with only one column, this parameter can be null.
4. *Value* – Any value we want to use as an argument to the selected operator.
5. *Values* – Certain operators (e.g. operator IN) work with collection of values. For these operators, tha last argument is a collection of values we want to use with the given operator. If you use operator which doesn't accept collection of values, this parameter can be null.

In case no restrictions are defined for a query, all the records from the selected table are returned.

Aliases and joining

In order to select data from more tables, SelectQuery API makes it possible to define aliases. Alias is just another name for the associated table which can be used to access columns of this table. To assign an alias, use **createAlias()** method. The first parameter of this method is the column of the main table which associates it with another table. The second is the name of the alias, which can be any reasonable string (such that it can be used as an alias in the SQL query – e.g. cannot contain spaces). When the alias is defined, we can access the fields of associated table using the dot notation (<alias_name>.<field_name>).

```
/*************************************************************
 * We want to select a history record for a given occurrence.    *
 * Table tHistory is associated with table tHistoryChange and    *
 * this table is associated with tOccurrences table (via foreign *
 * keys). The primary table we want to select from is tHistory,  *
 * we need to create aliases for tables tHistoryChange and       *
 * tOccurences.                                                  *
 *************************************************************/
// Select data from tHistory table
query = database.createQuery(HistoryRecord.class);
```

```
// Create aliases for other tables. See how the dot notation is used
query.createAlias("historyChange", "hc");
query.createAlias("hc.occurrence", "occ");
// Add restriction to CUNITVALUE column of tOccurence table
query.addRestriction(PlantloreConstants.RESTR_EQ,
                             "occ.unitValue", null, "unitValue", null);
```

Ordering the results

Ordering results is possible using addOrder() method defined in the *SelectQuery* interface. **addOrder()** method takes two arguments, the direction of ordering and the column to be used for ordering the results. Direction can be either ascending or descending.

Running a query

When the query is constructed, we can proceed to the execution. Query is executed by calling **executeQuery()** method defined in the DBLayer interface which takes an instance of SelectQuery we want to execute as an argument.

executeQuery() method returns unique id of result which is used for reading the results. This is necessary because more query results can be opened at the same time.

Working with the query results

Executing query doesn't fetch any data from the database. To get the selected data, either **next()** or **more()** method has to be used. These methods are defined in the DBLayer interface.

Method **next()** fetches next record from the result of the query. The only parameter of this method is the id of the result we want to read. **next()** returns an array of records read from the database. Although we are selecting a single row (record), very often associated records are fetched. For example, when selecting User record from *tUser* table, associated Right record (table *tRight*) is fetched. When retrieving results of a query, you have to know which records were fetched and cast the items in the returned array accordingly.

Method **more()** is used for retrieving an array of records from the result. (equivalent to multiple **next()** calls). Parameters of this method are the indexes of the first and last record to retrieve and of course an identifier of the result we want to read. The return value is a two dimensional array containg records in the same way as with the **next()** method.

To get the number of records returned by the query, **getNumRows()** method is available. The only parameter is the identifier of the result we are interested in. If the result is empty, value zero is returned.

Subqueries

In case you need to execute nested SELECT query (also known as subselect), DBLayer provides special implementation of *SelectQuery* interface for this purpose called *SubQuery*. *SubQuery* cannot be implemented as a regular *SelectQuery* since underlying Hibernate implementation requires you to use *DetachedCriteria* objects for subqueries and this is not possible to achiev using the *SelectQueryImplementation* class..

Subquery is created by calling **createSubQuery()** method from the DBlayer interface. Obtained *SelectQuery* can then be attached to another *SelectQuery* using a special restriction type (see

documentation of **SelectQuery.addRestriction()** method)

Classes and methods involved:

- interfaces SelectQuery and SubSelectQuery
- DBLayer.createQuery(), DBLayer.executeQuery(), DBLayer.closeQuery()
- DBLayer.createSubQuery()

Managing database users

In order to provide complete set of operations we must be able to manage users on the database level. This is not a very standardized part of the SQL standard and is left to individual database systems to implement how they choose. Therefore there is also no support for these operations in Hibernate.

To deal with this, DBLayer provides its own implementation of these operations using JDBC interface (The general layer structure is violated in this case). Currently, these operations are implemented only for PostgreSQL database system, but implementation for different systems should also be possible.

Roles of the users

There can be three types of users in Plantlore (details can be found in the documentation of the database schema):

- *Administrators* – with almost complete access to data
- *Common users* – with limited access to data
- *Special type for accessing the database from the web interface* – with very limited access to data.

Implementation of this user structure is done via different roles for different types of users. These roles are granted or revoked from users upon their creation or modification. There are 3 roles:

- *Plantlore_Role_Admin* – for users with administrative privileges
- *Plantlore_Role_User* – for common Plantlore users
- *Plantlore_Role_www* – for users accessing data through web interface

Names of database user

When user is created in Plantlore we create new database user as well. Name of this user consists of the prefix with database name (database this user can access) and the login name itself. If a person wants to access more databases on one database system, we create separate database user for each database. (they are differentiated by the prefix with the name of the database). This way person using plantlore can use one username for multiple databases since when logging in to a database, username provided by the user is combined with the database prefix and then used to for database system authentication.

Classes and methods involved:

- createUser(), dropUser(), alterUser() in DBLayer interface

Creating new database

Creating new database is another operation where direct usage of JDBC API is required. Currently it is implemented only for PostgreSQL database system. Creating new database constists of creating the database itself (CREATE DATABASE statement), creating appropriate database user and database roles and creating database tables. SQL statements for creating users, roles and tables are stored and loaded from files located in net.sf.plantlore.config.database package. See the following methods in the HibernateDBLayer for more details:

- **createDatabase()** - creating new database
- **executeSQLScript()** - executing SQL script. This method has to be called for script for creating database users and roles and script for creating database tables.

Plantlore Server and The Communication Layer

Plantlore Server and the Communication Layer [**CL**] are very tightly coupled. The Communication Layer mediates the connection between the Database Layer [**DBL**] and the Presentation Layer [**PL**]. The **PL** and the **DBL** may be running in different Java Virtual Machines [**JVM**] but do not necessarily have to. The **CL** can be disengaged during the local connection and save the overhead that comes with it.

The role of the Server is important only when establishing a new remote connection, and after that the Server just provides some necessary system resources, but is no longer of vital importance and doesn't play almost any role anymore.

The Technology

The technology used to achieve the desired level of transparency, simplicity and elegance is **RMI**. RMI stands for the Remote Method Invocation and it is a standard part of every JVM. The greatest advantage of RMI is that to the caller everything appears to be the same as if the calls he makes to remote objects were local. The only difference is that remote calls may throw RemoteException. If handled properly, it will pose no problem.

RMI Overview and Terminology

The Remote Method Invocation allows us to invoke methods of objects, that do not live in the same JVM as if they were only local objects. The RMI machinery is quite complex and it will not be covered here.

The use of this mechanism is simple and involves five steps:

1. A program **P** creates an object **O** in its **JVM**.
2. The RMI must become aware of existence of this objects, so that it can accept remote calls. The program **P** must **export** it = make RMI aware of it. If the object **O** is exported, it is called a **remote object**.
3. The program **P** should **bind** the remote objects to a **Naming service**, that is easily accessible, so that anyone can obtain the **remote references**. If an object is exported, a remote reference (or **stub**) is created. The stub is a proxy that contains information about the exported object and allows its owner to invoke calls on the remote object. Stubs can be passed via the network and can be available at different JVM, while the remote object never leaves the JVM in which it was created.
4. Clients that wish to call methods of the remote object **O** must obtain their stubs first. This is done by asking the Naming service (**lookup**), which returns the appropriate stub for the remote object.
5. Clients can call methods of the object via the stub.

The Client always has a **stub** and can use it to invoke methods of the **remote object**. Each method call is handled by the RMI – parameters are **marshalled** (packed, serialized) and send to the remote RMI, which must recreate those objects (unmarshall, deserialize) and fill them with sent values, perform the method call, marshall the result, send it back to the other RMI, which unmarshalls them and returns them as a result of the method call.

The Naming Service

The Naming service is a remote object, that is capable of storing pairs `(Name, Stub)` and returning the stub when asked for it by the associated name.

The Stub

A stub is generated for each remote object. Stubs are objects, they have their own classes. If we use the default Naming service, the stub's class is automatically (by RMI) downloaded, so that the stub (object) can be created on the local client.

If we for some reason do not wish to use the Naming service and have our own way of handing over the stub to the client, we must also ensure, that the client has an appropriate class, to be able to re-create the stub.

The Price

The price we have to pay is surprisingly low. First, everything that is passed as an argument of a remote call, must be serializable. Almost all objects that are available at java.lang and java.util are serializable. Making your own object serializable requires you to implement the "marker" interface Serializable. The interface is empty. And second, every method, that accepts a remote call, must throw a RemoteException. This way the RMI mechanism lets us know that something went wrong during the remote call. After all we work with the network.

The Idea

The PL and the DBL are connected. The PL calls methods of the DBL and DBL returns some values to the PL. This is how common work with the DBL would look like. Let's suppose we would be able to create a DBL in a different JVM than the PL. If we use the RMI in a clever way, PL may think that it works with a common object while the DBL will be in fact in a remote JVM operating with a remote database.

All we need is to create a mechanism, that will be able to create Database Layers in remote JVMs and return their stubs to Presentation Layers that will work with them remotely without knowing it. This mechanism together with the RMI can be called the Communication Layer.

Participants

Since the creation of the DBL will be complicated, we will create a **DatabaseLayerFactory** that will shield us from the creation of the DBL completely and just return either the local DBL or the stub that will allow the PL to make remote calls. The PL will not be able to tell the difference. The DatabaseLayerFactory must have its counterpart running in the remote JVM that will create the DBL for there and then pass its stub back to the DatabaseLayerFactory. We will denote the counterpart as **RemoteDatabaseLayerFactory**.

The DatabaseLayerFactory

The interface of the **DatabaseLayerFactory** is simple.

```
interface DBLayerFactory {
      DBLayer create(DBInfo settings);
      void destroy(DBLayer db);
}
```

In the **create()** method the DatabaseLayerFactory will decide whether the User wants a local

connection or a remote connection - this will be clear from the parameter **DBInfo**.

- If the User wants only a local connection, the new DBL is created in the local JVM, thus leaving the RMI out of the picture and saving the overhead that is connected with this technology.
- If the User wants a remote connection, the RemoteDatabaseFactory is contacted, asked to create the new DBL in the remote JVM and then the stub of the remote DBL is returned.

In order to secure a careful destruction of the DBL we provide the **destroy()** method. The principle is the same.

- If the destroyed DBL is local its destruction is local - only a method ensuring the proper cleanup of the database connection is called.
- If the destroyed DBL is remote, ie. if we have only the stub, we must contact the RemoteDatabaseLayerFactory which can destroy it properly. This is why the DatabaseLayerFactory must keep a list of all DBL it has created (or at least helped to create) in order to know which of them can destroy by itself and which must be destroyed by the RemoteDatabaseLayerFactory that created it.

The RemoteDatabaseLayerFactory

The interface of the RemoteDatabaseLayerFactory.

```
interface RemoteDBLayerFactory {
      DBLayer create();
      void destroy(DBLayer stub);
}
```

The purpose of the RemoteDatabaseLayerFactory is to

- create new Database Layers, make them accessible to the RMI so that they can accept remote calls and then return their stubs,
- unexport Database Layers, thus disabling the ability to receive remote calls, and to ensure the proper cleanup of the database connection.

Note that the DBL is not made available in the Naming service to anyone - it is export and becomes a remote object, but the stub is passed only to the caller. No one else will have access to it. Every caller will have its own DBL. In order to monitor the number of Database Layers currently created, the RemoteDatabaseLayerFactory may implement some policy and reject creation of new Database Layers if the policy would be violated.

The Server

The Server has the following interface.

```
public interface Server
      void start();
      void stop();
      ConnectionInfo[] getClients();
      void disconnect(ConnectionInfo client);
}
```

The purpose of the server is to

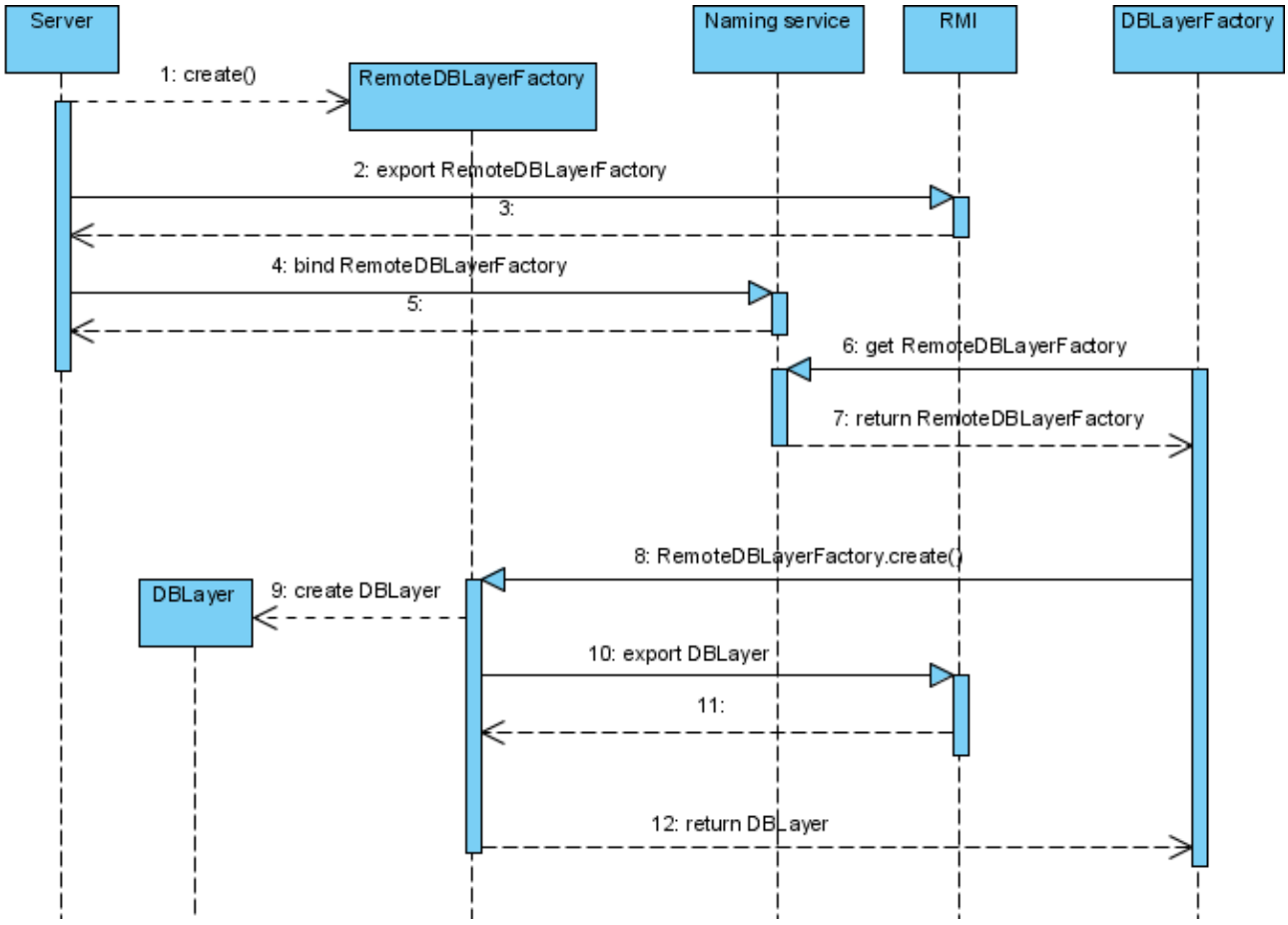
- create and export the RemoteDatabaseLayerFactory, so that it can accept remote calls, and bind it to the Naming service, so that it can be contacted by anyone,
- unbind the RemoteDatabaseLayerFactory from the Naming service rendering it inaccessible,

unexport it, thus preventing it from receiving remote calls, and instruct it to destroy all DatabaseLayers it has created,

- return a list describing all currently created Database Layers, which may be perceived as a list of currently connected clients,
- disconnect a client from the server, ie. instruct the RemoteDatabaseLayerFactory to destroy the DBL although the client does not wish it.

The Interaction of the Participants

The interaction during the Server startup and the new Database Layer creation.



The Guard

The Server must be accessible for others to control. The Server in fact exports itself, making itself a remote object, but it doesn't bind itself to the Naming service which means its stub will not be easily accessible.

Then the server creates another remote object, the Guard, that is exported and bound to the Naming service. The Guard contains the stub of the Server and if the Guard is asked, it can return the stub of the Server, thus efectivelly allowing the caller to control the Server by invoking its remote methods.

The interface of the Guard is:

```
interface Guard {
      Server certify(String authorizationInfo);
}
```

This is how even a remote administration of the Server is achieved. Anyone who wishes to control the Server, contacts the Guard, that is publicly available in the Naming service, calls its **certify()**

method and passes the authorization information. If the information is valid, the stub of the Server is returned.

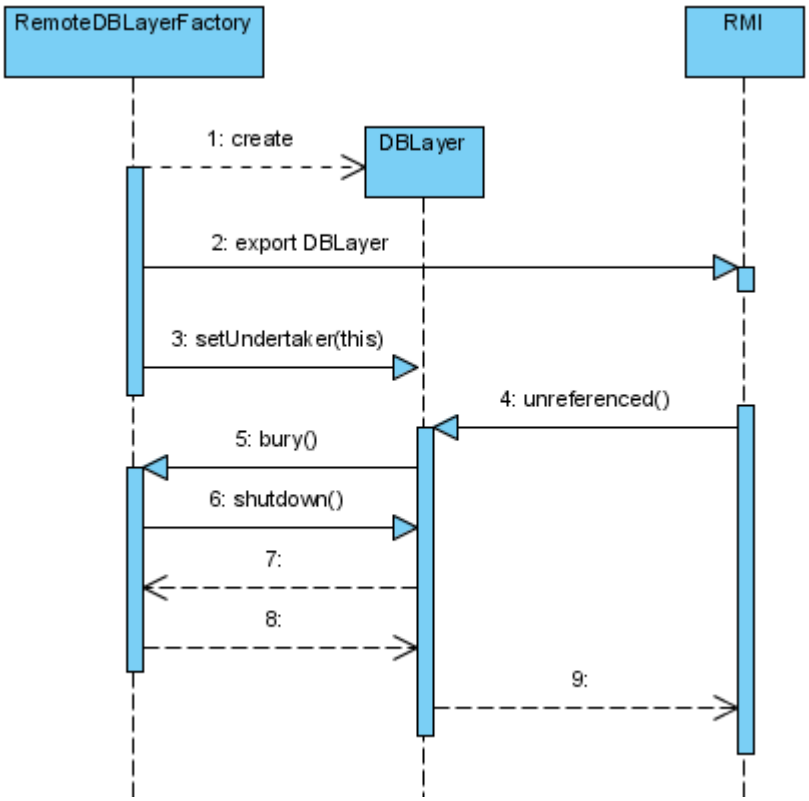
The Database Layer

It has been mentioned already that while the whole process is completely transparent to the PL, the DBL knows about it and must be prepared to take some extra measures. Since the DBL cannot allow the object SelectQuery to leave the JVM where the DBL lives, because if it did, then anyone could modify it and supply a malicious SelectQuery which would completely undermine the system of access rights. Therefore the SelectQuery is another remote object that is exported by the Database Layer, every time the DBL is asked to create it. The DBL must also destroy (unexport) the SelectQuery. This is what the DBL's methods **createQuery()** and **closeQuery()** do. The caller works in fact with stubs.

The Undertaker

The RMI allows the remote object to monitor the state of its remote references. If there are no remote references to the remote object left (either because clients discarded them properly or because they crashed), the method **unreferenced()** of the Unreferenced interface is called.

This mechanism is used to ensure a proper cleanup of the DBL even if the client crashes. Every remote DBL monitors the number of remote references and is equiped with an object that can destroy it properly if no stubs are left. This object is called the Undertaker.



The Destruction

Very similar actions must be taken if the DBL is to be destroyed, but in the inverse order. The DBL is unexported and its shutdown() method is called. The same goes for the RemoteDatabaseLayerFactory which is unbound from the Naming service, unexported and destroy() is called for every DBL that has not been destroyed yet.

Settings of the Server

The Server is configurable and it is possible to store the configuration in a file. The configuration is restored next time the Server is started. In case the configuration file is missing, the default hardwired configuration will be used.

The format of the file is based on XML. The file can be found in the Plantlore configuration directory under the name *plantlore.server.xml*.

Most of these settings can be set using the GUI except the number of connections per ip and the total number of connections. These settings are not documented in the User manual because they were not required to be part of the Server.

The format of the configuration file:

```
<config>
  <server>
    <port>1099</port>
    <connections>20</connections>
    <perip>3</perip>
    <database>
      <engine>postgresql</engine>
      <port>5432</port>
      <parameter></parameter>
    </database>
  </server>
</config>
```

MVC and the GUI communication layer

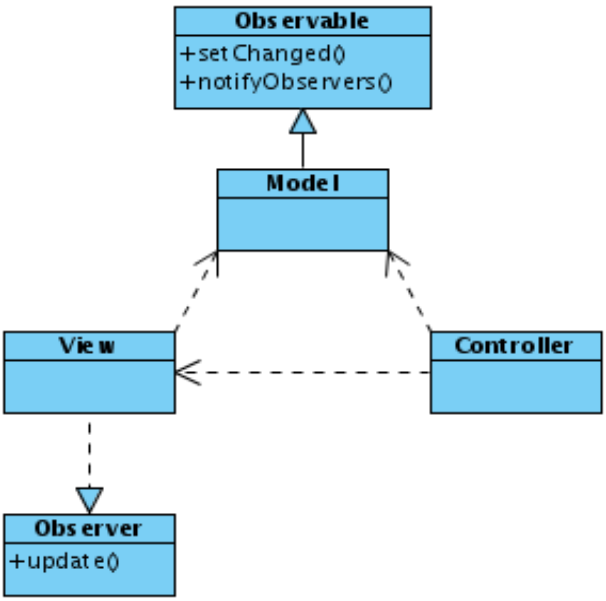
The graphical user interface of Plantlore is built upon a variation to the *Model View Controller* architecture. Each bigger dialog in Plantlore is built using at least one MVC class tripple.

The problem

Plantlore client is already quite a rich client from the view of the amount of dialogs and the complexity of communication between them. Most of the dialogs are quite large and writing all the code of one dialog into one class would result in a hard to understand unmaintainable code. On the other hand Plantlore is not so big to leverage all the flexibility of more advanced MVC based architectures like the Recursive MVC architecture for example. There however is a strong need to mediate information from one MVC triad to another.

MVCs

As opposed to for example the Java Swing we decided to implement every time all the three classes individually. This helps solve the problem of very long and complex source files and it is possible to tell quickly where to look for example for code that's responsible for handling user actions or for code that presents the GUI or for the code that does the actual computation and stores the application state.



Quick overview of MVC

MVC is usually denoted an architecture as opposed to a „simple“ design pattern. It is possible to find several design patterns in a single MVC. The most important one in Plantlore is the Observer design pattern. Model always extends the standard Java Observable class that is responsible for informing all interested Observers about any change to the model which the model wishes to expose. View on the other hand implements the standard Java Observable interface which practically means implementing an update() method. The update method is called each time the Model extending the Observable notifies the Observers using the Observable.notifyObservers() method. Usually it calles the Observable.setChanged() method right before so that it is sure that the

Observers really are notified.

The model usually does the computation and stores all information needed, the view usually presents the information fluently as it is notified from the model that it has changed and the controller's responsibility is to handle interaction with the user.

Another nice feature of MVC, apart from the code separation, is that it is easy to create different views of the same problem (model). Plantlore doesn't leverage this feature.

Problem using MVC

MVC was originally desinged to be one MVC for the whole application. This strategy doesn't suit any modern application any more. Therefore many variations of MVC exist that try to solve distinct problems of modern GUI application. For Plantlore client the main bottleneck was information exchange between the various MVC throughout the whole application.

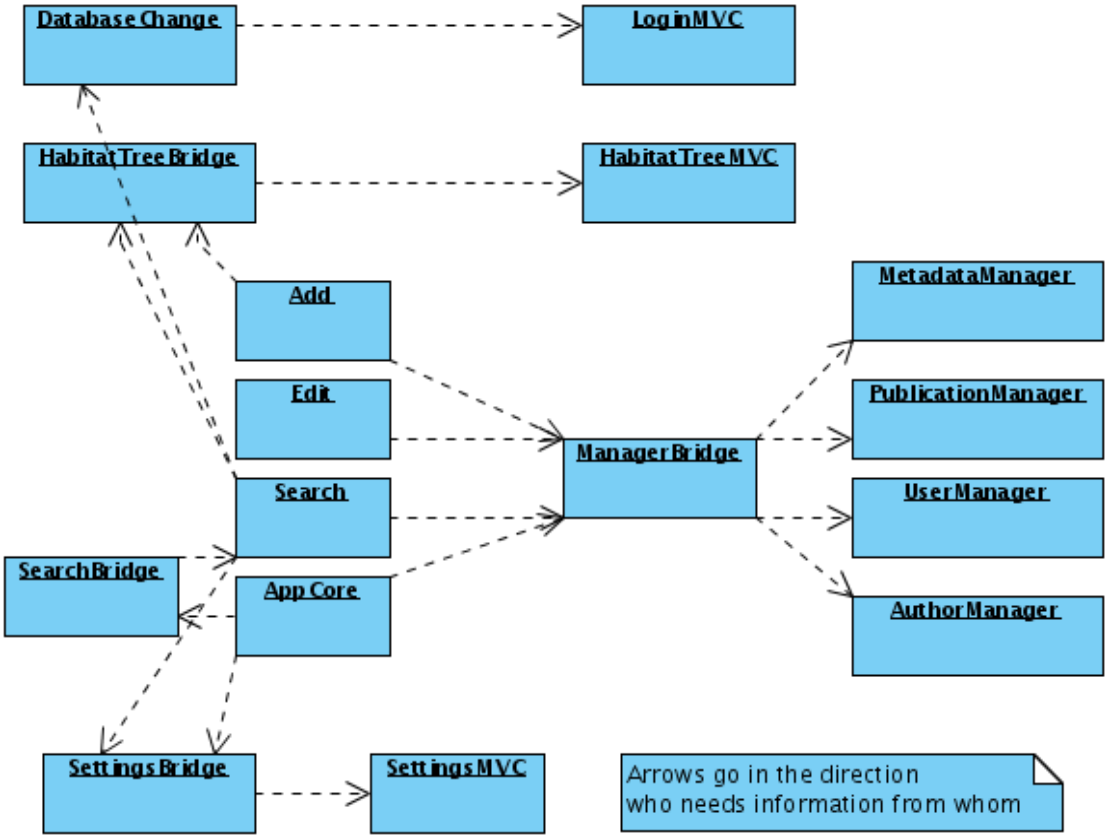
The „GUI communication layer“

The communication problem is solved in Plantlore using the Observer design pattern leveraging the fact that the models are Observables. Plantlore uses special Observers internally (with one exception) called bridges that work as information mediators from one MVC to the rest of the application.

Bridges

All the bridges live inside the main AppCoreCtrl controller. For example a change a user made in the Settings MVC is stored in the Settings model which notifies all Observers about that change. Usually there would only be one observer, the SettingsView. However, in Plantlore there is another called the SettingsBridge. The SettingsBridge observes the Setting model and propagates any information further to the application whenever needed. This is the only responsibility of the *Bridge classes – to propagate information from MVCs to the rest of the application.

Diagram of Bridge ↔ model dependencies


Database Change
LoginMVC
HabitatTreeBridge
HabitatTreeMVC
Add
Edit
Search
SearchBridge
App Core
ManagerBridge
MetadataManager
PublicationManager
UserManager
AuthorManager
SettingsBridge
SettingsMVC
Arrows go in the direction
who needs information from whom

Overview

Overview is an internal Plantlore nickname for the AppCore MVC. It displays and handles the main application JFrame. The AppCoreCtrl works as a central point in the application that coordinates the rest of the application using various Swing listneres and the Plantlore „bridges“ (see chapter about MVC and the GUI communication layer).

The AppCore model

The AppCore model initially loads from the database data required by AddEdit and Search dialogs. It would be very inefficient to load these data from the database each time the dialog is invoked. Therefore the data are loaded only when needed. The AppCore model also holds a so called OverviewTableModel which is the table model used for the main table in overview. The OverviewTableModel handles requests for displaying results of a new query, for moving one page further or back. It also implements the column settings function. The AppCore is informed by the SettingsBridge about the fact that the sequence of columns has changed and it propagates this information further to the OverviewTableModel. Between the AppCore and the OverviewTableModel there is another inter-layer, called the TableSorter that handles the sorting requests when user clicks on a column header in the overview.

The AppCore view

The AppCore view is modelled using the Matisse GUI editor included in Netbeans 5.0 and newer. Therefore it is very easy to actually change the number and location of various components in the view. The AppCore view observes changes in the model and updates itself accordingly. For example it updates the page count and result count upon a new search query.

The AppCore controller

The controller is the main coordination point in Plantlore as mentioned before. For detailed information read the chapter about MVC and the GUI communication layer.

Localization and internationalization

Localization of Plantlore is supported by the net.sf.plantlore.l10n.L10n class. This class works as a channel of Plantlore to the right properties file. Which property file to choose is determined in the main Plantlore class which either takes the via the Java Preferences class stored locale or takes the current locale and initializes with it the L10n class. Then the L10n class tries to find a property file for the given locale and falls back to a default English property if it doesn't find the one for the asked language (locale). When the client connects to a Plantlore server the DBLayer created on that server is also initialized with the clients locale. That way Plantlore is able to display even error messages in the desired locale.

Internationalization

Only parts of Plantlore are internationalized. Mainly the AddEdit and Search dialogs that use the Java NumberFormat class initialized with a proper locale. The current locale can be obtained from the L10n class.

Conventions

Structure of property keys reflects the package hierarchy. Mnemonics are denoted by an ampersand before a character in property files. Method *L10n.getMnemonic(String key)* returns the mnemonic for the given key. Method *L10n.getString(String key)* returns the value with the first ampersand omitted. Keys that have a „tooltip companion“ are denoted by a „TT“ suffix for example: Overview.ReloadTT. Dialog titles are denoted by a „Title“ suffix.

AddEdit and Search

Although almost the center of the application, creation of these dialogs didn't need so much programming skills as much the analytical skills. The whole AddEdit dialog has been refactored significantly at least two times during the development process due to misscommunication between the developers and the clients. Much tedious work has been put into the usability of these dialogs.

Architecture

Both (all three respectively) dialogs are based on the MVC architecture. They differ from the rest of the application only in their size and in the amount of GUI tweaks needed. First they were all based on the same View. But this showed as insufficient solution because of the for AddEdit useless distinction between basic and extended data. This distinction however proved useful in the Search dialog. There are usually just a few fields according to which the user wishes to search and the rest remains almost untouched. The AddEdit dialog also contains a built-in AddEditSettings MVC handling the enabling and disabling of AddEditView's fields.

Input checking

User has to work a lot with these dialogs and therefore it is likely he will make mistakes. Therefore special care has been taken to make the dialogs clever about allowed values in different fields. For most of the fields the **DocumentSizeFilter** class from package common is used to limit the length of entered text. Special DocumentListners have been created for the Altitude, Longitude and Latitude input fields. They are capable of recognizing valid number values even according to the current locale. If the user types some invalid text he immediately receives a red-color feedback – the font of the text turns from black to red. It returns to black after the user edits the text to make it a valid value. For the number recognition the Java **NumberFormat** is used. One tricky part to make it work is not to forget to set the format *setGroupingUsed()* to **false!** Otherwise the text woudl turn red after typing four numbers in a row even if more than four numbers are allowed in the integeral part of the number.

The Core

The core of the **AddEdit** dialog is the *AddEdit.storeRecord()* method that does the actual work with database using **DBLayer**. Very important is that it must conform to what history expects. It has to set the *cDeleted* columns to right values and call right *executeInsert** and *executeUpdate** methods so that history is or is not saved for that action, whatever is appropriate. The *storeRecord()* method uses helper method both from the **AddEdit** model itself and from the **DBLayerUtils**. These methods are simple record based manipulation helper methods. They consit for example: *getHabitatSharingOccurrences()*, *loadHabitatSharingOccurrences(), lookupPlant(), prepareNewOccurrence(), prepareOccurrenceUpdate(), DBLayerUtils.getObjectFor(),* etc.

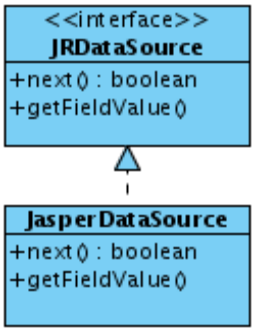
Activity diagram of the *AddEdit.storeRecord()* method follows:

Printing and Scheda

One of requirements on Plantlore was to be able to create schedas. Cards that were usually used in old-fashioned card indexes. From the technical point of view it is just a kind of report generation for data from the database. And for such a task there are many available tools for programmers. We have chosen the JasperReports library for being very flexible and reliable. This comes handy for printing too. Because we needed to create only one specialized data source and then use it for various reports.

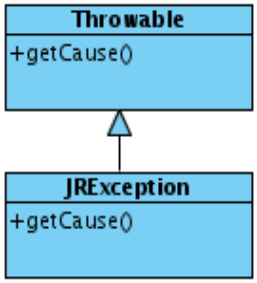
JasperReports

„JasperReports is a powerful open source Java reporting tool that has the ability to deliver rich content onto the screen, to the printer or into PDF, HTML, XLS, CSV and XML files.“ - a quote from the http://jasperforge.org/ website. To leverage JasperReports the best we have created a **JasperDataSource** which is a class implementing a **JRDataSource** a JasperReports data interface that makes any data available to the JasperReports engine. It's implementation is quite straightforward except the exception handling because of the Plantlore specific exception handling. The **JasperDataSource** is created initializing it with a current **DBLayer** and a **Selection** object containing a set of **Occurrences** to be printed (or created schedas from). In the JasperDataSource.next() method we iterate over the Occurrences values of which the JasperReports engine asks using the JRDataSource.getFieldValue().



The Exception Handling

It is only possible to throw a **JREception** from the interface. The problem is that we have to deal with **DBLayer** which can throw **DBLayerException** and **RemoteException**. This would be fine if we didn't have to deal with these exceptions carefully in a uniform way. Therefore we catch these two exception and pack them as a Throwable cause of a newly created **JRException**. Higher in the **AppCoreCtrl** or **PrintCtrl** / **Print** we extract the cause (*Throwable.getCause()*)and throw it again to be handled using the *DefaultExceptionHandler.handle()* method.



The Reports

One side of the problem is the data source the other, presentation, problem is the reports. These can be either written manually in an **XML** file format or fortunately exist **GUI** tools for rapid report creation. For Plantlore we used the **iReport** application to create the reports for Scheda and Printing. The reports are **XML** code combined with Java code which gives the user a great power in creating complex reports. These reports have to be then compiled to a compiled report. This can be done on the fly using the JasperReports engine but it has two drawbacks. One is that the user has to have a **JDK** installed and the second are possible security issues – it could be then possible to create malicious harmful reports. So Plantlore ships with the compiled reports packed into the application jar file. The reports are then loaded dynamically as a resource from the *net.sf.plantlore.client.resources* package.

Export

Export is one part of the Presentation Layer. Export is used to "take" the data from the database and store them in a file. This raises the need to be able to handle several different file formats, each of which has its special requirements. Therefore the Export framework follows the Builder Design Pattern.

The Technique

The Export allows the User to select the list of records that are to be exported and specify which attributes of the Occurrences are important. The list of selected records is represented by a simple wrapper of a set called the **Selection -** the records are identified by their unique database identifier, there is no need to store the whole records. The list of selected attributes is represented by another wrapper of a set called the **Projection**. The Projection allows us to quickly determine, whether certain attribute of the record was selected or not. Their names correspond with their purpose in the database language.

Export also makes use of the last select query that was executed by the User in the Search. If the User selected the records from a smaller subset, it is convenient to use this subset as a frame for the Selection. The last search query is also required in order to achieve the proclaimed functionality: nothing means everything i.e. if the User didn't select any record, everyting from the last Search will be exported.

After all information needed to start the export are gathered, they are supplied to an Export Task Factory, that can create Export tasks (Directors of the Builder Pattern), and appropriate Builders according to the selected file format.

The Participants

There are three participants that will be described below.

Builder

The interface contains several methods that are used to decompose the Occurrence data fetched from the database and build the output appropriately.

```
interface Builder {
      void header();
      void footer();
      void startRecord();
      void part(Record arg);
      void finishRecord();
}
```

The Occurrence record is quite complicated and there are M:N relations in the database model. In order to process the Occurrence properly, it must be supplied to the Builder in parts. Those parts could be easily identified using the Java's reflection API, therefore it is not necessary to have several methods for each part of the record.

The Builder usually takes the Projections in order to know which attributes should be exported and which should not.

A typical Builder implements the **header()**, **footer()**, **startRecord()** and **finishRecord()** methods in its own specific way, but the implementation of **part()** is usualy the same. In this method the whole record is decomposed and sent to the ouptput.

ExportTask

The Export task is the Director that manages the whole export. Export task requires a query that will be used to iterate over the last search query, the Selection that contains the selected records from the query (if any), and the Builder that constructs the output.

```
Builder build
Selection selection

run()
      build.header()
      for each record R in the last search query do
            if not selection contains R  continue

            build.startRecord()
            build.part( R )

            if R is Occurrence then
                  for each AuthorOccurrence AO associated with R do
                        build.part( AO )

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

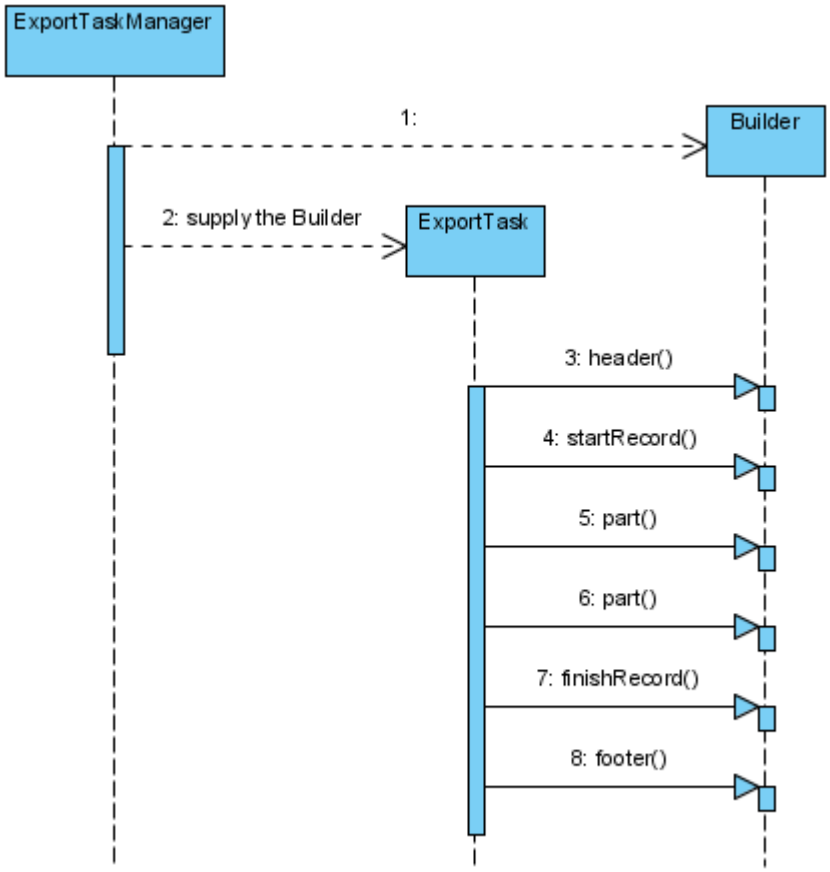
The pseudo-code above shows how exactly are methods of the Builder called.

ExportTaskManager

The ExportTaskManager is the Export Task factory. It is used to store the partial information about the future Export task and once all the information required is supplied, it can create a new Builder, that will be used to form the final output, and the Export task that will manage the whole export.

The Interaction

The creation of all participants and their interaction during Export.



ConcreteBuilder

The Application currently contains four different ConcreteBuilders, three of the are XML based. The last one is a comma-separated-values format.

XMLBased Builders

All XMLbased builders use the DOM4J to construct parts of the output. It is not possible to use the DOM4J to build the whole output, because the XML tree is kept in memory and it could result in extreme memory requirements.

A combined solution was introduced. Only one record is constructed using the DOM4J at a time. If another record is received, the previous one is written to the output. This way it is possible to handle virtually any number of records and still use the greatest advantage of the DOM4J - XML tree construction for Occurrence records.

Comma Separated Values Builder

The CSV Builder places each Occurrence record on a separate line. If there is more than one Author associated with the Occurrence record, the Occurrence is repeated for every Author. That is: an Occurrence having *n* authors will be repeated three times, each time with a different Author.

Occurrence Import

Occurrence Import is used to take the Occurrence data from a file and store them back into the database. Only one file format is currently supported.

The Technique

The Occurrence import must be capable of parsing files of virtually any size. That's why DOM4J is not a suitable choice. The SAX is used instead. SAX is based on a different principle- it uses callbacks to process the recognized XML elements and the contents between them. SAX is used because it is the only other option, it is not a wise choice to write another XML parser.

Because SAX is based on callbacks, the architecture was adapted to this behaviour. The input is parsed using SAX events and the record is gradually reconstructed from the file. If the Occurrence record is completely reconstructed, it is passed to the Record processor, that will insert it back into the database.

The Participants

The Record Processor

The record processor can process the supplied Occurrence record. Processing the Occurrence data requires special measures. The Occurrence data consist of several subrecords, and some of them must be well-defined. It is not possible to process an Occurrence record whose plant, territory, nearest bigger seat or phytochorion is not in the database already. The contents of these tables cannot be altered by the Occurrence Import. The User must use the Table Import in order to make changes to one of those tables. The record processor always makes an effort to find find a match in the database for every subrecord of the Occurrence - it will reuse the existing subrecords in the database whenever possible.

```
interface RecordProcessor {
      void processRecord(AuthorOccurrence...authorOccurrences);
}
```

The Occurrence Parser

The Occurrence Parser parses the input and reconstructs the Occurrence record. It uses SAX callbacks.

Every time a record is fully reconstructed, the Record Processor is invoked on this record. The Record Processor of the Occurrence Parser may be changed.

```
public interface OccurrenceParser {
      void setRecordProcessor(RecordProcessor processor);
      void startParsing();
}
```

The Occurrence Import Task

Import task is a Plantlore Task that must be dispatched accordingly.

The Occurrence Import task initiates the parsing routine. It has no other purpose. This is why in the implementation the Occurrence Import task and the Record Processor are merged together.

The Import Manager

The Import Manager is the Import Task factory. It is used to store the partial information about the future Import Task and once all the information required is supplied, it can create a new Record Processor, Parser and finally the Occurrence Import Task that can start the Parser.

The Interaction

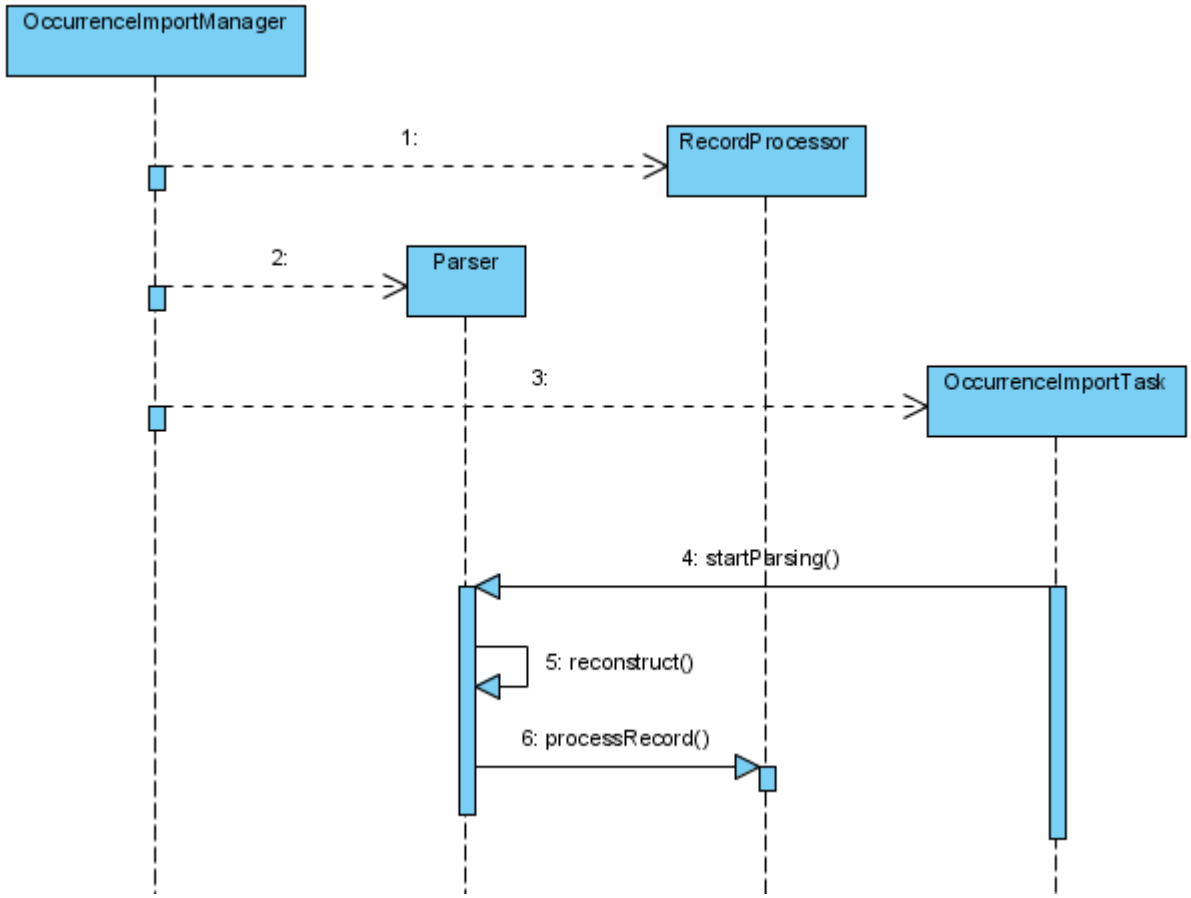


Table Import

Table Import is a procedure modifying otherwise immutable tables. It is not expected this kind of import will have to deal with large quantities of records because it is primarily intended for small changes and updates in the following tables: plants, nearest bigger seats, phytochoria and territories.

The Technique

Since the Table import was never meant to process excessive amounts of data, DOM4J can be safely used. It is not a problem to parse a file containing several thousands records. The framework resembles the "inverse" Builder pattern. Instead of a Builder there is a Parser. The Director does not pass records to the Builder to decompose them, but asks the Parser to return another record.

The format of the file is based on XML and allows adding, deleting and updating of the records.

Participants

The Table Parser

The Table Parser has somewhat similar interface to the Export Builder.

```
interface TableParser {
      boolean hasNext();
      DataHolder getNext();
      int getNumberOfRecords();
}
```

The Parser reads the file and returns records one by one if asked to.

The Table Import Task

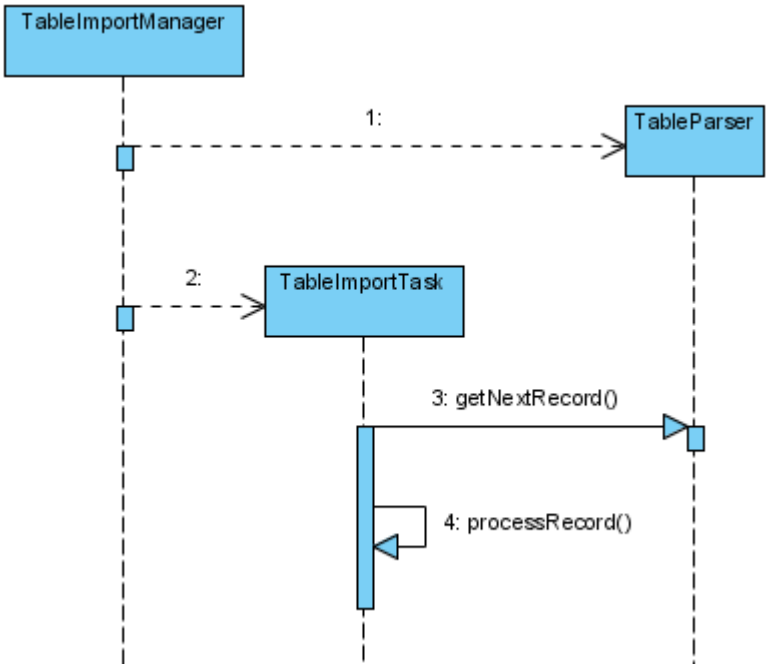
Table Import task is a Plantlore Task that must be dispatched accordingly.

The Table Import task acts as a Director in the Builder Design pattern but it asks the Parser for records instead of pass them to it. The Table Import task then processes these records accordingly. It can add, delete and update them, while maintaining the reference integrity of the database.

The Table Import Manager

The Table Import Manager is the Table Import Task factory. It is used to store the partial information about the future Table Import Task and once all the information required is supplied, it can create a new Table Parser and Table Import Task.

The Interaction



Settings

Plantlore is configurable to a certain level and has various kinds of variable information to be preserved between distinct runs.

MainConfig

MainConfig is a class managing the main configuration file of Plantlore. We have chosen to store it in XML format to be flexible enough for future extension. This configuration file, called plantlore.xml, is stored in the user's home directory called plantlore. On linux system we prepend a dot to make the directory hidden as it is the habit. This file contains a list of columns the User selected she wants to see in **AppCoreView**. And it also holds information of the **DBInfo** objects used by Login to store information needed to connect to and log on a server. For historical reasons the DBInfo information is enclosed in **<triplet>** tags.

MainConfig structure

As stated before it is an XML file. The root element is **<config>**. The **<config>** element holds exactly one **<overview>** element and one **<login>** element. The **<overview>** element then contains a **<columns>** tag which itself holds single **<column>** tags that bear the information about the columns the user chose. The **<login>** tag may contain arbitrary number of **<triplet>** where each triplet determines one database or server connection.

If a new part of Plantlore needs to store some information to the main configuration file the convention is to call the introduced element after the name of the package of that Plantlore part.

MainConfig example

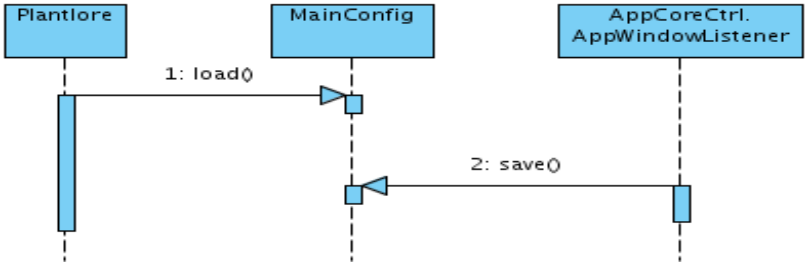
The **<column>** tags contain *net.sf.plantlore.client.overview.Column.Type.toString()* values.

```
<config>
  <overview>
    <columns>
      <column preferredSize="50">OCCURRENCE_ID</column>
      <column preferredSize="30">SELECTION</column>
      <column preferredSize="50">NUMBER</column>
      <column preferredSize="100">PUBLICATION_COLLECTIONNAME</column>
      <column preferredSize="100">PLANT_TAXON</column>
      <column preferredSize="100">AUTHOR</column>
      <column preferredSize="100">HABITAT_NEAREST_VILLAGE_NAME</column>
      <column preferredSize="50">OCCURRENCE_YEARCOLLECTED</column>
      <column preferredSize="100">PHYTOCHORION_NAME</column>
      <column preferredSize="150">HABITAT_DESCRIPTION</column>
      <column preferredSize="100">TERRITORY_NAME</column>
    </columns>
```

```
  </overview>
  <login auto="false">
    <triplet>
      <alias>PlantloreHibDataUTF</alias>
      <host>localhost</host>
      <port>1099</port>
      <database>
        <identifier>/data/plantloreHIBdataUTF.fdb</identifier>
      </database>
      <user>sysdba</user>
      <user>trentin</user>
    </triplet>
</login>
</config>
```

Loading and saving

The main configuration file is created if it doesn't exist yet. The **MainConfig** class expects the directory to be created already. It is loaded by the *Plantlore.loadConfiguration()* method upon Plantlore client start. It is saved upon the close of **AppCoreView JFrame** in the **AppCoreCtrl.AppWindowListener**.



Other settings

Plantlore also needs to store some petty quite wide-spread settings like for example the records per page setting in the AppCoreView or the locale in L10n class. For that purpose we have chosen the standard Java **Preferences** class. Implementation of this class is platform specific which means that on unix system the settings are stored somewhere in a file on the filesystem and on Windows systems these settings (can) go to the Windows registry.

Convention

The convention when using preferences is to always obtain the Preferences object for the class that is interested into storing the preferences. For example for L10n the call would be *Preferences.userNodeForPackage(L10n.class)*.

History

History of changes is recorded only when editing, inserting or deleting Occurrence, Author, Metadata, Publication and Habitats. Speaking of tables: tOccurences, tAuthorsOccurrences, tAuthors, tHabitats, tMetadata, tPublication, tHabitat, tVillages, tPhytochoria a tTerritories.

History can be accessed in two ways:

- The total history
  - includes operations insert, update, and delete performed on any table for which the history is recorded,
  - is accessible to the Administrator only; He can gain greater control over the database and the state of records,
  - the Administrator may undo all changes made to the database up to the selected date thus effectively restoring its previous state.
- History of one record only
  - is kept with every single record,
  - can be view and modified by Users who have the right to edit the record,
  - cannot be invoked from the Overview if the record was deleted, but it is still visible in the total history,
  - is meant to undo some unintentional changes.

Recording the history

The history is saved in the following tables: tHistory, tHistoryChange and tHistoryColumn. The detailed description of these tables can be found in the section describing the datbase model.

We will discuss the possible situations in below. If a column does not contain a value, it will be presumed to be null.

INSERT

- ***insert – a new Occurrence was inserted***
  - tHistoryChange.cRecordId = ID of the inserted record belonging to tOccurrences
  - tHistoryChange.cOperation = 1 (code for insert)
  - tHistoryChange.cWhen ... time of the operation
  - tHistoryChange.cWho … the user who performed the operation
  - tHistory.cOldRecordId = 0
- tHistory.cChangeId … foerign key to the tHistoryChange table
- tHistory.cColumnId … foreign key to the tHistoryColumn table (tHistoryColumn is used to find the ID of a record that has cTable = OCCURRENCE, cColumn = null).
- ***insert – a new Habitat was inserted***

- tHistoryChange.cRecordId = ID of the inserted record belonging to tHabitats
- tHistoryChange.cOperation = 1 (code for insert)
- tHistoryChange.cWhen ... time of the operation
- tHistoryChange.cWho … the user who performed the operation
- tHistory.cOldRecordId = 0
- tHistory.cChangeId, tHistory.cColumnId (tHistoryColumn is used to find the ID of a record that has cTable = HABITAT, cColumn = null).

- ***Insert – a new tPublication, tTerritories, tVillages, tPhytochoria, tMetadata, tAuthors was inserted***
  - tHistoryChange.cRecordId = id of the inserted record
  - tHistoryChange.cOperation = 1 (code for insert)
  - tHistoryChange.cWhen ... time of the operation
  - tHistoryChange.cWho … the user who performed the operation
  - tHistory.cOldRecordId = 0
  - tHistory.cChange, tHistory.cColumn (tHistoryColumn is used to find the ID of a record that has cTable = *tableName*, cColumn = null. *tableName* is the name of the table where the insert took place).
- ***Insert – a new tAuthorsOccurrences was inserted***
  1. ***invoked by insert of a new Occurrence record –*** history is not recorded
  2. ***inkoved by update of an existing Occurrence record***
     - tHistoryChange.cRecordId = ID of the inserted record in tAuthorOccurrence
     - tHistoryChange.cOperation = 2 (code for update)
     - tHistoryChange.cWhen ... time of the operation
     - tHistoryChange.cWho … the user who performed the operation
     - tHistory.cOldRecordId = 0
     - tHistory.cChangeId, tHistory.cColumnId tHistoryColumn is used to find the ID of a record that has cTable = AUTHOROCCURRENCE, cColumn = null)

  - When working with tAuthorOccurrences the problematic cases will be recognized by the tAuthorOccurrences.CDELETE a voláním executeInsertInTransactionHistory (neukládá historii):
    - cdelete = 0 the record is active (not deleted)
    - cdelete = 1 the record was marked as deleted when the update of the Occurrence took place
    - cdelete = 2 the record was marked as delete when the Occurrence was deleted
    - **executeInsertInTransactionHistory** – should be executed in case **1.**
    - **executeInsertInTransaction** - should be executed in case **2.**

UPDATE

- ***Update – update of tOccurrences occurred***
  - tHistoryChange.cRecordId = ID of the record containing the updated column (it can be only the tOccurrence.cId)
  - tHistoryChange.cOperation = 2 (code for update)
  - tHistoryChange.cWhen ... time of the operation
  - tHistoryChange.cWho … the user who performed the operation
  - tHistory.cOldRecordId .... this is a helper column that is used in case the

publication (tOccurrences.cPublicationId), metadata (tOccurrences.cMetadataId), or habitat (tOccurrences.tHabitat) was changed. If the change of the foreign key occurred, this column contains the value before the change took place. In case that another column was updated, the value is 0.
    - tHistory – cOldValue and cNewValue will contain the old and the new value.
    - tHistory.cChangeId, tHistory.cColumnId
    - tHistoryColumn – tHistoryColumn is used to find the ID of a record that has cTable = *tableName*, cColumn = *columnName*. *tableName* is the name of the table that was updated, *columnName* – is the name of the column that was updated.

- ***Update – update of tHabitat occurred (the Habitat will be changed for all Occurrences sharing it)***
    - tHistoryChange.cRecordId = ID of the record that contains the updated column (it can be just the tHabitat.cId)
    - tHistoryChange.cOperation = 2 (code for update)
    - tHistoryChange.cWhen ... time of the operation
    - tHistoryChange.cWho … he user who performed the operation
    - tHistory.cOldRecordId .... this is a helper column that is used in case the cTerritoryId, cPhytochorion or cNearestVilageId is changed. In case that another column was updated, the value is 0.
    - tHistory – cOldValue and cNewValue will contain the old and the new value.
    - tHistory.cChangeId, tHistory.cColumnId
    - tHistoryColumn – tHistoryColumn is used to find the ID of a record that has cTable = *tableName*, cColumn = *columnName*. *tableName* is the name of the table where the update took place, *columnName* – is the name of the column that was updated.

- ***Update - update of tHabitats occurred (the Habitat will be changed to the current Occurrence only = insert of another Habitat)***
    - this case is discussed above.
- ***Update – update in another table where the history is being recorded***
    - tHistoryChange.cRecordId = ID of the record containing the updated column
    - tHistoryChange.cOperation = 2 (code for edit)
    - tHistoryChange.cWhen ... time of the operation
    - tHistoryChange.cWho … he user who performed the operation
    - tHistory.cOldRecordId = 0
    - tHistory – cOldValue and cNewValue will contain the old and the new value.
    - tHistory.cChangeId, tHistory.cColumnId
    - tHistoryColumn – tHistoryColumn is used to find the ID of a record that has cTable = *tableName*, cColumn = *columnName*. *tableName* is the name of the table that was updated, *columnName* – the name of the column that was updated.

The list of columns for which the history is recorded

- tOccurrences
    - cYearCollected, cMontCollected, cDayCollected, cTimeCollected, cDataSource, cHerbarium, cNote, cHabitatId, cPlantId, cPublicationId, cMetadataId
    - it is **not** recorded for: cUnitIdDb, cUnitValue, cCreateWho, cCreateWhen (these four values should never change), cUpdateWho, cUpdateWhen ,

cIsoDateTimeBegin (this is a column whose value is composed from values of other columns)

- tAuthorOccurrence
  - cRole, cNote
- tHabitats
  - cQuadrant, cDescription, cCountry, cAltitude, cLatitude, cLongitude, cNote, cTerritoryId, cPhytochoriaId, cNearestVillageId
- tPhytochoria, tVillages, tTerritories, tPublications, tAuthor, tMetadata
  - all columns of the table in question except cDetete, cVersion, cCreateWho and tPublications.cReferenceCitation (this record comprise cCollectionName, cCollectionYearPublication, cJournalName and cJournalAuthorName)

DELETE

- ***Delete***
  - tHistoryChange.cRecordId = ID of the deleted record.
  - tHistoryChange.cOperation = 3 (code for delete)
  - tHistoryChange.cWhen ... time of the operation
  - tHistoryChange.cWho … the user who performed the operation
  - tHistory.cOldRecordId = 0
  - tHistory.cChangeId, tHistory.cColumnId
  - tHistoryColumn – tHistoryColumn is used to find the ID of a record that has cTable = *tableName*, cColumn = *null*. *tableName* is the nanme ot he column that was updated.
  - tHistory – the cOldValue and cNewValue will be NULL
- In case the Occurrence record is deleted it is imperative that you set for all its tAuthorOccurrences (such that ao.cOccurrenceId == o.cId) tAuthorOccurrence.cDelete to 2. Then you have to use the **executeInsertInTransactionHistory** function that does not save the information about changes of the record (the ifnormation about deletion of the record is sufficient enough)

Log4j logger introduction

Basics

First you get a logger from log4j usually by calling one of Logger's static methods. For example Logger.getRootLogger(), although this particular call is not recommended. Then you start sending logging requests to the logger. What the logger does with the requests depends on what kind of **Appender** you use. There are quite a few to choose from: FileAppender, ConsoleAppender, JDBCAppender, NTEventLogAppender, SMTPAppender, SocketAppender. You can easily guess the purpose of each of them from their names.

The format of the output depends on a so called **layout**. A layout defines the style and content of the output log.

Log4j loggers create a **parent-child hierarchy**. To illustrate, let's say you have a class called MyClass in a package called com.foo.bar. In this class, let's instantiate a logger by the name of com.foo.bar.MyClass. In this case, com.foo.bar.MyClass logger will be a *child* of the logger com.foo.bar -- if it exists. If it does not exist, the only ancestor of the com.foo.bar.MyClass logger will be the root logger provided by log4j (in case there is no logger by the name of com.foo or com).

Each logger has assinged a level. In case you don't provide one log4j assings to the logger the level of its parent. The root logger's default level is DEBUG.

There are **five basic logging levels** in log4j: DEBUG, INFO , WARN , ERROR , FATAL .

You have to send your logging requests using some logging level. The request is then passed to an appender only if the level of the requests is greater than or equal to the logging level of the logger.

Configuration

To work properly log4j must be set up first. The simplest way to do that is to run **BasicConfigurator.configure()**. That way all logging requests will be logged to standard output. More advanced configuration can be done using **PropertyConfigurator.configure(props)**, where the props variable is either a String pointing to a property file or a Properties object containing the log4j configuration properties.