Puzzle ITC GmbH
Maulbeerstrasse 15
CH-3011 Bern
Switzerland

Phone: +41 (0)31 398 44 77
Fax: +41 (0)31 398 44 78

www.puzzle.ch

openDBcopy - Developer Manual

http://opendbcopy.org

openDBcopy is an Open Source Project hosted at SourceForge
http://www.sourceforge.net/projects/opendbcopy/

Release	0.51
Date	26.07.04
Reference	user-manual-0.51

ch20040728w5874

Document History

Date	Version	Person	Comments
26/07/04	0.51	A. Smith	First issue
28/07/04	0.51	A. Smith	Further chapters added and reviewed

Table of Contents

1 Overview

openDBcopy is an open source project by Anthony Smith, Puzzle ITC GmbH. The project is hosted at Sourceforge and published under the terms of the GNU General Public License.

openDBcopy must be understood as a framework to configure and execute plugins. A plugin can do ANYTHING. Some plugins are database specific, others provide features to do file manipulations and transfers etc. There is no limitation of what a plugin can or could do.

The core framework itself is managed and checked in as one separate project in the CVS Repository at SourceForge. Each plugin is managed in a separate project path within the same CVS Repository.

Please use the public forum to post questions. There are also links to request new features and track bugs. The mailing lists have been deleted because except one person were posting questions using the forum.

For simple plugin development and deployment use an existing plugin.

Name	Website
openDBcopy at SourceForge	http://sourceforge.net/projects/opendbcopy/
openDBcopy Download	http://sourceforge.net/project/showfiles.php?group_id=91406
openDBcopy Forum	http://sourceforge.net/forum/?group_id=91406
openDBcopy Website	http://opendbcopy.org http://opendbcopy.ch
GNU General Public License	http://www.gnu.org/licenses/gpl.txt
Puzzle ITC Website	http://www.puzzle.ch (Puzzle provides a complete Sourceforge mirror :-)

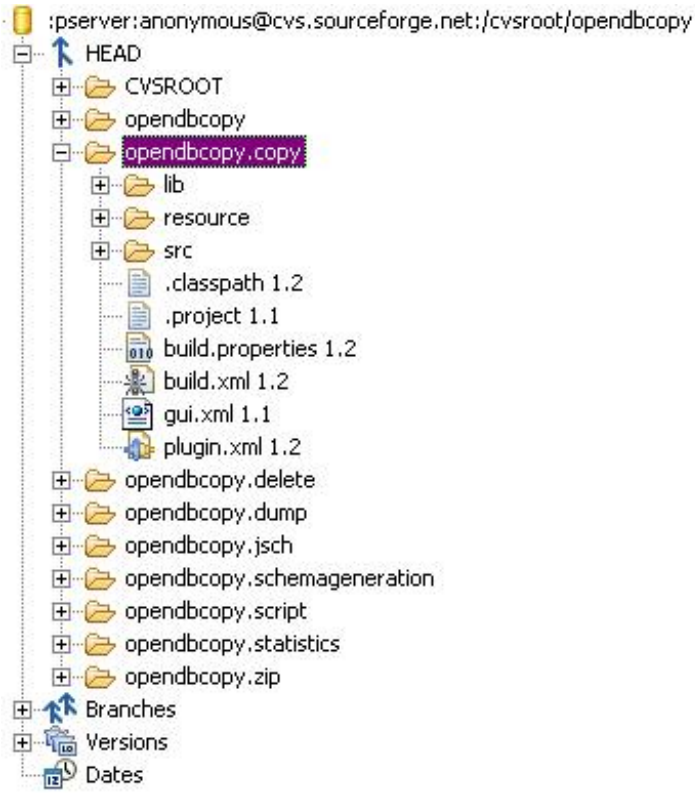
2 Software Requirements

- openDBcopy requires a Java Runtime Environment. If not yet installed, please download an actual Java Runtime Environment or Standard Developer Kit 1.4 from http://java.sun.com.
- To develop and / or debug openDBcopy or plugins you require a Java IDE. Am currently using Eclipse 3.0, see http://www.eclipse.org.
- OpenDBcopy Core Framework and Plugins are built and deployed using Apache Ant. Download Apache Ant (http://ant.apache.org) if you have not yet installed an actual version (1.5 or higher).
- An actual JDBC driver for each database system one plans to access.
- Other libraries required for new plugins

OpenDBcopy 0.5 has been tested using J2SDK 1.4.2_03.
The latest sources can be checked out from the CVS Repository. For details see the project's website.

3 openDBcopy Core Framework and Plugins on the Web

The openDBcopy core framework and plugins are managed and stored within separate directories in the CVS Repository at SourceForge.

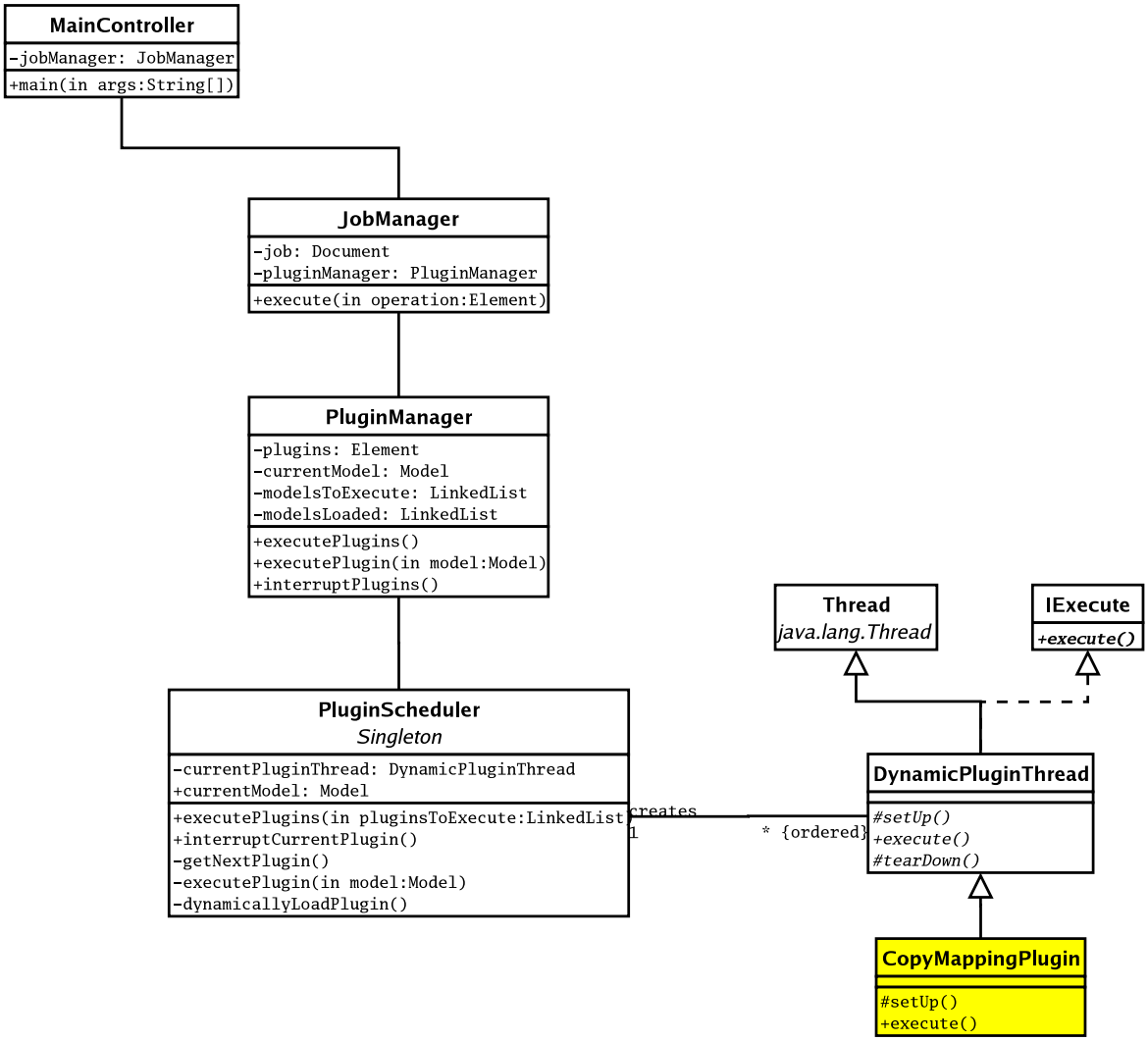


The core framework's directory is named opendbcopy. Each plugin's directory is prefixed with opendbcopy. E.g. opendbcopy.copy (as highlighted) contains the Copy Plugin for openDBcopy.

Please note that the sources are sometimes not commited before a new release is posted. If you require an update please do not hesitate to contact the developer team.

4 Architecture

Plugins are executed within separate Threads. The thread's lifecycle is managed by openDBcopy's core framework.



The above diagram is not complete. It shows some important classes with attributes and methods.

4.1 MainController

`MainController` is the main class to launch openDBcopy. It is the central class and parent of all top manager classes. openDBcopy is designed using the Model-View-Controller Pattern – MVC. The controller's job is to handle requests on the model and handover errors from the model to any view (caller of a request).

openDBcopy can be run in interactive or batch mode. Graphical User Interfaces (GUI) provide the ability to configure jobs and plugins, execute single plugins and plugin chains, also called jobs. All configuration parameters can be stored in xml files. Once a job is configured and ready for production, openDBcopy can be executed in batch mode, without loading any GUIs. This allows saving memory and place openDBcopy on a server not providing any Window Components (X Server). A job is stored in a single xml file and contains plugin metadata of one or several plugin models.

4.2 JobManager

`JobManager` is owned and created by `MainController`. `JobManager` is the owner of a job model. A job may contain zero to n plugins. A `JobManager` instance itself uses a `PluginManager` instance to manage plugins. `JobManager` provides methods to import and export a job configuration. An `execute()` method handles requests on the job model or handles requests over to the appropriate plugin model, which is part of the job model.

4.3 PluginManager

`PluginManager` is owned and created by `JobManager`. Its responsibility is to parse and dynamically load plugin resources from directories. After loading resources and metadata of plugins, `PluginManager` is able to add a plugin to one of two lists. One list contains plugins which are not bound to a job, only available in memory. A second list contains plugins which are part of a plugin chain – bound to a job – which can be saved as a single unit.
While `PluginManager` is responsible for loading plugin resources and models, it is not responsible for loading and managing Graphical User Interfaces of plugins. This is done by `PluginGuiManager`.

4.4 PluginScheduler

PluginScheduler is a singleton, owned by `PluginManager`. `PluginScheduler` receives a list of plugins to execute a plugin chain or a single plugin model. Its responsibility is to dynamically load a plugin thread, pass output parameters as input to following plugins, interrupt single plugins and plugin chains when requested and finally schedule plugins. A plugin which has been successfully executed is removed from `PluginScheduler`'s list. Not so from a plugin chain stored in `PluginManager`.

4.5 DynamicPluginThread

`DynamicPluginThread` extends `java.lang.Thread` and implements the Interface `IExecute`. `IExecute` only defines one method – `execute()`.
The final public `run()` method implemented by `DynamicPluginThread` calls `setUp()`, `execute()` and `tearDown()`. It also handles the state of a plugin.
Plugins implemented must overwrite the `execute()` method of `DynamicPluginThread`. Optionally `setUp()` and `tearDown()` methods of `DynamicPluginThread` can be overwritten. `setUp()` and `tearDown()` are called before, respectively after `execute()` has been successfully completed (no exception thrown). Use the `setUp()` method to read configuration parameters, setup database connections etc. The execute method is called after successful `setUp()`, e.g. configuration parameters were successfully read and validated, database connection(s) opened etc. Do not use the `tearDown()` method to close database connections as this method is only called after successful `execute()`. Use `tearDown()` for example to close file writers or whatever else. Close connections in a `finally` block of your `execute()` method body.

4.6 A Plugin Implementation

A plugin's thread class, e.g. `CopyMappingPlugin`, must extend DynamicPluginThread.

The only required method to implement is
`public final void execute() throws PluginException`

5 Plugin Components

Each openDBcopy plugin consists of the following components:

- Plugin Model
- Graphical User Interfaces
- Plugin Thread
- Language Resource Files

For an example please have a look at the following chapter.

5.1 Plugin Model

The model of a plugin is represented using a plugin.xml file and Java Model class. All parameters specified within such a plugin.xml file can be used when setting up, executing or tearing down a plugin. One can provide its own Java Model, extend existing ones or just use existing ones.

All plugin.xml files must contain the following elements

Element Name	Description
`conf`	This element contains configuration elements. openDBcopy provides a generic graphical user interface to display and manipulate plugin configuration parameters. See the next chapter, Generic Attributes for GUI Configuration for details.
`threads`	Future releases may provide the ability to specify more than one thread. Currently the element `threads` must contain one child-element called `thread`, specifying the thread class to execute and a `description` attribute.
`input`	Used to place output from a former plugin as input for the actual plugin
`output`	Used to store possible output of a plugin model

Example plugin model with minimum required elements (replace attribute values accordingly)

```
<plugin identifier="id" model_class="org.abc.ModelClass">
   <conf />
   <threads>
      <thread thread_class="org.abc.ThreadClass" description="key" />
   </threads>
   <input />
   <output />
</plugin>
```

5.1.1 Plugin Configuration

Using the graphical user interface `opendbcopy.gui.PluginConfiguration` to display and manipulate parameters is not mandatory. One can also provide its own graphical user interface(s) to manipulate data and configuration parameters. Within the conf element one can add child elements. To add parameters which can be configured using the generic GUI `PanelConfiguration`, the following attributes are currently supported, see an example below.

Attribute Name	Description
`type`	What GUI type is required to configure a parameter's value

Attribute Name	Description
`value`	May contain a default value or be empty
`required`	`true` or `false` – currently not tested automatically
`description`	key to resource entry in resource file (language specific descriptions)

The attribute `type` currently supports the following types. Type names are case insensitive.

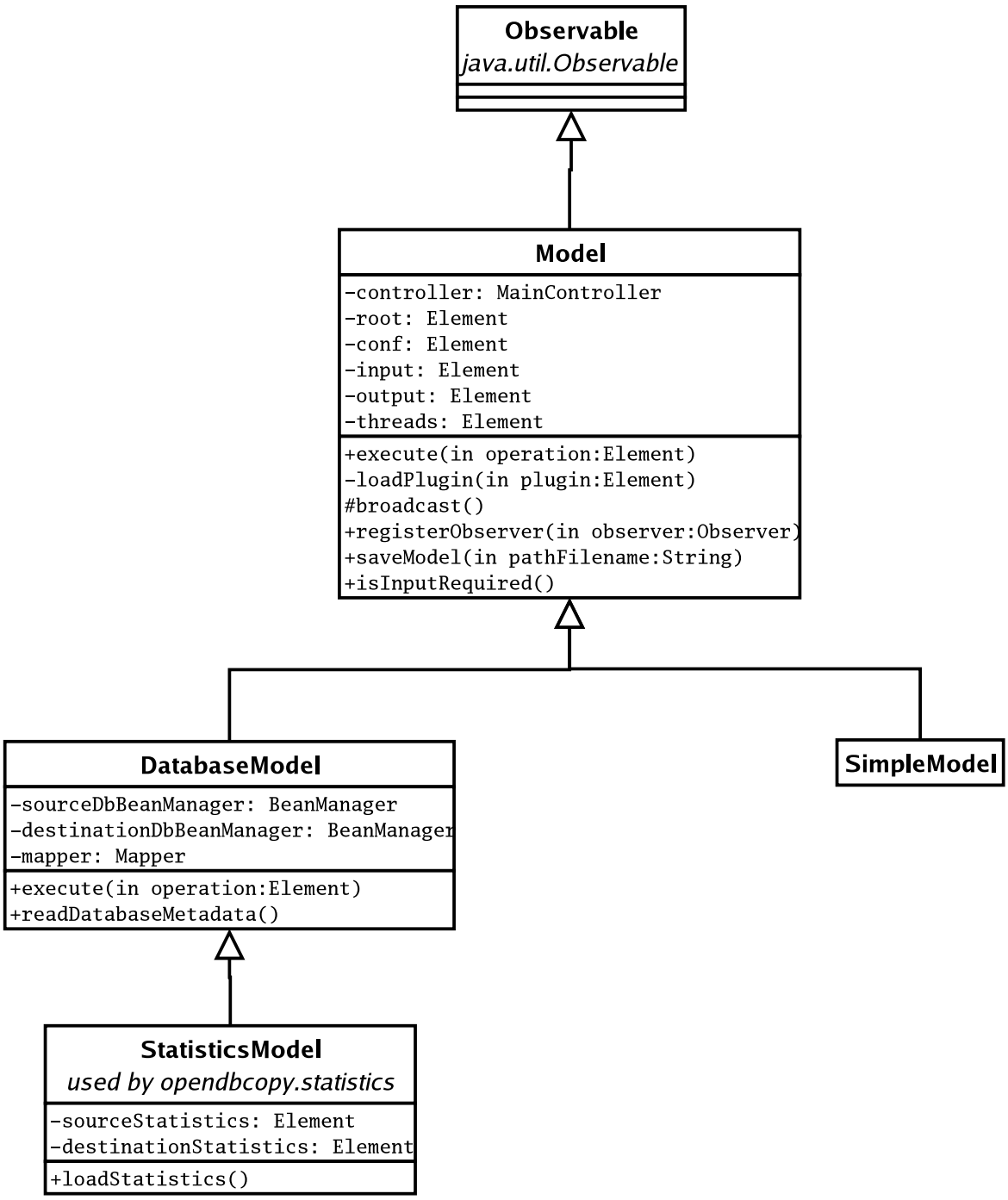
`type` values	Description
`string`	String parameters
`boolean`	Parameters which must be true or false
`int`	Integer value parameters. Tested when user enters a value.
`file_dir_filelists_selection`	Drop down box for selection of file, directory or filelist
`file`	File browser
`dir`	Directory browser
`hibernate_dialect`	Drop down box for Hibernate dialect selection (see http://www.hibernate.org/hib_docs/api/net/sf/hibernate/dialect/package-summary.html)

An example for selection of a directory. The default directory is „`dump`“, which may be relative or absolute (e.g. `/home/smitha/.opendbcopy/inout/dump`, `c:\temp`).

```
<conf>
  <!-- path may be relative or absolute -->
  <dir required="true" value="dump" type="dir"
       description="plugin.opendbcopy.dump.conf.outputDir" />
</conf>
```

5.1.2 Creating new Plugin Models

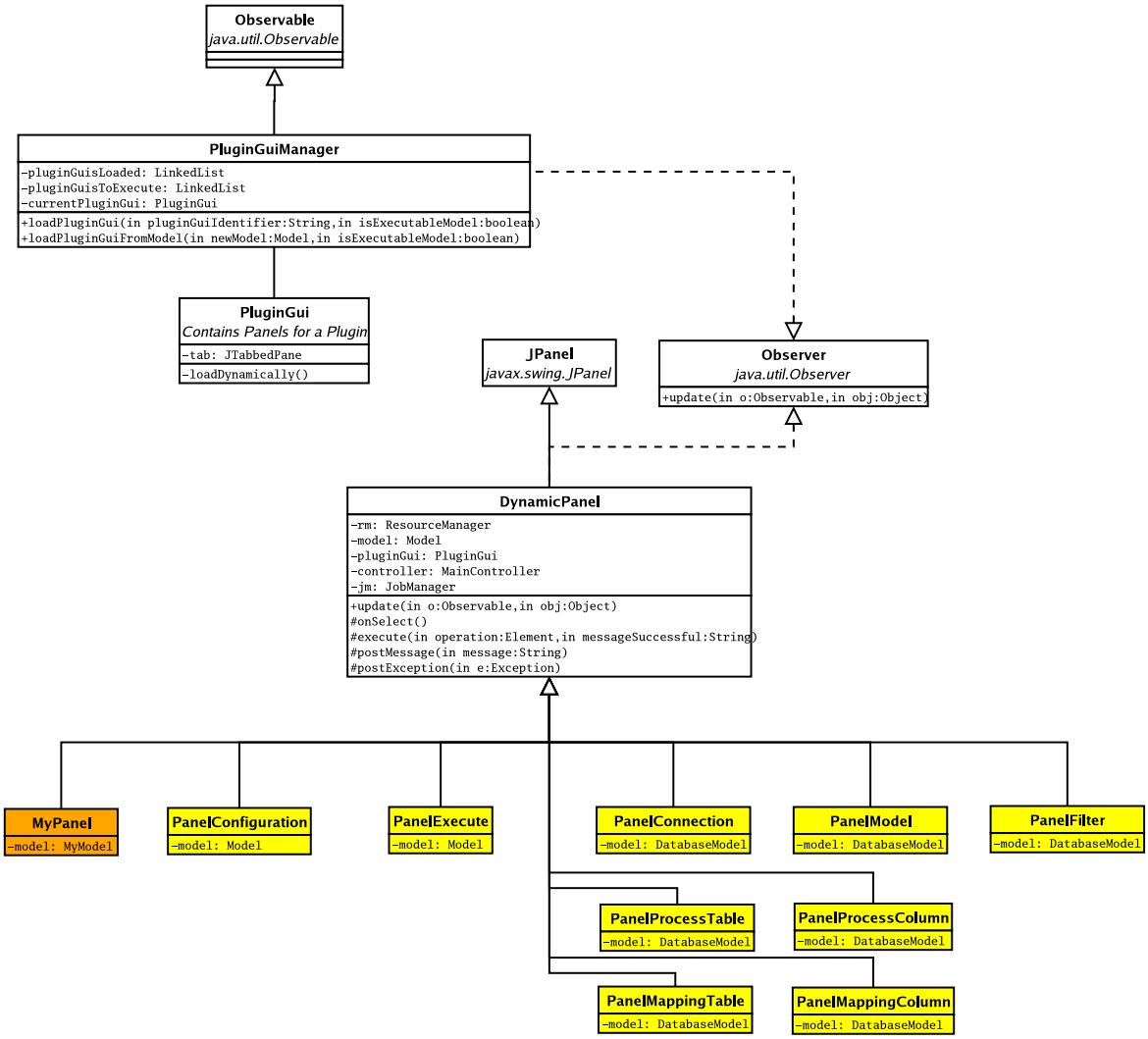
A plugins data container is a model. Operations on a model can be implemented in separate classes or the model itself. A model's file representation is `plugin.xml` providing the basic structure. Extending an existing Java Model class is not necessary as you can browse the model elements from top to bottom – using pure XML APIs. It is up to you to decide what is simpler and nicer. Providing methods to set and get data from the model is often more comfortable and more flexible for future extensions than directly parsing the XML tree.
The following diagram shows a simplified hierarchy of the model architecture. Please note that `DatabaseModel` provides tons of methods which cannot be shown on on simple diagram.



5.2 Graphical User Interfaces

Every plugin must provide a `gui.xml` file, even if no GUIs are required. In batch mode no GUIs are loaded at all – if not enabled. Use `gui.xml` to specify GUIs to load, order within wizard and GUI title. To develop your own GUIs and add those to the tab selection, extend the super class DynamicPanel.

The following diagram shows the relation of some important GUI classes. Note that one important attribute is telling openDBcopy if a GUI shall be registered as an observer of the observable model – PluginGuiManager. The diagram is on only an extract and not complete.



Boxes in yellow are standard GUIs which are used in standard plugins and can be used within your own plugins. To create your own GUIs, see orange box on the left hand side, extend the class `DynamicPanel`.

5.2.1 GUI Configuration

GUI configuration is stored in a file called `gui.xml`. Its structure looks like:

```
<gui identifier="id" display_order="numeric_value"
resource_name="resource_file_name_without_file_ending">
   <title value="key for title" />
   <panels>
      <panel title="key for first gui title">
         <class name="class name for first gui" register_as_observer="true" />
      </panel>
      <panel title="key for second gui title">
         <class name="class name for second gui" register_as_observer="false" />
      </panel>
   </panels>
</gui>
```

All `gui.xml` files must contain the following elements as children of the root element `gui`.

Element Name	Description
`title`	An attribute `value` contains the key for the language dependent title.
`panels`	May contain 0 to n panel elements

The element `gui` must contain the following attributes:

Attribute Name	Description
`identifier`	The unique identifier used within the `plugin.xml`
`display_order`	The `display_order` attribute must be a numeric number and allows to control the position of this plugin's title within the plugin selection menu.
`resource_name`	The resource property name for this plugin without file ending `.properties`.

A gui.xml only contains one element `panels` (plural). This `plugins` element may contain 0 to n elements named `plugin`. Each `plugin` element must provide a `title` attribute. The `title`'s attribute value is the key for the resources (language dependent). Each `panel` element must contain a child element named `class`.

The element class must contain the following attribute:

Attribute Name	Description
name	name must contain the java package and class name for the standard or custom GUI. This attribute is required.
register_as_observer	Use this attribute to tell PluginGui if this GUI shall be registered as observer of the model observed. This attribute is not mandatory.

5.2.2 Creating your own GUIs

When creating a new plugin it may be that you require new GUIs. To do such is very simple. openDBcopy already provides the framework to put a new GUI into a workflow wizard, register the new GUI as observer, provide a `next` button for your wizard etc. If your new plugin is of general interest, why not donate the source code back to the openDBcopy project?
Have a look at some other GUIs extending the `DynamicPanel` before you start creating your own GUIs to familiarise yourself. It is up to you if you want to overwrite the supermethods of `DynamicPanel`, such as the method `update(...)` or `onSelect()`. `onSelect()` is automatically called whenever a user clicks on the appropriate tab for this GUI. Overwrite this method when this GUI shall update its components, load data or whatever.

A word about GUI design using Swing. Generally you are free to use whatever `LayoutManager` you like. There is no limitation – pure Java. All new GUIs of openDBcopy will be replaced using `TableLayout` (see http://www.clearthought.info/software/TableLayout), if not yet already done as this `LayoutManager` is very good.

Have a look at how Exceptions are caught, respectively thrown. Do not catch Exceptions and handle them if it is not the GUIs responsibility. When throwing Exceptions those are finally caught by `FrameMain` and presented to the user and also logged properly.

5.3 Plugin Thread

Every plugin must implement a plugin thread class, extending the super class `DynamicPluginThread`. This plugin thread class must implement an `execute()` method. Additional methods, classes or other Java libraries, even external programmes can be used.
The plugin thread class together with optional classes and archives must be deployed as jar archive(s) or zip file(s) – preferrably jar file(s). Using the default plugin `build.xml` file, all class files including source code are added to one single jar file, including language resource files. When deploying a plugin to the openDBcopy environment, all jar files of the plugin's lib directory are deployed too.

5.4 Language Resources

Every plugin must provide at least one resource file. Resource files are normal `.properties` files, using `key` / `value` pairs.

The suffix of the resource file name specifies the language, e.g. „`de`“ for german („deutsch“). The default language resource file is expected to contain English key / value pairs, with no file suffix for English.

Resource values may contain parameters specified using `{0}`, `{1}`, `{...}` as placeholders within a value. The number specifies the parameter number, beginning at `0` (zero). When reading language dependent resource values containing dynamic parameters, such as numbers, names etc., parameters can be replaced by providing String arrays.

Here an example: `message.model.successful=capturing {0} done`
The key is `message.model.successful`. The English text for this key is `capturing {0} done`. The value within brackets `{0}` is variable.

To read the value according to the language currently set use the following code.

```
String[] param = { "testmodel" };
rm.getString("message.model.successful", param);
```

`rm` is the ResourceManager owned by MainController, which is available in any GUI extending `DynamicPanel`.

To read values without parameters use the following syntax:
`rm.getString("key");`

Please note that openDBcopy has an advanced `ResourceManager` implemented. One does not have to provide the bundle id as prefix for keys, `Locale` etc. A value for a given key is found if the key / value pair was loaded properly, independent of file. The only disadvantage of this model is uniqueness of keys which must be guaranteed to avoid overwriting of key / value pairs.

6 Example - The Copy Plugin

6.1 Overview

This plugin, called `opendbcopy.copy`, copies records of selected source tables and columns into selected destination tables.
If the underlying RDBMS supports Referential Integrity Constraints, the order of tables to process is respected, but this feature is part of the openDBcopy core framework and must not be implemented within this plugin. The plugin directly gets an ordered list of tables to process.
If copy errors occur, those are logged into comma separated value files, if enabled by the user. The plugin writes a new file for each table containing copy errors, including the record's values and error message.

6.2 Plugin DatabaseModel

This plugin uses the `DatabaseModel` class to hold and manipulate model data. `DatabaseModel` inherits from its superclass `Model`.

6.2.1 Plugin Configuration File

The plugin configuration file contains the following elements and attributes

```
<plugin identifier="copy" model_class="opendbcopy.plugin.model.database.DatabaseModel">
    <conf>
        <!-- path may be relative or absolute -->
        <dir required="true" value="" type="dir" description="plugin.opendbcopy.copy.conf.outputDirErrorLogs" />
        <log_error value="true" type="boolean" description="plugin.opendbcopy.copy.conf.logError" />
        <output>
             <filelist value="error_logs" type="string"
                       description="plugin.opendbcopy.copy.conf.output.filelistIdentifier" />
        </output>
    </conf>
    <threads>
          <thread thread_class="opendbcopy.plugin.copy.CopyMappingPlugin" description="title.plugin.opendbcopy.copy" />
    </threads>
    <input />
    <output />
    <source_db>
       <driver />
       <metadata />
       <connection />
       <catalog value="" />
       <schema value="%" />
       <table_pattern value="%" />
       <model />
    </source_db>
    <destination_db>
       <driver />
       <metadata />
       <connection />
       <catalog value="" />
       <schema value="%" />
       <table_pattern value="%" />
       <model />
    </destination_db>
    <mapping />
    <filter>
       <string name="trim" process="false" />
       <string name="remove_intermediate_whitespaces" process="false" />
       <string name="set_null" process="false" />
    </filter>
</plugin>
```

6.2.2 Plugin Attributes

Attribute Name	Description
`identifier`	„`copy`“ - a unique string identifier

Attribute Name	Description
model_class	„opendbcopy.plugin.model.database.DatabaseModel“ Model class providing elements for source database, destination database, database mapping and filter elements. Values set in the above file are default values. DatabaseModel can be used for single database mode by removing the element destination_db.

6.2.3 Conf Elements

Element Name	Description
dir	<dir required="true" value="" type="dir" description="plugin.opendbcopy.copy.conf.outputDirErrorLogs" /> Specify the output path for error log files • required = „true“ means this parameter must be set • value = „“ - no default directory set • type = „dir“ - pops up a directory browser when clicked by user • description - key for resource property
log_error	<log_error value="true" type="boolean" description="plugin.opendbcopy.copy.conf.logError" /> Shall errors occuring be logged or not • value = „true“ - by default errors are logged • type = „boolean“ - pops up a dialog box to select true or false • description - key for resource property
output filelist element	<filelist value="error_logs" type="string" description="plugin.opendbcopy.copy.conf.output.filelistIdentifier" /> If errors occur during the copy process, this output element contains a filelist with path and filenames to error log files for possible further processing of following plugins • value = „error_logs“ - a string name for filelist identification • type = „string“ - pops up a dialog allowing to specify a string • description - key for resource property

6.2.4 Threads Element(s)

Element Name	Description
thread	<thread thread_class="opendbcopy.plugin.copy.CopyMappingPlugin" description="title.plugin.opendbcopy.copy" /> Currently only one thread_class element is allowed per plugin. The element threads must contain an element called thread. • thread_class = „opendbcopy.plugin.copy.CopyMappingPlugin“ path and class name of the Thread class implementing the execute() method • description - key for resource property

6.2.5 Input / Output Elements

The `input` and `output` element are set at runtime. Do not use these elements to save configuration `input` or `output` as these elements may be overwritten by `PluginScheduler`.

6.2.6 Source & Destination Database Elements

The following elements are filled by reading database metadata and model(s). Database Metadata is read when testing a database connection. Models are read when a user clicks on Capture Source Model or Capture Destination Model. For more information about types, constants and metadata available please see the Java API, class `java.sql.DatabaseMetaData`.

Element Name	Description
`driver`	Contains information about the driver used for the appropriate database. Currently a driver's `name` and `version` are stored as attributes.

`metadata`	Contains further sub-elements:
	• `db_product_name` – e.g. „`Oracle`“, „`PostgreSQL`“
	• `db_product_version` – database product version
	• `catalog_separator` – separator between catalog name and table name – often this is a dot („`.`“)
	• `identifier_quote_string` - How are strings quoted
	• `catalog`
	If available lists all catalogs available as separate elements
	• `schema`
	If available lists all schemas available as separate elements
	• `type_info`
	lists all available data types specifying:
	  • `type_name` (e.g. „`VARCHAR2`“)
	  • `locale_type_name` (e.g. „`VARCHAR2`“ – normally the same as `type_name` or `null`)
	  • `data_type` (e.g. „`12`“ – `java.sql.Types` Constant)
	  • `precision` (e.g. „`4000`“)
	  • `literal_prefix` (e.g. „'“ - prefix for this data type)
	  • `literal_suffix` (e.g. „'“ - suffix for this data type)
	  • `nullable` („`true`“ or „`false`“ – is this data type nullable)
	  • `case_sensitive` (e.g. „`0`“ or „`1`“ – is this data type case sensitive)

`connection`	Contains attributes for: • `driver_class` (e.g. „`oracle.jdbc.driver.OracleDriver`“) • `url` (e.g. „`jdbc:oracle:thin:@localhost:1521:test`“) • `username` • `password`
`catalog`	Catalog selected – if supported
`schema`	Schema selected – if supported
`table_pattern`	Table pattern allows to specify an SQL like pattern to use when capturing table metadata, e.g. „`TA%`“ only captures tables beginning with `TA` followed by any character or none. See Java API for more details.

Element Name	Description
model	When a model is captured, this element contains an attribute capture_date. Capture a database model, save the plugin as job and then have a look at the XML file created. The model hierarchy should be self-explaining. Model contains children named table. A table element can be of type TABLE or VIEW, identified by the attribute table_type. An attribute process (true or false) allows to specify if a table shall be processed or not. Table contains column elements for each column in the underlying table / view column. These column elements contain attributes such as name, type_name, data_type, column_size, decimal_digits, nullable and process. When primary, foreign keys and indexes are enabled, appropriate elements are added to the table element. The attributes of the these elements are self explaining, comparing them with the DatabaseMetaData methods for reading primary, imported and exported keys as well as index information.

6.2.7 Mapping Element

This element contains information to map source tables to destination tables. Same for columns. This element's structure is likely to be changed in a future version to allow more complex mapping and merging rules to be defined.

6.2.8 Filter Elements

The structure of this element is likely to be changed in future versions. Currently it contains a string element for each string filter implemented. A name attribute identifies a string filter and a process attribute – true or false – allow to enable or disable appropriate string filter.

6.3 Graphical User Interfaces

gui.xml contains the panels used to specify and execute this plugin.
The root element gui must contain two elements. title and panels. Title must contain an attribute value specifying the resource key for the plugins title to show within the GUI of openDBcopy.
plugins can contain children named panel for the individual panels for this plugin. The order of these plugin elements within their plugins parent element specifies the order of the panels added within the tab in the GUI of openDBcopy, e.g. the first panel element is the first GUI in the wizard created etc.

A panel element must contain a class element. Element class must contain one attribute, name of the Panel class to load. The second attribute, register_as_observer (true or false) is not mandatory, as described in previous chapters.

The copy plugin uses 7 GUIs to configure, gather database metadata and finally execute the plugin. All except PanelConfiguration and PanelExecute are registered as observers of the model. PanelExecute itself contains its own registration mechanism – see code of PanelExecute for details.

The file looks like:

```
<gui identifier="copy" display_order="20" resource_name="plugin_opendbcopy_copy">
   <title value="title.plugin.opendbcopy.copy" />
   <panels>
      <panel title="title.plugin.opendbcopy.copy.panel.0">
          <class name="opendbcopy.gui.PanelConfiguration" />
      </panel>
```

```
      <panel title="title.plugin.opendbcopy.copy.panel.1">
          <class name="opendbcopy.gui.database.PanelConnection" register_as_observer="true" />
      </panel>
      <panel title="title.plugin.opendbcopy.copy.panel.2">
          <class name="opendbcopy.gui.database.PanelModel" register_as_observer="true" />
      </panel>
      <panel title="title.plugin.opendbcopy.copy.panel.3">
          <class name="opendbcopy.gui.database.dual.PanelMappingTable" register_as_observer="true" />
      </panel>
      <panel title="title.plugin.opendbcopy.copy.panel.4">
          <class name="opendbcopy.gui.database.dual.PanelMappingColumn" register_as_observer="true" />
      </panel>
      <panel title="title.plugin.opendbcopy.copy.panel.5">
          <class name="opendbcopy.gui.database.PanelFilter" register_as_observer="true" />
      </panel>
      <panel title="title.plugin.opendbcopy.copy.panel.6">
          <class name="opendbcopy.gui.PanelExecute" />
      </panel>
   </panels>
</gui>
```

6.3.1 GUI Attributes

Attribute Name	Description
`identifier`	„`copy`“ - a unique string identifier – must be the same as used within `plugin.xml`
`display_order`	„`20`“ Allows to specify the position of this plugin within the plugin menu selection. When adding new plugins leave some space between numbers for additions. Have a look at the other `display_order` numbers of other plugins.
`resource_name`	„`plugin_opendbcopy_copy`“ The name of the resource file name (language dependent texts). Note that the name specified must not contain the file ending `.properties` and the file itself must be accessible within the plugins classpath.

6.4 Plugin Thread

`CopyMappingPlugin` is the thread class of the Copy Plugin, see `opendbcopy.copy`. `CopyMappingPlugin` is the only class required to copy data from a source into a destination database, independent of underlying database management system. Therefore the deployed jar file currently only contains this one class file, `CopyMappingPlugin`. Resource properties files are added to this jar file too - into the jar archive's root. For simpler debugging of this plugin the source code is also added to the jar file. See the ant `jar target` for details.

The source code of the class `CopyMappingPlugin` is as follows:

```
/*
 * Copyright (C) 2004 Anthony Smith
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
 *
 * -------------------------------------------------------------------------
 * TITLE $Id: CopyMappingPlugin.java,v 1.2 2004/04/08 11:21:05 iloveopensource Exp $
 * -------------------------------------------------------------------------
 *
 * -------------------------------------------------------------------------*/
package opendbcopy.plugin.copy;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import opendbcopy.config.XMLTags;
import opendbcopy.connection.DBConnection;
import opendbcopy.connection.exception.CloseConnectionException;
import opendbcopy.controller.MainController;
import opendbcopy.filter.StringConverter;
import opendbcopy.plugin.model.DynamicPluginThread;
import opendbcopy.plugin.model.Model;
import opendbcopy.plugin.model.database.DatabaseModel;
import opendbcopy.plugin.model.exception.PluginException;
import opendbcopy.sql.Helper;
import opendbcopy.util.InputOutputHelper;

import org.jdom.Element;


/**
 * Copies records of selected source tables and columns into selected destination tables. If the underlying
 * RDBMS supports Referential Integrity Constraints, the order of tables to process is respected. If possible
 * errors shall be logged, those are logged into comma separated value files.
 *
 * @author Anthony Smith
 * @version $Revision: 1.2 $
 */
public class CopyMappingPlugin extends DynamicPluginThread {
    private static final String            fileType = "csv";
    private DatabaseModel     model; // this plugin's model
    private Connection        connSource;
    private Connection        connDestination;
    private Statement         stmSource;
    private PreparedStatement pstmtDestination;
    private ResultSet         rs;
    private StringBuffer      recordBuffer; // to hold records contents
    private StringBuffer      recordErrorBuffer; // in case of errors records are written to file
    private String            stmSelect = "";
    private String            stmInsert = "";
    private String            sourceTableName = "";
    private String            destinationTableName = "";
    private String            newLine = "";
    private File              outputPath = null;
    private String            fileName = "";
    private String            delimiter = "";
    private boolean           log_error = false;
    private boolean           errorLogSetup = false;
    private int               counterRecords = 0;
    private int               counterTables = 0;
    private List              processColumns;
    private List              processTables;
    boolean                   trimString = false;
    boolean                   trimAndRemoveMultipleIntermediateWhitespaces = false;
    boolean                   trimAndReturnNullWhenEmpty = false;

    /**
     * Creates a new CopyMappingPlugin object.
     *
     * @param controller DOCUMENT ME!
     * @param baseModel DOCUMENT ME!
     *
     * @throws PluginException DOCUMENT ME!
     */
    public CopyMappingPlugin(MainController controller,
                             Model          baseModel) throws PluginException {
        // Call the super constructor
          super(controller, baseModel);

          // cast the super base model into a specific database model
        this.model = (DatabaseModel) baseModel;
    }

    /**
     * Read configuration and setup database connections
     *
     * @throws PluginException DOCUMENT ME!
     */
    protected final void setUp() throws PluginException {
        // set the new line character as this differs from OS to OS
          newLine = MainController.lineSep;

        // get the plugin configuration
        Element conf = model.getConf();

        if (conf == null) {
            throw new PluginException("Missing conf element");
        }

        try {
          // set the output path selected by user
            outputPath = new File(conf.getChild(XMLTags.DIR).getAttributeValue(XMLTags.VALUE));

            // create the output directory if it does not yet exist
            if (!outputPath.exists()) {
                boolean mkDirOk = outputPath.mkdir();
```

```
                if (!mkDirOk) {
                    throw new PluginException("Could not create " + outputPath.getAbsolutePath());
                }
            }

            // shall errors be logged?
            log_error                   = Boolean.valueOf(conf.getChild(XMLTags.LOG_ERROR).getAttributeValue
(XMLTags.VALUE)).booleanValue();

            if (log_error) {
                recordBuffer          = new StringBuffer();
                recordErrorBuffer     = new StringBuffer();
            }

            // check string filters
            if (model.getStringFilterTrim().getAttributeValue(XMLTags.PROCESS).compareTo("true") == 0) {
                trimString = true;
            }

            if (model.getStringFilterRemoveIntermediateWhitespaces().getAttributeValue(XMLTags.PROCESS).compareTo("true")
== 0) {
                trimAndRemoveMultipleIntermediateWhitespaces = true;
            }

            if (model.getStringFilterSetNull().getAttributeValue(XMLTags.PROCESS).compareTo("true") == 0) {
                trimAndReturnNullWhenEmpty = true;
            }

            // get connections
            connSource          = DBConnection.getConnection(model.getSourceConnection());
            connDestination     = DBConnection.getConnection(model.getDestinationConnection());

            // extract the tables to copy
            processTables = model.getDestinationTablesToProcessOrdered();
        } catch (Exception e) {
            throw new PluginException(e);
        }

        // now set the number of tables that need to be copied
        model.setLengthProgressTable(processTables.size());
    }

    /**
     * Copies records of selected source tables and columns into selected destination tables.
     * If requested and occuring, errors are logged
     *
     * @throws PluginException DOCUMENT ME!
     */
    public final void execute() throws PluginException {
        Iterator itColumns;
        Element  tableProcess;
        Element  columnDestination;
        int      colCounter;
        Object   input;

        try {
            stmSource = connSource.createStatement();

            Iterator itProcessTables = processTables.iterator();

            ArrayList generatedFiles = new ArrayList();

            while (!isInterrupted() && itProcessTables.hasNext()) {
                tableProcess     = (Element) itProcessTables.next();

                sourceTableName          = tableProcess.getAttributeValue(XMLTags.SOURCE_DB);
                destinationTableName     = tableProcess.getAttributeValue(XMLTags.DESTINATION_DB);

                // file name for error logs
                if (log_error) {
                    fileName = destinationTableName + "_ERRORS" + "." + fileType;
                }

                // get the columns to process
                processColumns = model.getMappingColumnsToProcessByDestinationTable(destinationTableName);

                // setting record counter to minimum of progress bar
                model.setCurrentProgressRecord(0);
                model.setLengthProgressRecord(0);

                // Reading number of records for progress bar
                model.setLengthProgressRecord(Helper.getNumberOfRecordsFiltered(stmSource, model, XMLTags.SOURCE_DB,
sourceTableName));

                // get Select Statement for source model
                stmSelect     = Helper.getSelectStatement(model, sourceTableName, XMLTags.SOURCE_DB, processColumns);

                // get Insert Statement for destination model
                stmInsert     = Helper.getInsertPreparedStatement(model.getQualifiedDestinationTableName
(destinationTableName), processColumns);

                pstmtDestination = connDestination.prepareStatement(stmInsert);

                model.setCurrentProgressTable(counterTables);

                // Execute SELECT
```

```
                rs = stmSource.executeQuery(stmSelect);

                // do some logging
                model.setProgressMessage("Copying " + model.getQualifiedSourceTableName(sourceTableName) + " into " +
model.getQualifiedDestinationTableName(destinationTableName) + " ...");

                logger.info("Copying " + model.getQualifiedSourceTableName(sourceTableName) + " into " +
model.getQualifiedDestinationTableName(destinationTableName) + " ...");

                // while there are more records to process and the process is not interrupted by the user
                while (!isInterrupted() && rs.next()) {
                    colCounter     = 1;

                    // process columns
                    itColumns = processColumns.iterator();

                    while (itColumns.hasNext()) {
                        columnDestination     = model.getDestinationColumn(destinationTableName, ((Element)
itColumns.next()).getAttributeValue(XMLTags.DESTINATION_DB));

                        input = rs.getObject(colCounter);

                        if (input != null) {
                            input = applyStringFilters(input, Boolean.valueOf(columnDestination.getAttributeValue
(XMLTags.NULLABLE)).booleanValue());

                            if (input != null) {
                                if (log_error) {
                                    recordBuffer.append(input + delimiter);
                                }

                                pstmtDestination.setObject(colCounter, input);
                            } else {
                                if (log_error) {
                                    recordBuffer.append("null" + delimiter);
                                }

                                pstmtDestination.setNull(colCounter, Integer.parseInt(columnDestination.getAttributeValue
(XMLTags.DATA_TYPE)));
                            }
                        } else {
                            if (log_error) {
                                recordBuffer.append("null" + delimiter);
                            }

                            pstmtDestination.setNull(colCounter, Integer.parseInt(columnDestination.getAttributeValue
(XMLTags.DATA_TYPE)));
                        }

                        colCounter++;
                    }

                    // execute the prepared statement and log the error ... and continue without disturbing other
business
                    try {
                        // Execute INSERT
                        pstmtDestination.executeUpdate();
                        pstmtDestination.clearParameters();

                        counterRecords++;
                        model.setCurrentProgressRecord(counterRecords);
                    } catch (SQLException e) {
                        connDestination.rollback();

                        if (log_error && !errorLogSetup) {
                            initErrorLog(processColumns);
                        }

                        if (log_error) {
                            recordErrorBuffer.append(recordBuffer + e.toString() + newLine);
                        }
                    } finally {
                        // reset recordBuffer
                        if (log_error) {
                            recordBuffer = new StringBuffer();
                        }
                    }
                }

                if (!isInterrupted()) {
                    // commit INSERTs. Commit behaviour depends on RDBMS used
                    connDestination.commit();

                    // close the result set
                    rs.close();
                    logger.info(counterRecords + " records inserted into table " + destinationTableName);
                    counterRecords = 0;

                    // required in case of last table that had to be copied
                    counterTables++;
                    model.setCurrentProgressTable(counterTables);

                    // set processed
                    tableProcess.setAttribute(XMLTags.PROCESSED, "true");
                } else {
                    // rollback insert in case the user interrupts the process
                    connDestination.rollback();
```

```
                    // close the result set
                    rs.close();
                    counterRecords = 0;
                }

                if (log_error) {
                    if (recordErrorBuffer.length() > 0) {
                        // open file writer
                              File errorFile = new File(outputPath.getAbsolutePath() + MainController.fileSep +
fileName);
                        OutputStreamWriter fileWriter = new OutputStreamWriter(new FileOutputStream(errorFile),
MainController.getEncoding());
                        fileWriter.write(recordErrorBuffer.toString());
                        fileWriter.close();
                        generatedFiles.add(errorFile);

                        logger.error(errorFile + " contains records which could not be processed");
                        recordErrorBuffer     = new StringBuffer();
                        errorLogSetup         = false;
                    }
                }
            }

            stmSource.close();
            pstmtDestination.close();

            // close database connections
            DBConnection.closeConnection(connSource);
            DBConnection.closeConnection(connDestination);

            if (!isInterrupted()) {
                    if (generatedFiles != null && generatedFiles.size() > 0) {
                    File[] outputFiles = new File[generatedFiles.size()];
                    outputFiles = (File[]) generatedFiles.toArray(outputFiles);

                    Element outputConf = model.getConf().getChild(XMLTags.OUTPUT);

                    model.appendToOutput(InputOutputHelper.createFileListElement(outputFiles, outputConf.getChild
(XMLTags.FILELIST).getAttributeValue(XMLTags.VALUE)));
                    }

                logger.info(counterTables + " table(s) processed");
            }
        } catch (SQLException sqle) {
            throw new PluginException(sqle);
        } catch (Exception e1) {
            // clean up if required
            try {
                DBConnection.closeConnection(connSource);
                DBConnection.closeConnection(connDestination);
            } catch (CloseConnectionException e2) {
                // bad luck ... don't worry
            }

            throw new PluginException(e1);
        }
    }

    /**
     * If global string filters were selected by user, those are applied using this method
     *
     * @param in DOCUMENT ME!
     * @param returnNullWhenEmpty DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    private Object applyStringFilters(Object  in,
                                      boolean returnNullWhenEmpty) {
        if (in instanceof String || in instanceof Character) {
            if (trimAndRemoveMultipleIntermediateWhitespaces && trimAndReturnNullWhenEmpty) {
                return StringConverter.trimAndRemoveMultipleIntermediateWhitespaces(in, returnNullWhenEmpty);
            } else if (trimAndRemoveMultipleIntermediateWhitespaces && !trimAndReturnNullWhenEmpty) {
                return StringConverter.trimAndRemoveMultipleIntermediateWhitespaces(in, false);
            } else if (trimString && trimAndReturnNullWhenEmpty) {
                return StringConverter.trimString(in, returnNullWhenEmpty);
            } else if (trimString && !trimAndReturnNullWhenEmpty) {
                return StringConverter.trimString(in, false);
            } else {
                return in;
            }
        } else {
            return in;
        }
    }

    /**
     * Called to write a nice row header of column names in possible error logs
     *
     * @param processColumns DOCUMENT ME!
     */
    private void initErrorLog(List processColumns) {
        Iterator itColumns = processColumns.iterator();

        // set the column headings for the possible error log
        while (itColumns.hasNext()) {
            recordErrorBuffer.append(((Element) itColumns.next()).getAttributeValue(XMLTags.DESTINATION_DB) + delimiter);
```

```
        }

        recordErrorBuffer.append("ERROR" + newLine);
        errorLogSetup = true;
    }
}
```

6.5 Language Resource Files

The copy plugin provides resource files for English and German.
The source properties files are stored under `resource/bundle`.
The English properties file is named `plugin_opendbcopy_copy.properties`. The German version is named `plugin_opendbcopy_copy_de.properties`. The extension – `de` – is according to the Java Locale specification.

The English version looks as follows:

```
title.plugin.opendbcopy.copy=Copy data from a source into a destination database
title.plugin.opendbcopy.copy.panel.0=0. Plugin Configuration
title.plugin.opendbcopy.copy.panel.1=1. Database Connections
title.plugin.opendbcopy.copy.panel.2=2. Models
title.plugin.opendbcopy.copy.panel.3=3. Table Mapping
title.plugin.opendbcopy.copy.panel.4=4. Column Mapping
title.plugin.opendbcopy.copy.panel.5=5. Global String Filters
title.plugin.opendbcopy.copy.panel.6=6. Execute Plugin
plugin.opendbcopy.copy.conf.outputDirErrorLogs=Output dir for error logs
plugin.opendbcopy.copy.conf.logError=Log errors into comma separated files (csv) including
error message
plugin.opendbcopy.copy.conf.appendFileAfterRecords=After how many records shall disk I/O
occur? Depends on available memory and table width
plugin.opendbcopy.copy.conf.output.filelistIdentifier=Filelist identifier for error logs
```

Please note that the title for the panels in a tab must be specified for each plugin as the meaning of the same GUI may vary, depending on the plugin used.

7 Building, Testing and Deploying Plugins

OpenDBcopy and plugins are compiled and deployed using Apache Ant.

To test your plugin(s) your require an openDBcopy Core Framework environment. The most comfortable solution is to have openDBcopy Core Framework and Plugin Project(s) open and linked within the same Java IDE. Currently there are no JUnit test classes implemented to test openDBcopy Core Framework nor Plugins.

Each plugin provides a `build.xml` and `build.properties` file. The latter contains plugin specific details such as plugin name etc. Most plugins are released using an identical copy of `build.xml` as other plugins.
Therefore when starting a new plugin project, the simplest way to setup your environment is to copy an existing plugin project into a new directory and modify the new build.properties file. This file contains a hardcoded reference to your openDBcopy project location. You must modify this parameter, next to plugin identifier, author name etc.

The default build target is deploy. See the `build.xml` file and Apache Ant User Manual for further details.