PLANTLORE

Developer Manual

Charles University
Faculty of Mathematics and Physics
Prague, Czech Republic

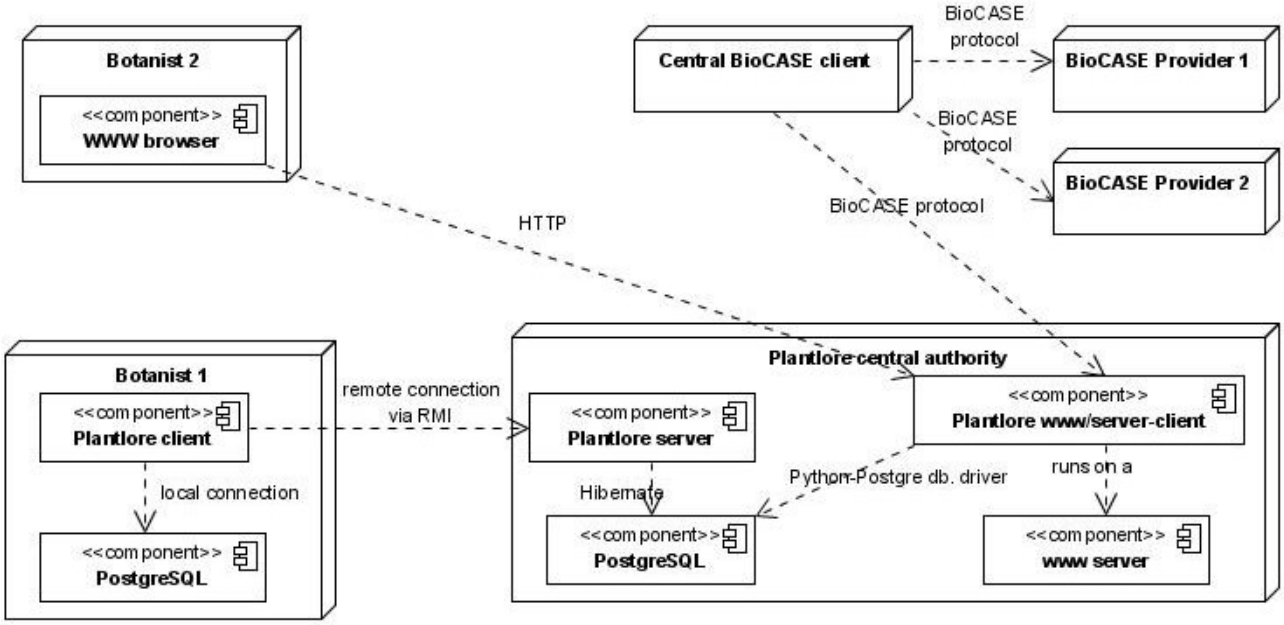
Supervisor: RNDr. Antonín Říha, CSc.
Lada Oberreiterová ladaob@seznam.cz
Jakub Kotowski fraktalek@gmx.net
Tomáš Kovařík tkovarik@gmail.com
Erik Kratochvíl

Table of Contents

1. System Architecture

The Plantlore system consists of three main parts which are the Plantlore Java Client, the Plantlore Java Server and the BioCASE-based Plantlore Web Client.

1.1. The Big Picture



Picture 1: Plantlore deployment diagram

As already introduced in the User Manual, Plantlore is expected to be run in a coordinated enviroment using a central authority. The central authority runs the main central database that is exposed to the world using the Plantlore Server and the Plantlore-BioCASE Web Client. Clients, usually botanists, can access the data using the Plantlore Client or the Plantlore Web Client. Another way to access the data would be using some global BioCASE search tool. The Plantlore Web Client is based upon the BioCASE Provider software which makes it possible for Plantlore to take part in the BioCASE network as a data provider node.

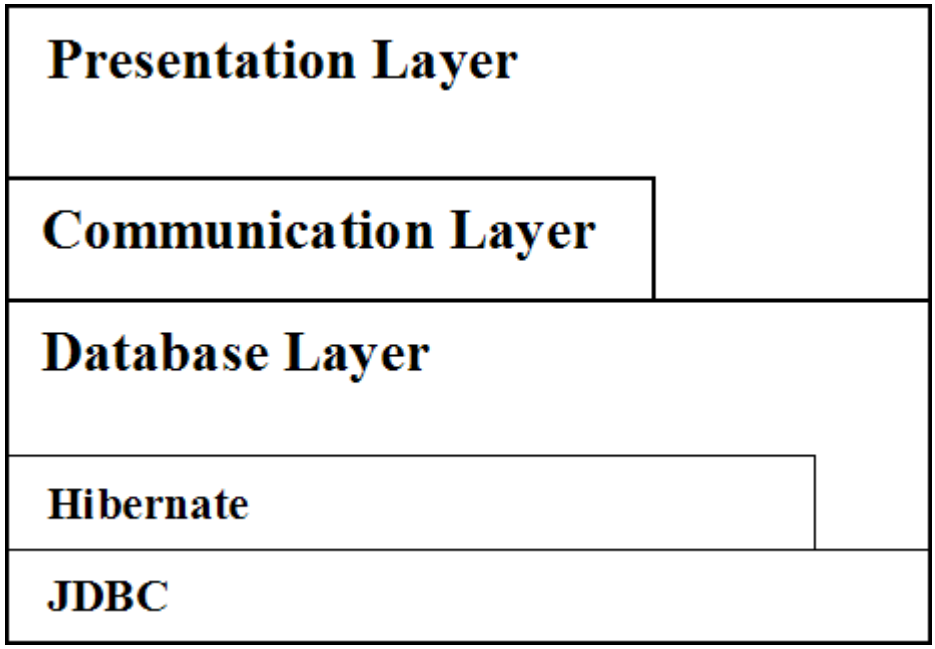
Standalone chapters are devoted to each of these Plantlore components, refer to them for further details. Next chapters introduce the core of the Plantlore system – the Plantlore Server and the Plantlore Client.

2. The Architecture of the Application

In this section we will introduce the architecture (design) of the whole Application that comprises three layers, and discuss their purpose and communication between these layers. Every layer will be described in detail in its own section of this Manual.

2.1. Introduction

The Application is be divided into three basic layers. These layer are depicted in the following picture.



Picture 2: Layers of the Application

2.2. Participants and Their Mutual Interaction

Every layer can only call the layer that is below it. A lower layer cannot call an upper one, it can just carry out requests passed by higher layers.

2.2.1. The Database Layer

The Database Layer serves as an intelligent encapsulation of the Hibernate ORM API and JDBC API. It provides a unified interface that can be safely used in combination with the Communication Layer. This is why several operations withing the Database Layer must be performed carefully, as we will see later. The Database Layer is responsible for the management of the database connection and carrying out all database operations.

The Database Layer also performs verification of sufficient user privileges when an operation is to be performed, and saves information about operations with the database which can be later used to restore the state of a record or of the whole database. There are several other security measures incorporated in the Database Layer that will be discussed in detail later. They include: safe creation of queries that do not allow the User to execute malicious SQL queries, and safe manipulation with records.

2.2.2. The Communication Layer

The Communication Layer is the mediator between the Presentation Layer and the Database Layer in case the remote connection is required, i.e. connection to a remote Database Layer. It ensures the communication over the network - passing the arguments and returning requested results.

The Communication Layer is based on the RMI. In fact it is the RMI that provides the network communication, argument passing, and results retrieval. The only non-trivial part is establishing the connection to the remote Database Layer at the beginning which is managed by the Presentation Layer.

The Communication Layer provides the same interface as the Database Layer so that the Presentation Layer can work with the Communication Layer in the same manner as if it worked with the Database Layer itself.

2.2.3. The Presentation Layer

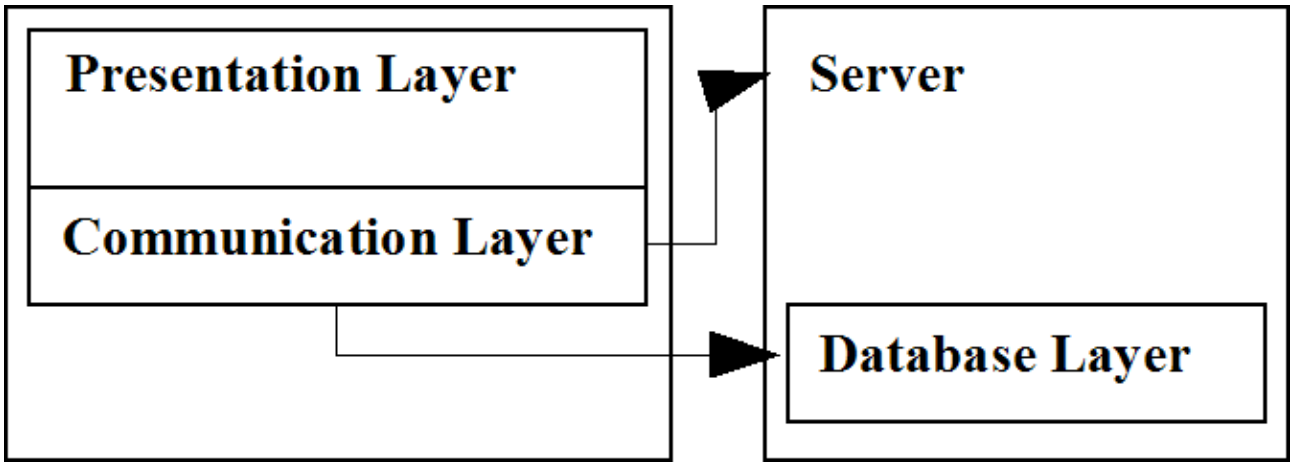
The Presentation Layer is the topmost layer. This layer consists of several parts that are known as the Plantlore Client. This layer provides most of the functionality - it comprises the GUI and the business logic of the Application. All parts of this layer are designed to use the Model-View-Controller architecture; all parts are responsible for carrying out different tasks in the Application.

As you can see from the Picture 2. the Presentation Layer can access the Database Layer directly without getting the Communication Layer involved. This happens during the local connection and can save the overhead connected with the Communication Layer.

In case the connection to a remote Database Layer is required, the Communication Layer is involved. The Presentation Layer establishes the connection to the Server and instructs it to create a remote Database Layer. Then the Communication Layer provides the connection for the Presentation Layer to use.

2.3. The Server

The Server was not introduced yet. Let's have a look at the architecture containing the Server.



Picture 3: Architecture with the Server

As you can see the Presentation Layer can establish a connection to the Server. The Server itself does not use Database Layers, it merely creates and manages them. The Presentation Layer asks the Server to create the remote Database Layer and then the Presentation Layer can work with it using

the Communication Layer.

The only responsibility of the Server is to create and manage its Database Layers including their proper destruction if something happens to the Client - if the Client crashes or if the network connection is lost.

3. MVC and the GUI Communication layer

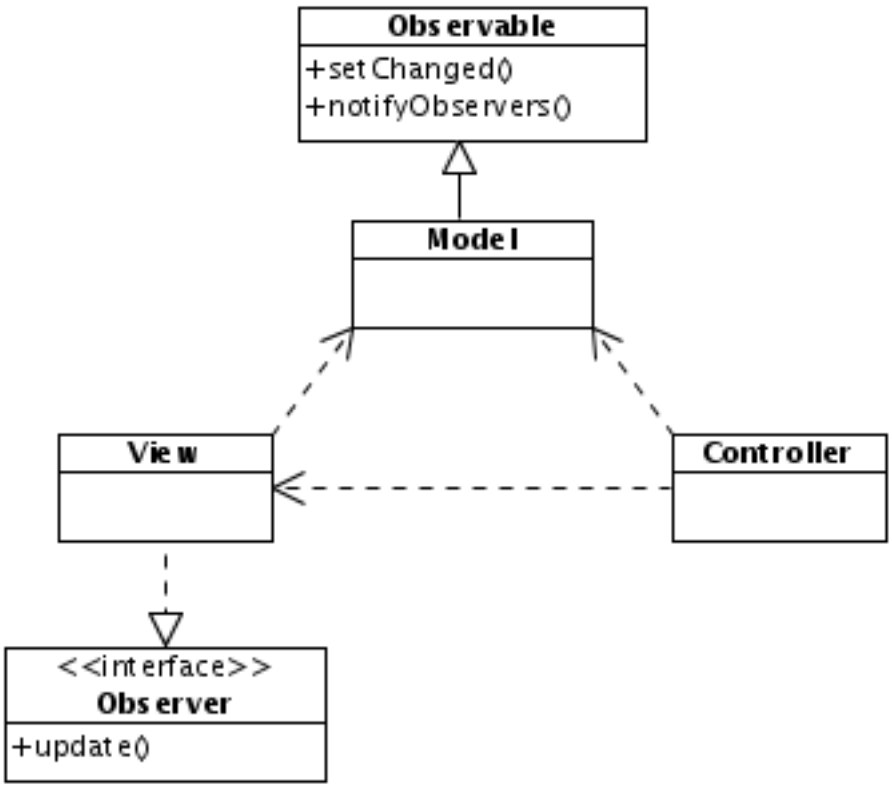
The graphical user interface of Plantlore is built upon a variation to the *Model View Controller* architecture. Every dialog in Plantlore is built using at least one MVC class triad.

3.1. Introduction

Plantlore client is already quite a rich client from the view of the amount of dialogs and the complexity of communication among them. Most dialogs are large and writing all the code of one dialog into one class would result in a difficult to understand code that would be hardly maintainable. On the other hand Plantlore is not so big to utilize all the flexibility of more advanced MVC based architectures like the Recursive MVC architecture for example. Still, there is a strong need to pass information from one MVC triad to another.

3.2. MVCs

We have decided to separate all three parts (the model, the view, and the controller) in their own files. This helps us solve the problem with very long and complex source files and it is possible to tell quickly where to look for the code that's responsible for handling user actions, or for the code that presents the GUI, or for the code that performs the actual computation and stores the application state.



Picture 4.: MVC Overview

3.2.1. Quick Overview of MVC

MVC is usually denoted as an architecture as opposed to a single design pattern. It is possible to find several design patterns in an MVC. One of the most important for Plantlore is the Observer design pattern. Models always extend the standard java.util.Observable class that is responsible for notification of all interested Observers about any change of the model which the model wishes to expose. View on the other hand implements the standard java.util.Observer interface which practically means implementing an **update()** method. The update method is called every time the model extending the Observable notifies its Observers using the **Observable.notifyObservers()** method. The **Observable.setChanged()** method has to be called right before the notification takes place so that it is sure that Observers really are notified. For details consult the Sun's JavaDocs for Observer and Observable classes.

The model usually performs the computation and stores all the information needed. The view presents the information continuously as it is notified by the model every time it is changed. 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 make use of this feature that much.

3.3. A Problem Using MVC

It was originally assumed that only one MVC would suffice for the whole application. This strategy doesn't suit almost any modern application any more. Therefore many variations of MVC attempt to solve various problems inherent in modern GUI designs. In the Plantlore Client the main problem was the information exchange among various MVCs throughout the whole Application.

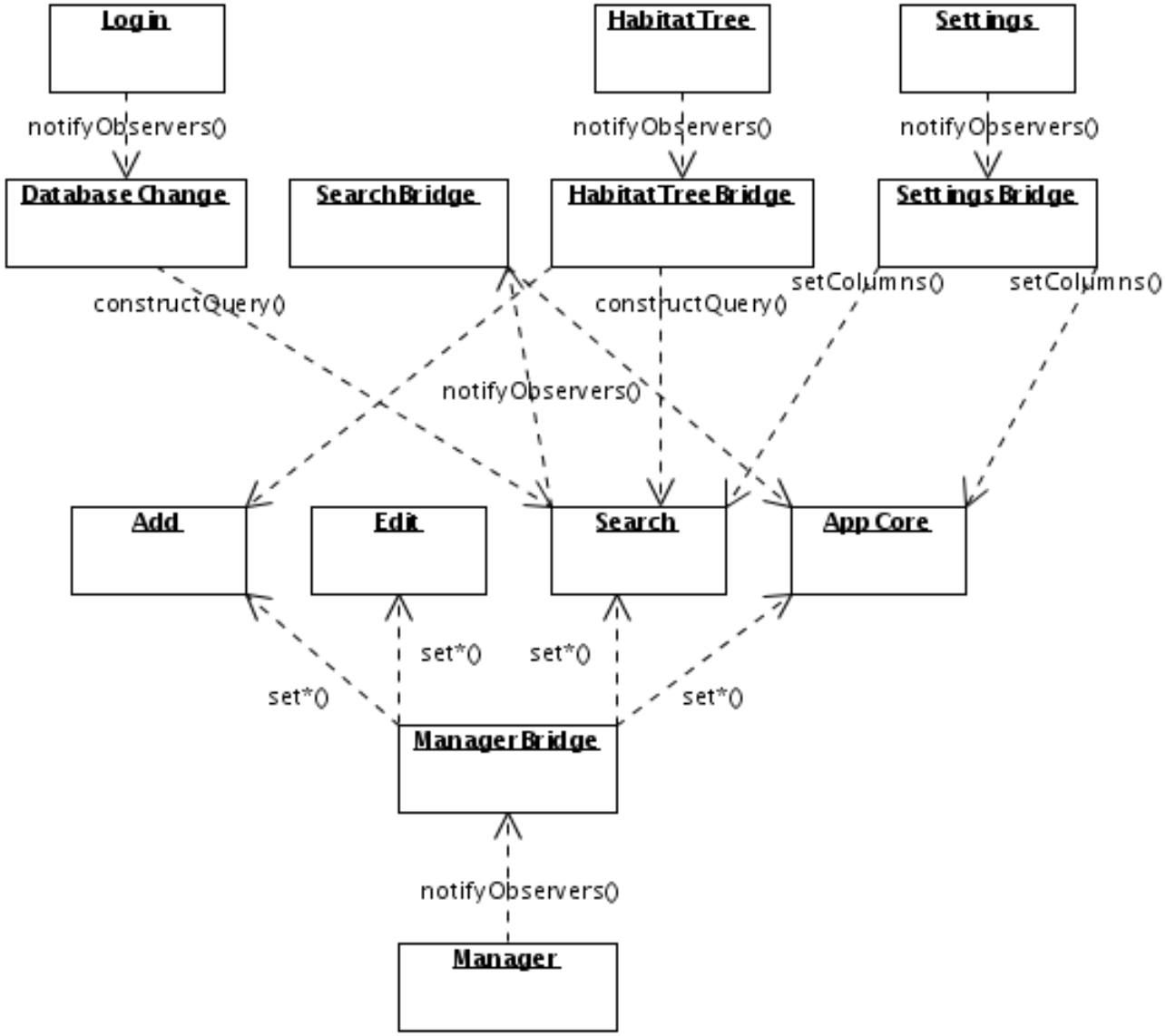
3.4. The „GUI Communication Layer“

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

3.4.1. Bridges

All the bridges live inside the main **AppCoreCtrl** controller. For example a change the User made in the Settings MVC is stored in the Settings model which notifies all Observers about that change. Usually there would be only one observer, the SettingsView. However, in the Plantlore Client there is another one called the SettingsBridge. The SettingsBridge observes the Settings 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.

3.4.2. Diagram of Bridges - Model Information Flow



Picture 5: Information flow. Arrow always begins at the source and points to the user or mediator of the information.

Only the main information paths are displayed on the diagram above. Some minor communication also takes place. One example would be the scheda headers settings that are stored in Settings model using the standard Java Preferences class and recalled in Scheda and Print models using the same class. Thus there is no direct information flow in this case.

The Manager object in the diagram stands for any and all of the AuthorManager, PublicationManager, MetadataManager, UserManager, Table Import, and Occurrence Import models. ManagerBridge calls mainly various setters of the Add, Edit, Search and AppCore models. For further details consult the JavaDocs and the sources of the models and bridges.

4. Database Layer

4.1. Introduction

This section 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 section 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 the Hibernate documentation at www.hibernate.org if necessary.

The Database Layer is used to cover the Hibernate API and provide other layers the API for comfortable work with the database. This layer is necessary because we do not want to call Hibernate directly from the rest of the sources and we need to do additional data processing before persisting data (such as access rights checks, saving history etc.)

In this section we will use the following terms when describing the database layer and the API it provides.

- *DBLayer* – is a short for the Database Layer as a whole (includes several classes listed below),
- *Client* – in this section the Client is a class using the DBLayer,
- *User* – is a person working with Plantlore,
- *Plantlore user* – is an entity in Plantlore identified by a username with the given privileges,
- *Database user* – is a User of the underlying database management system.

4.2. Participants

The Database layer in Plantlore is implemented in the following classes and interfaces. We will go through the most important functions of the DBLayer in the following paragraphs.

`middleware.DBLayer`	Interface defining the API for working with a database.
`server.HibernateDBLayer`	Implementation of the DBLayer interface using Hibernate.
`middleware.SelectQuery`	Interface defining the API for building SELECT queries.
`server.SelectQueryImplementation`	Implementation of SelectQuery interface using Hibernate criteria queries.
`server.SubQueryImplementation`	Implementation of SelectQuery interface for executing subqueries (subselects). Reasons for this implementation are explained later.

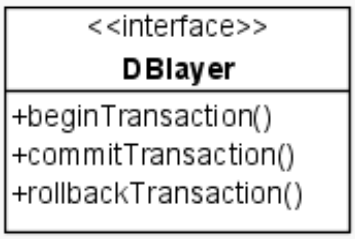
4.2.1. Initialization of the Database Connection

Picture 6: DBLayer

Initialization of the database connection occurs 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, account name and password). The initialization includes the Database User authentication and the Plantlore User authentication (against data in the tUser table). Later during the Application execution it is possible to obtain the connected User and His rights using **getUser()** and **getUserRights()** methods.

In the initialization phase Hibernate's 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 already connected to a database. Every method in the Database Layer working with the database has to check the existence of SessionFactory before executing anything. Information about the User that is currently logged in is stored as well and is used to check User's privileges when executing database operations.

4.2.2. Transaction Management Support



Picture 7: DBLayer

Hibernate ORM makes the JDBC transaction support available via its interface which is sufficient enough for short database transactions that are used in basic methods of the Database Layer and its Clients do not have to care about it, because they are completely transparent to them.

In case the need to execute multiple SQL queries arises (such as processing several subrecords of the Occurrence record), we have to provide interface to start a transaction, and commit and rollback it so that the Client has full control over the transaction.

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

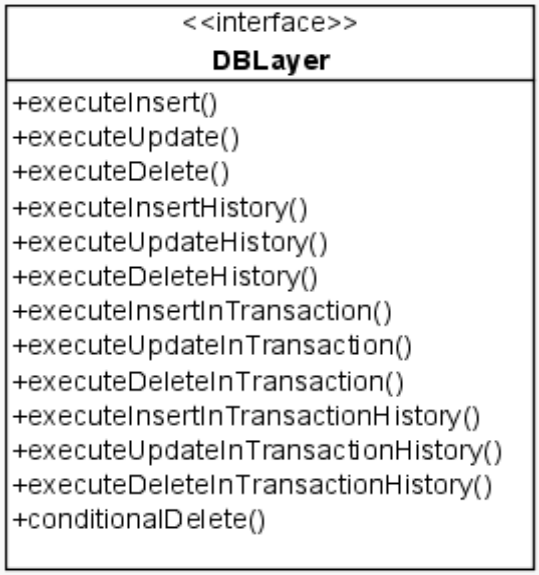
4.2.2.1. Application Transactions and Optimistic Row Level Locking

In many cases we have to deal with application transactions that often include a user input which might be problematic from the database point of view. One of the approaches known as the pesimistic locking is to lock database rows when they are requested for updating and unlock them after the update takes place. Firstly, this requires a lot of additional work by the database system and spawns number of problems. Secondly, the support for this behaviour on the database level is not universal (not all database systems implement SELECT FOR UPDATE statement) and Hibernate's support is even worse.

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

The downside of the optimistic approach is that updates made by the second transaction are completely lost. Since we don't expect Plantlore to run in a highly concurrent environment, using this method is feasible.

4.2.3. Executing INSERT/UPDATE/DELETE Statements



Picture 8: DBLayer

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, mostly due to the security reasons. These operations are:

4.2.3.1. Modification of the Occurrence Data Prior to Persisting

In order to keep the database in a consistent state and protect it from unauthorized or malicious 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 record.
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 to prevent data corruption when records are replicated and merged together.

4.2.3.2. Checking User Privileges

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

4.2.3.3. Saving History of Database Modifications

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

From the nature of this feature, it is reasonable to implement it on the DBLayer level so that we have complete control over the process. Detailed rules for saving history records are explained in the Section **History**.

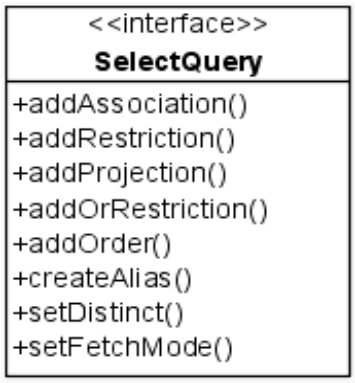
Only selected modifications of selected types of records are saved to history. Therefore DBLayer provides methods which do not modify the history. Refer to JavaDoc documentation for details.

4.2.3.4. Deleting Records

When the 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, most records are only marked as deleted (by setting “deleted” parameter of the record) and are not made directly 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 .

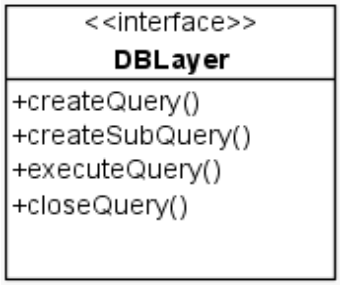
4.2.4. Executing SELECT Queries



Picture 9: SelectQuery

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 ourselves, 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.

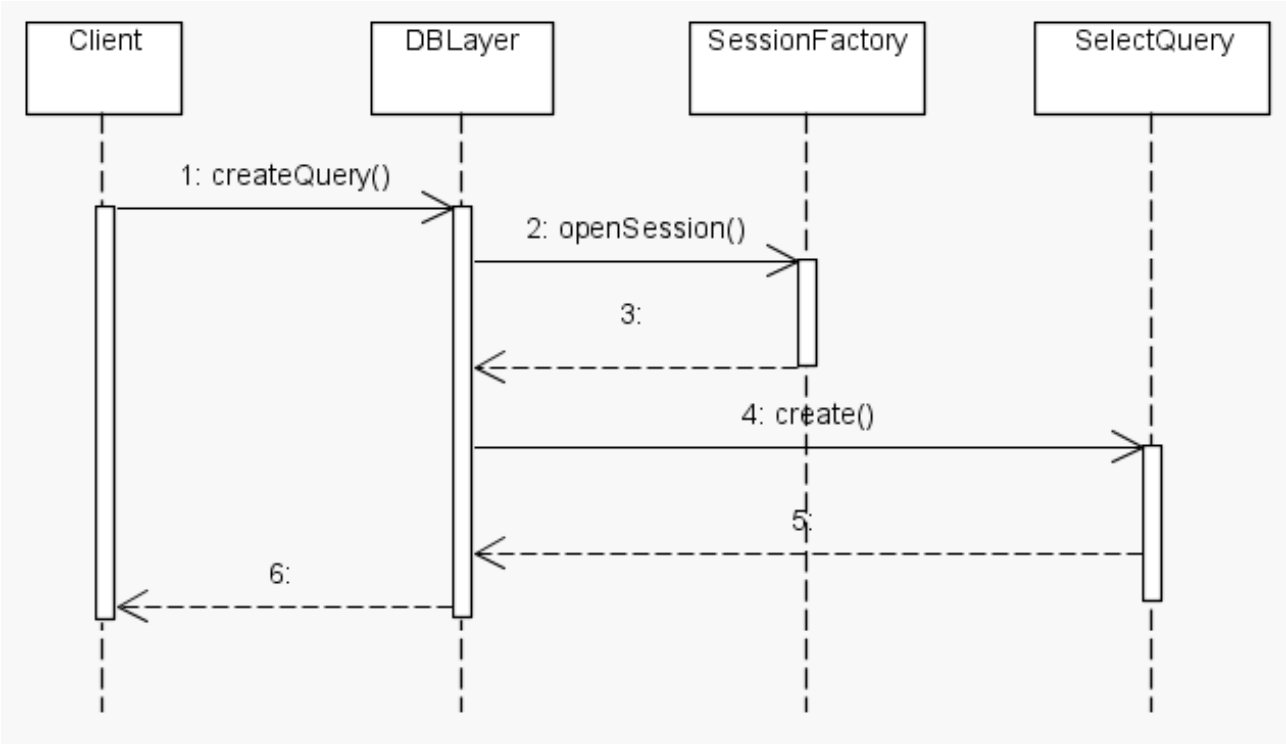
4.2.4.1. Constructing a Query



Picture 10: DBLayer

The 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.

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



Picture 11: Construction of a new SelectQuery

4.2.4.2. 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 the type of the projection and the name of the projected column. The names of columns are available as constants stored in the respective holder objects.

4.2.4.3. 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 five parameters:

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.
First property name	The name of the column used for operators which accept one or two columns as arguments.
Second property name	The name of the column used for operators which accept two columns as arguments. If the restriction works with only one column, this parameter

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.
	can be null.
Value	Any value we want to use as an argument for the selected operator.
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.

4.2.4.4. 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 the **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 the 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);
```

4.2.4.5. Ordering the Results

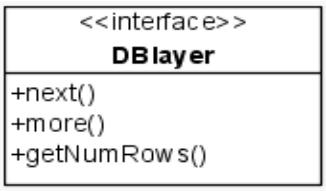
Ordering results is possible using the **addOrder()** method defined in the SelectQuery interface. The **addOrder()** method takes two arguments, the direction of ordering and the column to be used for ordering the results. Direction can be either ascending or descending.

4.2.4.6. Executing a Query

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

The **executeQuery()** method returns unique id of the result which is used for reading the results. It is necessary to distinguish between two results because more query results can be opened at the same time.

4.2.4.7. Working with Query Rresults



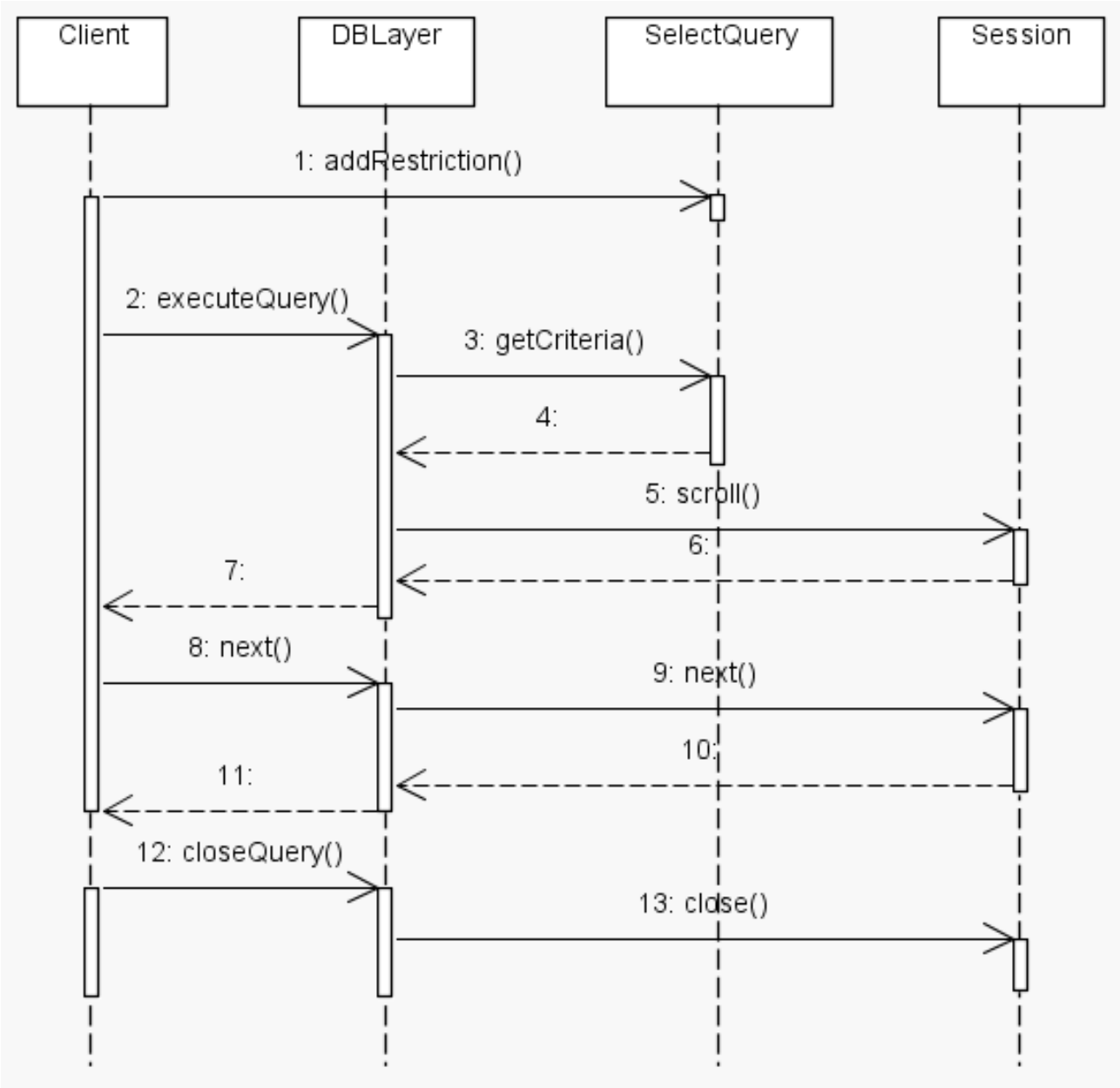
Picture 12: DBLayer

Executing a 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.

next() fetches the 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 the 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.

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 be retrieved 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 learn 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, 0 is returned.



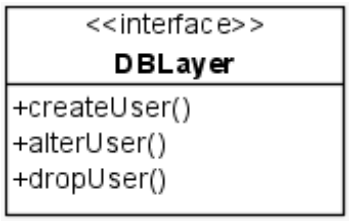
Picture 13: Working with QueryResults

4.2.4.8. Subqueries

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

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

4.2.5. Managing Database Users



Picture 14: DBLayer

In order to provide complete set of operations we must be able to manage Database and Plantlore Users on the database level. This is not a very standardized part of the SQL standard and it is left to individual database systems to implement as they see fit. Therefore there is next to no support for these operations in Hibernate.

To deal with this problem, the DBLayer provides its own implementation of these operations using JDBC interface (forcing us to abandon the general layer structure this time). Currently, these operations are implemented only for PostgreSQL database system, but implementation for different systems should also be possible.

4.2.5.1. 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 full access to data
Common users	With limited access to data
Special type for accessing the database from the web	With very limited access to data

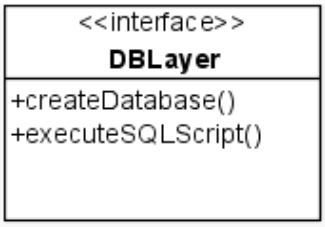
Implementation of this user structure is done via different roles for different types of users. These roles are granted or revoked from users upon their creation or modification. There are 3 roles:

Plantlore_Role_Admin	For Users with administrative privileges
Plantlore_Role_User	For common Plantlore Users
Plantlore_Role_www	For Uusers accessing data through Web Client

4.2.5.2. Names of Database Users

When a User is created in Plantlore we create new Database User as well. The name of this User consists of the prefix formed by the name of the database 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 distinguished by the prefix = the name of the database). This way the person using Plantlore can use the same username for multiple databases because when logging in to a database, the username provided by the User is combined with the database prefix and then used for the database system authentication.

4.2.6. Creating a New Database



Picture 15: DBLayer

Creating a 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 consists of creating the database itself (CREATE DATABASE statement), creating the appropriate Database User and database roles and creating database tables. SQL statements for creating Database Users, roles and tables are stored and loaded from files located in net.sf.plantlore.config.database package.

5. Tasks and the Dispatcher

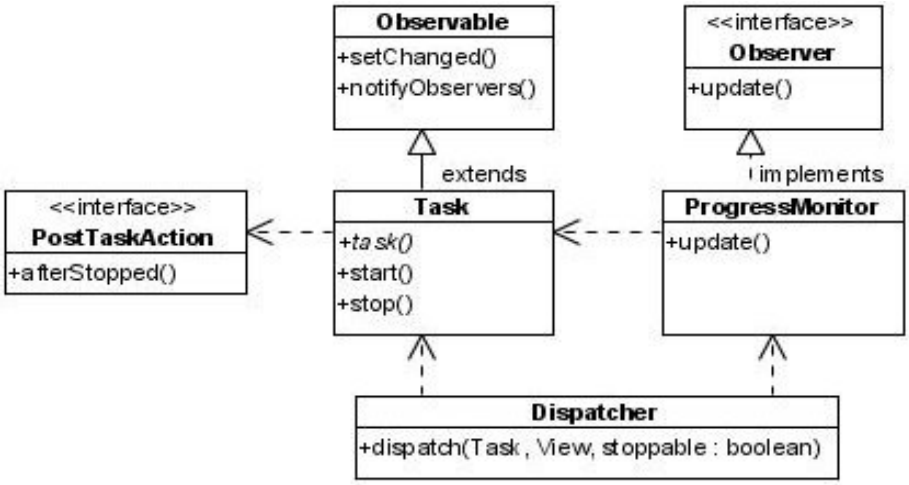
To solve the problem of the freezing Swing GUI when running long tasks the Task class was introduced. Task is an abstract class that executes the given code in a separate thread. To prevent problems when running multiple threads we introduced the Dispatcher class that actually starts (dispatches) each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

5.1. The Goal

Several goals lead us to the Task/Dispatcher solution. We had to present the User with a responsive GUI that not only doesn't freeze (does repaint itself regularly) but also keeps the User informed about the ongoing computation whenever possible. Another important issue was the "task-safety". We needed to prevent possible conflicts when running more tasks at the same time.

5.2. The Solution

The solution has three parts. The Task and Dispatcher classes and a Progress Monitor class. The Task takes care of the computation, the Dispatcher assures that at any time there is only one Task running when Dispatcher is used. The Progress Monitor watches the Task's progress and informs the User about it (or even offers cancellation of the task).



Picture 16: Task, Progress Monitor, Dispatcher

5.2.1. Task

Task builds upon the SwingWorker helper class and upon the Observer design pattern as implemented in standard Java. Task extends the Observer class. The user of the task, someone who wishes to perform a (potentially) long computation, just has to implement it's **task()** mehtod. It is the Task's user's responsibility to inform the Observers about progress of the computation from within the **task()** method. The actual new thread creation and return value handling is delegated to the SwingWorker class.

5.2.2. Progress Monitor

In this section we will 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 its progress. This is usually achieved by using some form of Swing's JProgressBar. Plantlore implements several Progress Monitor classes. The most 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 cancellation. It is used for example to inform the User about the progress of the Export Task. In other cases the ProgressBarManager is used.

5.2.3. Dispatcher

The 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()** method. If the Task should be stoppable then it uses the SimpleProgressBar2; otherwise it uses the ProgressBarManager. When called, the dispatcher checks whether other Task is already running and if it does it just returns, doing nothing. If no other Task is currently running it installs the proper progress monitor to the Task as an observer and starts the Task.

5.3. Incorporating it in the Application

To make sure the Dispatcher is able to fullfill it's responsibility it follows the Singleton design pattern. 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 after the Task stops. To this end a PostTaskAction interface has been introduced.

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.

5.3.1. Exception Handling

The Task's user in Plantlore typically has to use the Database Layer which can throw eithe the RemoteException or the DBLayerExcpetion at any time. 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 directly inform the User about the problem and possibly offer a solution. Progress Monitors do call the **DefaultExceptionHandler.handle()** to handle exceptions.

6. The Communication Layer

The Communication Layer is designed to be able to mediate the connection between a Presentation Layer and a Database Layer that will most likely "live" in different Virtual Machines. The Communication Layer uses the Remote Method Invocation, RMI.

Its greatest advantage is that it has the same interface as the Database Layer, so that the Presentation Layer, or at least most parts of it, are not aware of the network connection.

6.1. Introduction

First, we describe the RMI - how it works and its terminology.

6.1.1. How the RMI Works

The Remote Method Invocation allows us to invoke methods of objects, that do not "live" in the same JVM (i.e. that were not created 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 the Object can accept remote calls. The program **P** must **export** it = make RMI aware of it. If an object is exported, it is called a **remote object**.
3. The program **P** should **bind** the remote object to a **Naming service**, that is easily accessible, so that anyone can obtain the **remote reference**. 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 a remote object must obtain its stubs first. This is achieved 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 is a remote object, that is capable of

- storing pairs `[Name, Stub]` in a hash-table,
- and returning the stub when asked for it by the associated name.

For our purposes we use the default Naming Service Java provides - the **rmiregistry**.

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 downloaded automatically, so that the stub can be created on the local client.

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

6.1.2. What is Required

First, everything that is passed as an argument of a remote method call, must be serializable. Almost all objects that are available at java.lang and java.util are already serializable. Making your own object serializable requires you to implement the "marker" interface Serializable. The interface is empty.

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.

Third, every object, whose methods are to be called, must either implement the Remote interface or extend the UnicastRemoteObject. Both ways are equal, but if the object extends the UnicastRemoteObject, it is exported automatically the moment it is created. If it implements the Remote interface, we must export it ourselves.

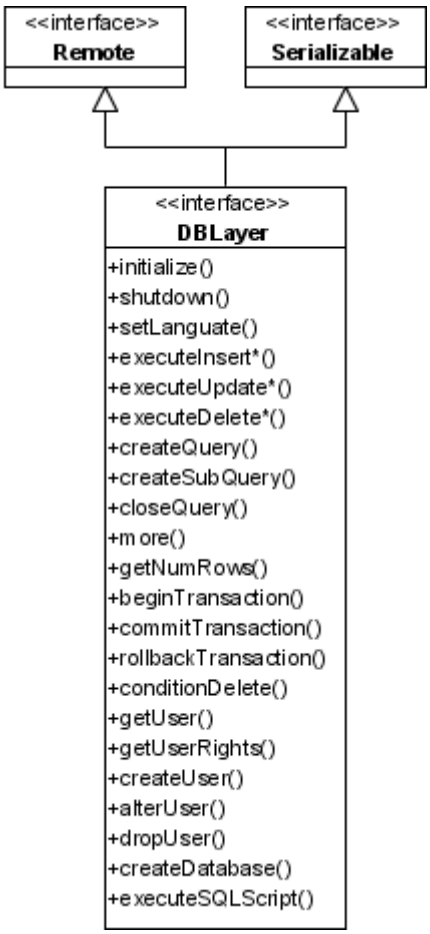
6.2. Participants

There is only one participant in this case - it is the Remote interface DBLayer. Everything else is handled by the RMI mechanism (assuming all objects were properly exported).

6.2.1. The DBLayer Interface

The DBLayer interface is located in the file

`net.sf.plantlore.middleware.DBLayer.java`.



Picture 17: DBLayer

This is the interface that is presented to the Presentation Layer. The Presentation Layer can use all its methods. The implementation of the interface is different: in the client JVM it is the stub and in the remote JVM it is the remote object.

6.3. Implementation

The particular implementation can differ but in our case it is the HibernateDBLayer.

6.3.1. HibernateDBLayer

This is the implementation of the DBLayer interface. For further details see the Section **Database Layer**.

6.4. Notes

Stubs must be generated manually for every remote object that will not be bound to the Naming Service. Those stubs must be made available to both client and server; the RMI mechanism would not work properly otherwise.

To generate a stub of a remote object you must use the rmi compiler (rmic).

7. The Server

In this Section we will discuss all participants on the Server side of the Application and their interaction when a new Database Layer must be created and properly destroyed. We well explain the mechanism of automatic Database Layer destruction if the Client fails to do so. And we will discuss the possibility to gain access to the Server in order to perform its administration.

7.1. Introduction

The Server is a remote object that comprises several parts responsible for the creation and management of its Database Layers including their destruction if something happens to the Client - if the Client crashes or if the network connection is lost.

Server can be controlled, even remotely, which is why the Server is a remote object itself.

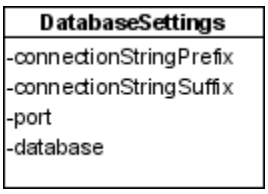
7.2. Participants

There are several participants and interfaces. They can be found in these two packages: net.sf.plantlore.middleware and net.sf.plantlore.server. The most important participants are the RemoteDBLayerFactory and the Server.

7.2.1. DatabaseSettings

The Database Settings is a simple holder object describing the settings of the database to which the RemoteDBLayerFactory should connect in order to create a new Database Layer. Database Settingsare in the file

`net.sf.plantlore.server.DatabaseSettings.java`.



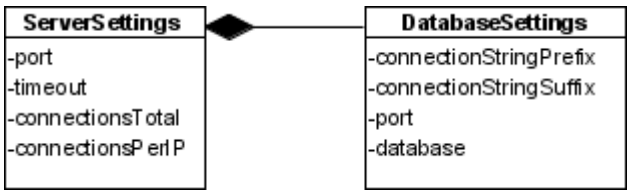
Picture 18:
Database Settings

There are only a few relevant settings that are supplied by the Administrator of the Server: the database type (engine), port and suffix (which is the database parameter). From these values the connectionPrefixString is created - it is the JDBC connection string and the suffix is the part the goes after the question mark (?). The rest of the settings required to initialize the Database Layer is supplied by the Client (i.e. the name of the database, the user name and the password).

7.2.2. ServerSettings

Server Settings configure the Server's policy to accept or reject a request to create another Database Layer, and configure the Server itself. Server Settings are in the file

`net.sf.plantlore.server.ServerSettings.java`.



Picture 19: Server Settings

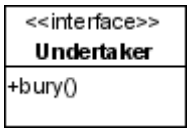
Part of the Server Settings is formed by the Database Settings. Then there is a port on which the rmiregistry with the RemoteDBLayerFactory should be available, total number of connections and maximum number of connections per IP.

7.2.3. 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.

The Undertaker denotes an object that can properly destroy stranded Database Layers, i.e. Database Layers whose clients crashed. After the time specified by the timeout in Server Settings is over and the Client does not respond, its Database Layer is destroyed. The Interface is in the file

`net.sf.plantlore.server.Undertaker.java`.

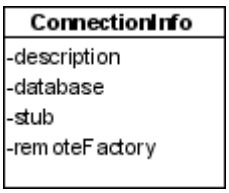


Picture 20: Undertaker

7.2.4. ConnectionInfo

Connection Info stores information about the created Database Layer (the remote object) and its stub and about the RemoteDBLayerFactory that created it. It can be found in the

`net.sf.plantlore.server.ConnectionInfo.java`.

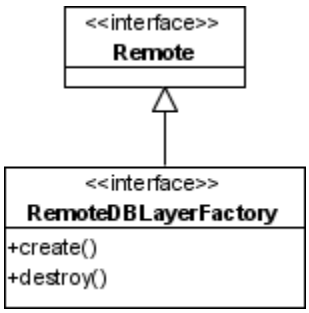


Picture 21: Connection Info

7.2.5. RemoteDBLayerFactory

This interface is located in the net.sf.plantlore.middleware.RemoteDBLayerFactory and those who implement it must be able to

1. create a new Database Layer, export it and return its stub,
2. unexport and destroy the Database Layer when given the stub.



Picture 22: Remote DBLayer Factory

Note that the Database Layer should not be publicly available in the Naming service to anyone - it should be exported and become a remote object, but the stub of the Database Layer should be returned to the caller only. No one else should be able to access it. Therefore every caller will have its own Database Layer. In order to monitor the number of Database Layers created, the RemoteDBLayerFactory may implement some policy and reject creation of new Database Layers if the policy would be violated.

The source is located in the file

`net.sf.plantlore.middleware.RemoteDBLayerFactory.java.`

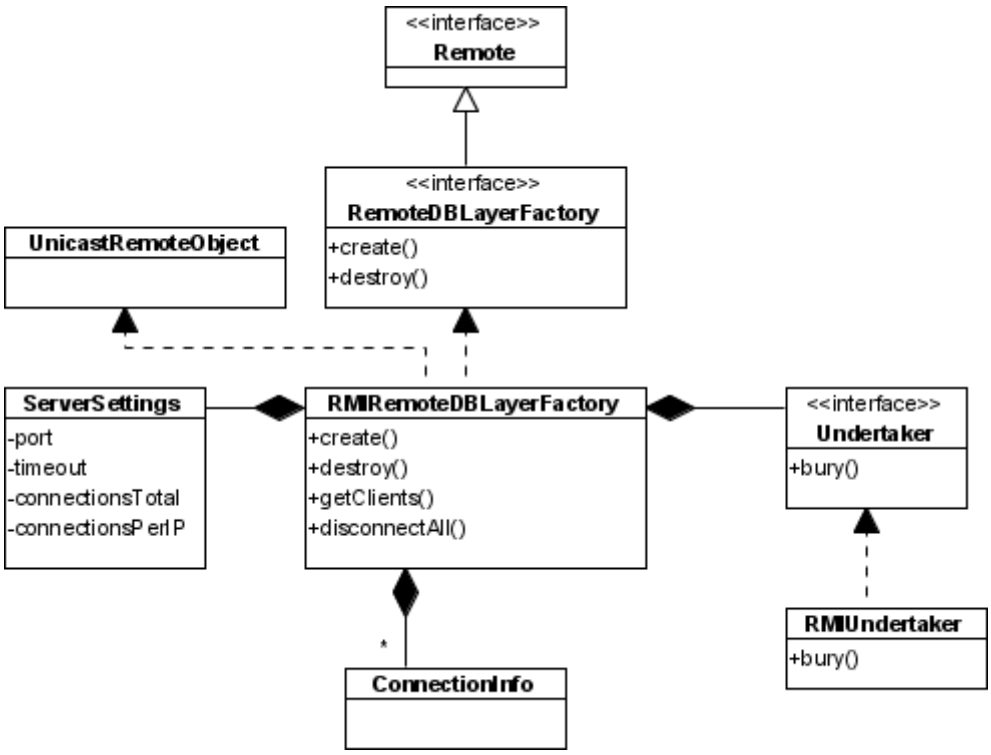
7.2.6. RMIRemoteDBLayerFactory

RMIRemoteDBLayerFactory is a particular implementation of the RemoteDBLayerFactory interface that can perform all the tasks that are delegated to it. It also implements its own policy to limit the total number of Database Layers created as well as the maximum number of Database Layers created for clients with the same IP address. If those numbers would be exceeded the creation of another Database Layer is rejected.

The class is equipped with its own implementation of the Undertaker that can get rid of Database Layers whose clients crashed or terminated without destroying their Database Layers properly.

The source is in the file

`net.sf.plantlore.server.RMIRemoteDBLayerFactory.java`

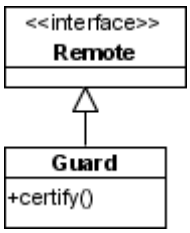


Picture 23: RMIRemote DBLayer Factory

7.2.7. Guard

Guard is an interface that provides access to the Server for its Administration since the Server should not be publicly available in the rmiregistry. The interface is in the file

`net.sf.plantlore.server.Guard.java`.

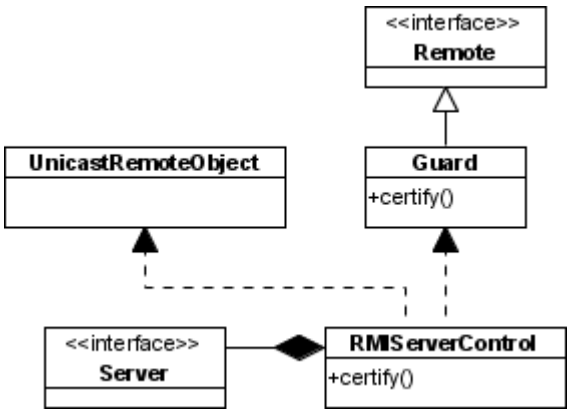


Picture 24: Guard

7.2.8. RMIServerControl

The RMIServerControl is the implementation of the Guard. It stores the stub of the Server as well as the password guarding the access to it. If the supplied certification information fits, the stub is returned. The source is avaliable at

`net.sf.plantlore.server.RMIServerControl.java`.



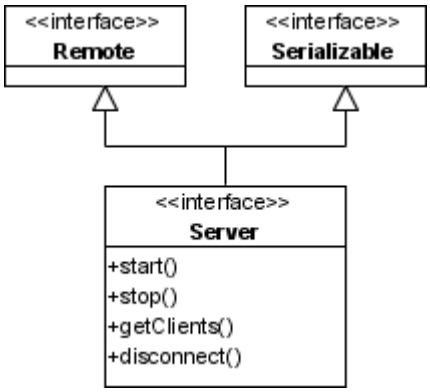
Picture 25: RMI Server Control

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 under a well known name, calls its **certify()** method, and passes the authorization information. If the information is valid, the stub of the Server is returned.

7.2.9. Server

The Server interface serves to control the Server - especially to start and terminate it. If the Server is asked to start, it should create a new RemoteDBLayerFactory, export it and bind it to the rmiregistry under a well known name. The Server itself should not be available in the rmiregistry, because anyone would be able to obtain it and terminate which is not desirable.

When the Server is asked to terminate it must unbind the RemoteDBLayerFactory from the rmiregistry, unexport it and instruct it to destroy all the Database Layers it has already created, so that the connection of all clients is terminated as well.



Picture 26: Server

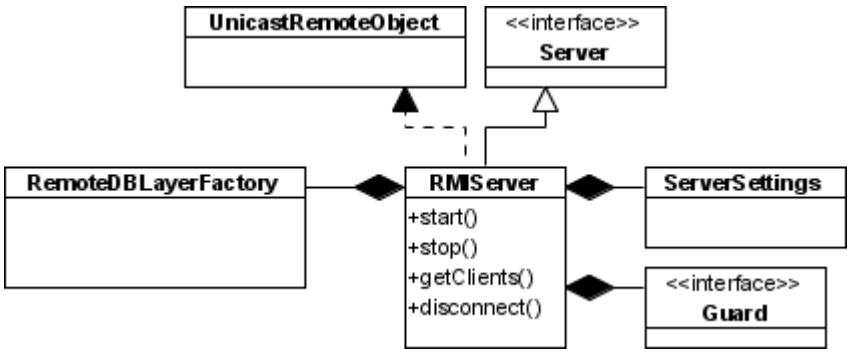
The Server may be able to return the list of currently connected clients (i.e. the list of Database Layers that have been created) and may be able to disconnect some of its clients (i.e. unexport and destroy the selected Database Layer).

The interface is in the file `net.sf.plantlore.server.Server.java`.

7.2.10. RMIServer

The RMIServer is the implementation of the Server interface. It does exactly all the things the interface specifies. RMIServer is in the file

`net.sf.plantlore.server.RMIServer.java`.



Picture 27: RMI Server

The RMIServre contains the RemoteDBLayerFactory that constructs and destroys DatabaseLayers, Guard that protects the access to the Server for administration and Server Settings that describe the configuration of the Server.

7.2.11. Database Layer

The Database Layer to be created. It is described in particular detail in the Section **Database Layer**. Although the RemoteDBLayerFactory creates and destroys it, it has no other interest in it.

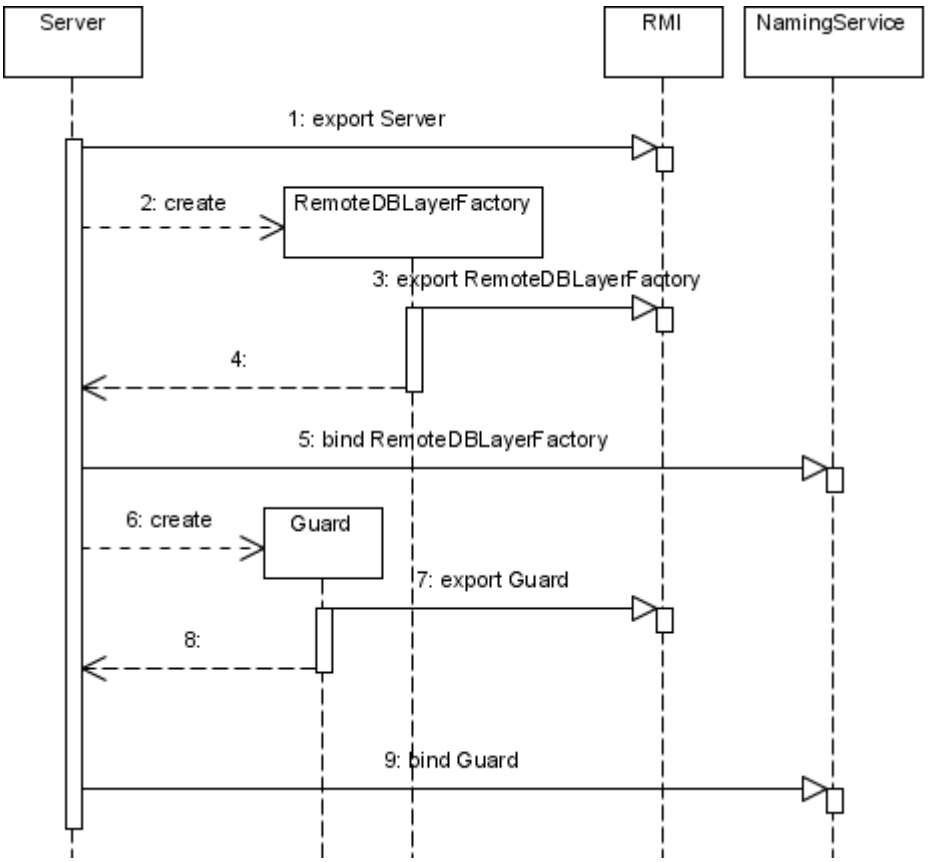
The Database Layer knows that it can be a remote object and must be prepared to take some extra measures. Since the Database Layer cannot allow the object SelectQuery to leave the JVM where the DBL lives, because if it did, then anyone could modify it and supply a malicious SelectQuery which would completely undermine the system of access rights. Therefore the Database Layer makes the SelectQuery another remote object that is exported by the Database Layer, every time the Database Layer is asked to create a new Query. The Database Layer must also destroy (unexport) the SelectQuery. This is what the Database Layer's methods **createQuery()** and **closeQuery()** do.

7.3. Interaction

In this section we provide the detailed information about the interaction between all participants in several situations.

7.3.1. Starting the Server

Starting the Server is a very straightforward operation. Note that the Server is indeed not bound to the rmiregistry (Naming Service) while both RemoteDBLayerFactory and Guard are.



Picture 28: Start of the Server

7.3.2. Stopping the Server

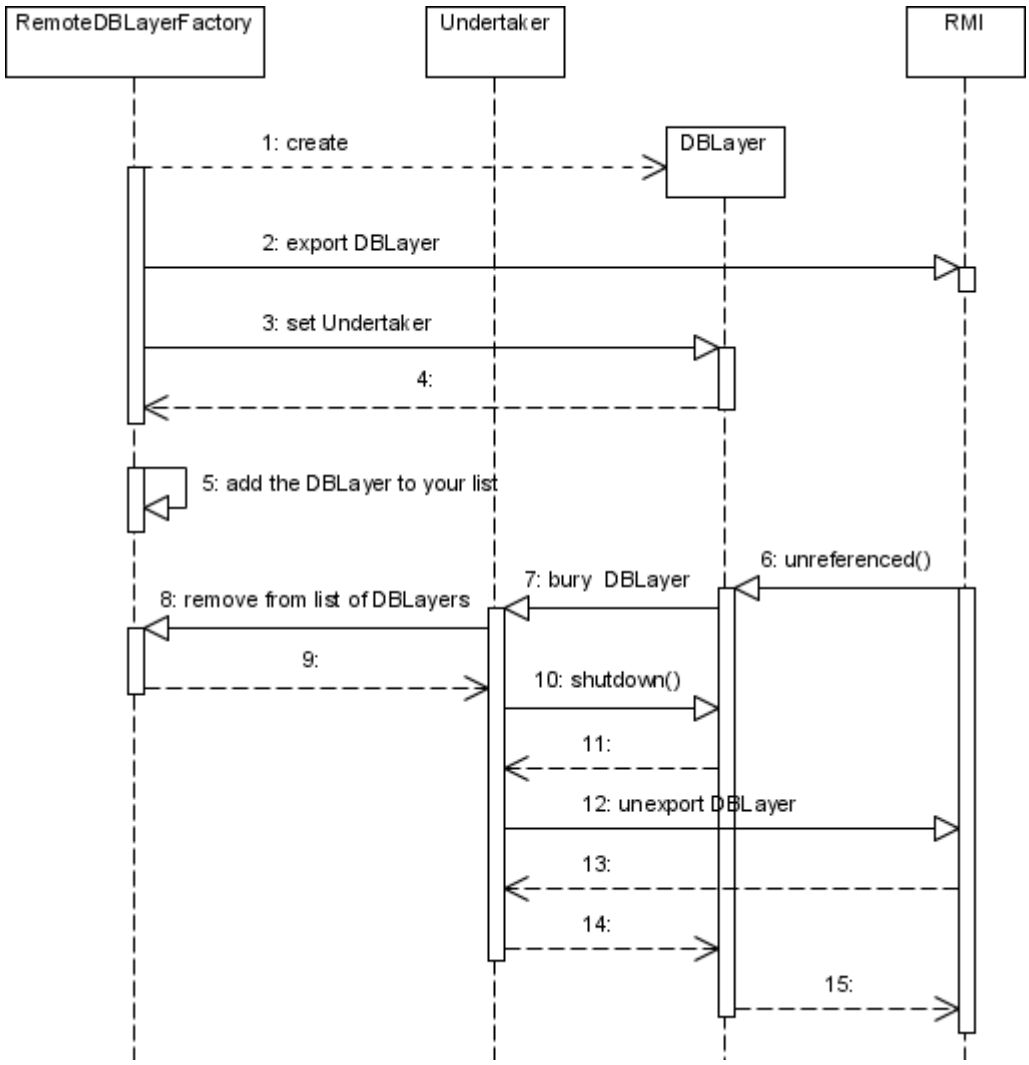
When the Server is to be stopped, it performs all operations but in the reverse order and their counterparts - unbind, unexport, and destroy.

7.3.3. Creating a New Database Layer and Destroying It

In this picture there you can see the process of creation of a new Database Layer, if the RemoteDBLayerFactory is asked to. It takes steps 1-5. The caller does not obtain the Database Layer itself (because it is the remote object) but its remote reference.

The process of proper destruction of a Database Layer is very similar to the process of its creation, but it is in the reverse order and counterparts of methods are used - unexport and destroy.

Steps 6-15. depict the situation when the network connection with the client has failed and the RMI considers the stub to be dead. In case there are no remote references (stubs) to the remote object Database Layer left, the RMI signals it to the Database Layer via the Unreferenced interface. In this interface the Database Layer calls its Undertaker that should take all steps necessary for its destruction. The Undertaker properly destroys the Database Layer by calling its shutdown() method and unexports it preventing it from accepting remote calls.



Picture 29: Creation of a new Database Layer and Its Destruction

8. The Server Manager

Server manager is a simple object that can

- create a new Server,
- connect to a running Server,
- manage the Server it has created or to which it has connected.

8.1. Introduction

The Server manager can create a new Server and give it the order to start, stop, disconnect a client, or return the list of currently connected clients. It simply makes use of the interface that the Server implements.

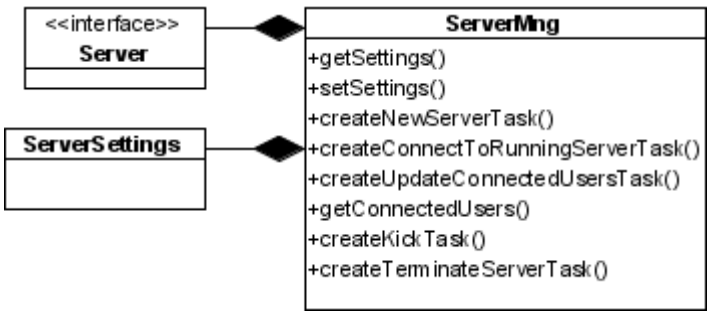
In case the Server is already running on some computer, the Server manager contacts the Guard, obtains the stub of the Server from it, and (because the Server is a remote object) can call the same methods as in the previous case. This means, that the Server can be stopped or have its clients disconnected remotely.

8.2. Participants

There is only one participant this time.

8.2.1. ServerManager

The Server Manager is capable of running a new Server with the specified Server Settings that can be stored and loaded from a file. The Server Manager is located in the file `net.sf.plantlore.server.manager.ServerMng.java`.



Picture 30: Server Manager

It works as a task factory. To learn more about tasks see the Section **Tasks and Dispatcher**. These tasks are very simple as they call methods of the Server, that has been either created or connected to (obtained from the Guard after certification).

8.3. Notes

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 - it supports up to 32 total connections and maximum of 3 connection per one IP address.

The file can be found in the Plantlore configuration directory under the name plantlore.server.xml. In this file there is a simple tree describing the Server Settings.

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>
```

9. Login and Logout

So that the User can work with a database, He must connect to it first. As we know the work with the database is possible via the Database Layer only. Furthermore, we can use the Communication Layer to work with a remote Database Layer, that can be created for us by a remote Server.

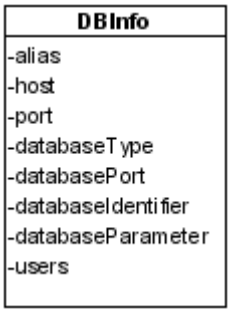
In this section we will discuss the process of a new Database Layer creation from the Client's perspective, i.e. how the client connects the Server and obtains a new Database Layer.

9.1. Participants

All participants can be found in the net.sf.plantlore.client.login and net.sf.plantlore.middleware package. Almost all the business logic of this operation is concentrated in the Login.java.

9.1.1. DBInfo

The Database Info, DBInfo in short, holds information about the database and where it is located. Its source is in the `net.sf.plantlore.client.login.DBInfo.java`.



Picture 31:
Database Info

Information from this holder object are presented to the User who can alter them at will. They are used when the creation of a new Database Layer is required.

9.1.2. DBLayerFactory

DBLayerFactory is an interface that allows the client to control a Database Layer Factory which is an object that shields other parts from the particular implementation of the Database Layer creation and destruction. There should be only one Database Layer Factory at a time. It can be found in `net.sf.plantlore.middleware.DBLayerFactory.java`.

Picture 32:
DBLayer Factory

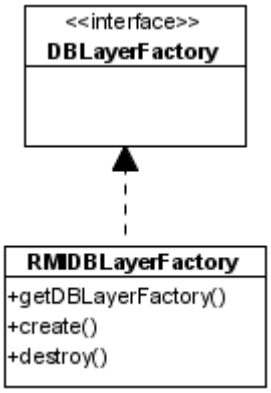
9.1.3. RMIDBLayerFactory

RMIDBLayerFactory can create and destroy Database Layers. It creates them based on the supplied

DBInfo. It creates them either in the current JVM if the database runs on the local computer, or contacts the RemoteDBLayerFactory and asks it to create a new Database Layer for it.

RMIDBlayerFactory is implemented as a Singleton in order to be sure there is only one instance in the Application at a time. Use the **getDBlayerFactory()** method to obtain an instance of RMIDBLayerFactory.

The source is in `net.sf.plantlore.middleware.RMIDBLayerFactory.java`.

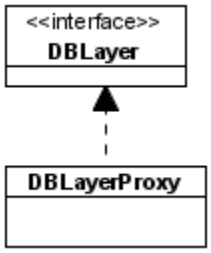


Picture 33: RMI DBLayer Factory

9.1.4. DBLayerProxy

The Database Layer Proxy is a wrapper of a spinoff of the Dabatase Layer. When the DBLayerProxy is asked to wrap a newly created Database Layer, it first creates its spin off, which is a safety object that ensures that every call will be performed in its own thread. And after that it wraps the spinoff and ads a verification that methods of the Database Layer are safe to call, which they generally are until the Logout is performed.

The source is in the `net.sf.plantlore.client.login.Login.java#DBLayerProxy`.



Picture 34: DBLayer Proxy

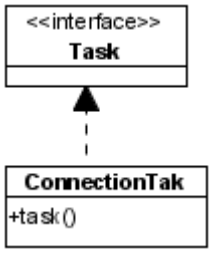
9.1.5. Connection Task

The Connection task extends the Task. To learn more about tasks see the Section **Tasks and Dispatcher**. It can create and initialize a new Database Layer using the supplied information and the DBLayerFactory. It instructs the DBLayerProxy to wrap the newly created Database Layer from the Database Layer Factory.

The second important task is to notify the Application that a new Database Layer has been created and instruct them to obtain it from Login. See the Section **The GUI Communication** for futher details.

The source is in the

`net.sf.plantlore.client.login.Login.java#ConnectionTask`.

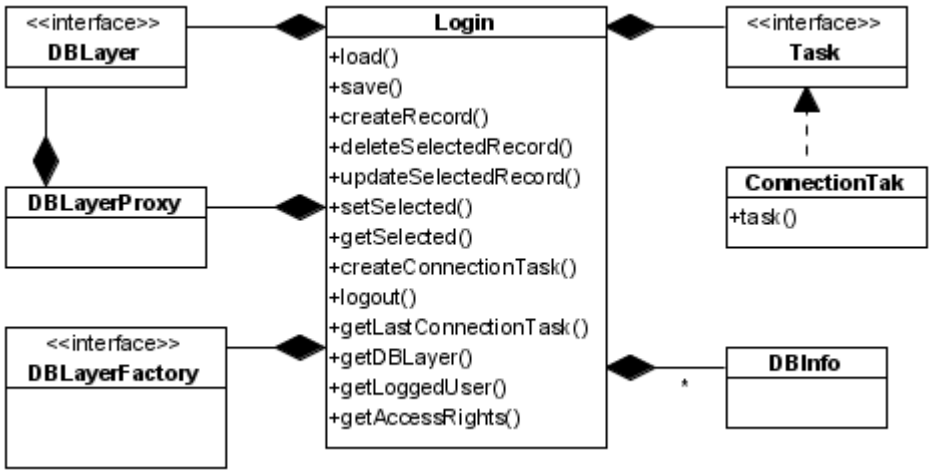


Picture 35: Connection Task

9.1.6. Login

Login acts mostly as a Connection Task factory. It stores information required for the task creation and creates it when it is asked to. Login is equipped with a Database Layer factory that is used to obtain the Database Layer in the Connection task and then it is wrapped by the DBLayerProxy. Every time Login is asked to return the newly created DBLayer, the DBLayerProxy is returned in its stead. All parts of the Presentation Layer work with this wrapper.

The source is in the `net.sf.plantlore.client.login.Login.java`.



Picture 36: Login

Login is also responsible for the management of DBInfos. It can create the DBInfo, alter it, or delete it.

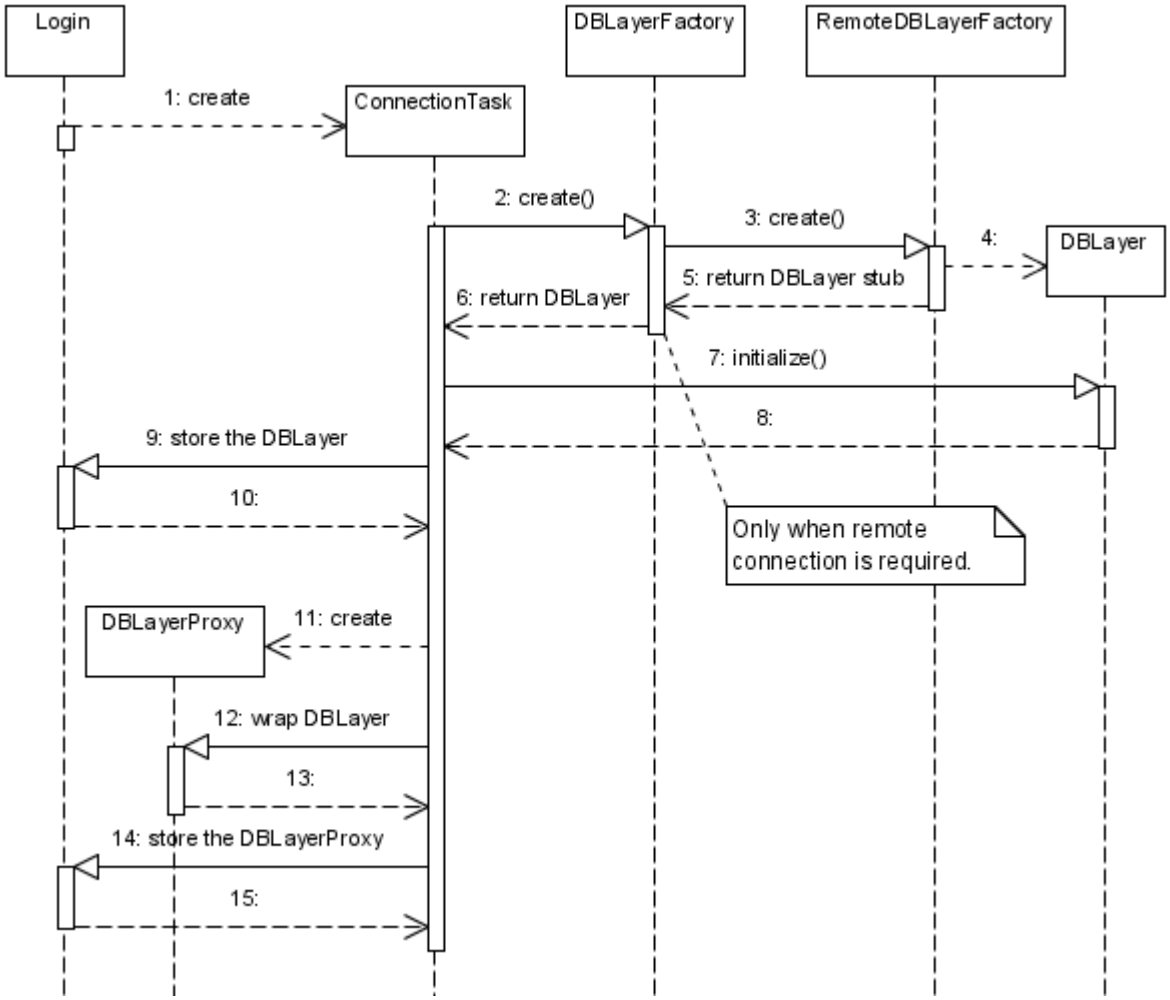
The last responsibility is to perform a proper logout, that is take the Database Layer as it was returned from the DBLayer Factory and ask that DBLayerFactory to destroy it. After that the Database Layer is not accessible.

9.2. Interaction

In this section we will depict the exact process of the new Database Layer creation and its destruction.

9.2.1. Creating and Initializing a New Database Layer

In the picture below there you can see how the remote Database Layer is created.



Picture 37: Creation and Initialization of the Database Layer

9.2.2. Destroying the Database Layer

The destruction of the Databse Layer.

The DBLayerProxy is instructed to wrap null which is a signal to reject all attempts to communicate with the wrapped Database Layer.

10. 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 listeners and the Plantlore „bridges“ (see chapter about MVC and the GUI communication layer).

10.1. The AppCore Model

The AppCore model initially loads data required by AddEdit and Search dialogs from the database. It would be very inefficient to load these data from the database every time the dialog is invoked. Therefore the loading takes place only when needed (when notified by the bridges). 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.

10.2. 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 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.

10.3. 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.

11. 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 correct properties file. Which property file to choose is determined in the main Plantlore class which obtaines them either via the Java Preferences class stored locale, or from the current locale, and initializes the L10n class with it. 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 Plantlore Client connects to a Plantlore Server the Database Layer created on that server is also initialized with the Client's locale. That way Plantlore is able to display even error messages spawned in the Database Layer in the desired locale.

11.1. 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.

11.2. Conventions

The structure of property keys reflects the package hierarchy. Mnemonics are denoted by ampersand before the desired character in property files. The method **L10n.getMnemonic()** returns the mnemonic for the given key. The method **L10n.getString()** 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.

12. Add, Edit and Search

The Add/Edit and Search dialogs belong to the most complex ones in Plantlore and not only from the programming perspective but mostly because they play a key role in day-to-day use and the work with them has to be very effective. The whole Add/Edit dialog has been refactored significantly at least twice during the development process. Much tedious work has been invested into the usability of these dialogs.

12.1. Introduction

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 proved an insufficient solution because of the useless distinction between basic and extended data. However, this distinction proved useful in the Search dialog. There are usually just a few conditions the User wishes to exploit to search and the rest remains almost untouched. The Add/Edit dialog also contains a built-in AddEditSettings MVC handling the enabling and disabling of AddEditView's fields.

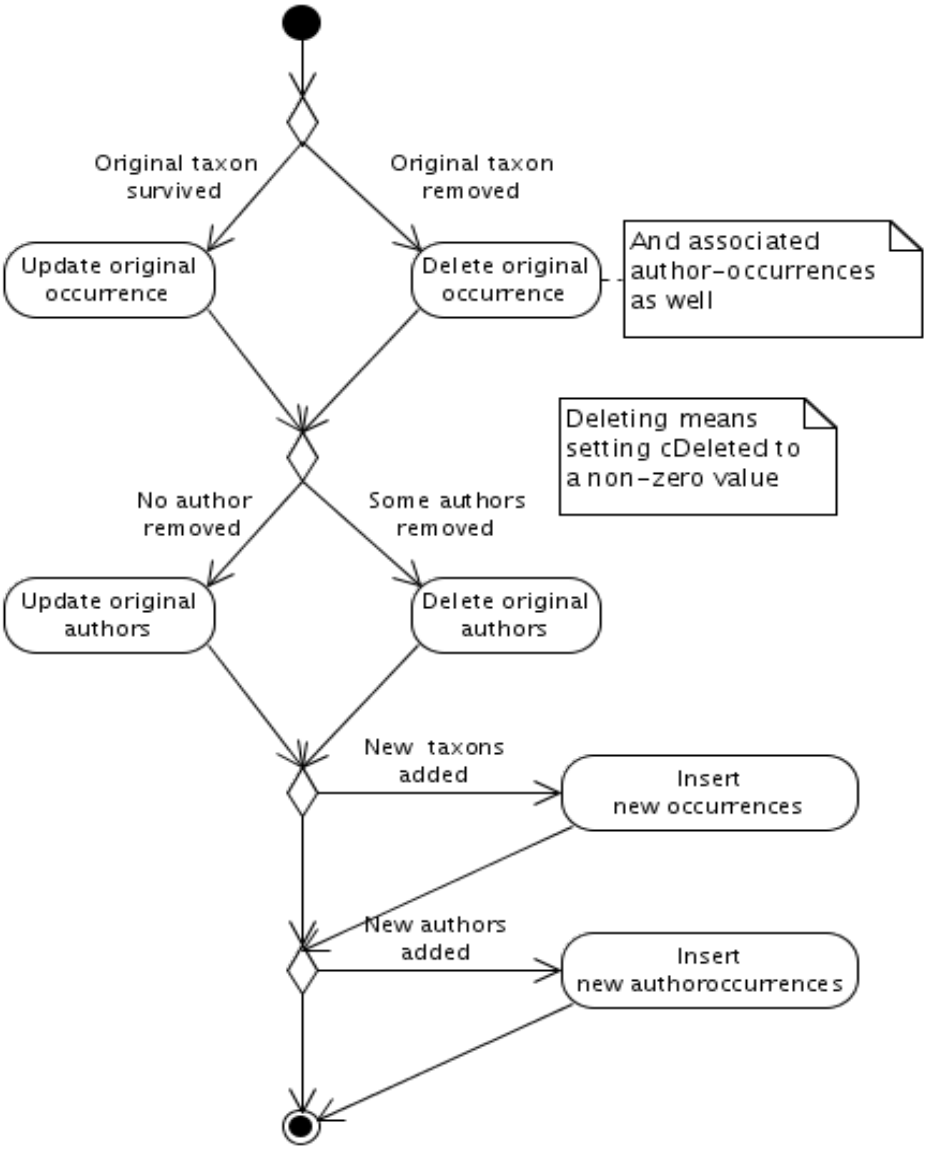
12.2. Input Checking

The User has to work a lot with these dialogs and therefore it is likely he will make mistakes. 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 the inserted text. Special DocumentListeners 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 would turn red after typing four numbers in a row even if more than four numbers are allowed in the integeral part of the number.

12.2.1. The Core

The core of the Add/Edit dialog is the **AddEdit.storeRecord()** method that does the actual work with the database using the Database Layer. It is very important that it must abide byt the rules posed by History. To learn more about these rules see the Section **History**. It has to set the cDeleted columns to proper values and call executeInsert* and executeUpdate* methods properly so that history is or is not saved for that action. The **storeRecord()** method uses helper method both from the Add/Edit model itself and from the DatabaseLayerUtils. These methods are simple record based manipulation helper methods. They include for example: **getHabitatSharingOccurrences()**, **loadHabitatSharingOccurrences()**, **lookupPlant()**, **prepareNewOccurrence()**, **prepareOccurrenceUpdate()**, **DBLayerUtils.getObjectFor()**, etc.

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



Picture 38: Storage of a Record

13. Export

Export is one part of the Presentation Layer. Export "takes" the data from the database and stores 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.

13.1. Introduction

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.

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.

13.2. The Participants

There are several participants; they can be found in the net.sf.plantlore.client.export package.

13.2.1. Builder

The interface contains several methods that are used to build the output appropriately. The source is in the file `net.sf.plantlore.client.export.Builder.java`.

Picture 39: Builder

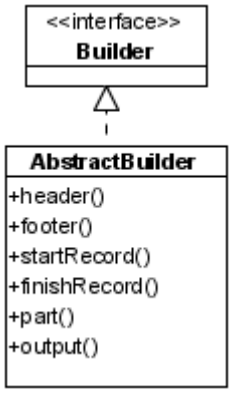
The Occurrence record is quite complicated and there are M:N relations in the database model. In order to process the Occurrence properly, it must be supplied to the Builder in parts. Those parts could be easily identified using the Java's reflection API, therefore it is not necessary to have several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

13.2.2. AbstractBuilder

AbstractBuilder adds (mostly empty) implementation to several methods of the interface and introduces new method that is used for the output. The advantage of the AbstractBuilder is that it decomposes the record into its attributes and uses the nethod **output()** to create the output. This way of implementing an operation is known as the Template Method Pattern.

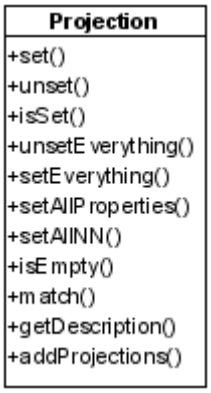


Picture 40: Abstract Builder

13.2.3. Projection

The Projection is a simple wrapper of a java.util.Collection. It provides a very easy way of setting and unsetting a column and testing, whether a column was selected or not. It can also apply the selected list of columns as projection for the supplied query. To learn more about projections see the Section **Database Layer**. After the projections are added the caller should obtain the list containing the information which columns were projected in what order.

The source is in the file `net.sf.plantlore.client.export.Projection.java`.

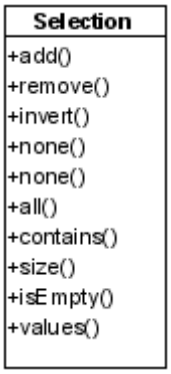


Picture 41: Projection

13.2.4. Selection

The Selection stores the list of selected records and allows us to quickly determine whether a record was selected or not.

The source is in the file `net.sf.plantlore.common.Selection.java`.



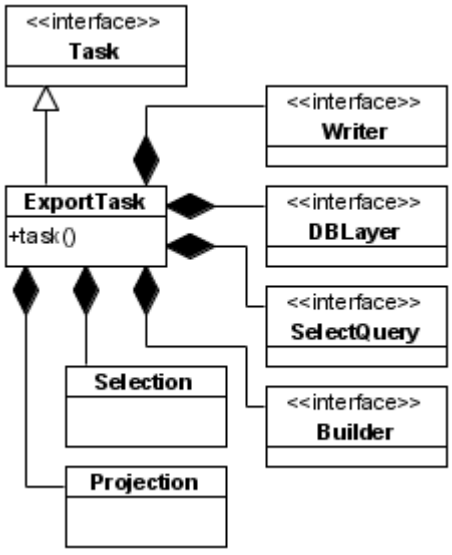
Picture 42: Selection

13.2.5. ExportTask

The Export task is the Director that manages the whole export. Export task requires the last Search query that will be used to iterate over, the Selection that contains the selected records from the query (if any), and the Builder that constructs the output. It also requires Projection in case it should start using projections. The Writer is the interface for file operations - in this file the output will be stored.

The Export Task is implemented as a Task. To learn more about tasks see the Section **Tasks and Dispatcher**.

The Export Task is in the `net.sf.plantlore.client.export.ExportTask.java` file.



Picture 43: Export Task

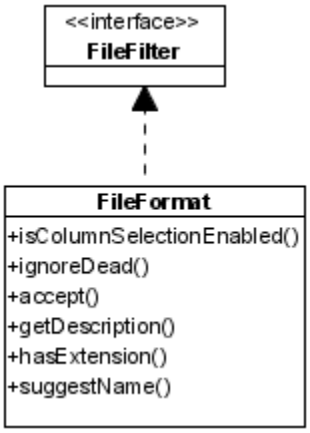
13.2.6. FileFormat

The File Format stores some information about a supported format of a file - whether all columns should be always exported, whether ignore records marked as dead, and the list of suitable extensions. The File Format implements File Filter which is a Java interface that must be

implemented if programmer wants the filter to be used in the JFileChooser component.

The File Format is in the

`net.sf.plantlore.client.export.component.FileFormat.java` file.



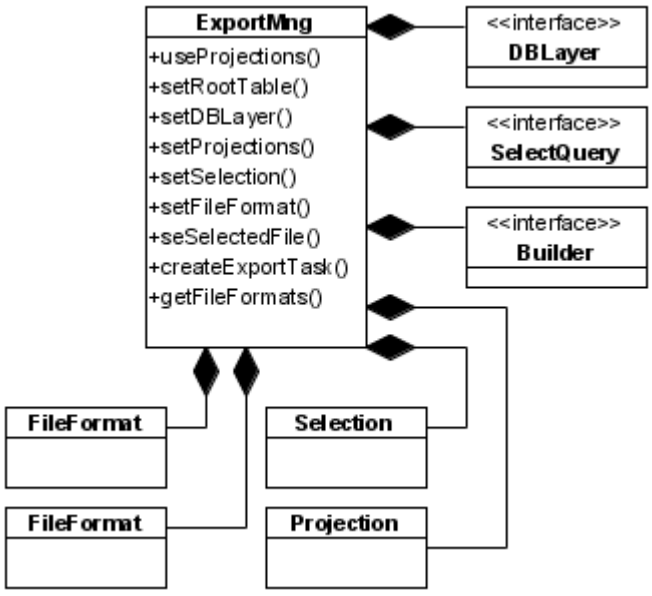
Picture 44: File Format

13.2.7. ExportManager

The Export Manager 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 decision which Builder will be used is based on the supplied File Format. The Export Manager also stores a list of all available File Formats that can be handled by Builders known to the Export Manager.

The source is in the `net.sf.plantlore.client.export.ExportMng.java` file.



Picture 45: Export Manager

13.2.8. 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. When 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.

These Builders implement the Builder interface and are in the

`net.sf.plantlore.client.export.builders` package.

13.2.9. Comma Separated Values Builder

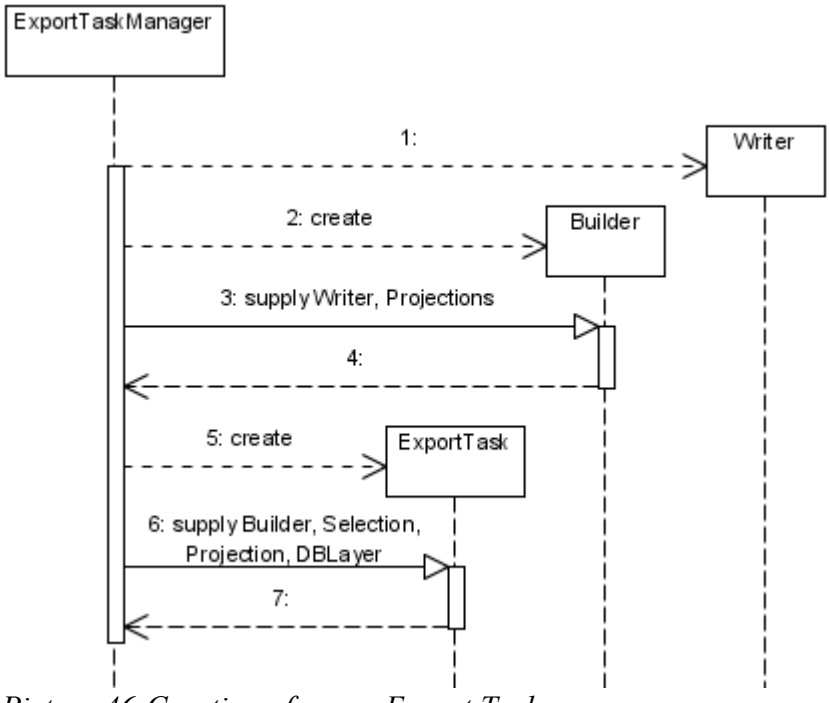
The CSV Builder places each Occurrence record on a separate line. If there is more than one Author associated with the Occurrence record, the Occurrence is repeated for every Author. That is: an Occurrence having *n* authors will be repeated three times, each time with a different Author.

This Builder extends the AbstractBuilder and can be found in the

`net.sf.plantlore.client.export.builders` package.

13.3. Interaction

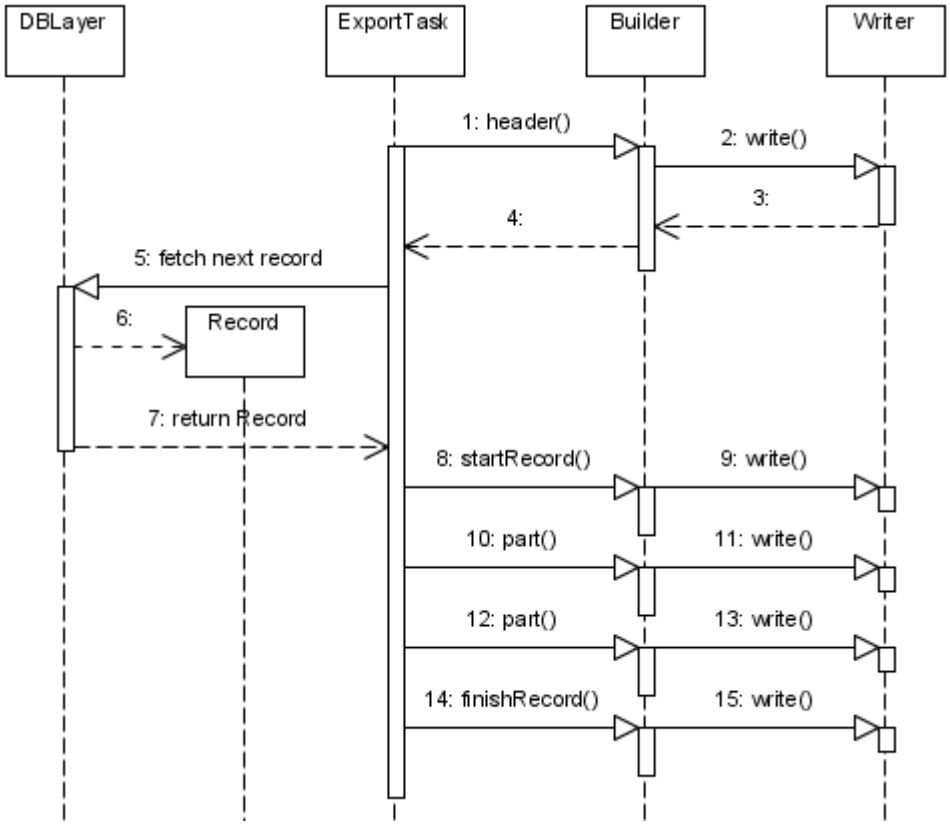
13.3.1. The Creation of All Participants



Picture 46:Creation of a new Export Task

13.3.2. Start of the Export Task

The start of the Export task - the Header is created and then repeatedly actions 5-15. are taken for every selected record.



Picture 47: Export Task in Progress

14. Occurrence Import

Occurrence Import is used to "take" the Occurrence data from a file and store them back into the database. The whole operation must be performed very carefully so that the integrity of data that are already in the database is not violated.

Occurrence Import data can be stored in XML format only, but the framework allows simple creation of new Parsers.

14.1. Introduction

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.

The most important difference between SAX and DOM is, that SAX does not construct any tree at all. It merely reports events that happened while parsing the file.

Because SAX is based on callbacks, the architecture must have been 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.

14.2. The Participants

14.2.1. Record Processor

The Record Processor must be able to 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 must also always make an effort to find a match in the database for every subrecord of the Occurrence - and reuse existing subrecords in the database whenever possible.

The Record Processor is in the file

`net.sf.plantlore.client.occurrenceimport.RecordProcessor.java`.

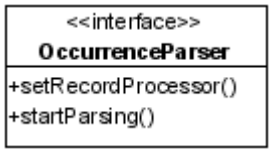
Picture 48: Record Processor

14.2.2. Occurrence Parser

The Occurrence Parser parses the input and reconstructs Occurrence records from it. Every time a record is fully reconstructed, the Record Processor must be invoked on this record.

The interface is in the file

`net.sf.plantlore.client.occurrenceimport.OccurrenceParser.java`.



Picture 49: Occurrence Parser

14.2.3. XML Occurrence Parser

The XML Occurrence Parser utitilizes the SAX framework. In order to do it, the XML Occurrence Parser must extend the Default Handler and override its methods. These methods are those callbacks called by SAX. There the Occurrence record is gradually reconstructed.

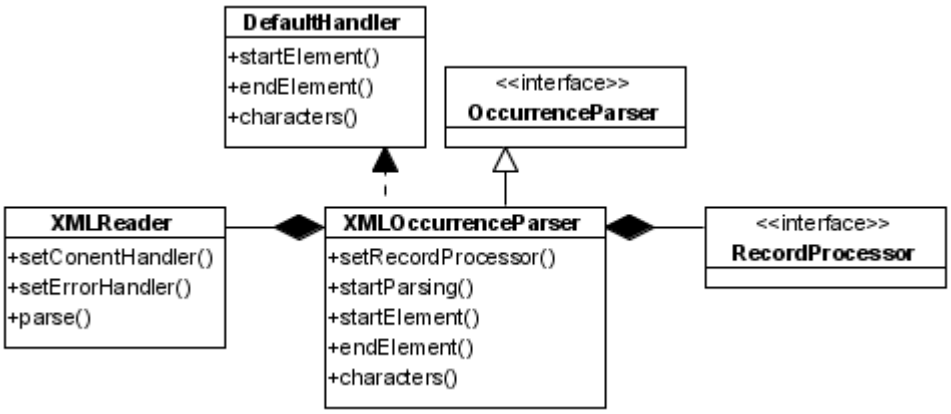
After the record is complete, it is handed over to the Record Processor.

The XML Reader is part of the SAX interface where our implementation of the DefaultHandler must be registered and where the parsing should start.

The source is in the file

`net.sf.plantlore.client.occurrenceimport.`

`parsers.XMLOccurrenceParser.java`



Picture 50: XML Occurrence Parser

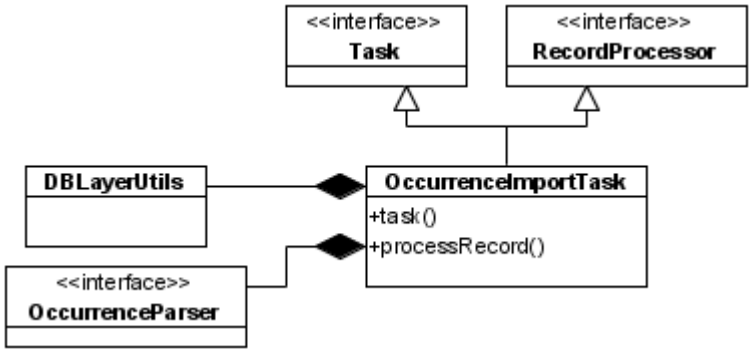
14.2.4. Occurrence Import Task

The Occurrence Import Task is a Plantlore Task that must be dispatched accordingly. To learn more about tasks see the Section **Tasks and Dispatcher**.

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 Occurrence Import Task is in the file

`net.sf.plantlore.client.occurrenceimport.OccurrenceImportTask.java`.

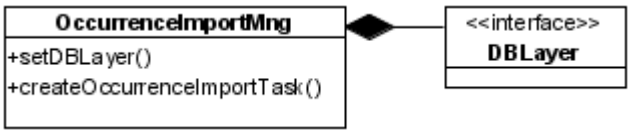


Picture 51: Occurrence Import Task

14.2.5. The Occurrence Import Manager

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

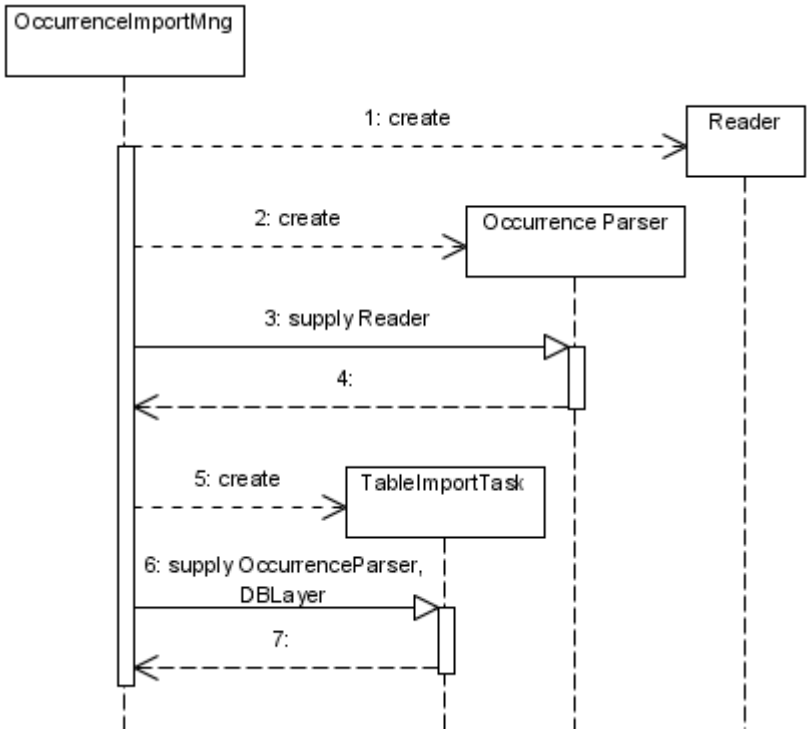
`net.sf.plantlore.client.occurrenceimport.OccurrenceImportMng.java`.



Picture 52: Occurrence Import Manager

14.3. The Interaction

14.3.1. Creating a New Occurrence Import Task



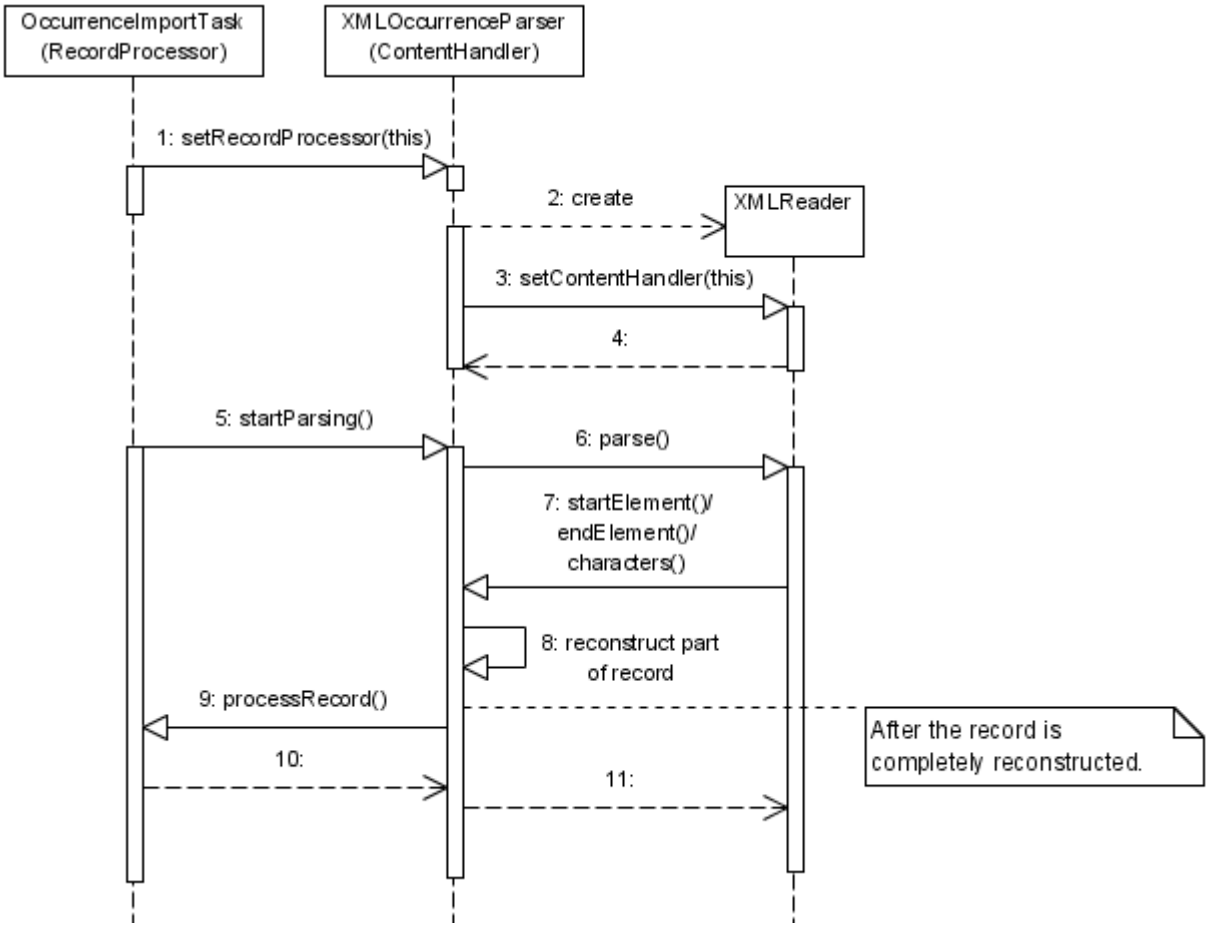
Picture 53: Creating a New Occurrence Import Task

14.3.2. Starting the Occurrence Import Task

The start of work of the Occurrence Import Task is quite complicated because several interfaces are merged together.

First, the XML Occurrence Parser is supplied a Record Processor. The XML Occurrence Parser creates a new XMLReader and registers itself as the Content Handler. The XMLReader will parse the input and send the SAX events back to the ContentHandler.

Second, start of the parsing is initiated from the Occurrence Import Task and comes to the XML Occurrence Parser who starts the XMLReader. The XMLReader starts parsing the input and reports events back to the XML Occurrence Parser which is also the Content Handler. The XML Occurrence Parser reconstructs the record and once the reconstruction is finished, the Occurrence Import task is called to process the record, because the Occurrence Import task also happens to be the Record Processor.



Picture 54: Starting the Occurrence Import Task

15. Import - Immutable Tables

Table Import is a procedure modifying otherwise immutable tables. This procedure is not expected 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, territories, and metadata.

15.1. Introduction

Since the Import to Immutable tables 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 an "inverse" Builder pattern: instead of a Builder there is a Parser. The difference is that the Director asks the Parser to return reconstructed records, and stores them into the database. It's the data flow what is inversed.

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

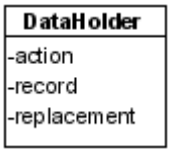
15.2. Participants

15.2.1. Data Holder

Data holder is designed for communication between the Parser and the Director (the Import task in this case). It stores the data retrieved from the file and the intention with this data.

The source is in the file

`net.sf.plantlore.client.tableimport.DataHolder.java`.



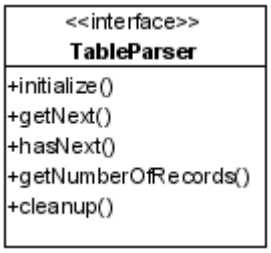
Picture 55: Data Holder

15.2.2. Table Parser

The Table Parser provides the interface for parsers retrieving data from tables. Table Parser must be able to fetch the next record if there is any. Implementation of other methods are optional.

The source is in the file

`net.sf.plantlore.client.tableimport.TableParser.java`.



Picture 56: Table Parser

15.2.3. Unified Table Parser

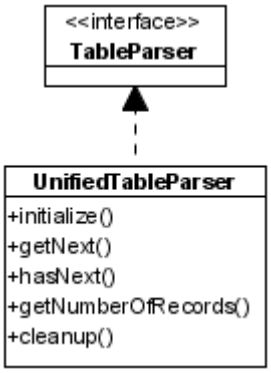
Unified Table Parser can parse and reconstruct any record from the Immutable table. This parser can be used to parse the difference list of plants, as well as nearest bigger seats, territories, villages, and metadata. This is achieved because of the Java reflection API. The **initialize()** method returns the table into which the records will belong, but it is not necessary as the type of the record may be identified any time.

The Unified Table Parser uses DOM4J to parse the XML file which is why all methods **hasNext()**, **getNext()**, and **getNumberOfRecords()** are supported.

The source code is located in the file

`net.sf.plantlore.client.tableimport.`

`parsers.UnifiedTableParser.java`.

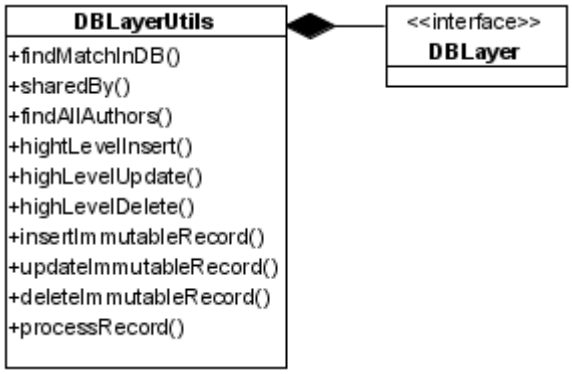


Picture 57: Unified Table Parser

15.2.4. DBLayerUtils

The Database Layer Utils is a set of high-level methods incorporating all rules for records inserting, updating, and deleting posed by the Database Model. It operates with the Database Layer and simplifies and unites most common operations.

The source is in the `net.sf.plantlore.common.DBLayerUtils.java`.



Picture 58: DBLayer Utils

15.2.5. Table Import Task

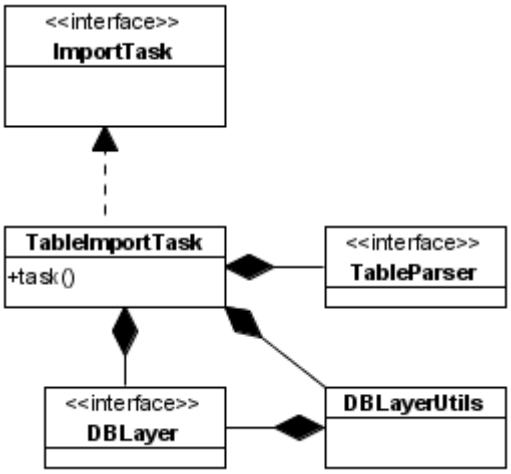
Table Import task is a Plantlore Task that must be dispatched accordingly. To learn more about tasks see the Section **Tasks and Dispatcher**.

The Table Import task acts as a Director in the Builder Design pattern but it asks the Parser for records instead of passing 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.

It also very carefully monitors the state of the heap, because DOM4J can be very memory hungry.

The source is in the

`net.sf.plantlore.client.tableimport.TableImportTask.java`.



Picture 59: Table Import Task

After the Table Import task finishes, the Application is notified so that changes made to the Immutable tables are backpropagated to the rest of the Application, so that the User is presented with up-to-date data. To study the propagation of changes see the Section **GUI Communication Layer**.

15.2.6. Table Import Manager

The Table Import Manager is the Table Import Task factory. It is used to store partial information

about the future Table Import Task and once all the information is supplied, it can create a new Table Parser and Table Import Task.

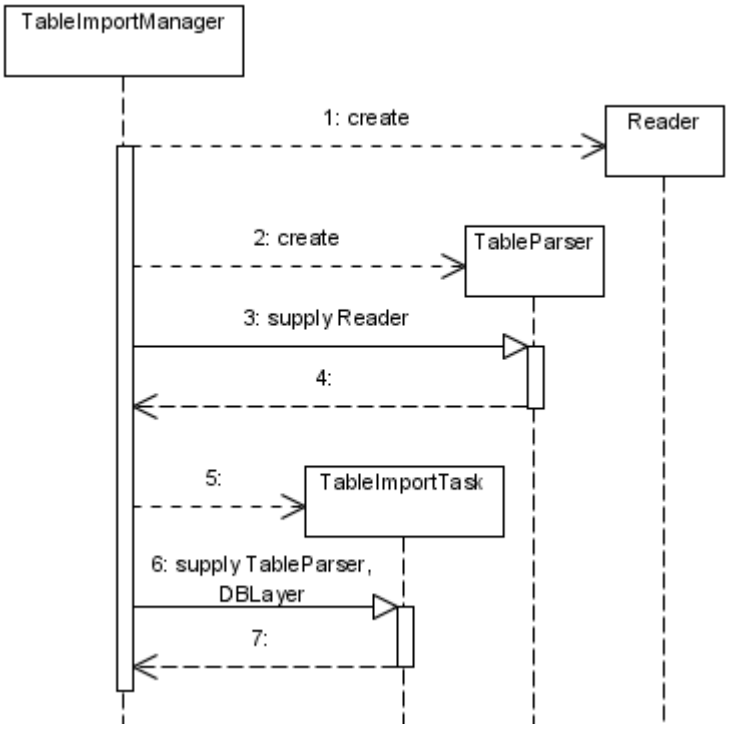
Picture 60: Table Import Manager

15.3. Interaction

In this section we describe the creation of a new Table Import Task and how the Table Import works.

15.3.1. Creation of a New Table Import Task

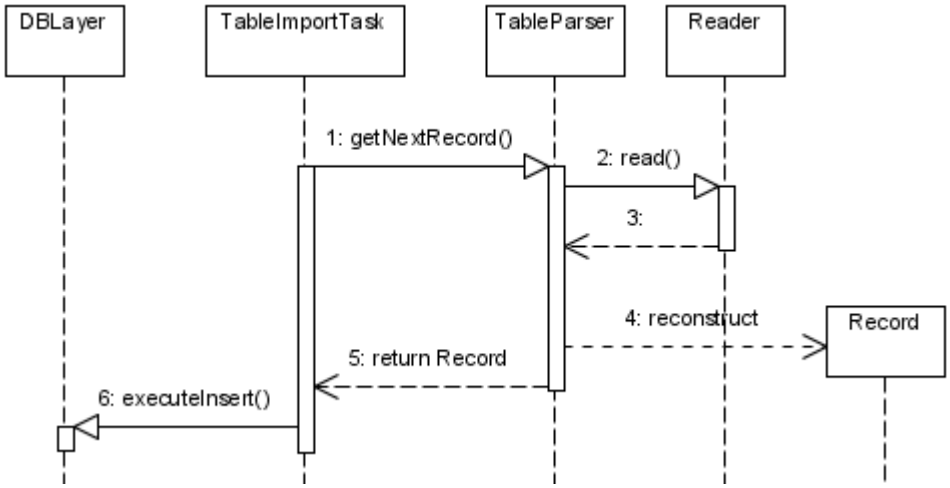
Creation of all participants.



Picture 61: Creating a New Table Import Task

15.3.2. Start of the Import Task

The next picture depicts the process of obtaining, reconstructing, and processing of a record.



Picture 62: Start of the Import Task

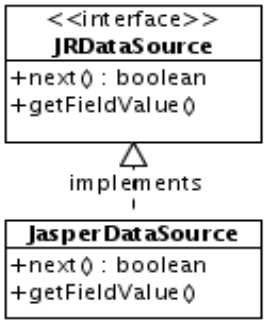
16. Printing and Scheda

One of the 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 problem 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 and we have already had some experience with this library. It also solves the printing because it offers a Printing dialog if desired.

16.1. 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. There are several ways to use the JasperReports library. It is always necessary to create a report. If you have a report you have to obtain a data source from which to get the data to fill the report with. JasperReports supports various datasources: direct JDBC connection using predefined SQL statements, XML datasource, arbitrary data source after implementing the JRDataSource interface.

To utilize JasperReports the their best we have created a JasperDataSource which is a class implementing the JRDataSource interface. The implementation is quite straightforward except the exception handling which will be described later. The JasperDataSource is initialized with the current Database Layer and the Selection object containing the set of Occurrences to be printed (or created schedas from). In the **JasperDataSource.next()** method we iterate over the Occurrences. Specific values of the records are retrieved by the JasperReports engine using the **JRDataSource.getFieldValue()** method.

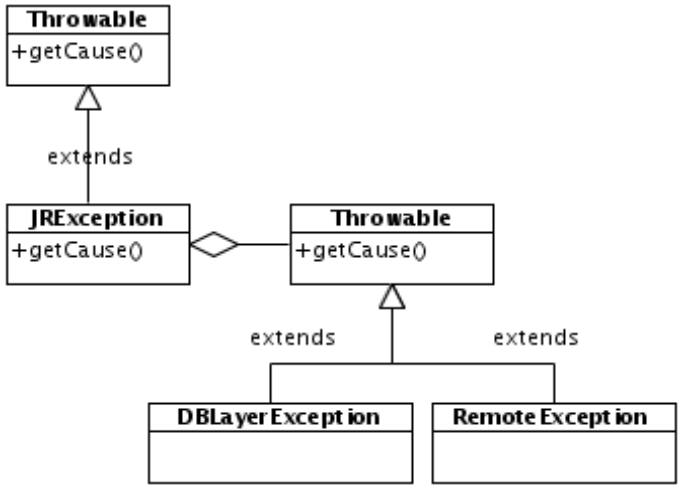


Picture 63: Jasper Data Source

16.1.1. The Exception Handling

It is only possible to throw a JRException from the interface. The problem is that we have to deal with the Database Layer 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 wrap them as a Throwable cause of a newly created JRException. Higher in the AppCoreCtrl or PrintCtrl / Print we extract the cause and throw it again to be handled further in Plantlore using the **DefaultExceptionHandler.handle()** method.

This solution is very similar to the solution used in the Occurrence import in combination with SAX.



Picture 64: Exception Handling in Scheda

16.1.2. The Reports

The reports can be either written manually in an XML file format or using existing GUI tools for rapid report creation. For Plantlore we used the iReport software to create the reports for Scheda and Printing. The reports comprise the XML code combined with Java code. That is a source of great power in creating complex reports. These reports have to be compiled to a so called compiled report. This could be done on the fly using the JasperReports engine but it has two drawbacks. The first is that the user has to have a JDK installed and the second are possible security issues – it could be possible to create malicious harmful reports. Therefore Plantlore contains already compiled reports packed into the application jar file. The reports are loaded dynamically as a resource from the `net.sf.plantlore.client.resources` package.

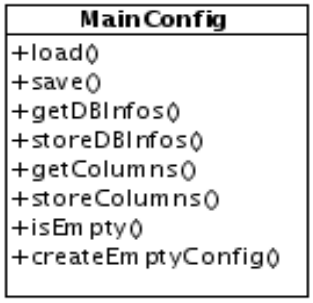
17. Settings

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

17.1. Participants

17.1.1. 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 extensions. This configuration file, called plantlore.xml, is stored within the User's home directory in its own directory called `plantlore`. On linux system we prepend a dot to make the directory hidden as it is the habit on this platform. The file contains a list of columns the User selected He wants to see in AppCoreView. It also holds information about 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.



Picture 65: MainConfig

17.1.1.1. MainConfig Structure

The root element of the main configuration file 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 <column> tags that bear the information about the columns the User chose. The <login> tag may contain arbitrary number of <triplet> tags 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.

17.1.1.2. MainConfig Example

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

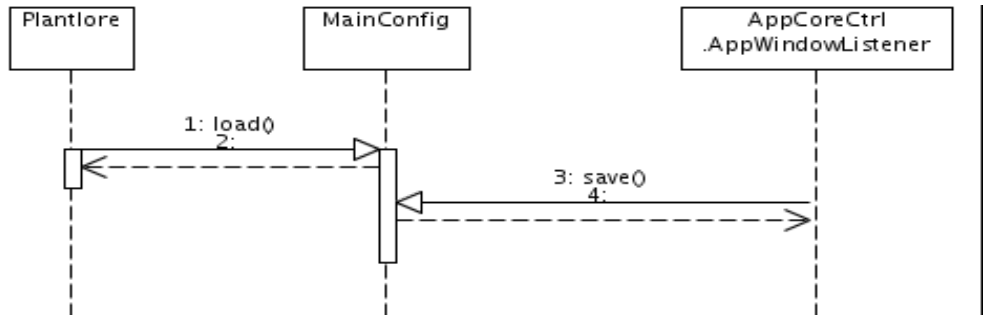
<config>

```
<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>
```

17.1.1.3. Loading and Saving

The main configuration file is created if it doesn't exist. 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.



Picture 66: Loading and Saving the Configuration

17.2. Other Settings

Plantlore also needs to store some petty and 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 to use 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.

17.2.1. Convention

The convention when using preferences is to always obtain the Preferences object for the class that

is interested in storing the preferences. For example for L10n the call would be **Preferences.userNodeForPackage(L10n.class)**.

18. History

History of changes monitors and records changes made to the records in the database during insert, update and delete. It is capable of restoring either the database or the record into its previous state.

18.1. Introduction

The History monitors all operations with the database. It is incorporated into the Database Layer. You can find it is called in every version of methods **executeInsert**, **executeUpdate**, **executeDelete** except those that contain the suffix History.

Thus the History is usually saved when working with the Application. There are several cases when History is not saved

1. During Table Import - data in the Immutable tables TPlants, TVillage, TPhytochoria, and TTerritoriess are modified without saving the changes; it is the Table Import that should be used to update the contents of those tables.
2. When adding/editing Users of the database.

The following table shows the list of tables and columns for which the History is recorded.

Table	*Columns*
tOccurrences	cYearCollected, cMontCollected, cDayCollected, cTimeCollected, cDataSource, cHerbarium, cNote, cHabitatId, cPlantId, cPublicationId, cMetadataId
tAuthorsOccurrences	cRole, cNote
tAuthors	cWholeName, cOrganization, cTelephoneNumber, cRole, cAddress, cEmail, cURL, cNote,
tHabitats	cQuadrant, cDescription, cCountry, cAltitude, cLatitude, cLongitude, cNote, cTerritoryId, cPhytochoriaId, cNearestVillageId
tMetadata	cTechnicalContactName, cTechnicalContactEmail, cTechnicalContactAddress, cContentContactAddress, cContentContactEmail, cContentContactAddress, cDataSetDetails, cDatasetTitle, cSourceInstitutionId, cSourceId, cOwnerOrganizationAbbrev, cRecordBasis, cBiotopeText

The changes are stored right into the databas. There are three tables that are used for it - they are tHistory, tHistoryChange and tHistoryColumn. The detailed description of these tables can be found the the Section **Database Model**. In short:

1. tHistoryColumn is a table with fixed contents. It contains pairs <Table, Column> for each table and column for which the History is recorded.
2. tHistoryChange contains details about the record that was changed (cRecordID) - the type of operation (cOperation), the date and time of the operation (cWhen), and the User who performed the operation (cWho).
3. tHistory contains information about the column that was changed (the cColumnId is a foreign key to the tHistoryColumn table - identifying the name of the table and the name of

the column that was changed) and contains both the new value and old value in that column (cOldValue and cNewValue). In order to know when the change occurred and who made it to which record, there is a reference to the record in the tHistoryChange (cChangedId).

18.2. Participants

There are two participants this time. The first one is the Database Layer itself that records the changes in all its methods starting with the prefix `**execute**` that do not have the suffix `**History**`.

The second one is History.java, that can restore the state of records in the database.

We will describe how exactly the History is saved.

18.2.1. Hibernate Database Layer

The Database Layer stores changes to the tHistory and tHistoryChange tables. We will discuss the values that can be in their columns.

The source is in the file `net.sf.plantlore.server.HibernateDBLayer.java`.

These columns **always** contain the same values:

Table.Column	*Value*
tHistoryChange.cWho	The User that performed the operation
tHistoryChange.cWhen	The date and time of the operation
tHistoryChange.cOperation	1 = insert; 2 = update; 3 = delete
tHistory.cChangeId	The cID of the record in the tHistoryChange that describes the record that has been modified (its cID, when did the change happen, who made the change, and the type of the change).

18.2.1.1. Insert

These columns always contain the same values when inserting a new Record:

Table.Column	*Value*
tHistoryChange.cRecordId	The cID of the inserted record.
tHistory.cColumnId	The cID of the record in the tHistoryColumn that satisfies: cTable = Record & cColumn = null (e.g. cTable = Habitat & cColumn = null)
tHistory.cOldRecordId	0
tHistory.cOldValue	null
tHistory.cNewValue	null

The only exception is the Insert of a new record into the tAuthorsOccurences table.

1. In case the insert occurs when inserting a new Occurrence, the History should not be recorded - **executeInsertHistory()** or **executeInsertInTransactionHistory()** should be used!
2. In case the insert occurs when adding a new AuthorOccurrence to an already existing Occurrence, the History should be recorded - **executeInsert()** or

executeInsertInTransaction() should be used. Besides tHistoriChange.cOperation = 2.

18.2.1.2. Update

These columns always contain the same values when updating an existing Record:

Table.Column	*Value*
tHistoryChange.cRecordId	The cID of the updated record.
tHistory.cColumnId	The cID of the record in the tHistoryColumn that satisfies: cTable = Record & cColumn = Name of the updated column. (e.g. cTable = Habitat & cColumn = Description)
tHistory.cOldRecordId	If the Record contains some foreign keys and the foreign key was updated to another foreign key, the original value of the foreign key is stored. Otherwise this column is 0.
tHistory.cOldValue	The old value contained in the column.
tHistory.cNewValue	The new value that replaced the old one.

18.2.1.3. Delete

These columns always contain the same values when deleting an existing Record:

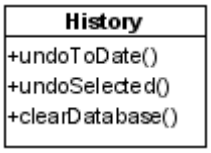
Table.Column	*Value*
tHistoryChange.cRecordId	The cID of the deleted record.
tHistory.cColumnId	The cID of the record in the tHistoryColumn that satisfies: cTable = Record & cColumn = null (e.g. cTable = Habitat & cColumn = null)
tHistory.cOldRecordId	0
tHistory.cOldValue	null
tHistory.cNewValue	null

Please note that deletion of AuthorOccurrences is not recorded by History, at least not directly. In case an Occurrence is marked as deleted (i.e. Occurrence.cDelete = 1), all associated AuthorOccurrences that have their cDelete = 0 should be marked as deleted but with the AuthorOccurrence.cDelete = 2. This way if the Occurrence is undeleted, it would be clear which AuthorOccurrences should be undeleted as well.

18.2.2. History

History can undo the changes made to the database. Most important methods are **undoToDate()** and **undoSelected()**. Because the list of stored changes can become very large, History provides methods to dispose of the whole History - **clearDatabase()**.

The source is in the file net.sf.plantlore.client.history.History.java.



Picture 67: History

19. Data managers

19.1. Introduction

Plantlore manages additional data associated with occurrences as well. These include: Authors, Publications, Metadata and Users. There is a special manager for each of these data types. All of these managers share common structure which is outlined in this section.

We will discuss only the model (of the MVC) of the data managers, for the description of MVC framework see the Section **MVC**.

19.2. Initialization

Every manager is created by the AppCore object and receives an instance of Database Layer from the AppCore as an argument to the model constructor. This instance of Database Layer is saved and used for executing database operations.

AppCore is also registered as an observer of the manager. This is neccessary in order to inform AppCore that the managed data (authors, publications etc.) has changed and Plantlore's internal caches of these records need to be refreshed.

19.3. Executing Database Operations

All of the database operations (insert, update, delete, search) are enclosed in a separate Task and executed in a new thread while GUI is showing a progress bar. Every manager keeps one database connection open for executing the search query and displaying the results). For further information see the JavaDoc documentation of the managers.

20. Conversion of Coordinates

In the Add/Edit dialog the User is offered a possibility to specify the location exactly by supplying the GPS coordinates. There are several valid GPS formats in the Czech republic:

1. World Geodetic Reference System 1984 (WGS84).

   This coordinate system is defined by a set of ground stations and locations of the GPS satellites. Its used is not limited for the Czech republic - it can be used anywhere on Earth.

2. The Coordinate System of the United Trigonometric Territorial Network (Souřadnicový systém Jednotné trigonometrické sítě katastrální, S-JTSK).

   This is a planar coordinate system. The latitude and longitude are projected on the Bessel's ellipsoid.

3. The Coordinate System 1942 (S-42).

   This is another planar system. This system is based on the Gauss Planar Conform Cylindric Projection which is based on the Krasov's ellipsoid.

The Plantlore Client supports these three coordinate systems and can convert coordinates between them. The default coordinate system is the WGS84.

Plantlore can convert from WGS-84 to S-JTSK and back, from WGS84 to S-42 and back. The conversion from S42 to S-JTSK and back is solved using the mutual system WGS84.

The particular algorithms are extremely sophisticated and require a thorough knowledge of calculus. The description of the conversion can be found in the literature.

21. 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

21.1. 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.***

21.2. 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 .

21.3. 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/.

21.4. 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.

21.5. 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.

21.6. 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.

21.7. Description of the Database Model Tables

21.7.1. 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.

21.7.2. 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.

21.7.3. 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.

21.7.4. 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.

21.7.5. 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.

21.7.6. 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.

21.7.7. 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).

21.7.8. 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.

21.7.9. 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.

21.7.10. 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.

21.7.11. 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".

21.7.12. 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.

21.7.13. 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.

21.7.14. 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.

21.7.15. 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.

21.7.16. 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.

21.8. 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.

21.9. 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.

22. BioCase Mapping

22.1. Mandatory Items

22.1.1. Darwin Core 2

- RecordSet/Record/CatalogNumber – *v0ccurrences.cId*
- RecordSet/Record/CollectionCode – *vMetadata.cDataSetTitle*
- RecordSet/Record/DateLastModified – *vMetadata.cDateModified*
- RecordSet/Record/InstitutionCode – *vMetadata.cOwnerOrganizationAbbrev*
- RecordSet/Record/ScientificName – *tPlants.cTaxon*

22.1.2. ABCD 1.20

- DataSets/DataSet/DatasetDerivations/DatasetDerivation/DateSupplied – *vMetadata.cDateCreate*
- DataSets/DataSet/OriginalSource/SourceInstitutionCode – *vMetadata.cSourceInstitutionId*
- DataSets/DataSet/OriginalSource/SourceLastUpdatedDate – *vMetadata.cDateModified*
- DataSets/DataSet/OriginalSource/SourceName – *vMetadata.cSourciId*
- DataSets/DataSet/Units/Unit/UnitID – v*Occurrences.cId*

22.1.2. ABCD 2.06

- DataSets/DataSet/ContentContats/Name – *vMetadata.cContentContactName*
- DataSets/DataSet/Metadata/Description/Representation/Title – *vMetadata.cDataSetTitle*
- DataSets/DataSet/Metadata/RevisionData/DateModified – *vMetadata.cDateModified*
- DataSets/DataSet/TechnicalContacts/TechnicalContact/Name – *vMetadata.cTechnicalContactName*
- DataSets/DataSet/Units/Unit/SourceID – *vMetadata.cSourceId*
- DataSets/DataSet/Units/Unit/SourceInstitutionID – *vMetadata.cSourceInstitutionId*
- DataSets/DataSet/Units/Unit/UnitID – v*Occurrences.cId*

22.2. Voluntary Items

22.2.1. Darwin Core 2

- RecordSet/Record/Genus – *tPlants.cGenus*
- RecordSet/Record/ScientificNameAuthorYear - *tPlants.cScientificNameAuthor*
- RecordSet/Record/Country - *vHabitats.cCoutry*
- RecordSet/Record/County - *tVillages.cName*
- RecordSet/Record/StateProvince - *tTerritories.cName*
- RecordSet/Record/Locality - *vHabitats.cDescription*
- RecordSet/Record/DecimalLatitude - *vHabitats.cLatitude*

- RecordSet/Record/DecimalLongitude - *vHabitats.cLongitude*
- RecordSet/Record/MinimumElevationMeters - *vHabitats.cAltitude*
- RecordSet/Record/YearCollected - *vOccurrences.cYearCollected*
- RecordSet/Record/MonthCollected - *vOccurrences.cMonthCollected*
- RecordSet/Record/DayCollected - *vOccurrences.cDayCollected*
- RecordSet/Record/TimeCollected - *vOccurrences.cTimeCollected*
- RecordSet/Record/Collector - v*AuthorCollected.cWholeName*
- RecordSet/Record/Note - *vOccurrences.cNote*

22.2.2. ABCD 1.2

- DataSets/DataSet/DatasetDerivations/DatasetDerivation/Description – *vMetadata.cDataSetDetails*
- DataSets/DataSet/DatasetDerivations/DatasetDerivation/Supplier/Addresses/Addres – *vMetadata.cTechnicalContactAddress*
- DataSets/DataSet/DatasetDerivations/DatasetDerivation/Supplier/EmailAddresses/EmailAddress – *vMetadata.cTechnicalContactEmail*
- DataSets/DataSet/Units/Unit/DateLastEdit – *vOccurrences.cUpdateWhen*
- DataSets/DataSet/Units/Unit/Gathering/GatheringAgent/Organization/OrganizationName – v*AuthorsCollectd.cOrganization*
- DataSets/DataSet/Units/Unit/Gathering/GatheringAgent/Person/PersonName – *tAuthors.cWholeName*
- DataSets/DataSet/Units/Unit/Gathering/GatheringSite/Altitude/MeasurementAtomized/MeasurementLowerValue – *vHabitats.cAltitude*
- DataSets/DataSet/Units/Unit/Gathering/GatheringSite/Country/CountryName – *vHabitats.cCountry*
- DataSets/DataSet/Units/Unit/Gathering/GatheringSite/LocalityText – *vHabitats.cDescription*
- DataSets/DataSet/Units/Unit/Gathering/GatheringSite/Notes – *vHabitats.cNote*
- DataSets/DataSet/Units/Unit/Gathering/GatheringSite/SiteCoordinateSets/SiteCoordinates/CoordinatesLatLong/LatitudeDecimal – *vHabitats.cLatitude*
- DataSets/DataSet/Units/Unit/Gathering/GatheringSite/SiteCoordinateSets/SiteCoordinates/CoordinatesLatLong/LongitudeDecimal – *vHabitats.cLongitude*
- DataSets/DataSet/Units/Unit/Gathering/Project/ProjectTitle – v*Metadata.cDataSetTitle*
- DataSets/DataSet/Units/Unit/Identifications/Identification/Identifier/IdentifierPersonName/PersonName – v*AuthorIdentified.cWholeName*
- DataSets/DataSet/Units/Unit/Identifications/Identification/TaxonIdentified/AuthorString – *tPlants.cScientificAuthorName*
- DataSets/DataSet/Units/Unit/Identifications/Identification/TaxonIdentified/NameAuthorYearString – *tPlants.cTaxon*
- DataSets/DataSet/Units/Unit/RecordBasis – *vMetadata.cRecordBasis*
- DataSets/DataSet/Units/Unit/UnitCollectionDomain/HerbariumUnit/ Exsiccatum – *vOccurrences.cHerbarium*
- DataSets/DataSet/Units/Unit/UnitReferences/UnitReference/ReferenceCitation – *vPublications.cCollectionName*
- DataSets/DataSet/Units/Unit/UnitReferences/UnitReference/ReferenceDetail – *vPublications.cReferenceDetail*
- DataSets/DataSet/Units/Unit/UnitReferences/UnitReference/URL – *vPublications.cURL*

22.2.3. ABCD 2.06

- DataSets/DataSet/ContentContats/Address – *vMetadata.cContentContactAddress*
- DataSets/DataSet/ContentContats/Email – *vMetadata.cContentContactEmail*
- DataSets/DataSet/Metadata/Description/Representation/@language - *„CZ_CS“*
- DataSets/DataSet/Metadata/Description/Representation/Details – *vMetadata.cDataSetDetails*
- DataSets/DataSet/Metadata/Owners/Owner/Addresses/Address – *vMetadata.cContentContactAddress*
- DataSets/DataSet/Metadata/Owners/Owner/EmailAddresses/EmailAddress – *vMetadata.cContentContactEmail*
- DataSets/DataSet/Metadata/Owners/Owner/Person/FullName - *vMetadata.cContentContactName*
- DataSets/DataSet/Metadata/RevisionData/Creators – vMetadata*.cSourceInstitutionId, vMetadata.cSourceId*
- DataSets/DataSet/Metadata/RevisionData/DateCreated – *vMetadata.cDateCreate*
- DataSets/DataSet/TechnicalContacts/TechnicalContact/Address – *vMetadata.cTechicalContactAddress*
- DataSets/DataSet/TechnicalContacts/TechnicalContact/Email – *vMetadata.cTechicalContacEmail*
- DataSets/DataSet/Units/Unit/DateLastEdited – *vOccurrences.cUpdateWhen*
- DataSets/DataSet/Units/Unit/Gathering/Agents/GhateringAgent/Organisation/name/Reservation/@language - *„CZ_CS“*
- DataSets/DataSet/Units/Unit/Gathering/Agents/GhateringAgent/Organisation/name/Reservation/Text – v*AuthorsRevission.cOrganization*
- DataSets/DataSet/Units/Unit/Gathering/Agents/GhateringAgent/Person/FullName – v*AuthorsCllected.cWholeName*
- DataSets/DataSet/Units/Unit/Gathering/Altitude/MeasurementOrFactAtomised/LowerValue – v*Habitats.cAltitude*
- DataSets/DataSet/Units/Unit/Gathering/Country/Name – v*Habitats.cCountry*
- DataSets/DataSet/Units/Unit/Gathering/DateTime/ISODateTimeBegin – *vOccurrences.cISODateTimeBegin*
- DataSets/DataSet/Units/Unit/Gathering/LocalityText - *vHabitats.cDescription*
- DataSets/DataSet/Units/Unit/Gathering/Notes – *vHabitats.cNote*
- DataSets/DataSet/Units/Unit/Gathering/Project/ProjectTitle – *vMetadata.cDataSetTitle*
- DataSets/DataSet/Units/Unit/Gathering/SiteCoordinateSets/SiteCoordinates/CoordinatesLatLong/LatitudeDecimal – *vHabitats.cLatutude*
- DataSets/DataSet/Units/Unit/Gathering/SiteCoordinateSets/SiteCoordinates/CoordinatesLatLong/LongitudeDecimal – *vHabitats.cLongitude*
- DataSets/DataSet/Units/Unit/HerbariumUnit/Exsiccatum – *vOccurrences.cHerbarium*
- DataSets/DataSet/Units/Unit/Identification/Identifiers/Identifier/PersonName/FullName – v*AuthorIdentified.cWholeName*
- DataSets/DataSet/Units/Unit/Identification/Result/TaxonIdentified/ScientificName/FullScientificNameString – *tPlants.cTaxon*
- DataSets/DataSet/Units/Unit/Identification/Result/TaxonIdentified/ScientificName/NameAtomised/Bacterial/GenusOrMonomial – *tPlants.cGenus*
- DataSets/DataSet/Units/Unit/RecordBasis – *vMetadata.cRecordBasis*
- DataSets/DataSet/Units/Unit/UnitReferences/UnitReference/CitationDetail –

vPublications.*cReferenceDetail*

- DataSets/DataSet/Units/Unit/UnitReferences/UnitReference/TitleCitation – *vPublications.cCollectionName*
- DataSets/DataSet/Units/Unit/UnitReferences/UnitReference/URI – *vPublications.cURI*

23. Log4j Logger Introduction

23.1. 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 assigned a level. In case you don't provide one log4j assigns 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.

23.2. 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.