Hibernate Tools Reference Guide

Version: 3.3.0.M5
1. Introduction .................................................................................................................. 1
   1.1. Key Features ........................................................................................................... 1
   1.2. Other relevant resources on the topic ................................................................. 2
2. Download and install Hibernate Tools ........................................................................ 3
   2.1. JBoss Tools ........................................................................................................... 3
   2.2. Eclipse IDE ........................................................................................................... 3
      2.2.1. Using Eclipse WTP ........................................................................................ 3
   2.3. Ant ......................................................................................................................... 4
3. Code generation architecture ......................................................................................... 5
   3.1. Hibernate Meta Model .......................................................................................... 5
   3.2. Exporters .............................................................................................................. 6
4. Eclipse Plugins ............................................................................................................. 7
   4.1. Introduction ........................................................................................................... 7
      4.1.1. Download base project .................................................................................... 7
   4.2. Creating a Hibernate Mapping File ....................................................................... 7
   4.3. Creating a Hibernate Configuration File ............................................................ 14
   4.4. Hibernate Console Configuration ....................................................................... 16
      4.4.1. Creating a Hibernate Console Configuration .................................................. 16
      4.4.2. Modifying a Hibernate Console Configuration ............................................... 28
      4.4.3. Closing Hibernate Console Configuration .................................................... 31
   4.5. Reverse Engineering and Code Generation ....................................................... 33
      4.5.1. Code Generation Launcher ............................................................................ 33
      4.5.2. Exporters ........................................................................................................ 36
   4.6. Hibernate Mapping and Configuration File Editor ............................................. 42
      4.6.1. Java property/class completion ...................................................................... 43
      4.6.2. Table/Column completion ............................................................................. 44
      4.6.3. Configuration property completion .................................................................. 44
   4.7. Structured Hibernate Mapping and Configuration File Editor ......................... 46
   4.8. JBoss Tools Properties Editor ............................................................................. 50
   4.9. Reveng.xml Editor ............................................................................................... 56
   4.10. Hibernate Console Perspective ......................................................................... 61
      4.10.1. Viewing the entity structure ......................................................................... 62
      4.10.2. Prototyping Queries ..................................................................................... 80
      4.10.3. Properties View ............................................................................................ 86
   4.11. Hibernate:add JPA annotations refactoring ..................................................... 88
   4.12. Enable debug logging in the plugins ................................................................. 94
      4.12.1. Relevant Resources Links ............................................................................ 94
   4.13. Hibernate support for Dali plugins in Eclipse WTP .......................................... 95
      4.13.1. Creating JPA project with Hibernate support .............................................. 95
      4.13.2. Generating DDL and Entities ...................................................................... 99
      4.13.3. Hibernate Annotations Support .................................................................. 102
      4.13.4. Relevant Resources Links ............................................................................ 110
5. Ant Tools ....................................................................................................................... 111
   5.1. Introduction ........................................................................................................... 111
5.2. The <hibernatetool> Ant Task ................................................................. 111
5.2.1. Basic examples ................................................................................. 113
5.3. Hibernate Configurations ................................................................. 113
5.3.1. Standard Hibernate Configuration (<configuration>) .................. 114
5.3.2. Annotation based Configuration (<annotationconfiguration>) ..... 115
5.3.3. JPA based configuration (<jpaconfiguration>) ............................... 116
5.3.4. JDBC Configuration for reverse engineering (<jdbcconfiguration>) 117
5.4. Exporters ......................................................................................... 118
5.4.1. Database schema exporter (<hbm2ddl>) ........................................ 119
5.4.2. POJO java code exporter (<hbm2java>) ....................................... 120
5.4.3. Hibernate Mapping files exporter (<hbm2hbmxml>) ................... 120
5.4.4. Hibernate Configuration file exporter (<hbm2cfgxml>) .............. 121
5.4.5. Documentation exporter (<hbm2doc>) .......................................... 122
5.4.6. Query exporter (<query>) ............................................................... 122
5.4.7. Generic Hibernate metamodel exporter (<hbmtemplate>) .......... 123
5.5. Using properties to configure Exporters ........................................... 124
5.5.1. <property> and <propertyset> .................................................... 124
5.5.2. Getting access to user specific classes ......................................... 124
6. Controlling reverse engineering ............................................................ 127
6.1. Default reverse engineering strategy ............................................... 127
6.2. hibernate.reveng.xml file ............................................................... 127
6.2.1. Schema Selection (<schema-selection>) ....................................... 129
6.2.2. Type mappings (<type-mapping>) ................................................. 129
6.2.3. Table filters (<table-filter>) .......................................................... 131
6.2.4. Specific table configuration (<table>) ........................................... 132
6.3. Custom strategy .............................................................................. 135
6.4. Custom Database Metadata ............................................................. 136
7. Controlling POJO code generation ......................................................... 137
7.1. The <meta> attribute ...................................................................... 137
7.1.1. Recommendations ...................................................................... 139
7.1.2. Advanced <meta> attribute examples ....................................... 142
Introduction

Hibernate Tools is a collection of tools for Hibernate 3 [http://www.hibernate.org/] and related projects. The tools provide Ant tasks and Eclipse plugins for performing reverse engineering, code generation, visualization and interaction with Hibernate.

1.1. Key Features

The table below lists the key features found in Hibernate Tools.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Generation through Ant Task</td>
<td>Allows to generate mapping or Java code through reverse engineering, schema generation and generation of other artifacts during the build process.</td>
<td>Chapter 5, Ant Tools</td>
</tr>
<tr>
<td>Wizards for creation purposes and code generation</td>
<td>A set of wizards are provided with the Hibernate Eclipse Tools™ to quickly create common Hibernate™ files such as configuration (cfg.xml) files, mapping files and reveng.xml as well. The Code Generation wizard helps by generating a series of various artifacts, and there is even support for completely reverse engineering an existing database schema.</td>
<td>Section 4.2, “Creating a Hibernate Mapping File”, Section 4.3, “Creating a Hibernate Configuration File”, Section 4.5.1, “Code Generation Launcher”</td>
</tr>
<tr>
<td>Mapping and Configuration files Editors</td>
<td>Support auto-completion and syntax highlighting. Editors also support semantic auto-completion for class names and property/field names, making it much more versatile than a normal XML editor.</td>
<td>Section 4.6, “Hibernate Mapping and Configuration File Editor”</td>
</tr>
<tr>
<td>Tools for organizing and controlling Reverse Engineering</td>
<td>The Code Generation wizard provides powerful functionality for generating a series of various artifacts such as domain model classes, mapping files, and annotated EJB3 entity beans, and the reveng.xml file editor provides control over this processes.</td>
<td>Section 4.5.1, “Code Generation Launcher”, Section 4.9</td>
</tr>
</tbody>
</table>
Chapter 1. Introduction

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hibernate Console</td>
<td>It is a new perspective in Eclipse which provides an overview of your Hibernate Console configurations, and were you also can get an interactive view of your persistent classes and their relationships. The console allows you to execute HQL queries against your database and browse the result directly in Eclipse.</td>
<td>“Reveng.xml Editor”</td>
</tr>
<tr>
<td>HQL Editor and Hibernate Criteria Editor</td>
<td>The editors are provided for writing, editing and executing HQL queries and criterias. They also have the ability to generate simple queries.</td>
<td>Section 4.10.2.1, “HQL Editor and Hibernate Criteria Editor”</td>
</tr>
<tr>
<td>Functional Mapping Diagram</td>
<td>Makes possible to visualize the structure of entities and the relationships between them.</td>
<td>Section 4.10.1.1, “Mapping Diagram”</td>
</tr>
<tr>
<td>Eclipse JDT integration</td>
<td>Hibernate Tools™ integrates into the Java code completion and build support for Java in Eclipse. This gives you HQL code completion inside Java code. Additionally, Hibernate Tools™ will display problem markers if your queries are not valid against the console configuration associated with the project.</td>
<td></td>
</tr>
</tbody>
</table>

1.2. Other relevant resources on the topic


Nightly documentation builds are available here [http://download.jboss.org/jbosstools/nightly-docs/].
Chapter 2.

Download and install Hibernate Tools

Hibernate Tools™ can be used "standalone" via Ant 1.6.x or fully integrated into an Eclipse + WTP based IDE, such as the case with JBDS/JBoss Tools™, or a default Eclipse + WTP installation. The following sections describe the install steps in these environments.

Note:
The Hibernate Tools 3.4.0™ (the current release version) requires Eclipse Helios (3.6).

2.1. JBoss Tools

JBoss Tools 3.4.0™ (the latest release) includes Hibernate Tools 3.3.0™ and thus no additional steps are required beyond downloading and installing JBoss Tools™. If you need to update to a newer version of the Hibernate Tools™ just follow the instructions in Section 2.2, “Eclipse IDE”.

2.2. Eclipse IDE

To install the Hibernate Tools™ into any Eclipse 3.6™ based IDE you can either use the JBoss Tools Update Site [http://download.jboss.org/jbosstools/updates/stable/].

Note:
If you need more detailed instructions on plugin installation and general usage of Eclipse™ then check out https://eclipse-tutorial.dev.java.net/ and especially https://eclipse-tutorial.dev.java.net/visual-tutorials/updatemanager.html which covers the use of the update manager.

2.2.1. Using Eclipse WTP

The Hibernate Tools™ plugins currently use WTP 3.x, which at this time is the latest stable release from the Eclipse Webtools™ project.

Because the WTP™ project has not always used proper versioning with their plugins, WTP™ plugins may be present in your existing Eclipse™ directory from other Eclipse™ based projects that are from an earlier WTP™ release but has either the same version number or higher. It is thus recommended that if you have issues with features provided by WTP™ to install the plugins on a clean install of Eclipse™ to ensure there are no version collisions.
2.3. Ant

To use the tools via Ant™ you need the hibernate-tools.jar file and associated libraries. The libraries are included in the distribution from the Hibernate™ website and the Eclipse™ update site. The libraries are located in the Eclipse™ plugins directory at /plugins/org.hibernate.eclipse.x.x.x/lib/tools/. These libraries are 100% independent from the Eclipse™ platform. How to use the Hibernate Tools™ via Ant™ tasks is described in Chapter 5, Ant Tools.
Code generation architecture

The code generation mechanism in Hibernate Tools™ consists of a few core concepts. This section explains their overall structure, which is the same for the Ant™ and Eclipse™ tools.

3.1. Hibernate Meta Model

The meta model is the model used by Hibernate Core™ to perform its object relational mapping. The model includes information about tables, columns, classes, properties, components, values, collections etc. The API is in the org.hibernate.mapping package and its main entry point is the Configuration class: the same class that is used to build a session factory.

The model represented by the Configuration class can be built in many ways:

- A Core configuration uses Hibernate Core™ and supports reading hbm.xml files, and requires a hibernate.cfg.xml file. This is referred to as Core in Eclipse and <configuration> in Ant.

- An Annotation configuration uses Hibernate Annotations™, supports hbm.xml files and annotated classes, and requires a hibernate.cfg.xml file. This is referred to as Annotations in Eclipse and <annotationconfiguration> in Ant.

- A JPA configuration uses a Hibernate EntityManager™ and supports hbm.xml files and annotated classes, and requires that the project has a META-INF/persistence.xml file in its classpath. This is referred to as JPA in Eclipse and <jpaconfiguration> in Ant.

- A JDBC configuration uses Hibernate Tools reverse engineering and reads its mappings via JDBC metadata + additional reverse engineering files (reveng.xml). Automatically used in Eclipse when doing reverse engineering from JDBC and referred to as <jdbcconfiguration> in Ant.

In most projects you will normally use only one of the Core, Annotation or JPA configuration and possibly the JDBC configuration if you are using the reverse engineering facilities of Hibernate Tools™.

Note:

Hibernate Tools has support for all kinds of Hibernate Configurations.

The following drawing illustrates the core concepts:
Chapter 3. Code generation ar...

Figure 3.1. Hibernate Core Concepts

The code generation is performed based on the Configuration model no matter which type of configuration has been used to create the meta model, and thus the code generation is independent on the source of the meta model and represented via Exporters.

3.2. Exporters

Code generation is performed by an Exporters. An Exporter is handed a Hibernate Meta Model represented as a Configuration instance, and it is then the job of the exporter to generate a set of code artifacts.

The tool provides a default set of Exporter's which can be used in both Ant and the Eclipse UI. Documentation for these Exporters can be found in Chapter 5, Ant Tools and Chapter 4, Eclipse Plugins.

Users can provide their own custom Exporter's, either through custom classes implementing the Exporter interface or simply through custom templates. This is documented at in Section 5.4.7, "Generic Hibernate metamodel exporter (<hbmtemplate>)."
Chapter 4.

Eclipse Plugins

This chapter will introduce you to the set of wizards and editors provided by Hibernate Tools™ within Eclipse, which are designed to simplify working with Hibernate™.

4.1. Introduction

Hibernate Eclipse Tools includes wizards for creating Hibernate mapping files, configuration files (.cfg.xml), reveng.xml files as well as wizards for adjusting Console Configuration and Code Generation. Special structured and XML editors and editors for executing HQL and Criteria queries are also provided in Hibernate Console. Refer to Section 1.1, “Key Features” to find all the benefits that are provided by these tools within Eclipse.

Note:

Please note that these tools do not try to hide any of Hibernates™ functionality; rather the tools make working with Hibernate™ easier. You are still encouraged to read the Hibernate Documentation [http://hibernate.org/docs] in order to be able to fully utilize Hibernate Tools™ and especially Hibernate™ itself.

4.1.1. Download base project

You can download the example projects that are used in this chapter.

A JPA base project is available on the documentation resources page [http://docs.jboss.org/tools/resources/] together with a base Java project [http://docs.jboss.org/tools/resources/TestHibernateproject_for_hibernate_jboss_tools.zip].

Also you need start the database [http://docs.jboss.org/tools/resources/GSG_database.zip].

Note:


4.2. Creating a Hibernate Mapping File

Hibernate mapping files are used to specify how your objects relate to database tables.

To create basic mappings for properties and associations, i. e. generate .hbm.xml files, Hibernate Tools provide a basic wizard which you can display by selecting File → New → Hibernate XML mapping file.
At first you will be asked to select a package or multiple individual classes to map. It is also possible to create an empty file: do not select any packages or classes and an empty .hbm file will be created in the specified location.

Using the depth control option you can define the dependency depth used when choosing classes.
Figure 4.1. Hibernate XML Mapping File Wizard
Chapter 4. Eclipse Plugins

The next wizard page lists the mappings to be generated. You can see the **Customers**, **Orders**, **Productlines** and **Products** classes added under depth control driving.
Figure 4.2. Mappings to be generated
This wizard page display a preview of the generated .hbm files.
Figure 4.3. Preview Generated Mapping Files
Clicking the Finish button creates the files.

### 4.3. Creating a Hibernate Configuration File

To be able to perform reverse engineering, prototype queries, and of course to simply use Hibernate Core a hibernate.properties or hibernate.cfg.xml file is needed. Hibernate Tools provides a wizard for generating the hibernate.cfg.xml file if you do not already have one.

This video illustrates the creation of a Hibernate Configuration File for an existing project

**Figure 4.4. Hibernate Configuration File creation video**

Start the wizard by clicking File → New → Other (Ctrl+N), then select Hibernate → Hibernate Configuration File (cfg.xml) and click the Next button.
**Creating a Hibernate Configuration File**

Figure 4.5. Hibernate Configuration File Wizard

<table>
<thead>
<tr>
<th><strong>Container:</strong></th>
<th>/TestHibernate/src</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>File name:</strong></td>
<td>hibernate.cfg.xml</td>
</tr>
<tr>
<td><strong>Session factory name:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Database dialect:</strong></td>
<td>HSQL</td>
</tr>
<tr>
<td><strong>Driver class:</strong></td>
<td>org.hsqldb.jdbcDriver</td>
</tr>
<tr>
<td><strong>Connection URL:</strong></td>
<td>jdbc:hsqldb:hsqldb://localhost/database/db</td>
</tr>
<tr>
<td><strong>Default Schema:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Default Catalog:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Username:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Password:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Create a console configuration</strong></td>
<td>✓</td>
</tr>
</tbody>
</table>
Note:
The contents in the comboboxes for the JDBC driver class and JDBC URL change automatically, depending on the Dialect and actual driver you have chosen.

Enter your configuration information in this dialog. Details about the configuration options can be found in *Hibernate Reference Documentation* [http://docs.jboss.org/ejb3/app-server/Hibernate3/reference/en/html_single].

Click the Finish button to create the configuration file, and after optionally creating a Console configuration, the hibernate.cfg.xml file will be automatically opened in an editor. The last option, Create Console Configuration, is enabled by default and when enabled, it will automatically use the hibernate.cfg.xml file for the basis of a Console configuration.

### 4.4. Hibernate Console Configuration

A Console configuration describes how the Hibernate plugin should configure Hibernate and what configuration files and classpaths are needed to load the POJO's, JDBC drivers etc. It is required to make use of query prototyping, reverse engineering and code generation. You can have multiple named console configurations. Normally you would just need one per project, but it is definitely possible to create more if required.

#### 4.4.1. Creating a Hibernate Console Configuration

You can create a console configuration by running the Console Configuration Wizard, shown in the following screenshot. The same wizard will also be used if you are coming from the hibernate.cfg.xml wizard and had enabled the Create Console Configuration option.

Note:
The wizard will look at the current selection in the IDE and try to auto-detect the appropriate settings, which you then can approve or modify to suit your needs.

The dialog consists of five tabs:

- **Main**, which displays the basic and required settings
The following table describes the available settings on the **Main** tab. The wizard can automatically detect the default values for most of the settings if you started the wizard with the relevant Java project or resource selected.
### Table 4.1. Hibernate Console Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The unique name of the console configuration</td>
<td>Name of the selected project</td>
</tr>
<tr>
<td>Type</td>
<td>Choose between &quot;Core&quot;, &quot;Annotations&quot; and &quot;JPA&quot;. Note that the two latter</td>
<td>Core and Hibernate Version: 3.5</td>
</tr>
<tr>
<td></td>
<td>requires running Eclipse IDE with a JDK 5 (or newer) runtime, otherwise you</td>
<td></td>
</tr>
<tr>
<td></td>
<td>will get class loading and version errors.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>You will also need to select the version of Hibernate your application</td>
<td></td>
</tr>
<tr>
<td></td>
<td>server has (3.5, 3.6 or 4.0). Select a version from the drop-down box</td>
<td></td>
</tr>
<tr>
<td></td>
<td>labelled Hibernate Version.</td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td>The name of a Java project whose classpath should be used in the console</td>
<td>Name of the selected project</td>
</tr>
<tr>
<td></td>
<td>configuration</td>
<td></td>
</tr>
<tr>
<td>Database connection</td>
<td>DTP provided connection that you can use instead of what is defined in the</td>
<td>[Hibernate Configured connection]</td>
</tr>
<tr>
<td></td>
<td>cfg.xml and JPA persistence.xml files. It is possible to use an already</td>
<td></td>
</tr>
<tr>
<td></td>
<td>configured Hibernate or JPA connection, or specify a new one here.</td>
<td></td>
</tr>
<tr>
<td>Property file</td>
<td>Path to a hibernate.properties file</td>
<td>First hibernate.properties file found in the selected project</td>
</tr>
<tr>
<td>Configuration file</td>
<td>Path to a hibernate.cfg.xml file</td>
<td>First hibernate.cfg.xml file found in the selected project</td>
</tr>
<tr>
<td>Persistence unit</td>
<td>Name of the persistence unit to use</td>
<td>No default value (let Hibernate Entity Manager find the persistence unit or it</td>
</tr>
<tr>
<td></td>
<td>can be defined manually using the Browse button)</td>
<td></td>
</tr>
</tbody>
</table>
Tip:
The two latter settings are usually not required if you specify a project that has a `/hibernate.cfg.xml` or `/META-INF/persistence.xml` file in its classpath.

- The **Options** tab for the optional settings
Chapter 4. Eclipse Plugins

Figure 4.7. Options Tab of the Console Configuration Wizard
The next table describes the Hibernate Console Configuration options available on the Options tab.

**Table 4.2. Hibernate Console Configuration Options**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Auto detected value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database dialect</td>
<td>Define a database dialect. It is possible either to enter your own value or choose from list.</td>
<td>No default value</td>
</tr>
<tr>
<td>Naming strategy</td>
<td>Fully qualified class name of a custom NamingStrategy. Only required if you use a special naming strategy.</td>
<td>No default value</td>
</tr>
<tr>
<td>Entity resolver</td>
<td>Fully qualified class name of a custom EntityResolver. Only required if you have special XML entity includes in your mapping files.</td>
<td>No default value</td>
</tr>
</tbody>
</table>

- **Classpath** for classpath
Figure 4.8. Specifying Classpath in Hibernate Console Configuration
The following table specifies the parameters of the **Classpath** tab of the wizard.

**Table 4.3. Hibernate Console Configuration Classpath**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Auto detected value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classpath</td>
<td>The classpath for loading POJO and JDBC drivers; only needed if the default classpath of the Project does not contain the required classes. Do not add Hibernate core libraries or dependencies, they are already included. If you get ClassNotFound errors then check this list for possible missing or redundant directories or JAR files.</td>
<td>Empty</td>
</tr>
<tr>
<td>Include default classpath from project</td>
<td>When enabled the project classpath will be appended to the classpath specified above</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

- **Mappings** for additional mappings
Figure 4.9. Specifying additional Mappings in Hibernate Console Configuration

Parameters of the **Mappings** tab of the Hibernate Console Configuration wizard are explained below:

**Table 4.4. Hibernate Console Configuration Mappings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Auto detected value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping files</td>
<td>List of additional mapping files that should be loaded. Note: A <em>hibernate.cfg.xml</em> or <em>persistence.xml</em> can also contain mappings. Thus if these are duplications here, you will get &quot;Duplicate mapping&quot; errors when using the console configuration.</td>
<td>empty</td>
</tr>
</tbody>
</table>

• and the last tab **Common**
Figure 4.10. Common Tab of the Console Configuration Wizard
Chapter 4. Eclipse Plugins

The **Common** tab allows you to define the general aspects of the launch configuration including storage location, console encoding and some others.

Clicking the **Finish** button creates the configuration and shows it in the Hibernate Configurations view.
Figure 4.11. Console Overview
4.4.2. Modifying a Hibernate Console Configuration

When you created a Hibernate Console Configuration you can modify it in two ways:

- Right-click on the configuration in the Hibernate Configurations View and select Edit Configuration, or just double-click on the Console Configuration item.
Modifying a Hibernate Console Configuration

Figure 4.12. Opening Edit Configuration Wizard
You will then see the **Edit Configuration Wizard**, which is similar to **Create Console Configuration**, described in Section 4.4.1, “Creating a Hibernate Console Configuration”.

- Use the **Properties** view to modify the Console Configuration properties.

---

**Figure 4.13. Properties View**
The following table describes the available settings in the Properties view. Most properties can be changed by left clicking on them, but some can not.

**Table 4.5. Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Is Changeable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additional mapping files</td>
<td>Count of additional mapping files that should be loaded.</td>
<td>False</td>
</tr>
<tr>
<td>Configuration file</td>
<td>Path to a <code>hibernate.cfg.xml</code> file</td>
<td>False</td>
</tr>
<tr>
<td>Connection</td>
<td>DTP provided connection that you can use instead of what is in the <code>cfg.xml</code> and JPA <code>persistence.xml</code> files. It is possible to use either an already configured Hibernate or JPA connection, or specify a new one here.</td>
<td>True</td>
</tr>
<tr>
<td>Name</td>
<td>The unique name of the console configuration</td>
<td>True</td>
</tr>
<tr>
<td>Project</td>
<td>The name of a Java project which classpath should be used in the console configuration</td>
<td>True</td>
</tr>
<tr>
<td>Properties file</td>
<td>Path to a <code>hibernate.properties</code> file</td>
<td>False</td>
</tr>
<tr>
<td>Type</td>
<td>Choose between &quot;CORE&quot;, &quot;ANNOTATIONS&quot; and &quot;JPA&quot; according to the method of relational mapping you want to use. Note, the two latter requires running Eclipse IDE with a JDK 5 runtime, otherwise you will get class loading and/or version errors.</td>
<td>True</td>
</tr>
</tbody>
</table>

### 4.4.3. Closing Hibernate Console Configuration

To close **Hibernate Console Configuration** you need to right-click your configuration and choose the **Close Configuration** option.
Figure 4.14. Close Hibernate Console Configuration
When closing the configuration the connection with database will be closed, JAR libs will be unlock (for Windows) and other resources will set as free.

4.5. Reverse Engineering and Code Generation

Hibernate provides "click-and-generate" reverse engineering and code generation facilities. This allows you to generate a range of artifacts based on database or an existing Hibernate configuration, be that mapping files or annotated classes. Some of these are POJO Java source files, Hibernate .hbm.xml files, hibernate.cfg.xml generation and schema documentation.

To start working with this process, start the Hibernate Code Generation tool which is available from the toolbar via the Hibernate icon or via the Run → Hibernate Code Generation menu item.

4.5.1. Code Generation Launcher

When you click on the Open Hibernate Code Generation Dialog... option the standard Eclipse launcher dialog will appear. In this dialog you can create, edit and delete named Hibernate code generation "launchers".

![Figure 4.15. Getting Hibernate Code Generation Launcher](image)
Figure 4.16. Hibernate Code Generation Launcher
The first time you create a code generation launcher you should give it a meaningful name, otherwise the default prefix New_Generation will be used.

Tip:

The "At least one exporter option must be selected" warning indicates that for this launcher to work you need to select an exporter on the Exporter tab. When an exporter has been selected the warning will disappear.

The dialog also has the standard Refresh and Common tabs that can be used to configure which directories should be automatically refreshed and various general settings for launchers, such as saving them in a project for sharing the launcher within a team.

On the Main tab you see the following fields:

Table 4.6. Code generation "Main" tab fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Console Configuration</td>
<td>The name of the console configuration that should be used when generating code</td>
</tr>
<tr>
<td>Output directory</td>
<td>The default location where all output will be written to. It's possible to enter absolute directory path, for example - d:/temp. Be aware that existing files will be overwritten, so be sure to specify the correct directory.</td>
</tr>
<tr>
<td>Reverse engineer from JDBC Connection</td>
<td>If enabled, the tools will reverse engineer the database defined in the connection information in the selected Hibernate Console Configuration, and generate code based on the database schema. If not enabled, the code generation will be based on the existing mappings specified in the Hibernate Console configuration.</td>
</tr>
<tr>
<td>Package reveng.xml</td>
<td>The package name here is used as the default package name for any entities found when reverse engineering</td>
</tr>
<tr>
<td>Package reveng. strategy</td>
<td>Path to a reveng.xml file. A reveng.xml file allows you to control certain aspects of the reverse engineering process such as how JDBC types are mapped to Hibernate types, and which tables are included and excluded from the process (which is especially important). Clicking the Setup button allows you to select an existing reveng.xml file, or create a new one. See more details about the reveng.xml file in Chapter 6, Controlling reverse engineering.</td>
</tr>
<tr>
<td>Package reveng. strategy</td>
<td>If the reveng.xml file does not provide enough customization you can provide your own implementation of a ReverseEngineeringStrategy. The class needs to be in the classpath of the Console Configuration, otherwise you will get class not found exceptions. See Section 6.3, “Custom strategy” for details and an example of a custom strategy.</td>
</tr>
</tbody>
</table>
### Chapter 4. Eclipse Plugins

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate basic typed composite ids</td>
<td>When a table that has a multi-column primary key a <code>&lt;composite-id&gt;</code> mapping will always be created. If this option is enabled and there are matching foreign-keys, each key column is still considered a 'basic' scalar (string, long, etc.) instead of a reference to an entity. If you disable this option a <code>&lt;key-many-to-one&gt;</code> instead. Note: a <code>&lt;many-to-one&gt;</code> property is still created, but is simply marked as non-updatable and non-insertable.</td>
</tr>
<tr>
<td>Detect optimistic lock columns</td>
<td>Automatically detect optimistic lock columns. Controllable via reveng. strategy; the current default is to use columns named VERSION or TIMESTAMP.</td>
</tr>
<tr>
<td>Detect many-to-many tables</td>
<td>Automatically detect many-to-many tables. Controllable via reveng. strategy.</td>
</tr>
<tr>
<td>Detect one-to-one associations</td>
<td>Reverse engineering detects one-to-one associations via primary key and both the hbm.xml file and annotation generation generates the proper code for it.</td>
</tr>
<tr>
<td></td>
<td>The detection is enabled by default (except for Seam 1.2 and Seam 2.0) reverse engineering. For Hibernate Tools generation there is a checkbox to disable this feature if it is not required.</td>
</tr>
<tr>
<td>Use custom templates</td>
<td>If enabled, the Template directory will be searched first when looking up the templates, allowing you to redefine how the individual templates process the hibernate mapping model.</td>
</tr>
<tr>
<td>Template directory</td>
<td>A path to a directory with custom templates</td>
</tr>
</tbody>
</table>

### 4.5.2. Exporters

The **Exporters** tab is used to specify the type of code that should be generated. Each selection represents an Exporter that is responsible for generating the code, hence the name.
Figure 4.17. Selecting Exporters
The following table provides a short description of the various exporters. Remember you can add and remove any Exporters depending on your needs.

**Table 4.7. Code generation "Exporter" tab fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain code</td>
<td>Generates POJO's for all the persistent classes and components found in the given Hibernate configuration.</td>
</tr>
<tr>
<td>DAO code</td>
<td>Generates a set of DAO's for each entity found.</td>
</tr>
<tr>
<td>Hibernate XML Mappings</td>
<td>Generate mapping (hbm.xml) files for each entity.</td>
</tr>
<tr>
<td>Hibernate XML Configuration</td>
<td>Generate a hibernate.cfg.xml file. Used to keep the hibernate.cfg.xml file updated with any newly discovered mapping files.</td>
</tr>
<tr>
<td>Schema Documentation (.html)</td>
<td>Generates a set of HTML pages that documents the database schema and some of the mappings.</td>
</tr>
<tr>
<td>Generic Exporter (hbmtemplate)</td>
<td>Fully customizable exporter that can be used to perform custom generation.</td>
</tr>
<tr>
<td>Schema Export (.ddl)</td>
<td>Generates the appropriate SQL DDL and allows you to store the result in a file or export it directly to the database.</td>
</tr>
</tbody>
</table>

Each Exporter listens to certain properties which can be setup in the Properties section where you can add and remove predefined or customer properties for each of the exporters. The following table lists the time of writing predefined properties:

**Table 4.8. Exporter Properties**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdk5</td>
<td>Generate Java 5 syntax</td>
</tr>
<tr>
<td>ejb3</td>
<td>Generate EJB 3 annotations</td>
</tr>
<tr>
<td>for_each</td>
<td>Specifies for which type of model elements the exporter should create a file and run through the templates. Possible values are: entity, component, configuration</td>
</tr>
<tr>
<td>template_path</td>
<td>Custom template directory for this specific exporter. You can use Eclipse variables.</td>
</tr>
<tr>
<td>template_name</td>
<td>Name for template relative to the template path</td>
</tr>
<tr>
<td>outputdir</td>
<td>Custom output directory for this specific exporter. You can use Eclipse variables.</td>
</tr>
<tr>
<td>file_pattern</td>
<td>Pattern to use for the generated files, relatively for the output dir. Example: {package-name}/{class-name}.java.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>dot.executable</td>
<td>Executable to run GraphViz (only relevant, but optional for Schema documentation)</td>
</tr>
<tr>
<td>drop</td>
<td>Output will contain drop statements for the tables, indices and constraints</td>
</tr>
<tr>
<td>delimiter</td>
<td>If specified the statements will be dumped to this file</td>
</tr>
<tr>
<td>create</td>
<td>Output will contain create statements for the tables, indices and constraints</td>
</tr>
<tr>
<td>scriptToConsole</td>
<td>The script will be output to Console</td>
</tr>
<tr>
<td>exportToDatabase</td>
<td>Executes the generated statements against the database</td>
</tr>
<tr>
<td>outputFileName</td>
<td>If specified the statements will be dumped to this file</td>
</tr>
<tr>
<td>haltOnError</td>
<td>Halts the build process if an error occurs</td>
</tr>
<tr>
<td>format</td>
<td>Applies basic formatting to the statements</td>
</tr>
<tr>
<td>schemaUpdate</td>
<td>Updates a schema</td>
</tr>
</tbody>
</table>

To add a property to the chosen Exporter click the **Add** button in the **Properties** section. In the resulting dialog you should select the property from the proposed list and the value for it.
Create, manage, and run configurations
Select or configure a code generation

Figure 4.18. Adding the Property for Schema Export (.ddl)
Tip:
If the property is a directory, it is possible to browse directories in the Value field.

Figure 4.19. Specifying the Property Value
4.6. Hibernate Mapping and Configuration File Editor

The Hibernate Mapping File editor provides XML editing functionality for the `hbm.xml` and `cfg.xml` files. The editor is based on the Eclipse WTP tools and extends its functionality to provide Hibernate specific code completion.

Figure 4.20. XML Editing Functionality

```xml
<ns name="demo.Customers" table="CUSTOMERS" schema="PUBLIC">
  <id name="customernumber" type="int">
    <column name="CUSTOMERNUMBER" />
    <generator class="assigned" />
  </id>
  <many-to-one name="employees" class="demo.Employees" fetch="select">
    <column name="SALESREPREMPLYEENUMBER" />
  </many-to-one>
  <property name="customercname" type="string">
    <column name="" />
  </property>
  <property name="ph">
    <column name="" />
  </property>
  <property name="ci">
    <column name="" />
  </property>
  <property name="po">
    <column name="" />
  </property>
  <property name="co">
    <column name="" />
  </property>
  <property name="cr">
    <column name="" />
  </property>
  <set name="orderses" table="ORDERS" inverse="true" lazy="true" fetch="select">
    <key>
      <column name="CUSTOMERNUMBER" not-null="true" />
    </key>
    <one-to-many class="demo.Orders" />
  </set>
</ns>
```
4.6.1. Java property/class completion

Package, class, and field completion is enabled for relevant XML attributes. The auto-completion tool detects its context and limits the completion for a tag (such as `<property>`) and only shows the properties and fields available in the enclosing `<class>`, `<subclass>` etc. It is also possible to navigate from the hbm.xml files to the relevant classes and fields in your Java code.

Figure 4.21. Navigation Functionality

This is done via the standard hyperlink navigation functionality in Eclipse. By default this is done by pressing F3 while the cursor is on a class or field, or by pressing Ctrl and the mouse button.

For Java completion and navigation to work the file needs to reside inside an Eclipse Java project, otherwise the feature is not available.

Note:

Java completion does not require a Hibernate console configuration to be used.
4.6.2. Table/Column completion

Table and column completion is also available for all table and column attributes.

Important:
Table and Column completion requires a properly configured Hibernate console configuration, and this configuration should be the default for the project where the hbm.xml resides.

You can check which console configuration is selected in the Properties of a project under the Hibernate Settings page. When a proper configuration is selected it will be used to fetch the table and column names in the background.

Note:
Currently it is not recommended to use this feature on large databases since it does not fetch the information iteratively. This will be improved in future versions.

4.6.3. Configuration property completion

Code completion for the value of \<property\> name attributes are available when editing the cfg.xml file.
Figure 4.23. Property Completion
4.7. Structured Hibernate Mapping and Configuration File Editor

The structured editor represents a file in a tree form. It also provides a way to modify the structure of the file and its elements with the help of tables provided on the right-hand area.

To open any mapping file in the editor, select **Open With → Hibernate 3.0 XML Editor** from the context menu of the file. The editor is shown in the following image:
Structured Hibernate Mapping and Configuration File Editor

Figure 4.24. Structured hbm.xml Editor
For the configuration file you should select **Open With → Hibernate Configuration 3.0 XML Editor**.
Structured Hibernate Mapping and Configuration File Editor

Figure 4.25: Structured cfg.xml Editor
4.8. JBoss Tools Properties Editor

The editor is designed to edit properties files. It contains two tabs: the Properties (UI) tab and the Source tab for manual editing.

For hibernate.properties files the JBoss Tools Properties Editor provides content assist for both Hibernate properties and values. You can make use of the content assist while editing the file in the Source view and in the Properties view of the editor.

To add the property in the Properties view, click the Add button.
Figure 4.26: Adding the Property
In the **Name** field press `Ctrl+Space` to invoke the content assist. It will suggest `hibernate.` which is the prefix for all hibernate properties. After selecting `hibernate.` and invoking the content assist again, other prefixes and properties are displayed as the proposals, with a description for each one.

![Content Assist for Properties Names](image)

**Figure 4.27. Content Assist for Properties Names**

When invoking the content assist in the **Value** field, it also provides a list of proposals.
Figure 4.28. Content Assist for Properties Values

In the Source view of the editor, content assist can also be invoked both for properties names and values:
Figure 4.29. Content Assist in the Source view
Figure 4.30: Content Assist in the Source view
4.9. Reveng.xml Editor

A reveng.xml file is used to customize and control how reverse engineering is performed by the tools. The plugins provide an editor to assist in editing this file.

The editor is intended to allow easy definition of type mappings, table include and excludes, and specific override settings for columns, e.g. define an explicit name for a column when the default naming rules are not applicable.

Note:

Not all the features of the .reveng.xml file are exposed or fully implemented in the editor, but the main functionality is there. To understand the full flexibility of the reveng.xml file, please see Section 6.2, “hibernate.reveng.xml file”.

The editor is activated as soon as a .reveng.xml file is opened. To create an initial reveng.xml file the Reverse Engineering File Wizard can be started by pressing Ctrl+N and then selecting Hibernate → Hibernate Reverse Engineering File (reveng.xml).
Figure 4.31. Choose Reverse Engineering File Wizard

Or you can get it via the Code Generation Launcher by checking the appropriate section in the Main tab of the Getting Hibernate Code Generation Wizard (see Figure 4.15, “Getting Hibernate Code Generation Launcher”).

The following screenshot shows the Overview page where the appropriate console configuration is selected (it is auto-detected if Hibernate 3 support is enabled for the project).
Figure 4.32. Overview Page

The **Table Filter** page allows you to specify which tables to include and exclude. Clicking the **Refresh** button shows the tables from the database that have not yet been excluded.
The Type Mappings page is used to specify type mappings from JDBC types to any Hibernate type (including user types) if the default rules are not applicable. To see the database tables press the Refresh button underneath. For more information on type mappings please see the Section 6.2.2, “Type mappings (<type-mapping>)” section.
The Table and Columns page allows you to explicitly set which details (e.g. which hibernate-type and property-name) should be used in the reverse engineered model. For more details on how to configure the tables while reverse engineering read Section 6.2.4, "Specific table configuration (<table>)."
The Hibernate Console Perspective combines a set of views which allow you to see the structure of your mapped entities and classes, edit HQL queries, execute the queries, and view the results.

**Figure 4.35. Table and Columns Page**

**4.10. Hibernate Console Perspective**

The Hibernate Console Perspective combines a set of views which allow you to see the structure of your mapped entities and classes, edit HQL queries, execute the queries, and view the results.
To use this perspective you need to create a Hibernate Console Configuration (see Section 4.4, “Hibernate Console Configuration” for more information).

4.10.1. Viewing the entity structure

To view your new configuration and entity or class structure, switch to the Hibernate Configurations View. Expanding the tree allows you to browse the class or entity structure, as well as view the relationships.
Figure 4.36. Hibernate Console Perspective
Chapter 4. Eclipse Plugins

The Console Configuration does not dynamically adjust to changes performed in mappings and Java code. To reload the configuration select the configuration and click the Reload button in the view toolbar or in the context menu.

It is possible to open source and mapping files for objects showed in the Hibernate Configurations View. Just bring up the context menu for an object and select Open Source File to see the appropriate Java class or Open Mapping File to open a .hbm.xml file.
Viewing the entity structure

Figure 4.37 Opening Source for Objects
4.10.1.1. Mapping Diagram

In order to visualize how entities are related, as well as view their structures, a **Mapping Diagram** is provided. It is available by right clicking on the entity you want view a mapping diagram for and then selecting **Mapping Diagram**.

![Figure 4.38. Mapping Diagram](image-url)
To make Mapping Diagram usage easier you can use the Rules, Grid, Snap to Geometry checkboxes in the View menu.

![View menu](image)

**Figure 4.39. View menu**

If you select the Rules checkbox, the view print page scale will be added to the page. The numbers on the scale displays its size in inches. If you click on the scale a Ruler Guide will appear on the diagram. You can connect any diagram item to it. To connect the items you should move their tops to the Ruler Guide. And while moving the ruler guide, the items will be moved together with it as a whole.
Figure 4.40. Moving the Ruler guide
If you select the **Grid** checkbox, a grid will appear on the diagram.
Figure 4.41. Grid on Mapping diagram
The **Snap to Geometry** checkbox allows you to align the diagram items with the grid.

For better navigating through the diagram use the **Outline view**, which is available in the structural and graphical modes.
Figure 4.42. Navigating in the Structural Mode
To switch between the view modes, use the buttons in the top-right corner of the **Outline view**.

![Outline view](image)

**Figure 4.43. Navigating in the Graphical Mode**

The options in the context menu of the mapping diagram are listed in the next table.

**Table 4.9. Context Menu Options of the Mapping Diagram**

<table>
<thead>
<tr>
<th>Icon</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>Show</td>
<td>Hide connections</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Property Mappings</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Class Mappings</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Associations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Foreign key constraints</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>Select All</td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>Auto layout</td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>Export as Image</td>
<td></td>
</tr>
</tbody>
</table>
When you open the context menu for an item in the diagram, it differs quite significantly from the one described before.
Figure 4.44. Context Menu in Mapping Item
The next table describes the additional options found in the mapping items context menu:

### Table 4.10. Extra Options in the Context Menu of Mapping Item

<table>
<thead>
<tr>
<th>Icon</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>Open Source File</td>
<td>Opens the source file for a chosen object or element. The selected element will be highlighted in the open file.</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>Open Mapping File</td>
<td>Opens a mapping file for a chosen object or element. The selected element will be highlighted in the open file.</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>Show</td>
<td>Hide shape(s)</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>Expand</td>
<td>Collapse shape(s)</td>
</tr>
</tbody>
</table>

**Tip:**

All these context menu options are also available in the Outline view.

The following table lists the available keyboard shortcuts.

### Table 4.11. Hibernate Mapping Diagram Shortcut Keys

<table>
<thead>
<tr>
<th>Command</th>
<th>Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scroll the diagram content</td>
<td>Ctrl+Shift+arrows</td>
</tr>
<tr>
<td>Collapse or Expand selected item(s)</td>
<td>Enter</td>
</tr>
<tr>
<td>Show or Hide selected item(s)</td>
<td>+</td>
</tr>
<tr>
<td>Sort items in alphabetical order or return to the initial state</td>
<td>Space</td>
</tr>
<tr>
<td>Navigate between the items</td>
<td>Arrows</td>
</tr>
</tbody>
</table>

It is possible to save the diagram in the Eclipse workspace. Select **File → Save As**, and the wizard will ask you to specify the location within you project where you wish to save the file, and provide the name for the diagram. The default name is the item’s names concatenated with the ampersand symbols. The file is saved with the `.hibernate` extension.
Figure 4.45. The Diagram saved in the Workspace

Note:

If you restart Eclipse with the mapping diagram opened, the mapping diagram will be restored with the message like on the figure below. To view the diagram content, you should refresh the view.
Hibernate Console Configuration is not loaded cause Hibernate...
Figure 4.47. The Diagram after Refreshing
There are some useful commands in the toolbar.

Figure 4.48. The Diagram View Toolbar

They are described in the table below.

Table 4.12. Command in Diagram View Toolbar

<table>
<thead>
<tr>
<th>Icon</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>🔄</td>
<td>Refresh Visual Mapping</td>
<td>It will update the Mapping Diagram if Console Configuration was changed.</td>
</tr>
<tr>
<td>🔍</td>
<td>Zoom Box</td>
<td>Used to define scale of the diagram. It is also used for printing Mapping Diagrams. If you want to print the whole diagram to one page, you need select the Page option in the Zoom Box drop down list.</td>
</tr>
<tr>
<td>🚁</td>
<td>Auto layout</td>
<td>Used to arrange all diagram items in a standard manner.</td>
</tr>
<tr>
<td>🔥</td>
<td>Show</td>
<td>Hide connections</td>
</tr>
<tr>
<td>💾</td>
<td>Expand</td>
<td>Collapse</td>
</tr>
<tr>
<td>✅</td>
<td>Show</td>
<td>Hide shape(s)</td>
</tr>
</tbody>
</table>

4.10.2. Prototyping Queries

Queries can be prototyped by entering them into the HQL or Criteria Editor. To execute a query click the green run button in the editor toolbar, or press Ctrl+Enter.

4.10.2.1. HQL Editor and Hibernate Criteria Editor

To open the query editors right-click your projects Console Configuration and select HQL Editor (or Hibernate Criteria Editor).
Tip:

If the context menu items are disabled then you need at first to create a Session Factory. That is done by expanding the Session Factory node.

When they are opened, the editors they should automatically detect the chosen Console Configuration.

To generate a predefined query for any entity (or any entity child node) listed in the Session Factory you should double-click it. This will open the HQL Editor with the associated query.
Choosing **HQL Editor** in the context menu for any entity (or any entity child node) will also open the HQL Editor with the associated query. If you select **Hibernate Criteria Editor** in the context menu, it will open **Hibernate Criteria Editor** with the associated criteria.

![Figure 4.50. Generating Simple Queries](image)

It is also possible to copy a portion of code from a `.java` file into the **HQL** or **Criteria editor**. To do this make use of the Quick Fix option (**Ctrl+1**).
You can also update the original Java code with changes made in the HQL or Criteria editor. For that you should save your HQL/Criteria query and submit the replacement code when prompted by the confirmation dialog.
Figure 4.52. Updating Java Code

In addition, you can pin the HQL editor and Criteria editor as a tab in the Hibernate Query Result view. For that you need click on the Stick result to one tab button (

In this state query executions results will be shown in one tab (no more will be opened).

You are able to rename the Hibernate Query Result tab. Click the tab, and type a new name in the Property View → Tab name field.
4.10.2.2. Error Handling

Errors raised during the creation of the **Session Factory**, or when executing the queries (e.g. if your configuration or query is incorrect), will be shown in a message dialog or inclined in the view that detected the error. You may get more information about the error in the **Error Log** view on the right pane.

Results of a query will be shown in the **Hibernate Query Result** view and details of possible errors (syntax errors, database errors, etc.) can be seen in the **Error Log** view.

**Note:**

HQL queries are executed by default using the `list()` function, and without any row limit could return a large result set. You may run out of memory. To avoid this you can enter a value in the **Max results** field to reduce the number of elements that are returned.

4.10.2.3. Dynamic Query Translator

If the **Hibernate Dynamic Query Translator** view is visible, it will show the generated SQL for a HQL query while you write in the **HQL Editor**.
Figure 4.54. Hibernate Dynamic Query Translator View

The translation is performed each time you stop typing in the editor. If there are errors in the HQL code the parse exception will be shown embedded in the view.

4.10.3. Properties View

As you can see in the figure below, when clicking on class or entity the Properties view shows the number of query results as well as the execution time.
Figure 4.55. Properties View

It also displays the structure of any persistent object selected in the Hibernate Query Results View. Editing is not yet supported.

Figure 4.56. Properties View for Selected Object
You can also use Properties view when clicking on the configuration itself in Hibernate Configuration view (see Section 4.4.2, “Modifying a Hibernate Console Configuration”).

### 4.11. Hibernate: add JPA annotations refactoring

Using this wizard you can add the following Hibernate annotations to a class: @Column, @Entity, @ManyToOne, @OneToMany, @OneToOne, @ManyToMany, @MappedSuperclass, @Id, @GeneratedValue, @Version

- @Column is added to all String properties.
- @Entity is always declared before any class where it has not yet been defined.
- @ManyToOne, @OneToMany, @OneToOne, @ManyToMany - these annotations are declared according to the classes hierarchy.
- @MappedSuperclass is added to abstract superclasses.
- @Id, @GeneratedValue are only added automatically to the properties under the name “Id”, where they have not yet been defined.
- @Version is declared in case you select optimistic locking (see Section 4.11, “Hibernate: add JPA annotations refactoring” [92]).

**Note:**

This section doesn't cover the definitions of the Hibernate annotations. For more information read the Hibernate Annotations Documentation [http://docs.jboss.org/hibernate/stable/annotations/reference/en/html/].

To open this wizard you should right click the class you want to add the annotations to and select Source → Generate Hibernate/JPA annotations from the context menu. You will see the Hibernate: add JPA annotations dialog.
Figure 4.57. Starting Hibernate: add JPA annotations dialog
The top section of this dialog lists all the classes that will be passed through refactoring. Next to the class you have selected, this list also displays its superclasses and the classes that the objects present in the current class as properties. If you want to add new classes or packages to the list, you should click the Back button. This will display the Add classes and packages page.
Figure 4.58. Add classes and packages page
Here you can add additional classes or entire packages, and you can limit the dependencies depth by selecting the depth control option (you can find more information on this option in Section 4.2, “Creating a Hibernate Mapping File”). When you are finished click the Next button and you will be returned to The following classes will be changed page.

By default the tags are added to the fields of selected classes. You can change this option to Getters in the Preferred location of Annotations drop down list, which results in the annotations being added to the getter methods. If you choose Auto select from class preference then the annotations are added according to the position of the majority of the existing annotations.

If it is necessary to map your String properties to the columns that length differ from the default value (255), change the Default string length field and the @Column(length = your length) annotation will be created for every String property.

You can add optimistic locking capabilities to an entity bean by selecting the Enable optimistic locking checkbox. This operation will add the version property to all the selected classes. The property will be also annotated with @Version, and a getter and setter will be created. If the property is already exists, it won't be created, but the getters and setters will be generated. If there is already @MappedSuperclass annotation with @Version in the base class of the current class, @Version is not inserted into the current class.

After defining all the required settings click the Next button.
Hibernate: add JPA annotations refactoring

Figure 4.59. Hibernate: add JPA annotations view
Chapter 4. Eclipse Plugins

The view represents two windows: one with the source code and the second with refactored one. With the help of the buttons you can quickly navigate between the differences in the code. If you don't agree with some changes you can't undo them but you can remove the class from the list of classes that need refactoring.

![Changes to be performed](image)

Figure 4.60. List of classes that need refactoring

To apply the changes click the Finish button.

4.12. Enable debug logging in the plugins

It is possible to configure the Eclipse plugin to route all logging performed by the plugins and Hibernate code itself to the Error Log view in Eclipse.

The Error Log view is very useful tool to use when solving any problems which appear in the Hibernate Tools plugins. You can use it if there are troubles setting up a Hibernate Console Configuration.

This is done by editing the hibernate-log4j.properties file in the org.hibernate.eclipse/ directory or JAR. This file includes a default configuration that only logs WARN and above to a set of custom appenders (PluginFileAppender and PluginLogAppender). You can change these settings to be as verbose or silent as you please. See the Hibernate Documentation [http://www.hibernate.org/5.html] for more information on logging categories and Log4j documentation.

4.12.1. Relevant Resources Links

More information on how to configure logging via a Log4j property file can be found in the Log4j documentation [http://supportweb.cs.bham.ac.uk/docs/tutorials/docsystem/build/tutorials/log4j/log4j.html].
4.13. Hibernate support for Dali plugins in Eclipse WTP

Starting from version 3.0.0 Alpha1, JBoss Tools™ Hibernate plugins support Eclipse Dali integration, which makes it possible to use a Hibernate as a complete JPA development platform.

4.13.1. Creating JPA project with Hibernate support

When starting a new JPA project by selecting New → Other → JPA → JPA Project (or simply New → JPA Project in the JPA Perspective), the first wizard page is shown in the image below.
Figure 4.61. Starting JPA Project
You can select a target runtime and change the project configuration, or you can leave everything as it is.

On the JPA Facet page you should choose Hibernate as a target platform. Also select the proper database connection, if it is defined, or add a new one by clicking the Add connection link.

Clicking the Finish button will generate the project.
Chapter 4. Eclipse Plugins

Figure 4.62. Targeting at Hibernate Platform
4.13.2. Generating DDL and Entities

By enabling Hibernate platform specific features you can now generate DDL and Entities. To do that select the JPA Tools → Generate Tables from Entities/Generate Entities from Tables options in the context menu of your JPA project.

![Generate DDL/Entities](image)

**Figure 4.63. Generate DDL/Entities**

**Note:**

Remember to put the appropriate database driver to the classpath of your project.

The Generate Entities wizard will first ask you to choose the directory where all output will be written.
Figure 4.64. Generate Entities Wizard
To generate entities you can use:

- A Hibernate Console Configuration (proposed by default)

  To select this option make sure that the **Use Console Configuration** checkbox is selected and select a configuration from the **Console configurations** list box.

**Figure 4.65. Generate Entities Wizard**

- Or a DTP connection directly
To select this option uncheck the **Use Console Configuration** option and adjust the database settings.

The options you define in the **Generate Entities Wizard** can also be set with the **Generate DDL wizard**. The **Generate DDL** wizard also allows you automatically generate DDL for the database.

![Figure 4.66. Generate DDL Wizard](image)

In this way you can enable Hibernate runtime support in Eclipse JPA projects.

### 4.13.3. Hibernate Annotations Support

Hibernate Annotations are also supported in Dali Java Persistence Tools. The following annotations are integrated with the **JPA Details** view:

- **Id Generator annotations** - `@GenericGenerator` and `@GeneratedValue`
// default package

import java.util.HashSet;

/**
 * Customers generated by hbm2java
 */
@Entity
@Table(name = "CUSTOMERS", schema = "PUBLIC")
@GenericGenerator(strategy = "identity", name = "system-identity")
public class Customers implements java.io.Serializable {

    private int customernumber;
    private Employees employees;
    private String customername;
    private String contactlastname;
    private String contactfirstname;
    private String phone;
    private String addressline1;
    private String addressline2;
    private String city;
    private String state;
    private String postalcode;
    private String country;
    private Double creditlimit;
    private Set<Orders> orderses = new HashSet<Orders>(0);
    private Set<Payments> paymentses = new HashSet<Payments>(0);

    public Customers() {
    }

    public Customers(int customernumber, String customername, String contactlastname, String contactfirstname, String phone, String addressline1, String addressline2, String city, String state, String postalcode, String country, Double creditlimit) {
    }
}
Figure 4.68. @GeneratedValue support in Dali

- Property annotations - @DiscriminatorFormula, @Generated, @Index
// default package

import java.util.HashSet;

/**
 * Customers generated by hbm2java
 */
@Entity
@Table(name = "CUSTOMERS", schema = "PUBLIC")
@GenericGenerator(strategy = "identity", name = "system-identity")
public class Customers implements java.io.Serializable {
    @Id
    @GeneratedValue(generator="system-identity", strategy
    private int customernumber;
    private Employees employees;
    private String customername;
    private String contactlastname;
    private String contactfirstname;
    @Index(name = "phoneIndex")
    private String phone;
    private String addressline1;
    private String addressline2;
    private String city;
    private String state;
    private String postalcode;
    private String country;
    private Double creditlimit;
    private Set<Orders> orderses = new HashSet<Orders>(0);
    private Set<Payments> paymentses = new HashSet<Payments>(0);

    public Customers() {
}

Attribute 'phone' is mapped as default (basic).
• Mapping Queries annotations - @NamedQuery and @NamedNativeQuery

![Diagram of Add New Named Query Dialog with Hibernate Support]

Figure 4.70. Add New Named Query Dialog with Hibernate Support
import java.util.HashSet;

/**
 * Customers generated by hbm2java
 */
@Entity
@Table(name = "CUSTOMERS", schema = "PUBLIC")
@GenericGenerator(strategy = "identity", name = "system-identity")
@NamedQuery(name = "NamedQueryHibernate",
    query = "select n from Customers where n.creditlimit >= :mincreditlimit")
public class Customers implements java.io.Serializable {

@Id
@GeneratedValue(strategy = "system-identity")
private int customernumber;

@OneToOne()
private Employees employees;

private String customername;

private String contactlastname;

private String contactfirstname;

@Index(name = "phoneIndex")

@Entity

Queries

NamedQueryHibernate

Name: NamedQueryHibernate

Query: select n from Customers where n.creditlimit >= :mincreditlimit

Index
• **Association annotations in an embeddable object** (@OneToOne, @ManyToOne, @OneToMany or @ManyToMany)


```java
package org.example;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Embeddable;
import javax.persistence.ManyToOne;

@Embeddable
public class Country implements Serializable {
    private static final long serialVersionUID = 1L;
    private String iso2;

    @ManyToOne State state;
    @Column(name="countryName") private String name;

    public String getIso2() { return iso2; }
    public void setIso2(String iso2) { this.iso2 = iso2; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Figure 4.72. Hibernate Support for Embeddable Object
4.13.4. Relevant Resources Links

Ant Tools

This chapter demonstrates how to use Hibernate Tools via Ant tasks.

5.1. Introduction

The hibernate-tools.jar file, available from tools.hibernate.org [http://www.hibernate.org/subprojects/tools.html], contains the core code for Hibernate Tools™. It is used as the basis for both the Ant tasks described in this document and the Eclipse plugins available as part of JBoss Developer Studio. The hibernate-tools.jar file is located in your Eclipse plugins directory at /plugins/org.hibernate.eclipse.x.x.x/lib/tools/hibernate-tools.jar.

This jar is 100% independent from the Eclipse platform and can thus be used independently of Eclipse.

Note:

There may be incompatibilities with respect to the hibernate3.jar bundled with the tools and your own JAR. To avoid any confusion it is recommended that you use the hibernate3.jar and hibernate-annotations.jar files bundled with the tools when you want to use the Ant tasks. Do not worry about using the JAR’s from a later version of Hibernate (e.g. Hibernate 3.2) with a project using an earlier version of Hibernate (e.g. a Hibernate 3.1) since the generated output will work with previous Hibernate 3 versions.

5.2. The <hibernatetool> Ant Task

To use the Ant tasks you need to have the hibernatetool task defined. That is done in your build.xml file by inserting the following XML (assuming the JARs are located in the lib directory):

```xml
<path id="toolslib">
  <path location="lib/hibernate-tools.jar" />
  <path location="lib/hibernate3.jar" />
  <path location="lib/freemarker.jar" />
  <path location="${jdbc.driver.jar}" />
</path>
<taskdef name="hibernatetool"
  classname="org.hibernate.tool.ant.HibernateToolTask"
  classpathref="toolslib" />
```
Chapter 5. Ant Tools

This `<taskdef>` defines an Ant task called `hibernatetool` which now can be used anywhere in your Ant `build.xml` files. It is important to include all the Hibernate Tools™ dependencies as well as the JDBC driver.

Notice that to use the annotation based Configuration you must [get a release](http://www.hibernate.org/).

When using the `hibernatetool` task you have to specify one or more of the following:

```xml
<hibernatetool
    destdir="defaultDestinationDirectory"
    templatepath="defaultTemplatePath"
>
    <classpath ...>
    <property key="propertyName" value="value"/>
    <propertyset ...>
    (<configuration ...>[annotationconfiguration ...]|
     <jpaconfiguration ...>[jdbcconfiguration ...])
    (<hbm2java>,<hbm2cfgxml>,<hbmtemplate>,...)
</hibernatetool>
```

**Table 5.1. Hibernatetool attributes**

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Definition</th>
<th>Attribute use</th>
</tr>
</thead>
<tbody>
<tr>
<td>destdir</td>
<td>Destination directory for files generated with the exporters</td>
<td>Required</td>
</tr>
<tr>
<td>templatepath</td>
<td>A path used for looking up user-edited templates</td>
<td>Optional</td>
</tr>
<tr>
<td>classpath</td>
<td>A classpath to be used to resolve resources, such as mappings and user types</td>
<td>Optional, but very often required</td>
</tr>
<tr>
<td>property (and propertyset)</td>
<td>Used to set properties that control the exporters. Mostly relevant for providing custom properties to user defined templates</td>
<td>Optional</td>
</tr>
<tr>
<td>configuration (annotationconfiguration, jpaconfiguration, jdbcconfiguration)</td>
<td>One of four different ways of configuring the Hibernate Meta Model must be specified</td>
<td></td>
</tr>
<tr>
<td>hbm2java (hbm2cfgxml, hbmtemplate, etc.)</td>
<td>One or more of the exporters must be specified</td>
<td></td>
</tr>
</tbody>
</table>
5.2.1. Basic examples

The following example shows the most basic setup for generating POJOs via `<hbm2java>` from a normal `hibernate.cfg.xml` file. The output will be placed in the `${build.dir}/generated` directory.

```xml
<hibernatetool destdir="${build.dir}/generated">
  <classpath>
    <path location="${build.dir}/classes"/>
  </classpath>

  <configuration configurationfile="hibernate.cfg.xml"/>
  <hbm2java/>
</hibernatetool>
```

The following example is similar, but now we are performing multiple exports from the same configuration. We are exporting the schema via `<hbm2ddl>`, generating some DAO code via `<hbm2dao>`, and finally running some custom code generation via `<hbmtemplate>`. This is again from a normal `hibernate.cfg.xml` file, and the output is still placed in the `${build.dir}/generated` directory.

The example also shows how a classpath is specified, which is useful when you have custom user types or some mappings that are required to be looked up as a classpath resource.

```xml
<hibernatetool destdir="${build.dir}/generated">
  <classpath>
    <path location="${build.dir}/classes"/>
  </classpath>

  <configuration configurationfile="hibernate.cfg.xml"/>
  <hbm2ddl/>
  <hbm2dao/>
  <hbmtemplate
    filepattern="{package-name}/I{class-name}Constants.java"
    templatepath="${etc.dir}/customtemplates"
    template="myconstants.vm"
  />
</hibernatetool>
```

5.3. Hibernate Configurations

Hibernatetool™ supports four different Hibernate configurations: A standard Hibernate configuration (`<configuration>`), Annotation based configuration
(<annotationconfiguration>), JPA persistence based configuration (<jpaconfiguration>) and a JDBC based configuration (<jdbcconfiguration>) used when reverse engineering.

Each can be used to build a Hibernate Configuration object, from which a set of exporters can be run in order to generate various output formats.

**Note:**

Output can be anything, e.g. specific files, statements execution against a database, error reporting or anything else that can be done in Java code.

The following sections describe what the various configurations can do, as well as listing their individual settings.

### 5.3.1. Standard Hibernate Configuration (<configuration>)

A <configuration> tag is used to define a standard Hibernate configuration. A standard Hibernate configuration reads the mappings from a cfg.xml file and/or a fileset.

```xml
<configuration
    configurationfile="hibernate.cfg.xml"
    propertyfile="hibernate.properties"
    entityresolver="EntityResolver classname"
    namingstrategy="NamingStrategy classname"
>
    <fileset...>
</configuration>
```

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Definition</th>
<th>Attribute use</th>
</tr>
</thead>
<tbody>
<tr>
<td>configurationfile</td>
<td>The name of a Hibernate configuration file, e.g. hibernate.cfg.xml.</td>
<td>Optional</td>
</tr>
<tr>
<td>propertyfile</td>
<td>The name of a property file, e.g. hibernate.properties.</td>
<td>Optional</td>
</tr>
<tr>
<td>entity-resolver</td>
<td>Name of a class that implements org.xml.sax.EntityResolver. Used if the mapping files require custom entity resolver.</td>
<td>Optional</td>
</tr>
<tr>
<td>namingstrategy</td>
<td>Name of a class that implements org.hibernate.cfg.NamingStrategy. Used for setting up the naming strategy in Hibernate which controls the automatic naming of tables and columns. In JPA projects</td>
<td>Optional</td>
</tr>
</tbody>
</table>
5.3.1.1. Example

This example shows an example where no hibernate.cfg.xml file exists, and a hibernate.properties file and fileset is used instead.

Note:

Hibernate will still read any global hibernate.properties files available in the classpath, but the specified properties file here will override those values for any non-global property.

```xml
<hibernatetool destdir="${build.dir}/generated">
  <configuration propertyfile="{etc.dir}/hibernate.properties">
    <fileset dir="${src.dir}">
      <include name="**/*.hbm.xml"/>
      <exclude name="**/*Test.hbm.xml"/>
    </fileset>
  </configuration>
</hibernatetool>
```

5.3.2. Annotation based Configuration

( <annotationconfiguration> )

An <annotationconfiguration> tag is used when you want to read the metamodel from EJB3 or Hibernate Annotations based POJO's.
Important:

To use an <annotationconfiguration> tag remember to place the JAR files required to use Hibernate annotations in the classpath of the <taskdef>, i.e. hibernate-annotations.jar and hibernate-commons-annotations.jar.

The <annotationconfiguration> tag supports the same attributes as the <configuration> tag, with the exception being that the configurationfile attribute is now required as that is where an AnnotationConfiguration gets the list of classes and packages it should load.

Thus the minimal usage is:

```xml
<hibernatetool destdir="${build.dir}/generated">
  <annotationconfiguration
    configurationfile="hibernate.cfg.xml"/>

  <!-- list exporters here -->
</hibernatetool>
```

5.3.3. JPA based configuration (<jpaconfiguration>)

A <jpaconfiguration> tag is used when you want to read the metamodel from JPA or Hibernate Annotation where you want to use the auto-scan configuration as defined in the JPA specification (part of EJB3). In other words, when you do not have a hibernate.cfg.xml, but instead have a setup where you use a persistence.xml file packaged in a JPA compliant manner.

The <jpaconfiguration> tag will try and auto-configure itself based on the available classpath, e.g. look for the META-INF/persistence.xml file.

The persistenceunit attribute can be used to select a specific persistence unit. If no persistenceunit attribute is specified it will automatically search for one, and if a unique one is found, use it. However, having multiple persistence units will result in an error.

To use a <jpaconfiguration> tag you will need to specify some additional JARs from the Hibernate EntityManager in the <taskdef> section of the hibernatetool. The following demonstrates a full setup:

```xml
<path id="ejb3toolslib">
  <path refid="jpatoolslib"/> <!-- ref to previously defined toolslib -->
  <path location="lib/hibernate-annotations.jar"/>
  <path location="lib/hibernate-entitymanager.jar"/>
  <path location="lib/jboss-archive-browsing.jar"/>
</path>
```
5.3.4. JDBC Configuration for reverse engineering
(<jdbcconfiguration>)

A <jdbcconfiguration> tag is used to perform reverse engineering of a database from a JDBC connection.

This configuration works by reading the connection properties either from a hibernate.cfg.xml file or a hibernate.properties file with a fileset.

The <jdbcconfiguration> tag has the same attributes as a <configuration> tag, plus the following additional attributes:

```
<jdbcconfiguration>
  ...
  packagename="package.name"
  revengfile="hibernate.reveng.xml"
  reversestrategy="ReverseEngineeringStrategy classname"
  detectmanytomany="true|false"
  detectoptimisticlock="true|false"
...
```
Chapter 5. Ant Tools

Table 5.3. Jdbcconfiguration attributes

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Definition</th>
<th>Attribute use</th>
</tr>
</thead>
<tbody>
<tr>
<td>packagename</td>
<td>The default package name to use when mappings for classes are created</td>
<td>Optional</td>
</tr>
<tr>
<td>revengfile</td>
<td>The name of a property file, e.g. hibernate.properties</td>
<td>Optional</td>
</tr>
<tr>
<td>reversestrategy</td>
<td>Name of a class that implements org.hibernate.cfg.reveng.ReverseEngineeringStrategy. Used for setting up the strategy the tools will use to control the reverse engineering, e.g. naming of properties, which tables to include or exclude etc. Using a class instead of (or as addition to) a reveng.xml file gives you full programmatic control of the reverse engineering.</td>
<td>Optional</td>
</tr>
<tr>
<td>detectManytoMany</td>
<td>true, tables which are pure many-to-many link tables will be mapped as such. A pure many-to-many table is one which primary-key contains exactly two foreign-keys pointing to other entity tables and has no other columns.</td>
<td>Default: true</td>
</tr>
<tr>
<td>detectOptimisticLock</td>
<td>true, columns named VERSION or TIMESTAMP with appropriate types will be mapped with the appropriate optimistic locking corresponding to &lt;version&gt; or &lt;timestamp&gt;.</td>
<td>Default: true</td>
</tr>
</tbody>
</table>

5.3.4.1. Example

Here is an example using a <jdbcconfiguration> tag to generate Hibernate XML mappings via <hbm2hbmxml>. The connection settings used here are read from a hibernate.properties file, but they could also have been defined in a hibernate.cfg.xml file.

```xml
<hibernatetool>
    <jdbcconfiguration propertyfile="etc/hibernate.properties" />
    <hbm2hbmxml destdir="${build.dir}/src" />
</hibernatetool>
```

5.4. Exporters

Exporters do the actual job of converting the Hibernate metamodel into various artifacts, mainly code. The following section describes the current supported set of exporters in the Hibernate Tool™ distribution. It is also possible to implement user defined exporters, which is done through the <hbmtemplate> exporter.
5.4.1. Database schema exporter (<hbm2ddl>)

<hbm2ddl> lets you run schemaexport and schemaupdate which generates the appropriate SQL DDL and allow you to store the result in a file or export it directly to the database. Remember that if a custom naming strategy is needed it is defined in the configuration element.

```xml
<hbm2ddl
  export="true|false"
  update="true|false"
  drop="true|false"
  create="true|false"
  outputfilename="filename.ddl"
  delimiter=";"
  format="true|false"
  haltonerror="true|false"
>
```

Table 5.4. Hbm2ddl exporter attributes

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Definition</th>
<th>Attribute use</th>
</tr>
</thead>
<tbody>
<tr>
<td>export</td>
<td>Executes the generated statements against the database</td>
<td>Default: true</td>
</tr>
<tr>
<td>update</td>
<td>Try and create an update script representing the &quot;delta&quot; that is, between what is in the database and what the mappings specify. Ignores create and update attributes. (Do <em>not</em> use against production databases, as there are no guarantees that the proper delta can be generated, nor that the underlying database can actually execute the required operations).</td>
<td>Default: false</td>
</tr>
<tr>
<td>drop</td>
<td>Output will contain drop statements for the tables, indices and constraints</td>
<td>Default: false</td>
</tr>
<tr>
<td>create</td>
<td>Output will contain create statements for the tables, indices and constraints</td>
<td>Default: true</td>
</tr>
<tr>
<td>outputfilename</td>
<td>If specified the statements will be dumped to this file</td>
<td>Optional</td>
</tr>
<tr>
<td>delimiter</td>
<td>If specified the statements will be dumped to this file</td>
<td>Default: &quot;;&quot;</td>
</tr>
<tr>
<td>format</td>
<td>Apply basic formatting to the statements</td>
<td>Default: false</td>
</tr>
<tr>
<td>haltonerror</td>
<td>Halt build process if an error occurs</td>
<td>Default: false</td>
</tr>
</tbody>
</table>

5.4.1.1. Example

Below is a basic example of using <hbm2ddl>, which does not export to the database but simply dumps the SQL to a file named sql.ddl.
<hibernatetool destdir="${build.dir}/generated">
  <configuration configurationfile="hibernate.cfg.xml"/>
  <hbm2ddl export="false" outputfilename="sql.ddl"/>
</hibernatetool>

5.4.2. POJO java code exporter (<hbm2java>)

<hbm2java> is a Java code generator. Options for controlling whether JDK 5 syntax can be used and whether the POJO should be annotated with EJB3/Hibernate Annotations.

<hbm2java
  jdk5="true|false"
  ejb3="true|false"
>

Table 5.5. Hbm2java exporter attributes

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Definition</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdk</td>
<td>Code will contain JDK 5 constructs such as generics and static imports</td>
<td>False</td>
</tr>
<tr>
<td>ejb3</td>
<td>Code will contain EJB 3 features, e.g. using annotations from javax.persistence and org.hibernate.annotations</td>
<td>False</td>
</tr>
</tbody>
</table>

5.4.2.1. Example

Here is a basic example using <hbm2java> to generate POJO’s that utilize JDK5 constructs.

<hibernatetool destdir="${build.dir}/generated">
  <configuration configurationfile="hibernate.cfg.xml"/>
  <hbm2java jdk5="true"/>
</hibernatetool>

5.4.3. Hibernate Mapping files exporter (<hbm2hbmxml>)

<hbm2hbmxml> generates a set of .hbm files. It is intended to be used together with a <jdbcconfiguration> when performing reverse engineering, but can be used with any kind of configuration e.g. to convert from annotation based POJO’s to a hbm.xml file.
5.4.3.1. Example

Basic usage of `<hbm2hbmxml>`.

```xml
<hibernatetool destdir="${build.dir}/generated">
  <configuration configurationfile="hibernate.cfg.xml"/>
  <hbm2hbmxml/>
</hibernatetool>
```

`<hbm2hbmxml>` is normally used with a `<jdbcconfiguration>` like in the above example, but any other configuration can also be used to convert between the different ways of performing mappings. Here is an example of that, using an `<annotationconfiguration>`.

Note: Not all conversions are implemented (contributions welcome), so some hand editing might be necessary.

```xml
<hibernatetool destdir="${build.dir}/generated">
  <annotationconfiguration configurationfile="hibernate.cfg.xml"/>
  <hbm2hbmxml/>
</hibernatetool>
```

5.4.4. Hibernate Configuration file exporter (<`hbm2cfgxml`>)

`<hbm2cfgxml>` generates a `hibernate.cfg.xml` file. It is intended to be used together with a `<jdbcconfiguration>` when performing reverse engineering, but it can be used with any kind of configuration. The `<hbm2cfgxml>` will contain the properties that are used and adds mapping entries for each mapped class.

```xml
<hbm2cfgxml
```
Table 5.6. Hbm2cfgxml exporter attribute

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Definition</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ejb3</td>
<td>The generated cfg.xml will have &lt;mapping class=&quot;..&quot;/&gt; opposed to &lt;mapping resource=&quot;...&quot;/&gt; for each mapping.</td>
<td>False</td>
</tr>
</tbody>
</table>

5.4.5. Documentation exporter (<hbm2doc>)

<hbm2doc> generates HTML documentation similar to Javadoc for the database schema et.al.

<hibernatetool>
<configuration configurationfile="hibernate.cfg.xml"/>
<query>from java.lang.Object</query>
</hibernatetool>

5.4.6. Query exporter (<query>)

<query> is used to execute HQL query statements and optionally redirects the output to a file. It can be used for verifying the mappings and for basic data extraction.

<query
  destfile="filename">
  <hql>[a HQL query string]</hql>
</query>

Currently one session is opened and used for all queries, which are executed via the list() method. In the future more options might become available, like executing executeUpdate(), use named queries and etc.

5.4.6.1. Examples

The simplest usage of <query> will execute the query without dumping to a file. This can be used to verify that queries can be performed successfully.

<hibernatetool>
<configuration configurationfile="hibernate.cfg.xml"/>
<query>from java.lang.Object</query>
</hibernatetool>

Multiple queries can be executed by nested <hql> elements. In this example we also let the output be dumped to the queryresult.txt file.
Note:
Currently the dump is performed by calling the `toString()` function on each element.

```xml
<hibernatetool>
  <configuration configurationfile="hibernate.cfg.xml"/>
  <query destfile="queryresult.txt">
    <hql>select c.name from Customer c where c.age > 42</hql>
    <hql>from Cat</hql>
  </query>
</hibernatetool>
```

5.4.7. Generic Hibernate metamodel exporter (<hbmtemplate>)

Below is an example of a generic exporter that can be controlled by a user provided template or class.

```xml
<hbtemplate
  filepattern="{package-name}/{class-name}.ftl"
  template="somename.ftl"
  exporterclass="Exporter classname"
/>
```

Note:
Previous versions of the tools used Velocity™. We are now using Freemarker™, which provides much better exception and error handling.

5.4.7.1. Exporter via <hbtemplate>

The following is an example of reverse engineering via a `<jdbcconfiguration>` tag and the use of a custom Exporter via the `<hbtemplate>` tag.

```xml
<hibernatetool destdir="${destdir}"
  <jdbcconfiguration
    configurationfile="hibernate.cfg.xml"
    packagename="my.model"/>
  <!-- setup properties -->
  <property key="appname" value="Registration"/>
```
5.4.7.2. Relevant Resources Links

You can read more about Velocity [http://velocity.apache.org/] and Freemarker [http://freemarker.org/] to find out why using the latter is better or refer to Max Andersens discussion on the topic in “A story about FreeMarker and Velocity” [http://in.relation.to/2110.lace;jsessionid=3462F47B17556604C15DF1B96572E940].

5.5. Using properties to configure Exporters

Exporters can be controlled by user properties. These user properties are specified via a <property> or <propertyset> tag, and each exporter will have access to them directly in the templates and via Exporter.setProperties().

5.5.1. <property> and <propertyset>

The <property> tag allows you bind a string value to a key. The value will be available in the templates via the $<key> tag. The following example will assign the string value "true" to the variable $descriptors.

```
<property key="descriptors" value="true"/>
```

Usually using the <property> tag is enough when specifying the properties required by the exporters. Still, the Ant tools supports the notion of a <propertyset> which is used for grouping a set of properties. More about the functionality of <propertyset> is can be found in the Ant manual [http://ant.apache.org/manual/].

5.5.2. Getting access to user specific classes

It is possible for the templates to access user classes by specifying a "toolclass" in the properties.

```
<property key="hibernatetool.sometool.toolclass" value="x.y.z.NameOfToolClass"/>
```

Placing the above <property> tag in the <hibernatetool> tag or inside any exporter will automatically create an instance of x.y.z.NameOfToolClass which will be available in the
Getting access to user specific classes

templates as \$sometool. This is useful to delegate logic and code generation to Java code instead of placing such logic in the templates.

5.5.2.1. Example

Here is an example that uses the <hbmtemplate> tag together with the <property> tag, which will be available to the templates and exporter.

Note:
This example actually simulates what the <hbm2java> tag does.

```xml
<hibernatetool destdir="${build.dir}/generated">
  <configuration configurationfile="etc/hibernate.cfg.xml"/>
  <hbmtemplate templateprefix="pojo/"
    template="pojo/Pojo.ftl"
    filepattern="{package-name}/{class-name}.java">
    <property key="jdk5" value="true" />
    <property key="ejb3" value="true" />
  </hbmtemplate>
</hibernatetool>
```
Controlling reverse engineering

When using the `<jdbcconfiguration>` tag, the Ant task will read the database metadata and then reverse engineer the database schema into a normal Hibernate Configuration. It is from this object (e.g. `<hbm2java>`) that other artifacts, such as `.java` and `.hbm.xml`, can be generated.

To govern this process Hibernate™ uses a reverse engineering strategy. A reverse engineering strategy is mainly called to provide more Java like names for tables, column and foreign keys into classes, properties and associations. It is also used to provide mappings from SQL types to Hibernate™ types.

The strategy can be customized by the user. This can be done by providing a custom reverse engineering strategy should the default strategy does not include the required functionality, or simply define a small component of the strategy and delegate the rest to the default strategy.

Further in this chapter we will discuss how you can configure the process of reverse engineering, what the default reverse engineering strategy includes, as well as some custom concepts.

### 6.1. Default reverse engineering strategy

The default strategy uses a collection of rules for mapping JDBC artifact names to Java artifact names. It also provide basic type mappings from JDBC types to Hibernate™ types. It is the default strategy that uses the `packagename` attribute to convert a table name into a fully qualified class name.

### 6.2. hibernate.reveng.xml file

A `hibernate.reveng.xml` file can provide a finer degree of control of the reverse engineering process. In this file you can specify type mappings and table filtering. This file can be created by hand (it's just basic XML) or you can use the `Section 4.9, “Reveng.xml Editor”`, which provides a specialized editor.

**Note:**

Many databases have case-sensitive names, so if a table does not match, and you are sure it is not excluded by a `<table-filter>`, check that the case matches. Most databases stores table names in upper case.

Below you can see an example of a `reveng.xml` file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering
    SYSTEM "http://hibernate.sourceforge.net/hibernate-reverse-engineering-3.0.dtd" >
```
Chapter 6. Controlling revers...

```xml
<hibernate-reverse-engineering>

<type-mapping>
  <!-- jdbc-type is name for java.sql.Types -->
  <sql-type jdbc-type="VARCHAR" length='20' hibernate-type="SomeUserType" />
  <!-- length, scale and precision can be used to specify the mapping precisely -->
  <sql-type jdbc-type="VARCHAR" length='1' hibernate-type="yes_no" />
  <!-- the type-mappings are ordered. This mapping will be consulted last, 
      thus overridden by the previous one if precision=1 for the column -->
  <sql-type jdbc-type="NUMERIC" precision='1' hibernate-type="boolean" />
  <sql-type jdbc-type="NUMERIC" hibernate-type="long" />
</type-mapping>

<!-- BIN$ is recycle bin tables in Oracle -->
<table-filter match-name="BIN$.*" exclude="true" />

<!-- Exclude DoNotWantIt from all catalogs/schemas -->
<table-filter match-name="DoNotWantIt" exclude="true" />

<!-- exclude all tables from the schema SCHEMA in catalog BAD. -->
<table-filter match-catalog="BAD" match-schema="SCHEMA" match-name=".*" exclude="true" />

<!-- table allows you to override/define how reverse engineering 
    is done for a specific table -->
<table name="ORDERS">
  <primary-key>
    <!-- setting up a specific id generator for a table -->
    <generator class="sequence">
      <param name="table">seq_table</param>
    </generator>
  </primary-key>
  <column name="NAME" property="orderName" type="string" />
  <!-- control many-to-one and set names for a specific named foreign key constraint -->
  <foreign-key constraint-name="ORDER_CUST">
    <many-to-one property="customer"/>
    <set property="orders"/>
  </foreign-key>
  <!-- can also control a pure (shared pk) one-to-one -->
  <foreign-key constraint-name="ADDRESS_PERSON">
    <one-to-one exclude="false"/>
  </foreign-key>
</table>
</hibernate-reverse-engineering>
```
6.2.1. Schema Selection (<schema-selection>)

The <schema-selection> tag is used to determine which schemas the reverse engineering will try and process.

By default the reverse engineering will read all schemas and then use the <table-filter> tag to decide which tables are reverse engineered and which are not. This makes it easy to get started but can be inefficient on databases with many schemas.

With the <schema-selection> tag it is thus possible to limit which schemas are processed, which in turn can significantly speed-up the reverse engineering. The <table-filter> tag is still used to then decide which tables will be included and excluded.

Note:
If no <schema-selection> tag is specified, the reverse engineering works as if all schemas should be processed. This is equal to: <schema-selection/>, which in turn is equal to: <schema-selection match-catalog=".*" match-schema=".*" match-table=".*"/>

6.2.1.1. Examples

The following will process all tables from "MY_SCHEMA".

```
<schema-selection match-schema="MY_SCHEMA"/>
```

It is possible to have multiple schema-selection’s to support multi-schema reading, or to limit the processing to very specific tables. The following example processes all tables in "MY_SCHEMA", a specific "CITY" table plus all tables that start with "CODES_" in "COMMON_SCHEMA".

```
<schema-selection match-schema="MY_SCHEMA"/>
<schema-selection match-schema="COMMON_SCHEMA" match-table="CITY"/>
<schema-selection match-schema="COMMON_SCHEMA" match-table="CODES_.*"/>
```

6.2.2. Type mappings (<type-mapping>)

The <type-mapping> section specifies how the JDBC types found in the database should be mapped to Hibernate types. e.g. java.sql.Types.VARCHAR with a length of 1 should be mapped to the Hibernate type yes_no, or java.sql.Types.NUMERIC should generally just be converted to the Hibernate type long.

```
<type-mapping>
```
Chapter 6. Controlling revers...

The number of attributes specified and the sequence of the sql-type tags are important. This is because Hibernate™ will search for the most specific first, and if no specific match is found it will seek from top to bottom when trying to resolve a type mapping.

6.2.2.1. Example

The following is an example of a type-mapping which shows the flexibility and importance of the ordering of the type mappings.

```xml
<type-mapping>
  <sql-type jdbc-type="NUMERIC" precision="15" hibernate-type="big_decimal"/>
  <sql-type jdbc-type="NUMERIC" not-null="true" hibernate-type="long" />
  <sql-type jdbc-type="NUMERIC" not-null="false" hibernate-type="java.lang.Long" />
  <sql-type jdbc-type="VARCHAR" length="1" not-null="true" hibernate-type="java.lang.Character"/>
  <sql-type jdbc-type="VARCHAR" hibernate-type="your.package.TrimStringUserType"/>
  <sql-type jdbc-type="VARCHAR" length="1" hibernate-type="char"/>
  <sql-type jdbc-type="VARCHAR" hibernate-type="string"/>
</type-mapping>
```

The following table shows how this affects an example table named CUSTOMER:

### Table 6.1. sql-type examples

<table>
<thead>
<tr>
<th>Column</th>
<th>jdbc-type</th>
<th>length</th>
<th>precision</th>
<th>not-null</th>
<th>Resulting hibernate-type</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>INTEGER</td>
<td>10</td>
<td></td>
<td>true</td>
<td>int</td>
<td>Nothing is defined for INTEGER. Falling back to default behavior.</td>
</tr>
<tr>
<td>NAME</td>
<td>VARCHAR</td>
<td>30</td>
<td></td>
<td>false</td>
<td>your.package.TrimStringUserType</td>
<td>Type type-definition matches length=30 and</td>
</tr>
</tbody>
</table>
6.2.3. Table filters (<table-filter>)

The <table-filter> tag lets you specify matching rules for performing general filtering and setup of tables, e.g. let you include or exclude specific tables based on the schema or even a specific prefix.

<table>
<thead>
<tr>
<th>Column</th>
<th>jdbc-type</th>
<th>length</th>
<th>precision</th>
<th>not-null</th>
<th>Resulting hibernate-type</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL</td>
<td>VARCHAR</td>
<td>1</td>
<td></td>
<td>false</td>
<td>char</td>
<td>not-null=false, but type-mapping matches the 2 mappings which only specifies VARCHAR. The type-mapping that comes first is chosen. Even though there is a generic match for VARCHAR, the more specific type-mapping for VARCHAR with not-null=&quot;false&quot; is chosen. The first VARCHAR sql-type matches in length but has no value for not-null and thus is not considered.</td>
</tr>
<tr>
<td>CODE</td>
<td>VARCHAR</td>
<td>1</td>
<td></td>
<td>true</td>
<td>java.lang.Character</td>
<td>The most specific VARCHAR with not-null=&quot;true&quot; is selected</td>
</tr>
<tr>
<td>SALARY</td>
<td>NUMERIC</td>
<td>15</td>
<td></td>
<td>false</td>
<td>big_decimal</td>
<td>There is a precise match for NUMERIC with precision 15</td>
</tr>
<tr>
<td>AGE</td>
<td>NUMERIC</td>
<td>3</td>
<td></td>
<td>false</td>
<td>java.lang.Long</td>
<td>type-mapping for NUMERIC with not-null=&quot;false&quot;</td>
</tr>
</tbody>
</table>
6.2.4. Specific table configuration (<table>)

The `<table>` tag allows you to explicitly define how a table should be reverse engineered. It allows control over the naming of a class for the table, provides a way to specify which identifier generator should be used for the primary key and more.

```xml
<table
  catalog="catalog_name"
  schema="schema_name"
  name="table_name"
  class="ClassName"
>
  <primary-key.../>
  <column.../>
  <foreign-key.../>
</table>
```

### Table 6.3. Table attributes

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Definition</th>
<th>Attribute use</th>
</tr>
</thead>
<tbody>
<tr>
<td>catalog</td>
<td>Catalog name for a table. It has to be specified if you are reverse engineering multiple catalogs or if it is not equal to hibernate.default_catalog.</td>
<td>Optional</td>
</tr>
<tr>
<td>schema</td>
<td>Schema name for a table. It has to be specified if you are reverse engineering multiple schemas or if it is not equal to hibernate.default_schema.</td>
<td>Optional</td>
</tr>
</tbody>
</table>
Specific table configuration (<table>)

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Definition</th>
<th>Attribute use</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name for a table.</td>
<td>Required</td>
</tr>
<tr>
<td>class</td>
<td>The class name for a table. Default name is a CamelCase version of the table name.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

### 6.2.4.1. <primary-key>

A `<primary-key>` tag allows you to define a primary-key for tables that do not have one defined in the database, and more importantly it allows you to define which identifier strategy should be used (even for preexisting primary-key's).

```xml
<primary-key
  <generator class="generatorname">
    <param name="param_name">parameter value</param>
  </generator>
  <key-column...>
</primary-key>
```

#### Table 6.4. Primary-key attributes

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Definition</th>
<th>Attribute use</th>
</tr>
</thead>
<tbody>
<tr>
<td>generator/class</td>
<td>Defines which identifier generator should be used. The class name is any hibernate short hand name or fully qualified class name for an identifier strategy.</td>
<td>Optional</td>
</tr>
<tr>
<td>generator/param</td>
<td>Allows to specify which parameter with a name and value should be passed to the identifier generator.</td>
<td>Optional</td>
</tr>
<tr>
<td>key-column</td>
<td>Specifies which column(s) the primary-key consists of. A key-column is same as column, but does not have the exclude property.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

### 6.2.4.2. <column>

With a `<column>` tag it is possible to explicitly name the resulting property for a column, to redefine what JDBC and/or Hibernate type a column should be processed as, and to completely exclude a column from processing.

```xml
<column
  name="column_name"
  jdbc-type="java.sql.Types type"
  type="hibernate_type"
  property="propertyName"
  exclude="true|false"
/>
```
6.2.4.3. <foreign-key>

The `<foreign-key>` tag has two purposes. The first is to define foreign-keys in databases that does not support them or do not have them defined in their schema. The second is to define the name of the resulting properties (many-to-one, one-to-one and one-to-many's).

```xml
<foreign-key
    constraint-name="foreignKeyName"
    foreign-catalog="catalogName"
    foreign-schema="schemaName"
    foreign-table="tableName"
>
    <column-ref local-column="columnName" foreign-column="foreignColumnName"/>
    <many-to-one
        property="aPropertyName"
        exclude="true|false"/>
    <set
        property="aCollectionName"
        exclude="true|false"
    
    <one-to-one
        property="aPropertyName"
        exclude="true|false"/>
    <inverse-one-to-one
        property="aPropertyName"
        exclude="true|false"/>
</foreign-key>
```

Table 6.6. Foreign-key attributes

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Definition</th>
<th>Attribute use</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint-name</td>
<td>Name of the foreign key constraint. Important when naming many-to-one, one-to-one and set. It is the constraint-name</td>
<td>Required</td>
</tr>
<tr>
<td>name</td>
<td>Column name</td>
<td>Required</td>
</tr>
<tr>
<td>jdbc-type</td>
<td>Which jdbc-type this column should be processed as. A value from java.sql.Types, either numerical (e.g. 93) or the constant name (e.g. TIMESTAMP).</td>
<td>Optional</td>
</tr>
<tr>
<td>type</td>
<td>Which hibernate-type to use for this specific column</td>
<td>Optional</td>
</tr>
<tr>
<td>property</td>
<td>What property name will be generated for this column</td>
<td>Optional</td>
</tr>
<tr>
<td>exclude</td>
<td>Set to true if this column should be ignored</td>
<td>default: false</td>
</tr>
</tbody>
</table>
### Attribute name | Definition | Attribute use
--- | --- | ---
foreign-catalog | Name of the foreign table’s catalog. (Only relevant if you want to explicitly define a foreign key). | Optional
foreign-schema | Name of the foreign table’s schema. (Only relevant if you want to explicitly define a foreign key). | Optional
foreign-table | Name of the foreign table. (Only relevant if you want to explicitly define a foreign key). | Optional
column-ref | Defines the foreign-key constraint between a local-column and foreign-column name. (Only relevant if you want to explicitly define a foreign key). | Optional
many-to-one | Defines that a many-to-one should be created and the property attribute specifies the name of the resulting property. Exclude can be used to explicitly define that it should be created or not. | Optional
set | Defines that a set should be created based on this foreign-key and the property attribute specifies the name of the resulting (set) property. Exclude can be used to explicitly define that it should be created or not. | Optional
one-to-one | Defines that a one-to-one should be created and the property attribute specifies the name of the resulting property. Exclude can be used to explicitly define that it should be created or not. | Optional
inverse-one-to-one | Defines that an inverse one-to-one should be created based on this foreign-key and the property attribute specifies the name of the resulting property. Exclude can be used to explicitly define that it should be created or not. | Optional

### 6.3. Custom strategy

It is possible to implement a user strategy. Such a strategy must implement `org.hibernate.cfg.reveng.ReverseEngineeringStrategy`. It is recommended that you use the `DelegatingReverseEngineeringStrategy` and provide a public constructor which takes another `ReverseEngineeringStrategy` as an argument. This will allow you to only implement the relevant methods and provide a fall back strategy. An example is shown below of a custom delegating strategy that converts all column names ending with "PK" into a property named "id".

```java
public class ExampleStrategy extends DelegatingReverseEngineeringStrategy {
    public ExampleStrategy(ReverseEngineeringStrategy delegate) {
        super(delegate);
    }
}
```
public String columnToPropertyName(TableIdentifier table, String column) {
    if(column.endsWith("PK")) {
        return "id";
    } else {
        return super.columnToPropertyName(table, column);
    }
}

6.4. Custom Database Metadata

By default the reverse engineering is performed using the JDBC database metadata API. This is done via the class org.hibernate.cfg.reveng.dialect.JDBCMetaDataDialect, which is an implementation of org.hibernate.cfg.reveng.dialect.MetaDataDialect.

The default implementation can be replaced with an alternative implementation by setting the hiberanatetool.metadatadialect property to a fully qualified class name for a class that implements JDBCMetaDataDialect.

This can be used to provide database specific optimized metadata reading. If you create an optimized metadata reader for your database it will be a very welcome contribution.
Chapter 7.

Controlling POJO code generation

When using the `<hbm2java>` tag or the Eclipse plugin to generate POJO Java code you have the ability to control certain aspects of the code generation process. This is primarily done with the `<meta>` tag in the mapping files. The following section describes the possible `<meta>` tags and their use.

7.1. The `<meta>` attribute

The `<meta>` tag is a simple way of annotating the hbm.xml file with information, so tools have a natural place to store and read information that is not directly related to the Hibernate core.

As an example, you can use the `<meta>` tag to tell the `<hbm2java>` tag to only generate "protected" setters, have classes always implement a certain set of interfaces, have them extend a certain base class and more.

The following example shows how to use various `<meta>` attributes and the resulting Java code.

```xml
<class name="Person">
    <meta attribute="class-description">Javadoc for the Person class
    @author Frodo</meta>
    <meta attribute="implements">IAuditable</meta>
    <id name="id" type="long">
        <meta attribute="scope-set">protected</meta>
        <generator class="increment"/>
    </id>
    <property name="name" type="string">
        <meta attribute="field-description">The name of the person</meta>
    </property>
</class>
```

The above hbm.xml file will produce something like the following (the code has been abbreviated for clarity). Notice the Javadoc comment and the protected set methods:

```java
// default package
import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;

/**
 * Javadoc for the Person class
 */
```
Chapter 7. Controlling POJO c...

```java
@Author Frodo

public class Person implements Serializable, IAuditable {

    public Long id;
    public String name;

    public Person(java.lang.String name) {
        this.name = name;
    }

    public Person() {
    }

    public java.lang.Long getId() {
        return this.id;
    }

    protected void setId(java.lang.Long id) {
        this.id = id;
    }

    /**
     * The name of the person
     */
    public java.lang.String getName() {
        return this.name;
    }

    public void setName(java.lang.String name) {
        this.name = name;
    }
}
```

Table 7.1. Supported meta tags

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class-description</td>
<td>inserted into the Javadoc for classes</td>
</tr>
<tr>
<td>field-description</td>
<td>inserted into the Javadoc for fields and properties</td>
</tr>
<tr>
<td>interface</td>
<td>If true, an interface is generated instead of a class.</td>
</tr>
<tr>
<td>implements</td>
<td>interface the class should implement</td>
</tr>
<tr>
<td>extends</td>
<td>class that the current class should extend (ignored for subclasses)</td>
</tr>
<tr>
<td>generated-class</td>
<td>overrule the name of the actual class generated</td>
</tr>
</tbody>
</table>
## Recommendations

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope-class</td>
<td>scope for class</td>
</tr>
<tr>
<td>scope-set</td>
<td>scope for setter method</td>
</tr>
<tr>
<td>scope-get</td>
<td>scope for getter method</td>
</tr>
<tr>
<td>scope-field</td>
<td>scope for actual field</td>
</tr>
<tr>
<td>default-value</td>
<td>Default initialization value for a field.</td>
</tr>
<tr>
<td>use-in-tostring</td>
<td>Include this property in the <code>toString()</code> method.</td>
</tr>
<tr>
<td>use-in-equals</td>
<td>Include this property in the <code>equals()</code> and <code>hashCode()</code> methods. If no use-in-equals is specified, no equals or hashcode method will be generated.</td>
</tr>
<tr>
<td>gen-property</td>
<td>Property will not be generated if false (use with care).</td>
</tr>
<tr>
<td>property-type</td>
<td>Overrides the default type of property. Use this with any tag's to specify the concrete type instead of just <code>Object</code>.</td>
</tr>
<tr>
<td>class-code</td>
<td>Extra code that will inserted at the end of the class</td>
</tr>
<tr>
<td>extra-import</td>
<td>Extra import that will inserted at the end of all other imports</td>
</tr>
</tbody>
</table>

Attributes declared via the `<meta>` tag "inherited" inside an `hbm.xml` file by default.

What does that mean? As an example if you want to have all your classes implement `IAuditable` then you just add `<meta attribute="implements">IAuditable</meta>` in the top of the `hbm.xml` file, just after `<hibernate-mapping>`. Now all classes defined in that `hbm.xml` file will implement `IAuditable`.

**Note:**

This applies to all `<meta>`-tags. Thus it can also be used to specify that all fields should be declared `protected`, instead of the default `private`. This is done by adding `<meta attribute="scope-field">protected</meta>` just under the `<class>` tag, and all fields of that class will be protected.

To avoid having a `<meta>` tag inherited then you can specify `inherit = "false"` for the attribute. For example `<meta attribute = "scope-class" inherit = "false">public abstract</meta>` will restrict the "class-scope" to the current class, not the subclasses.

### 7.1.1. Recommendations

The following are some good practices to employ when using `<meta>` attributes.
7.1.1.1. Dangers of a class level use-in-string and use-in-equals meta attributes when using bi-directional associations

In the following example we have two entities with a bi-directional association between them and define the use-in-string and use-in-equals meta attributes at the class scope level the meta attributes:

```xml
<hibernate-mapping>
  <class name="Person">
    <meta attribute="use-in-tostring">true</meta>
    <meta attribute="use-in-equals">true</meta>
    ...
  </class>
</hibernate-mapping>
```

Here is the Event.hbm file:

```xml
<hibernate-mapping>
  <class name="events.Event" table="EVENTS">
    <meta attribute="use-in-tostring">true</meta>
    <meta attribute="use-in-equals">true</meta>
    <id name="id" column="EVENT_ID">
      <generator class="native"/>
    </id>
    <property name="date" type="timestamp" column="EVENT_DATE"/>
    <property name="title"/>
    <set name="participants" table="PERSON_EVENT" inverse="true">
      <key column="EVENT_ID"/>
      <many-to-many column="PERSON_ID" class="events.Person"/>
    </set>
  </class>
</hibernate-mapping>
```

In this situation the `<hbm2java>` tag will assume you want to include all properties and collections in the `toString()` and `equals()` methods. This can result in infinite recursive calls.

To remedy this you have to decide which side of the association will include the other part (if at all) in the `toString()` and `equals()` methods. Therefore it is not a good practice to define these meta attributes at the class scope, unless you are defining a class without bi-directional associations.

Instead it is recommended that the meta attributes are defined at the property level, like so:

```xml
<hibernate-mapping>
  <class name="events.Event" table="EVENTS">
```

140
and for `Person`:

```xml
<hibernate-mapping>
  <class name="Person">
    <meta attribute="class-description">Javadoc for the Person class</meta>
    <author>Frodo</author>
    <meta attribute="implements">IAuditable</meta>
    <id name="id" type="long">
      <meta attribute="scope-set">protected</meta>
      <meta attribute="use-in-tostring">true</meta>
      <generator class="increment"/>
    </id>
    <property name="name" type="string">
      <meta attribute="field-description">The name of the person</meta>
      <meta attribute="use-in-tostring">true</meta>
    </property>
  </class>
</hibernate-mapping>
```

### 7.1.1.2. Be aware of putting at class scope level `<meta>` attribute `use-in-equals`

Only attributes with business meaning (e.g. the name, social security number, etc, but no generated id's) should be referenced when calculating the return value for the `equal()` and `hashCode()` methods.
This is important because Java's hashbased collections, such as java.util.Set, rely on equals() and hashcode() being correct and not changing for objects in the set; this can be a problem if the id gets assigned for an object after you inserted it into a set.

Therefore automatically configuration of the generation of equals() and hashCode() methods specifying the <meta> attribute use-in-equals at class scope level could be a dangerous decision that could produce unexpected side-effects.


### 7.1.2. Advanced <meta> attribute examples

This section shows an example for using meta attributes (including user specific attributes) together with the code generation features in Hibernate Tools™.

The example shown below automatically inserts some pre and post conditions into the getter and setter methods of the generated POJO.

#### 7.1.2.1. Generate pre/post-conditions for methods

With <meta attribute="class-code"> you can add additional methods on a given class. However, such <meta> attributes can not be used at a property scope level and Hibernate Tools does not provide such <meta> attributes.

A possible solution for this is to modify the Freemarker templates responsible for generating the POJOs. If you look inside the hibernate-tools.jar archive, you can find the template pojo/PojoPropertyAccessor.ftl.

As its name indicates, this file is used to generate property accessors for POJOs.

Extract the PojoPropertyAccessor.ftl file into a local folder e.g. ${hbm.template.path}, respecting the whole path, for example: ${hbm.template.path}/pojo/PojoPropertyAccessor.ftl.

The contents of the file will be something like this:

```plaintext
<#foreach property in pojo.getAllPropertiesIterator()>
  ${pojo.getPropertyGetModifiers(property)}
  ${pojo.getJavaTypeName(property, jdk5)}
  ${pojo.getGetterSignature(property)}() {
    return this.$property.name;
  }

  ${pojo.getPropertySetModifiers(property)} set
  ${pojo.getPropertyName(property)} (${pojo.getJavaTypeName(property, jdk5)} ${property.name}) {
    this.$property.name = ${property.name};
  }
</#foreach>
```
We can add pre and post conditions on our set method generation just by adding a little Freemarker syntax to the above source code:

```freemarker
<#foreach property in pojo.getAllPropertiesIterator()>
    ${pojo.getPropertyGetModifiers(property)}
    ${pojo.getJavaTypeName(property, jdk5)}
    ${pojo.getGetterSignature(property)}()
    {
        return this.${property.name};
    }

    ${pojo.getPropertySetModifiers(property)} void set
    ${pojo.getPropertyName(property)}
    (${pojo.getJavaTypeName(property, jdk5)} ${property.name})
    {
        <#if pojo.hasMetaAttribute(property, "pre-cond")>
            ${c2j.getMetaAsString(property, "pre-cond","\n")}
        </#if>
        this.${property.name} = ${property.name};
        <#if pojo.hasMetaAttribute(property, "post-cond")>
            ${c2j.getMetaAsString(property, "post-cond","\n")}
        </#if>
    }
</#foreach>
```

Now if in any .hbm.xml file we define the <meta> attributes: pre-cond or post-cond, and their contents will be generated into the body of the relevant set method.

As an example let us add a pre-condition for the name property which will prevent the Person class from having an empty name. To achieve this we have to modify the Person.hbm.xml file like so:

```xml
<hibernate-mapping>
    <class name="Person">
        <id name="id" type="long">
            <generator class="increment"/>
        </id>
        <property name="firstName" type="string">
            <meta attribute="pre-cond">
                if ((firstName != null) && (firstName.length() == 0)) { 
                    throw new IllegalArgumentException("firstName can not be an empty String");
                }
            </meta>
        </property>
    </class>
</hibernate-mapping>
```
Finally we have to generate the `Person.java` class. For this we can use either Eclipse or Ant, as long as you remember to set or fill in the `templatepath` setting. For Ant we configure the `<hibernatetool>` task via the `templatepath` attribute as in:

```xml
<target name="hbm2java">
  <taskdef name="hibernatetool"
    classname="org.hibernate.tool.ant.HibernateToolTask"
    classpathref="lib.classpath"/>
  <hibernatetool destdir="${hbm2java.dest.dir}"
    templatepath="${hbm.template.path}">
    <classpath>
      <path refid="pojo.classpath"/>
    </classpath>
    <configuration>
      <fileset dir="${hbm2java.src.dir}">
        <include name="**/*.hbm.xml"/>
      </fileset>
    </configuration>
  </hibernatetool>
</target>
```

Invoking the target `<hbm2java>` will generate file `Person.java` in `${hbm2java.dest.dir}`:

```java
// default package
import java.io.Serializable;
public class Person implements Serializable {

    public Long id;

    public String name;
```
In conclusion, this document is intended to introduce you to Hibernate plugin specific features related to tools both for the Eclipse and Ant tasks.

In *Chapter 4, Eclipse Plugins* you've learned about a set of wizards for creating Mapping files, Configuration files, Console Configurations, become familiar with Mapping and Configuration files editors, tooling for organizing and controlling Reverse Engineering, Hibernate Console and Mapping diagrams.

The rest chapters have explored the use of the Hibernate Tools™ via Ant tasks.

Please visit [JBoss Tools Users Forum](http://www.jboss.com/index.html?module=bb&op=viewforum&f=201) to leave questions or/and suggestions on the topic. Your feedback is always appreciated.