Hibernate Tools Reference Guide

Version: 3.2.0.Beta

Introduction

Hibernate Tools is a toolset for *Hibernate 3* [http://www.hibernate.org/6.html] and *related projects* [http://www.hibernate.org/27.html]. The tools provide Ant tasks and Eclipse plugins for performing reverse engineering, code generation, visualization and interaction with Hibernate.

1.1. Key Features

First, we propose to look through the list of key features that you can benefit from if you start using Hibernate Tools.

Table 1.1. Key Functionality for Hibernate Tools

Feature	Benefit	Chapter
Code Generation through Ant Task	Allows to execute mapping or Java code generation from reverse engineering, schema generation and generation of other artifacts during the build process.	*ant task*
Wizards for creation purposes and code generation	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 revenge.xml as well. Code Generation wizard helps to generate a series of various artifacts, there is even support for completely reverse engineer an existing database schema.	*hibernate mapping file* *hibernate configuration file* *code generation*
Mapping and Configuration files Editors	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.	*mapping and configuration files editors*
Tools for organizing and controlling Reverse Engineering	Code Generation wizard provides powerful functionality for generating a series of various artifacts like domain model classes, mapping files, annotated EJB3 entity beans, etc. and reveng.xml file editor allows to control this processes.	*code generation* *reveng.xml editor*
Hibernate Console	It is a new perspective in Eclipse which provides an overview of your Hibernate Console configurations, 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.	*hibernate console*
HQL Editor and Hibernate	The editors are intended for writing, editing and executing HQL queries and criterias. They also support the functionality for generating simple queries.	*hql and hibernate*

Feature	Benefit	Chapter
Criteria Editor		*criteria editors*
Functional Mapping Diagram	Makes possible to visualize structure of entities and relationships between them.	*mapping diagram*
Eclipse JDT integration	Hibernate Tools integrates into the Java code completion and build support of Java in Eclipse. This gives you code completion of HQL inside Java code. Additionally, Hibernate Tools will add problem markers if your queries are not valid against the console configuration associated with the project.	

1.2. Other relevant resources on the topic

All JBoss Developer Studio/JBoss Tools release documentation you can find at*http://docs.jboss.org/tools* [http://docs.jboss.org/tools/] in the corresponding release directory.

There is some extra information about Hidernate on *JBoss Wiki page.* [http://www.jboss.org/community/wiki/JBossHibernate3]

The latest documentation builds are available at *http://download.jboss.org/jbosstools/nightly-docs* [http://download.jboss.org/jbosstools/nightly-docs/].

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 JBDS/JBoss Tools, or a default Eclipse + WTP installation. The following sections describe the install steps in these environments.

Note:

The Hibernate Tools 3.3.0 (the current release version) requires Eclipse Galileo 3.5.

2.1. JBoss Tools

JBoss Tools 3.1.0 (the latest release) includes Hibernate Tools 3.3.0 and thus nothing is required besides downloading and installing JBoss Tools. If you need to update to a newer version of the Hibernate Tools just follow the instructions in the Eclipse IDE section.

2.2. Eclipse IDE

To install the Hibernate Tools into any Eclipse 3.5 based IDE you can either use *JBoss Tools Update Site* [http://download.jboss.org/jbosstools/updates/stable/] or install it manually.

If you want to install the Hibernate Tools distribution manually you need to:

- Download from www.eclipse.org:
  - birt-report-framework-2_5_0.zip
  - birt-wtp-integration-sdk-2_3_2.zip
  - dtp-sdk_1.7.0.zip
  - eclipse-SDK-3.5-win32.zip
  - emf-runtime-2.5.0.zip
  - GEF-SDK-3.5.0.zip
  - org.eclipse.swtbot.eclipse.test-2.0.0.371-dev-e35.zip
  - org.eclipse.swtbot.eclipse-2.0.0.340-dev.zip
  - site-1.6.2.zip

- tptp.sdk-TPTP-4.6.0.zip
- wtp-sdk-R-3.1-20090616035105.zip
- xsd-runtime-2.5.0.zip

- Then you should unpack these files into Eclipse install folder.
- From *hibernate.org-Download Overview* [https://www.hibernate.org/6.html] download *Hibernate Tools* [http://downloads.sourceforge.net/project/jboss/JBossTools/JBossTools3.1.0.CR1/HibernateTools-3.3.0.v200912250601M-H198-CR1.zip] .
- Unpack Hibernate Tools in eclipse/dropins folder

Note:

If you need more detailed instructions on plugins 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 using the update manager.

2.2.1. Usage of 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 not always have had proper versioning of their plugins there might exist WTP plugins 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 WTP provided features to try and 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* and associated libraries. The libraries are included in the distribution from the Hibernate website and the Eclipse updatesite. 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 these via ant tasks are described in the *Ant Tools* chapter.

Code generation architecture

The code generation mechanism in the Hibernate Tools consists of a few core concepts. This section explains their overall structure which are 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 `org.hibernate.mapping` 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 build in many ways. The following list the currently supported ones in Hibernate Tools.

- A Core configuration uses Hibernate Core and supports reading *hbm.xml* files, requires a *hibernate.cfg.xml* . Named core in Eclipse and `<configuration>` in ant.
- An Annotation configuration uses Hibernate Annotations and supports *hbm.xml* and annotated classes, requires a *hibernate.cfg.xml* . Named annotations in Eclipse and `<annotationconfiguration>` in ant.
- A JPA configuration uses a Hibernate EntityManager and supports *hbm.xml* and annotated classes requires that the project has a *META-INF/persistence.xml* in its classpath. Named 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 named `<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:

No matter which Hibernate Configuration type you are using Hibernate Tools supports them.

The following drawing illustrates the core concepts:

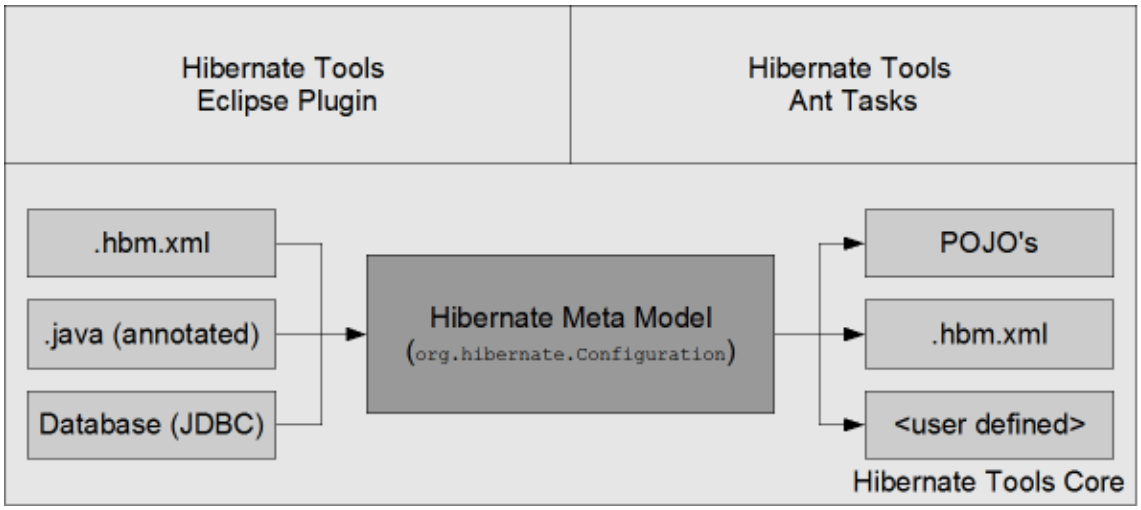

Figure 3.1. Hibernate Core Concepts

The code generation is done based on the Configuration model no matter which type of configuration have 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 done in so called 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 tools provides a default set of Exporter's which can be used in both Ant and the Eclipse UI. Documentation for these Exporters is in the *Ant Tools* and *Eclipse Plugins* chapters.

Users can provide their own customer Exporter's, either by custom classes implementing the Exporter interface or simply be providing custom templates. This is documented at *Section 5.4.7, "Generic Hibernate metamodel exporter (<hbmtemplate>)"*

Eclipse Plugins

This chapter will introduce you to the functionality that Hibernate Tools provide within Eclipse. That is a set of wizards and editors for simplifying the work with Hibernate.

4.1. Introduction

Hibernate Eclipse Tools include wizards for creating Hibernate mapping files, configuration files (.cfg.xml), revenge.xml as well as wizards for adjusting Console Configuration and Code Generation. Special structured and XML editors, editors for executing HQL and Criteria queries are also provided in Hibernate Console. Refer to *Key Features* section to find all benefits that you can take advantage of while using the tools within Eclipse.

Note:

Please note that these tools do not try to hide any functionality of Hibernate. The tools make working with Hibernate easier, but you are still encouraged/required to read the *Hibernate Documentation* [http://www.hibernate.org/5.html] to fully utilize Hibernate Tools and especially Hibernate it self.

4.1.1. Download base project

You can download example projects which are used for this chapter.

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

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

Note:

How to run database you can know in *Getting Started Guide* [http://docs.jboss.org/tools/3.0.1.GA/en/GettingStartedGuide/html/first_seam.html#start_dev_db].

4.2. Creating a Hibernate Mapping File

Hibernate mapping files are used to specify how your objects are related 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 bring up by navigating *New > Hibernate XML mapping file*.

At first you'll be asked to select a package or multiple individual classes to map. It's also possible to create an empty file, don't select any packages or classes and an empty .hbm will be created in the specified location

With depth control option you can define dependences depth for choosing classes (it means to set level of references which is used to collect linked classes to the selection).

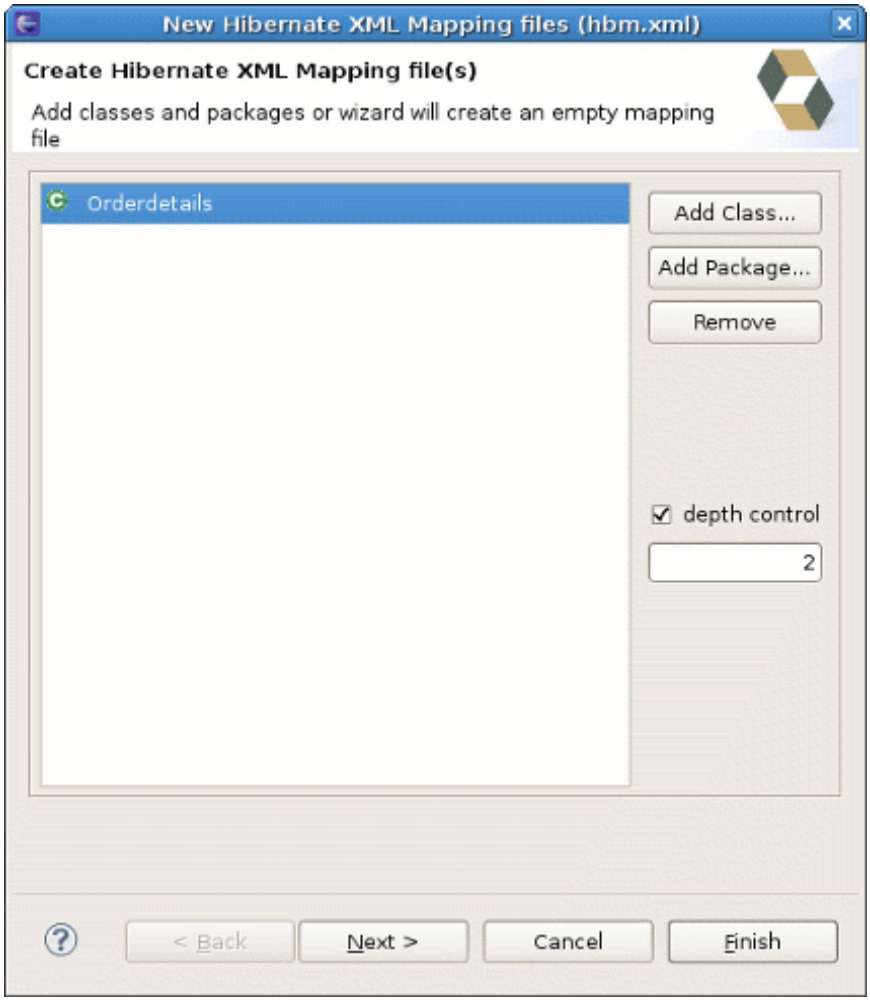

Figure 4.1. Hibernate XML Mapping File Wizard

The next wizard page lists the mappings to be generated. As you see Customers, Orders, Productlines and Products classes added under depth control driving.

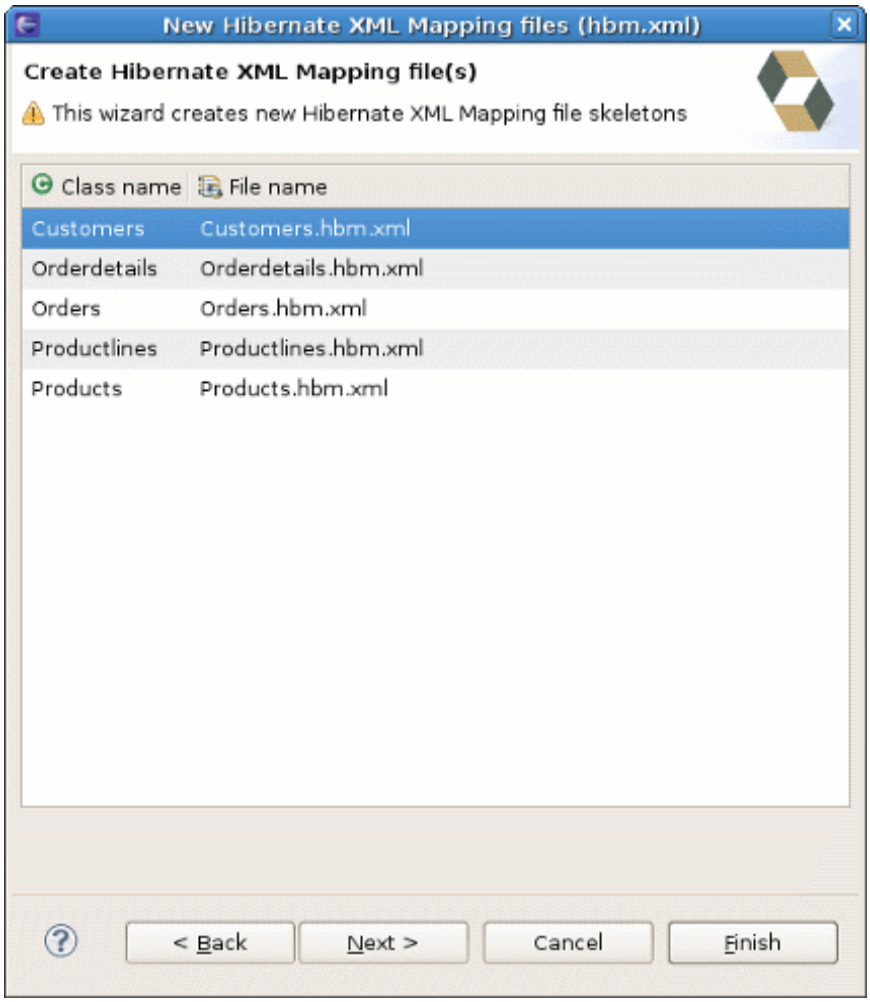

Figure 4.2. Mappings to be generated

This wizard page outputs a generated .hbm files preview.

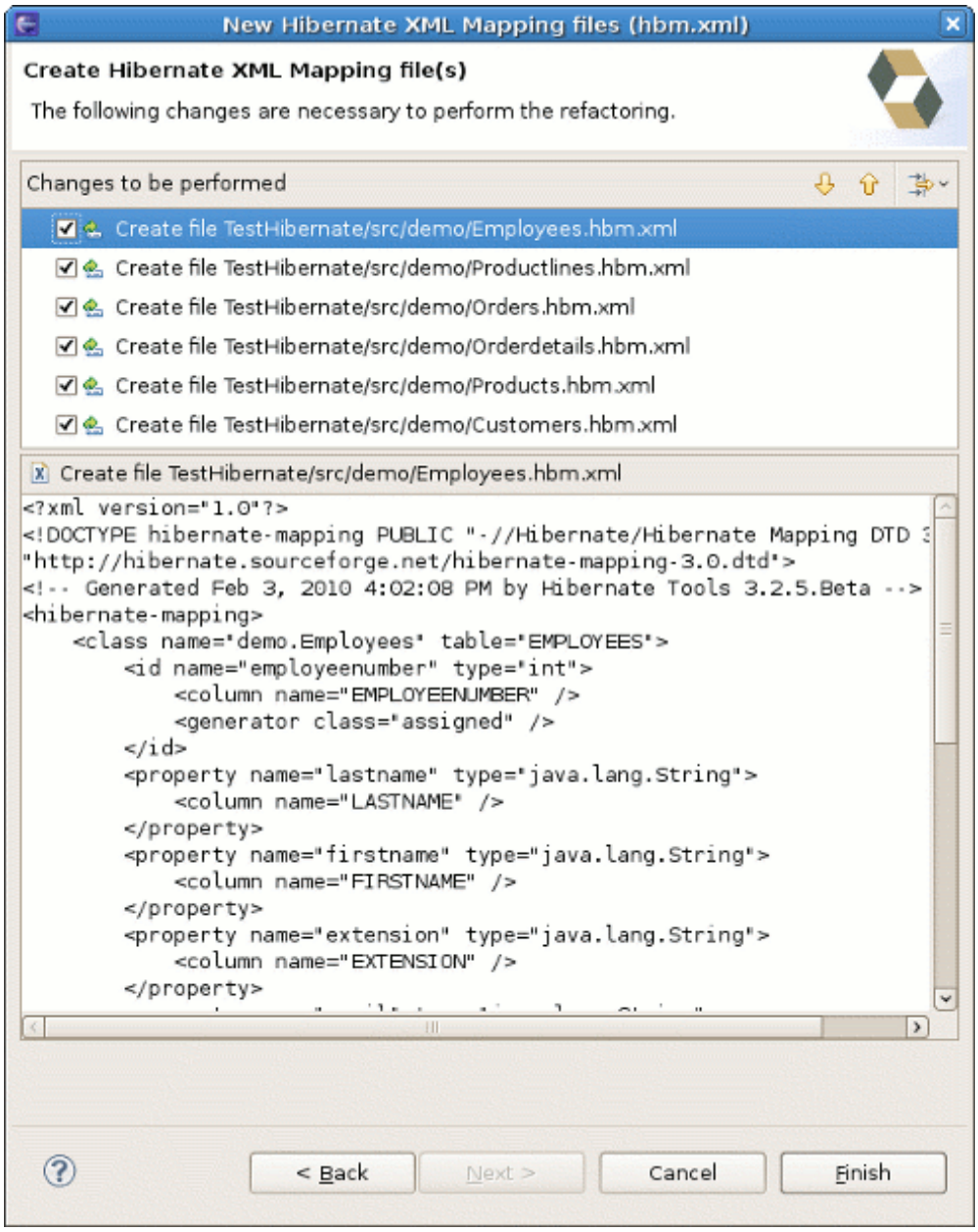

Figure 4.3. Preview Generated Mapping Files

Pressing Finish creates the files.

4.3. Creating a Hibernate Configuration File

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

Start the wizard by clicking *New > Other (Ctrl+N)*, then *Hibernate > Hibernate Configuration File (cfg.xml)* and press *Next* or on a web Seam project in the Web Projects view *WebContent -> New*

-> File -> Hibernate Configuration 3.0 . After selecting the wanted location for the *hibernate.cfg.xml* file, you will see the following page:

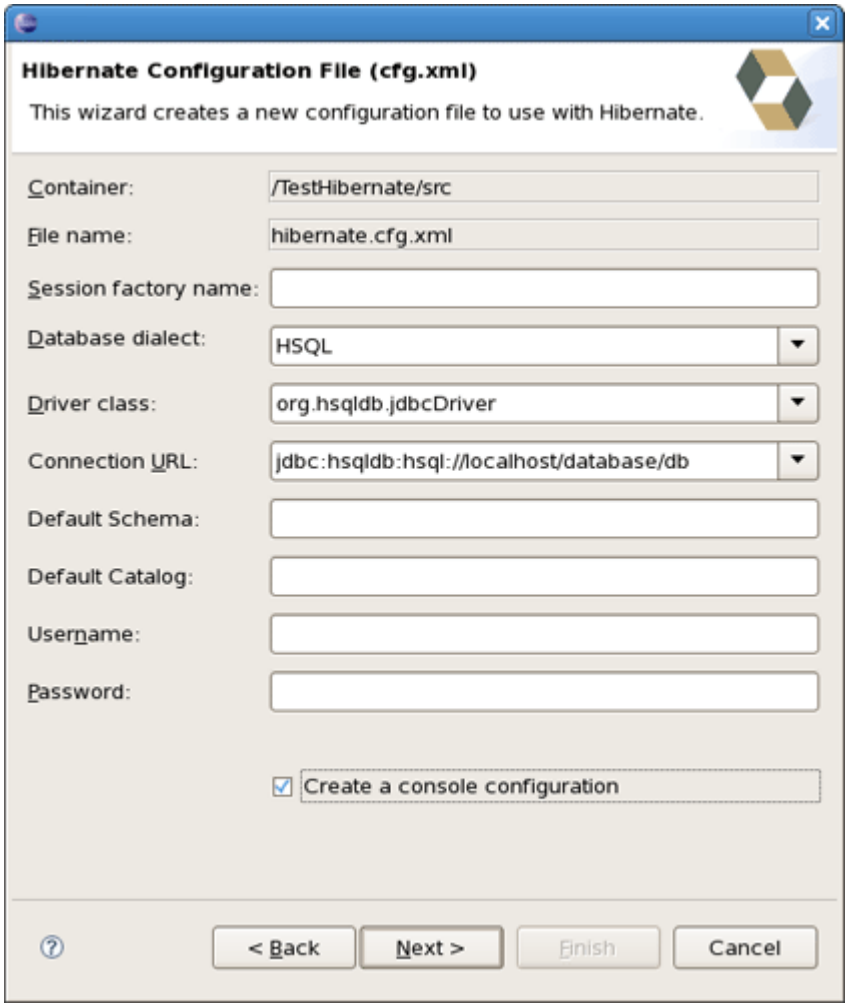

Figure 4.4. Hibernate Configuration File Wizard

Note:

The contents in the combo boxes 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].

Press *Finish* to create the configuration file, after optionally creating a Console configuration, the *hibernate.cfg.xml* 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* 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, including which classpath are needed to load the POJO's, JDBC drivers etc. It is required to make usage of query prototyping, reverse engineering and code generation. You can have multiple named console configurations. Normally you would just need one per project, but more is definitely possible if your project requires this.

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 *Create Console Configuration* .

Note:

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

The dialog consists of five tabs:

- *Main* for the basic/required settings

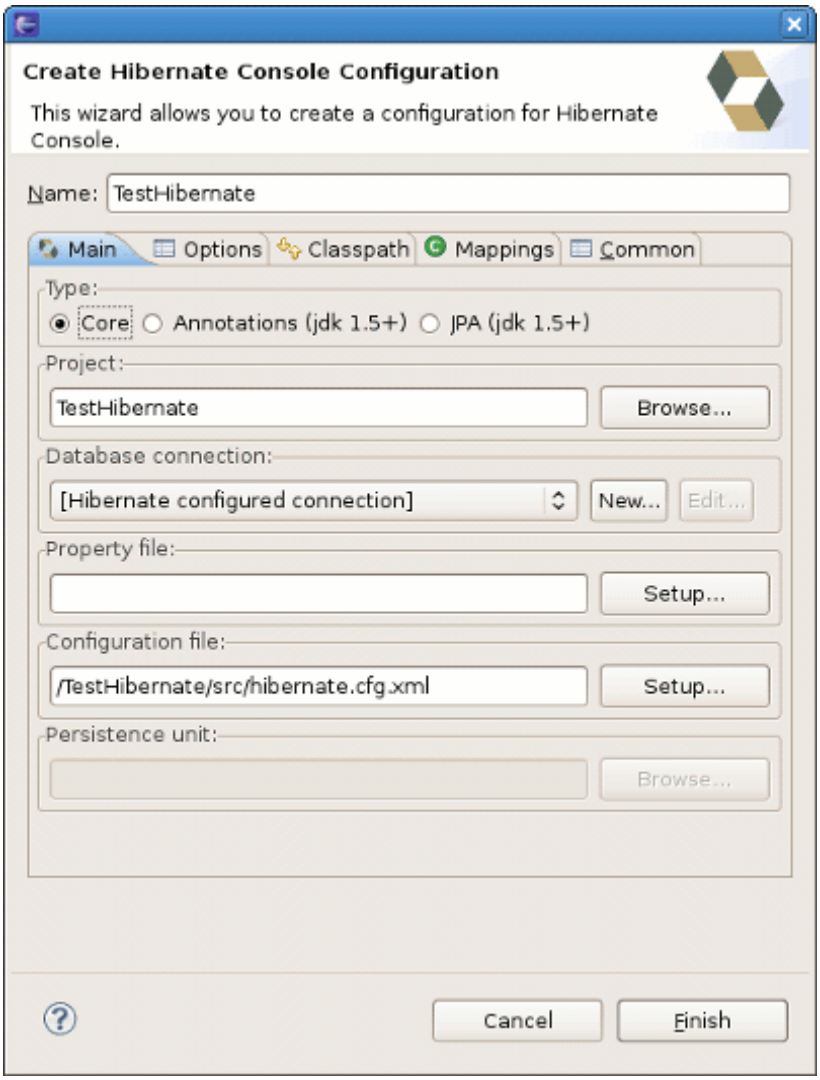

Figure 4.5. Creating Hibernate Console Configuration

The following table describes the available settings on the *Main* tab. The wizard can automatically detect the default values for most of them if you started the wizard with the relevant java project or resource selected.

Table 4.1. Hibernate Console Configuration Parameters

Parameter	Description	Auto detected value
Name	The unique name of the console configuration	Name of the selected project
Type	Choose between "Core", "Annotations" and "JPA". Note that the two latter requires running Eclipse IDE with a JDK 5 runtime, otherwise you will get classloading and/or version errors.	No default value

Parameter	Description	Auto detected value
Project	The name of a java project which classpath should be used in the console configuration	Name of the selected project
Database connection	DTP provided connection that you can use instead of what is in cfg.xml and jpa persistence.xml. It's possible to use either already configured hibernate or JPA connection or specify a new one here.	[Hibernate Configured connection]
Property file	Path to a hibernate.properties file	First hibernate.properties file found in the selected project
Configuration file	Path to a hibernate.cfg.xml file	First hibernate.cfg.xml file found in the selected project
Persistence unit	Name of the persistence unit to use	No default value (let Hibernate Entity Manager find the persistence unit or it can be defined manually using Browse button)

Tip:

The two latter settings are usually not required if you specify a project and it has or in its project classpath.

- *Options* for the additional/optional settings

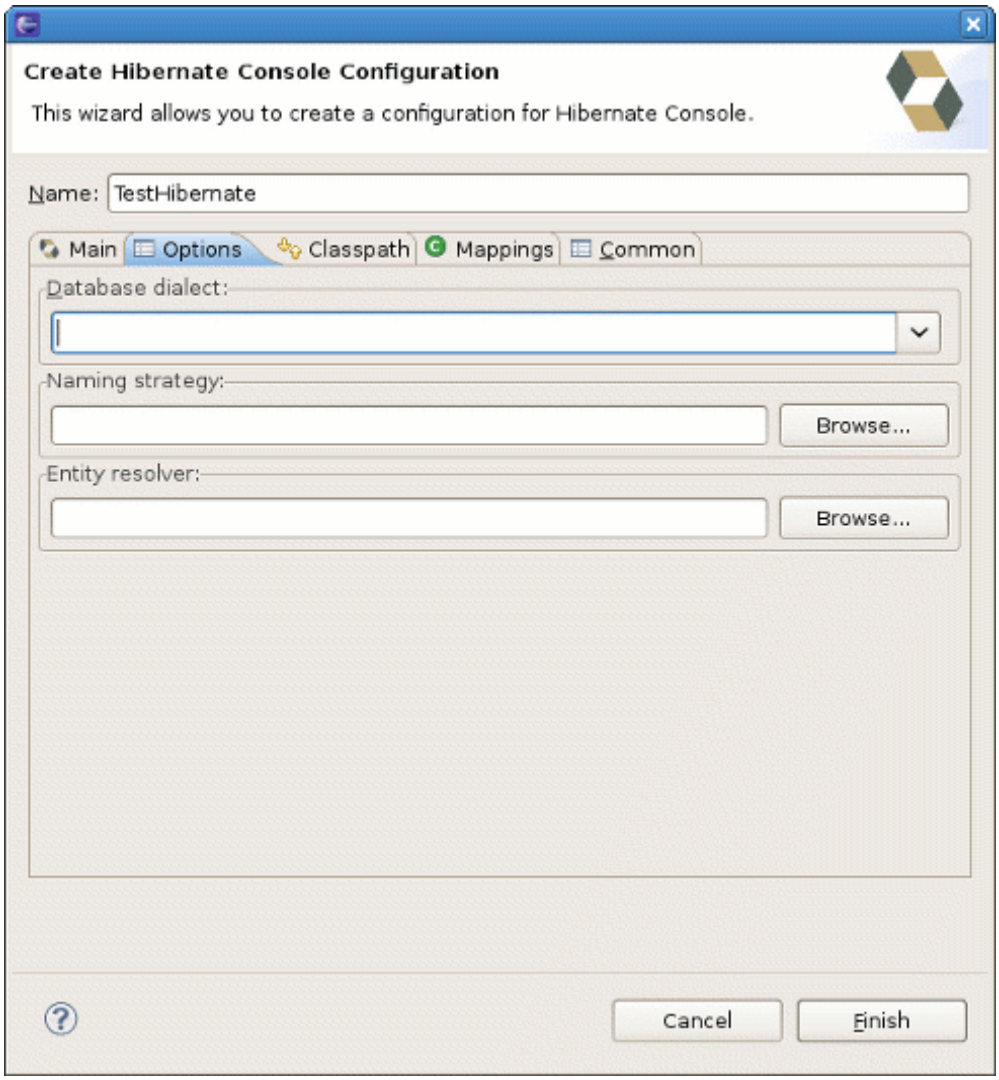

Figure 4.6. Options Tab of the Console Configuration Wizard

The next table describes Hibernate Console Configuration options available on the *Options* tab.

Table 4.2. Hibernate Console Configuration Options

Parameter	Description	Auto detected value
Database dialect	Define a database dialect. It's possible either to write your value or choose from list.	No default value
Naming strategy	Fully qualified classname of a custom NamingStrategy. Only required if you use a special naming strategy.	No default value
Entity resolver	Fully qualified classname of a custom EntityResolver. Only required if you have special xml entity includes in your mapping files.	No default value

- *Classpath* for classpath

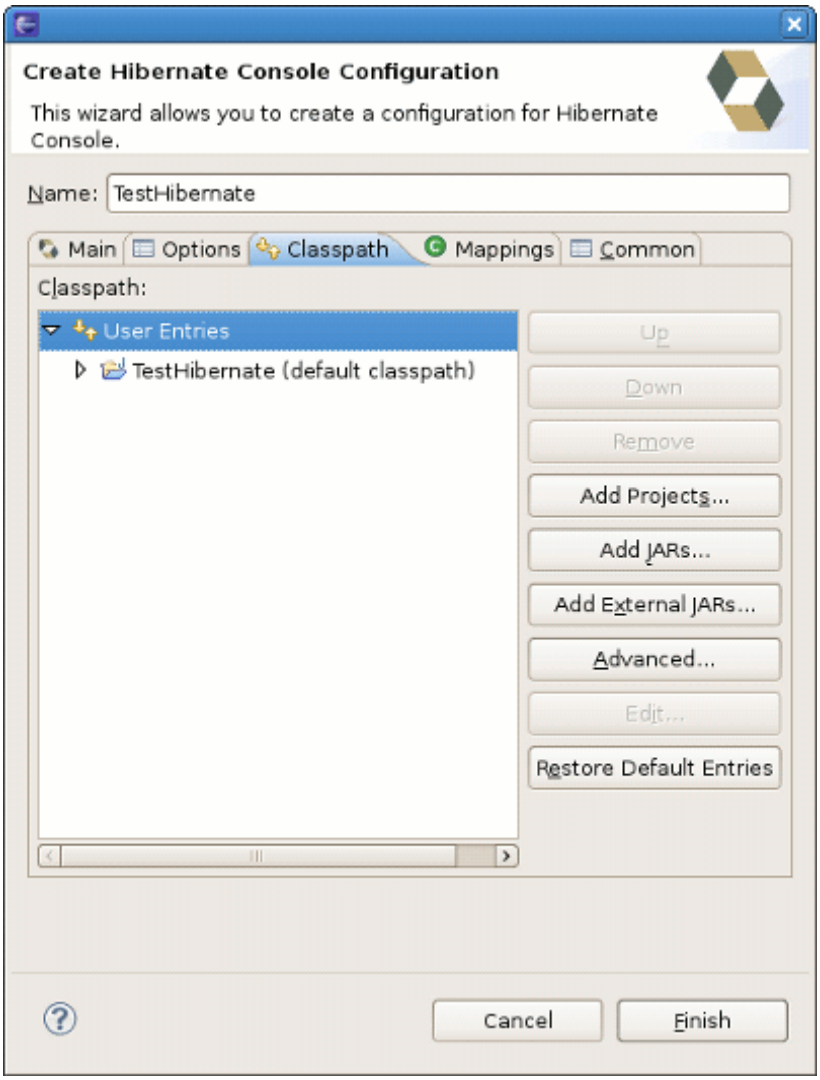

Figure 4.7. Specifying Classpath in Hibernate Console Configuration

The following table specifies the parameters of the Classpath tab of the wizard.

Table 4.3. Hibernate Console Configuration Classpath

Parameter	Description	Auto detected value
Classpath	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/jars.	Empty
Include default classpath from project	When enabled the project classpath will be appended to the classpath specified above	Enabled

- *Mappings* for additional mappings

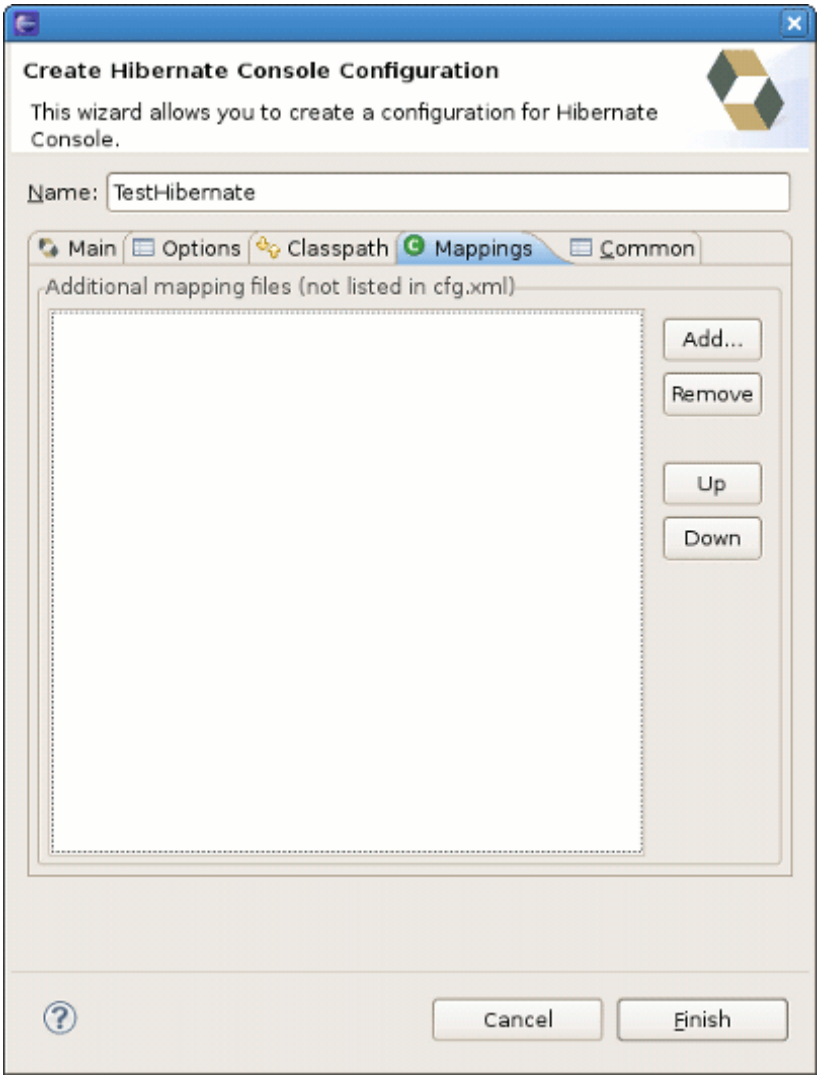

Figure 4.8. Specifying additional Mappings in Hibernate Console Configuration

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

Table 4.4. Hibernate Console Configuration Mappings

Parameter	Description	Auto detected value
Mapping files	List of additional mapping files that should be loaded. Note: A hibernate.cfg.xml or persistence.xml can also contain mappings. Thus if these are duplications here, you will get "Duplicate mapping" errors when using the console configuration.	empty

- and the last tab *Common*

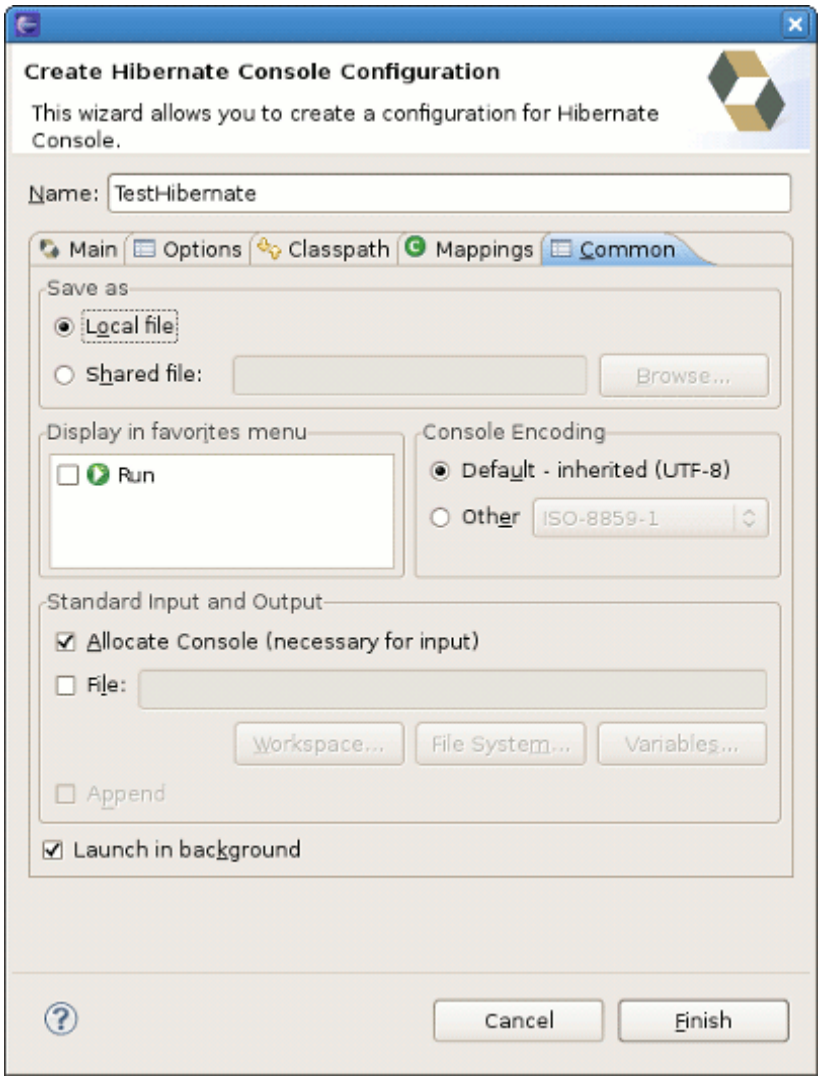

Figure 4.9. Common Tab of the Console Configuration Wizard

It allows to define general aspects of the launch configuration including storage location, console encoding and some others.

Clicking *Finish* creates the configuration and shows it in the Hibernate Configurations view.

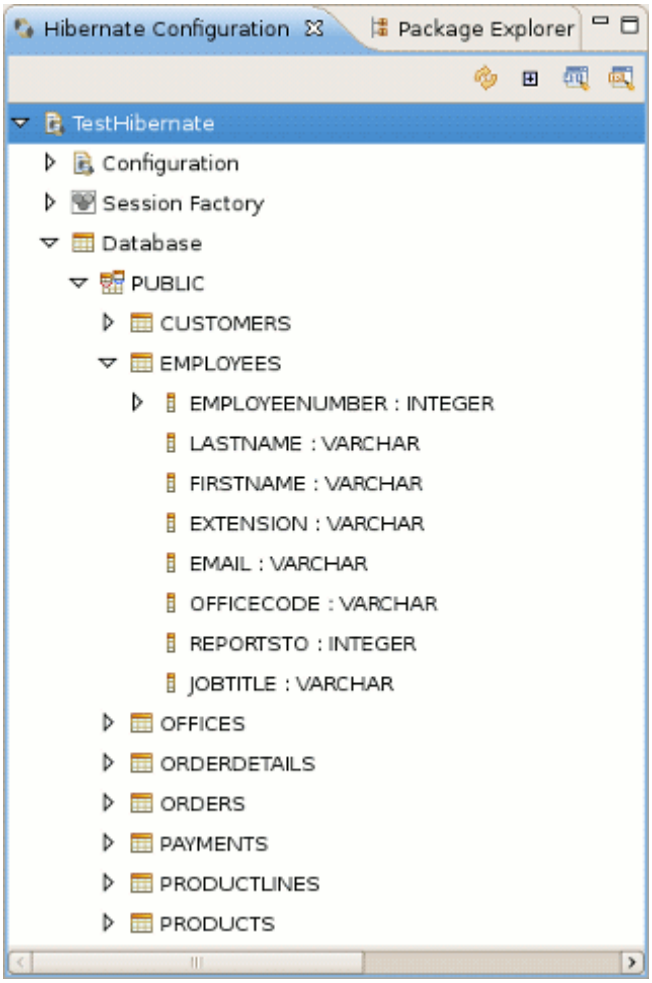

Figure 4.10. Console Overview

4.4.2. Modifying a Hibernate Console Configuration

When you created a hibernate console configuration you can modify it in 2 ways:

- right-click on the configuration in *Hibernate Configurations View->Edit Configuration* or just double-click on Console Configuration item.

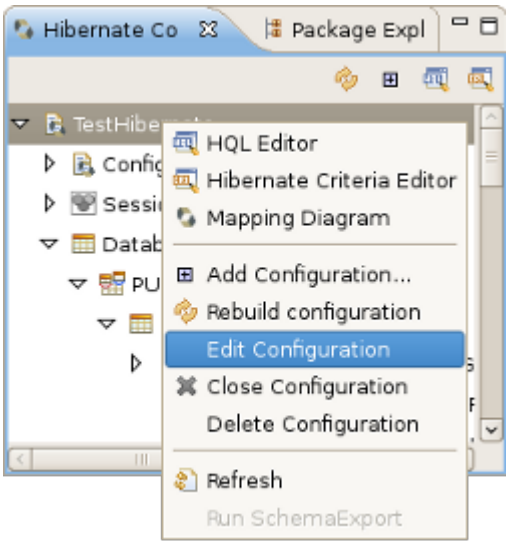

Figure 4.11. Opening Edit Configuration Wizard

After clicking you will see the Edit Configuration Wizard that is similar to *Create Console Configuration* ,described in *Creating a Hibernate Console Configuration section*.

- use Properties view for modifying some of Console Configuration properties.

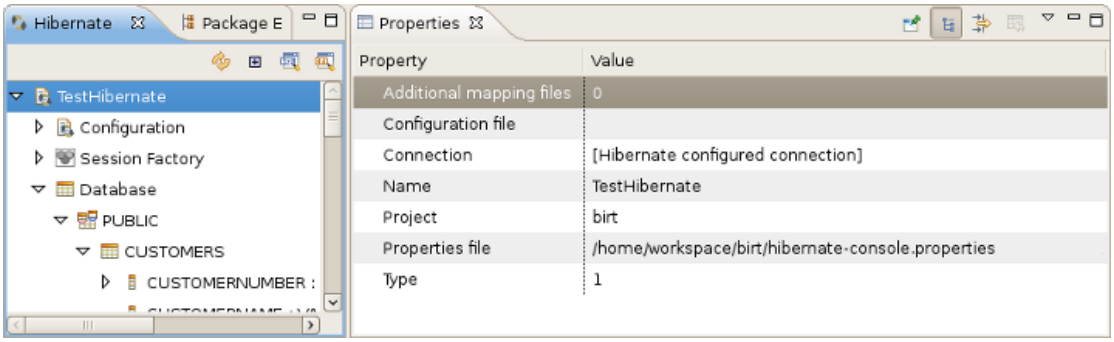

Figure 4.12. Properties View

The following table describes the available settings in the Properties view. Most properties are changeable by left click but some are not.

Table 4.5. Properties

Property	Description	Is Changeable
Additional mapping files	List of additional mapping files that should be loaded.	False
Configuration file	Path to a hibernate.cfg.xml file	False

Property	Description	Is Changeable
Connection	DTP provided connection that you can use instead of what is in cfg.xml and jpa persistence.xml. It's possible to use either already configured hibernate or JPA connection or specify a new one here.	True
Name	The unique name of the console configuration	True
Project	The name of a java project which classpath should be used in the console configuration	True
Properties file	Path to a hibernate.properties file	False
Type	Choose between "CORE", "ANNOTATIONS" and "JPA" 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 classloading and/or version errors.	True

4.4.3. Closing Hibernate Console Configuration

To close Hibernate Console Configuration you need do right-click your configuration and choose Close Configuration option

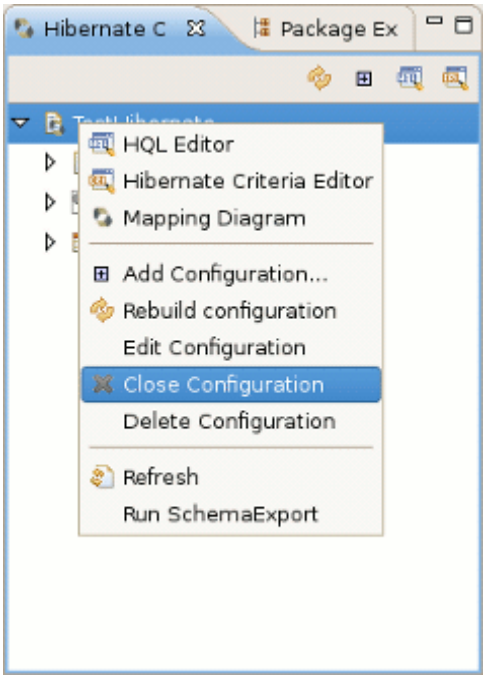

Figure 4.13. Close Hibernate Console Configuration

While closing 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

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

To start working with this process, start the Hibernate Code Generation which is available in 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 *Open Hibernate Code Generation Dialog...* the standard Eclipse launcher dialog will appear. In this dialog you can create, edit and delete named Hibernate code generation "launchers".

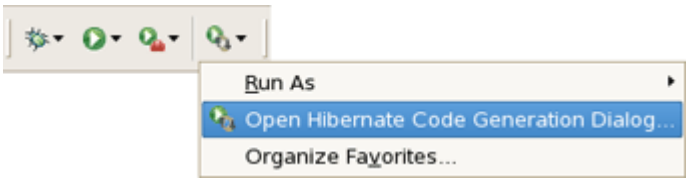

Figure 4.14. Getting Hibernate Code Generation Wizard

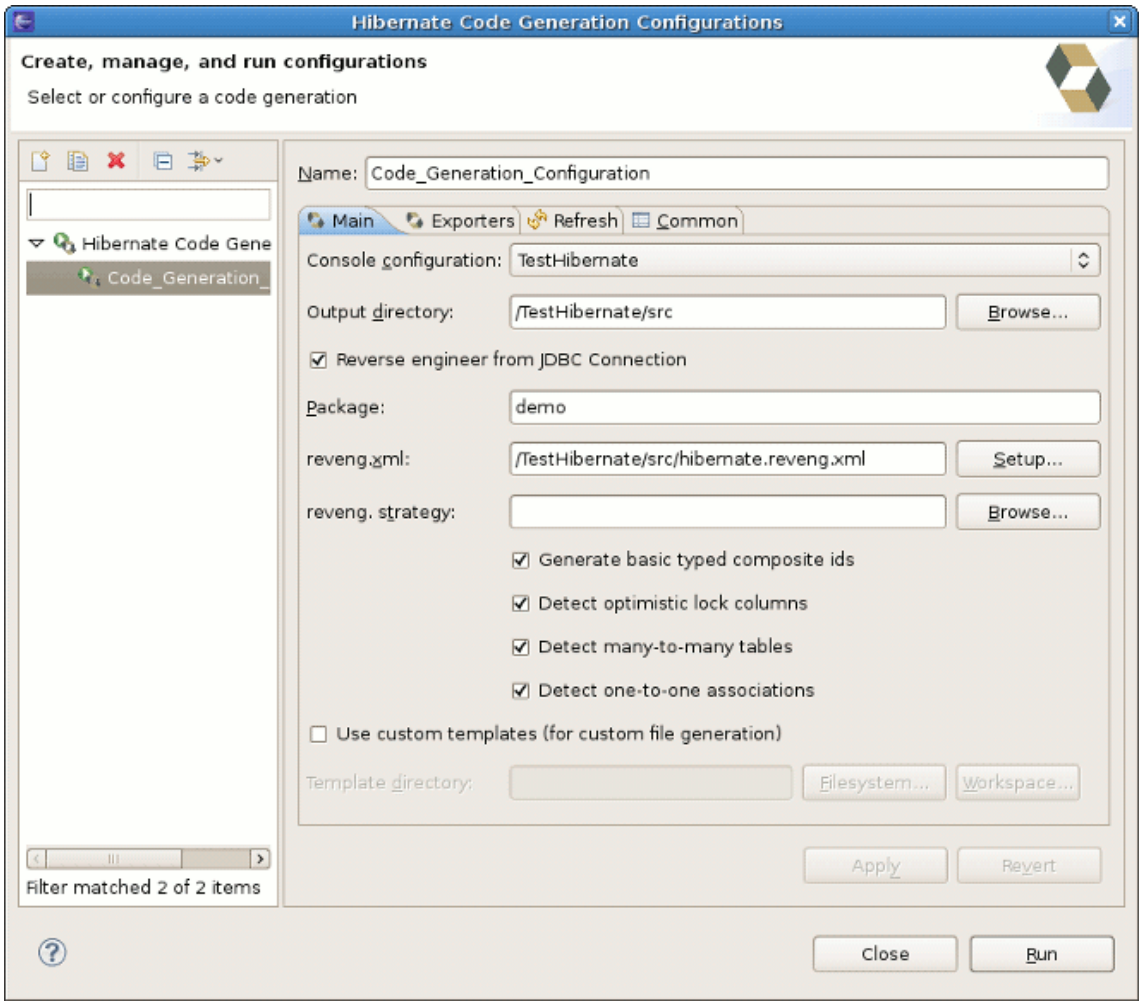

Figure 4.15. Hibernate Code Generation Wizard

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" is just a warning stating that for this launch 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 have the standard tabs *Refresh* and *Common* that can be used to configure which directories should be automatically refreshed and various general settings 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

Field	Description
Console Configuration	The name of the console configuration which should be used when code generating
Output directory	Path to a directory where all output will be written by default. 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.
Reverse engineer from JDBC Connection	If enabled, the tools will reverse engineer the database available via the connection information in the selected Hibernate Console Configuration and generate code based on the database schema. If not enabled, the code generation will just be based on the mappings already specified in the Hibernate Console configuration.
Package	The package name here is used as the default package name for any entities found when reverse engineering
reveng.xml	Path to a reveng.xml file. A reveng.xml file allows you to control certain aspects of the reverse engineering. e.g. how jdbc types are mapped to hibernate types and especially important which tables are included/ excluded from the process. Clicking "setup" 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*.
reveng. strategy	If reveng.xml does not provide enough customization you can provide your own implementation of an 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.
Generate basic typed composite ids	A table that has a multi-column primary key a <composite-id> 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 `<key-many-to-one>` instead. Note: a `<many-to-one>` property is still created, but is simply marked as non-updatable and non-insertable.
Detect optimistic lock columns	Automatically detect optimistic lock columns. Controllable via reveng. strategy; the current default is to use columns named VERSION or TIMESTAMP.
Detect many-to-many tables	Automatically detect many-to-many tables. Controllable via reveng. strategy.
Detect one-to-one associations	Reverse engineering detects one-to-one associations via primary key and both hbm.xml and annotation generation generates the proper code for it.

Field	Description
	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 if not wanted.
Use custom templates	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.
Template directory	A path to a directory with custom templates

4.5.2. Exporters

The *Exporters* tab is used to specify which type of code that should be generated. Each selection represents an Exporter that is responsible for generating the code, hence the name.

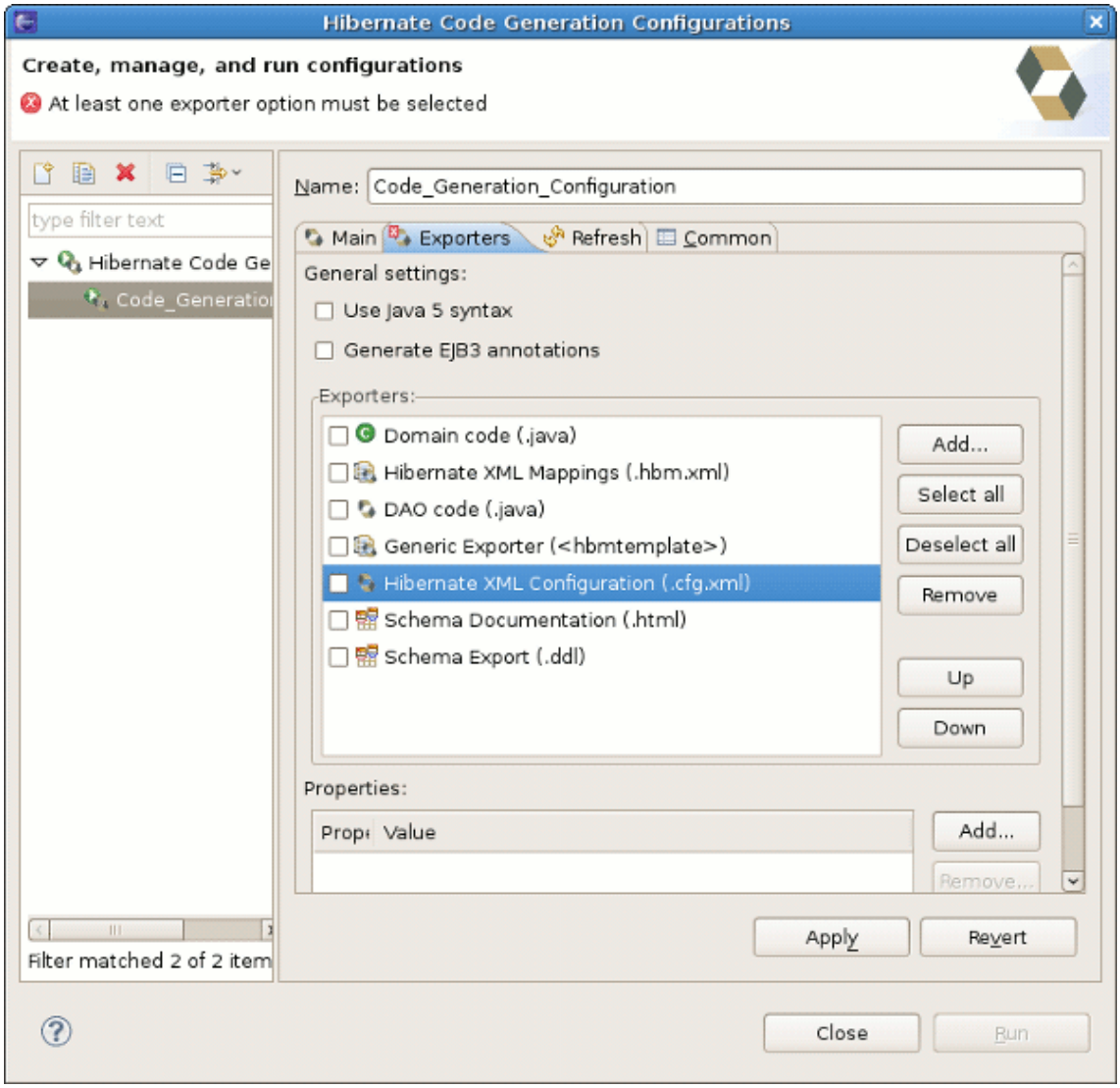

Figure 4.16. Selecting Exporters

The following table describes in short the various exporters. Remember you can add/remove any Exporters depending on your needs.

Table 4.7. Code generation "Exporter" tab fields

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

Each Exporter listens to certain properties and these can be setup in the *Properties* section where you can add/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

Name	Description
jdk5	Generate Java 5 syntax
ejb3	Generate EJB 3 annotations
for_each	Specifies for which type of model elements the exporter should create a file and run through the templates. Possible values are: entity, component, configuration
template_path	Custom template directory for this specific exporter. You can use Eclipse variables.
template_name	Name for template relative to the template path
outputdir	Custom output directory for this specific exporter. You can use Eclipse variables.
file_pattern	Pattern to use for the generated files, relatively for the output dir. Example: {package-name}/{class-name}.java .
dot.executable	Executable to run GraphViz (only relevant, but optional for Schema documentation)

Name	Description
drop	Output will contain drop statements for the tables, indices and constraints
delimiter	If specified the statements will be dumped to this file
create	Output will contain create statements for the tables, indices and constraints
scriptToConsole	The script will be output to Console
exportToDatabase	Executes the generated statements against the database
outputFileName	If specified the statements will be dumped to this file
haltOnError	Halts the build process if an error occurs
format	Applies basic formatting to the statements
schemaUpdate	Updates a schema

To add a property to the chosen Exporter click the *Add* button in the Properties section. In the appeared dialog you should select the property from the proposed list and the value for it.

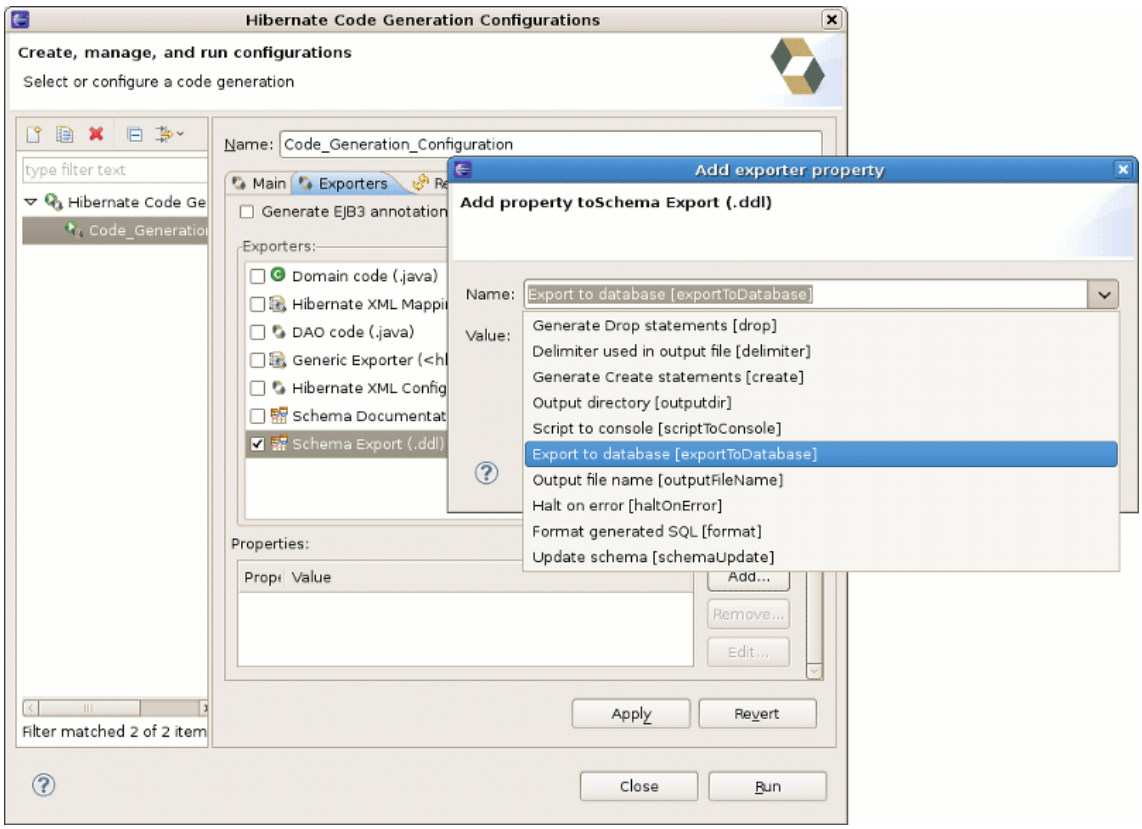

Figure 4.17. Adding the Property for Schema Export (.ddl)

Tip:

If the property is a directory, it is possible to browse directories in the Value field.

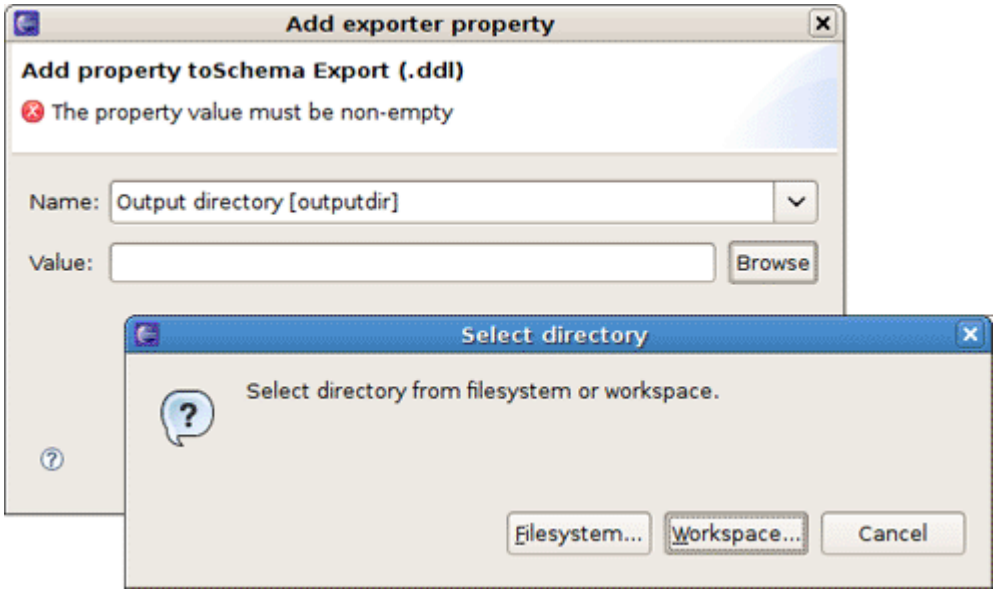

Figure 4.18. Specifying the Property Value

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.

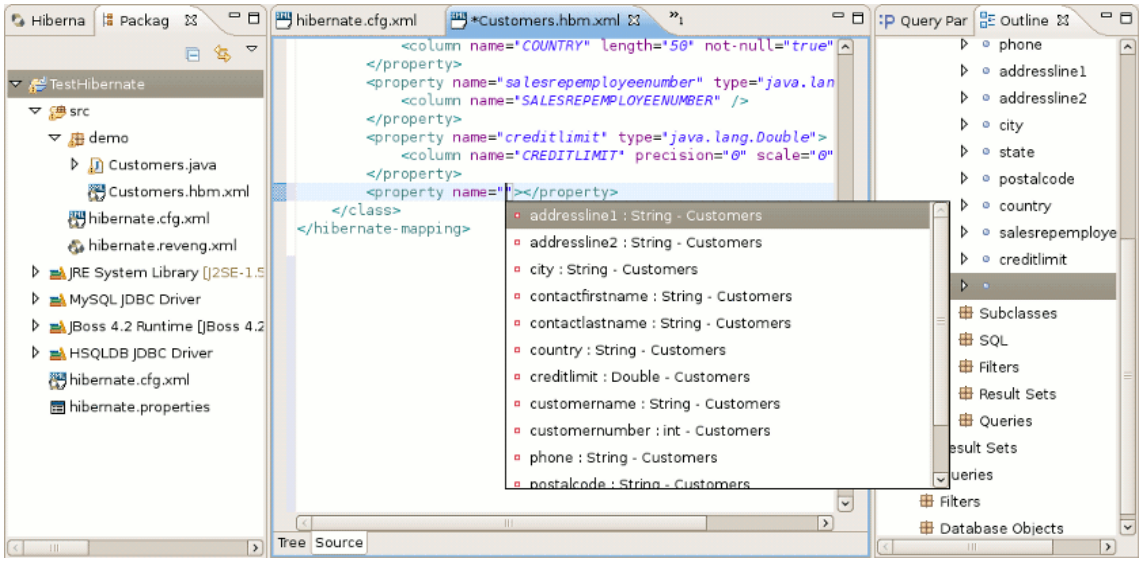

Figure 4.19. XML Editing Functionality

4.6.1. Java property/class completion

Package, class, and field completion is enabled for relevant XML attributes. The auto-completion detects its context and limits the completion for e.g. `<property>` and only shows the properties/fields available in the enclosing `<class>` , `<subclass>` etc. It is also possible to navigate from the *hbm.xml* files to the relevant class/field in java code.

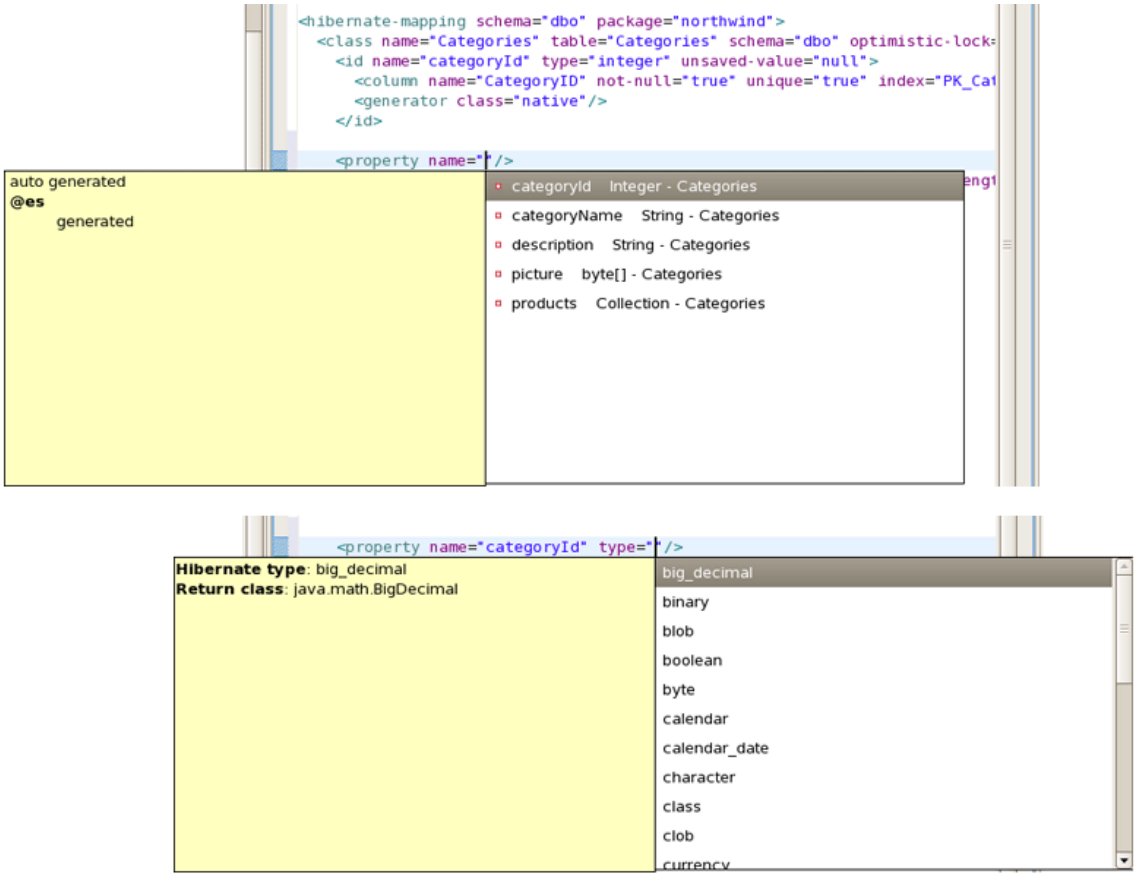

Figure 4.20. Navigation Functionality

This is done via the standard hyperlink navigation functionality in Eclipse; per default it is done by pressing F3 while the cursor is on a class/field or by pressing *Ctrl* and the mouse button to perform the same navigation.

For java completion and navigation to work the file needs to reside inside an Eclipse Java project, otherwise no completion will occur.

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.

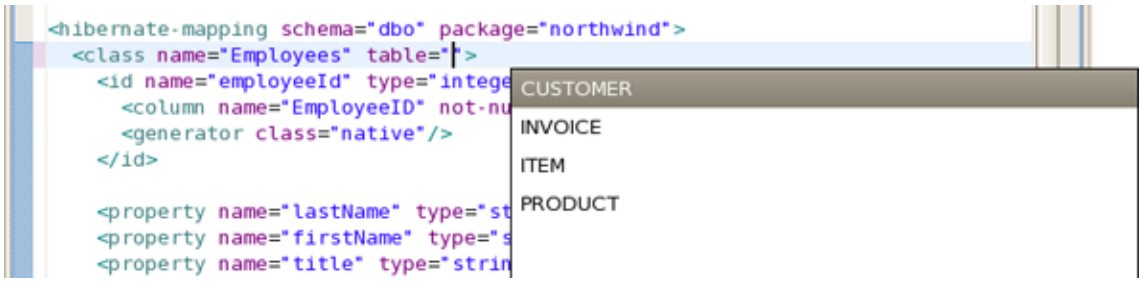

Figure 4.21. Table and Column Completion

Important:

Table/Column completion requires a proper 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 under the Properties of a project and look under the *Hibernate Settings* page. When a proper configuration is selected it will be used to fetch the table/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. It will be improved in future versions.

4.6.3. Configuration property completion

In *cfg.xml* code completion for the value of `<property> name` attributes is available.

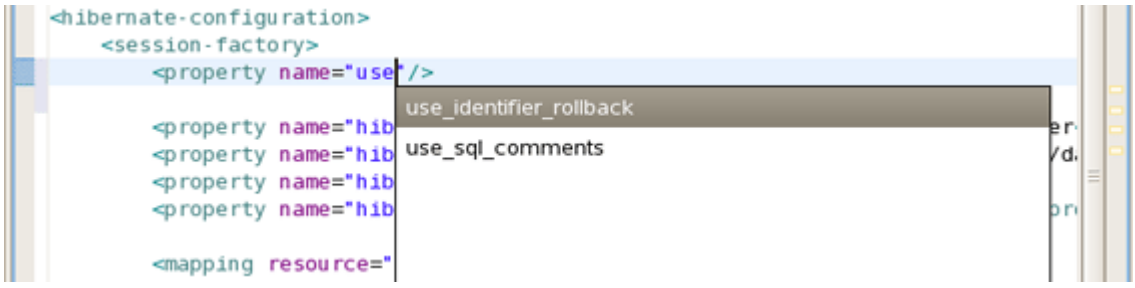

Figure 4.22. Property Completion

4.7. Structured Hibernate Mapping and Configuration File Editor

The structured editor represents the file in the tree form. It also allows 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, choose *Open With > Hibernate 3.0 XML Editor* option from the context menu of the file. The editor should look as follows:

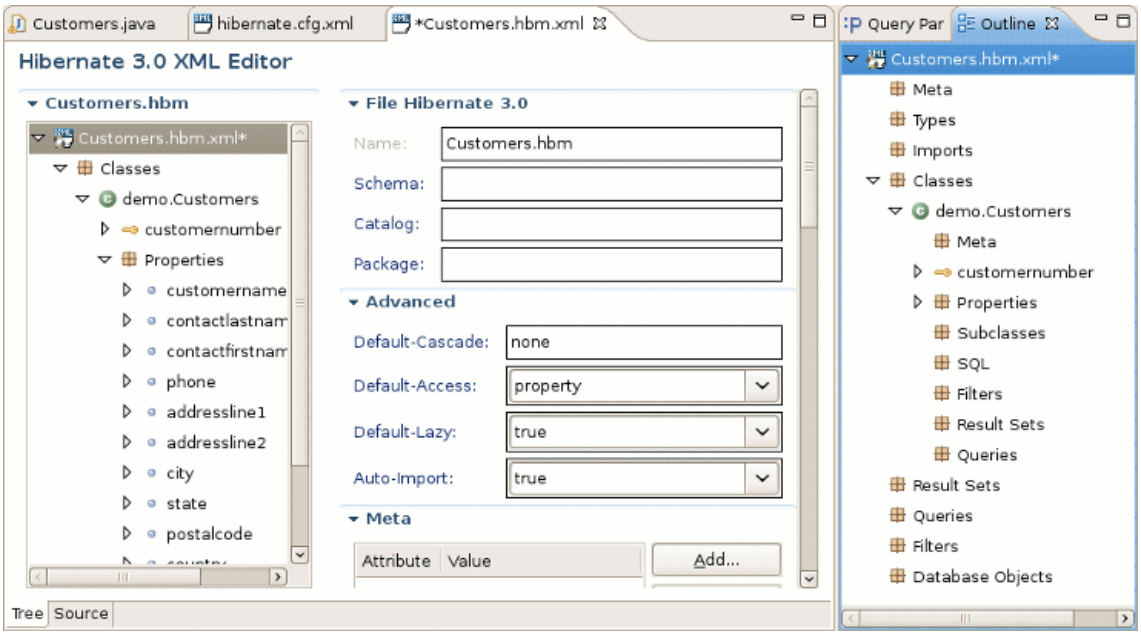

Figure 4.23. Structured hbm.xml Editor

For the configuration file you should choose *Open With > Hibernate Configuration 3.0 XML Editor* option.

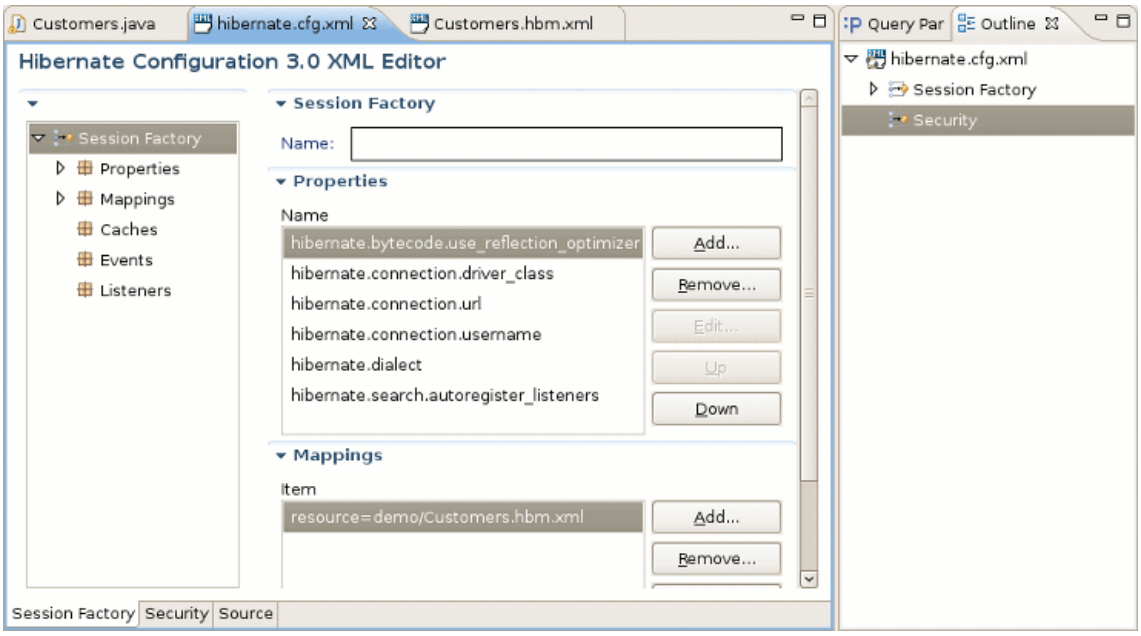

Figure 4.24. Structured cfg.xml Editor

4.8. JBoss Tools Properties Editor

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

For `hibernate.properties` file JBoss Tools Properties Editor provides content assist which is available both for hibernate properties and property 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.

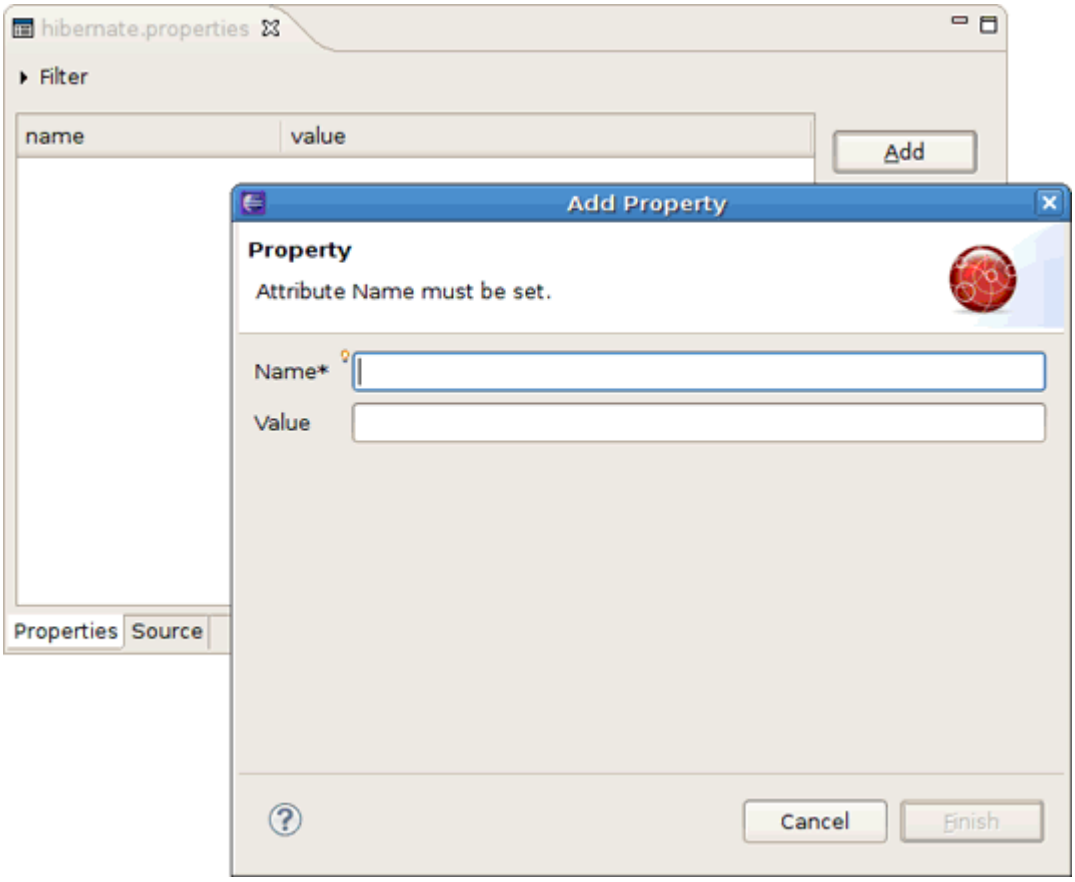

Figure 4.25. Adding the Property

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 of each one.

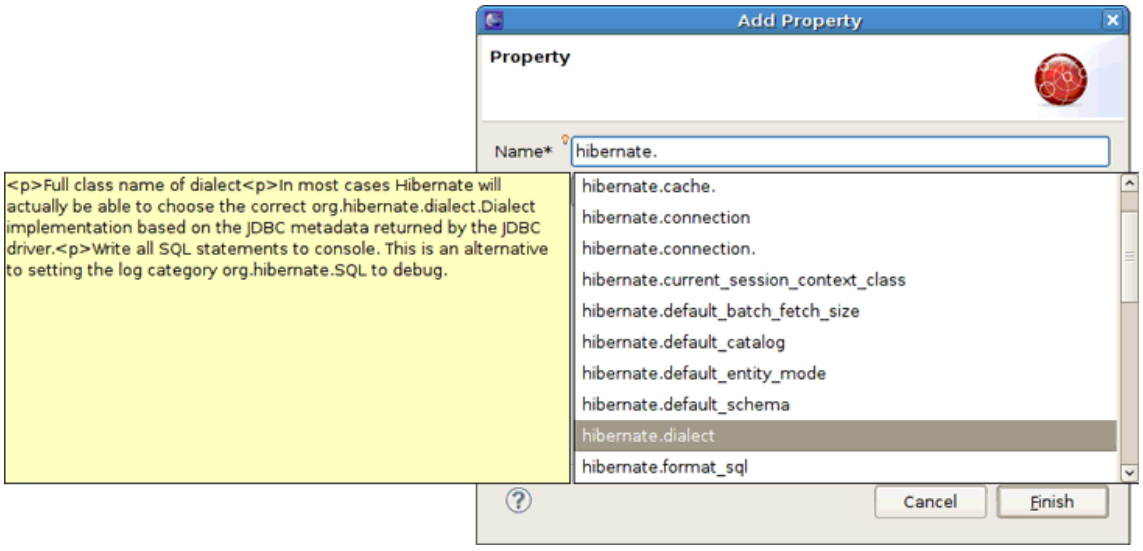

Figure 4.26. Content Assist for Properties Names

When invoking the content assist in the *Value* field, it also provides a list of proposals.

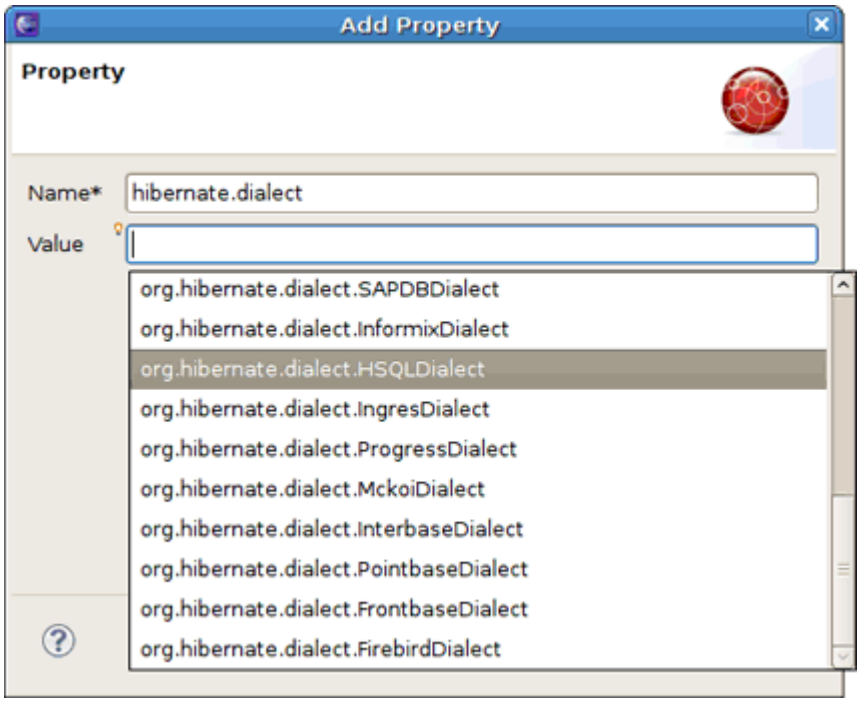

Figure 4.27. Content Assist for Properties Values

In the Source view of the editor, content assist also could be invoked both for properties names and values:

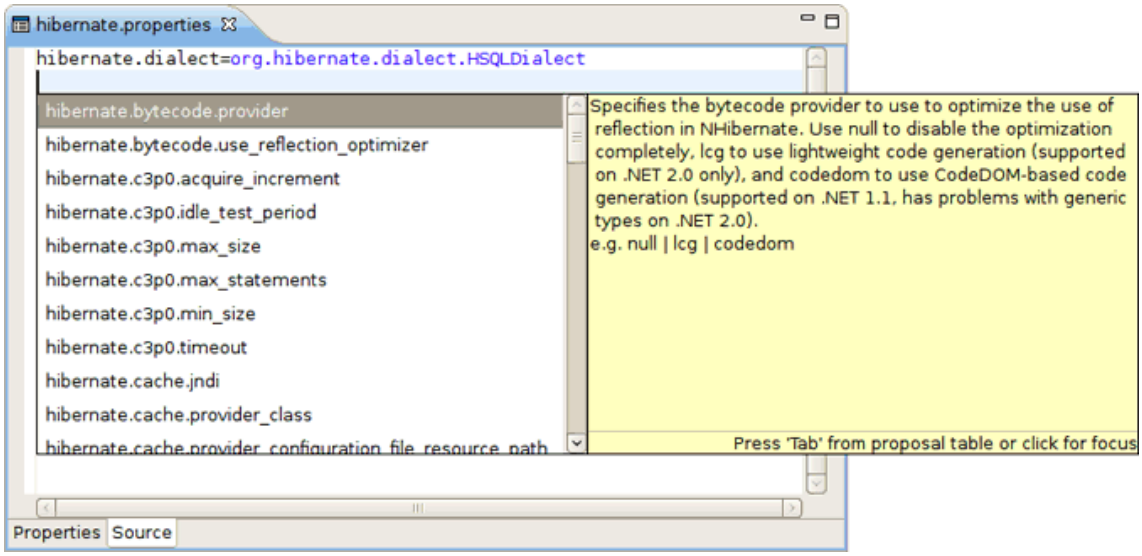

Figure 4.28. Content Assist in the Source view

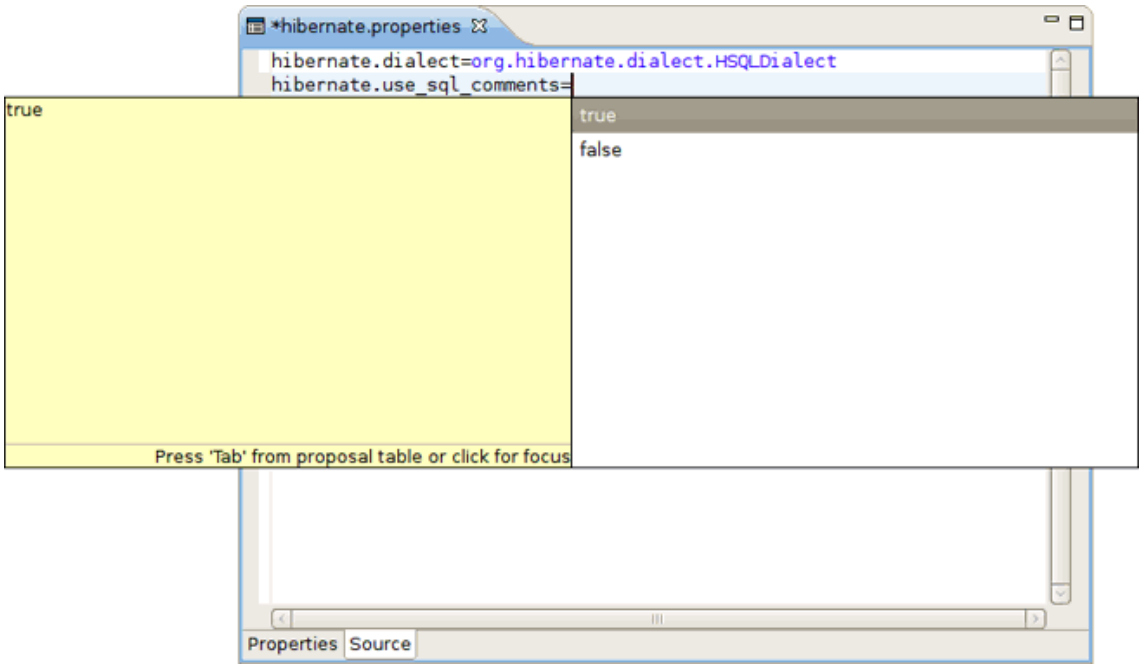

Figure 4.29. Content Assist in the Source view

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 ease the editing of this file and hence used to configure the reverse engineering process.

The editor is intended to allow easy definition of type mappings, table include/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* , please see *Section 6.2, "hibernate.reveng.xml file"*

The editor is activated as soon as an *.reveng.xml* file is opened. To get an initial *reveng.xml* file the Reverse Engineering File Wizard can be started via *Ctrl+N* and *Hibernate > Hibernate Reverse Engineering File (reveng.xml)* then.

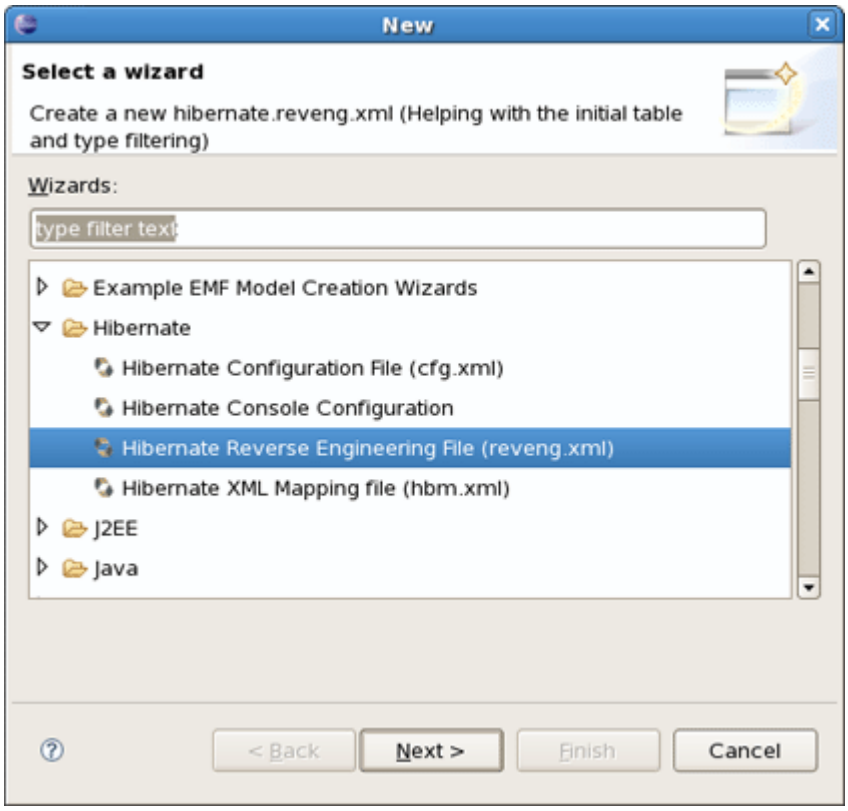

Figure 4.30. ChooseReverse Engineering File Wizard

Or you can get it via the Code Generation Launcher by checking the proper section in the *Main* tab of the *Hibernate Code Generation Wizard*.

The following screenshot shows the *Overview* page where the wanted console configuration is selected (auto-detected if Hibernate 3 support is enabled for the project)

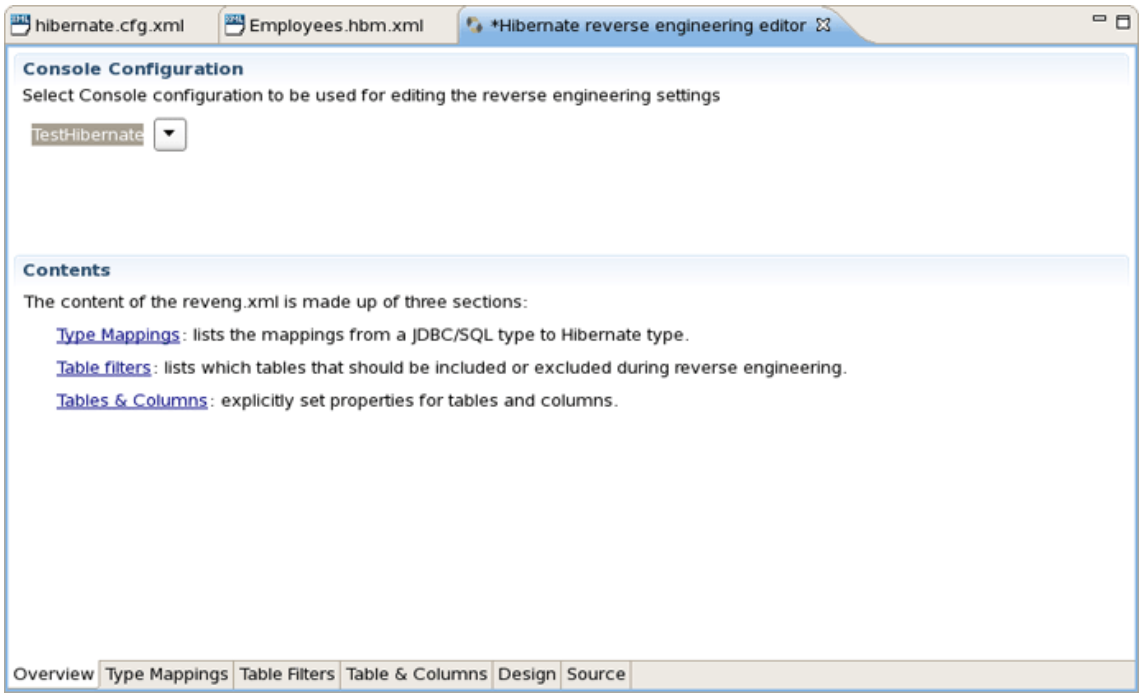

Figure 4.31. Overview Page

The *Table Filter* page allows you to specify which tables to include and exclude. Pressing *Refresh* shows the tables from the database that have not yet been excluded.

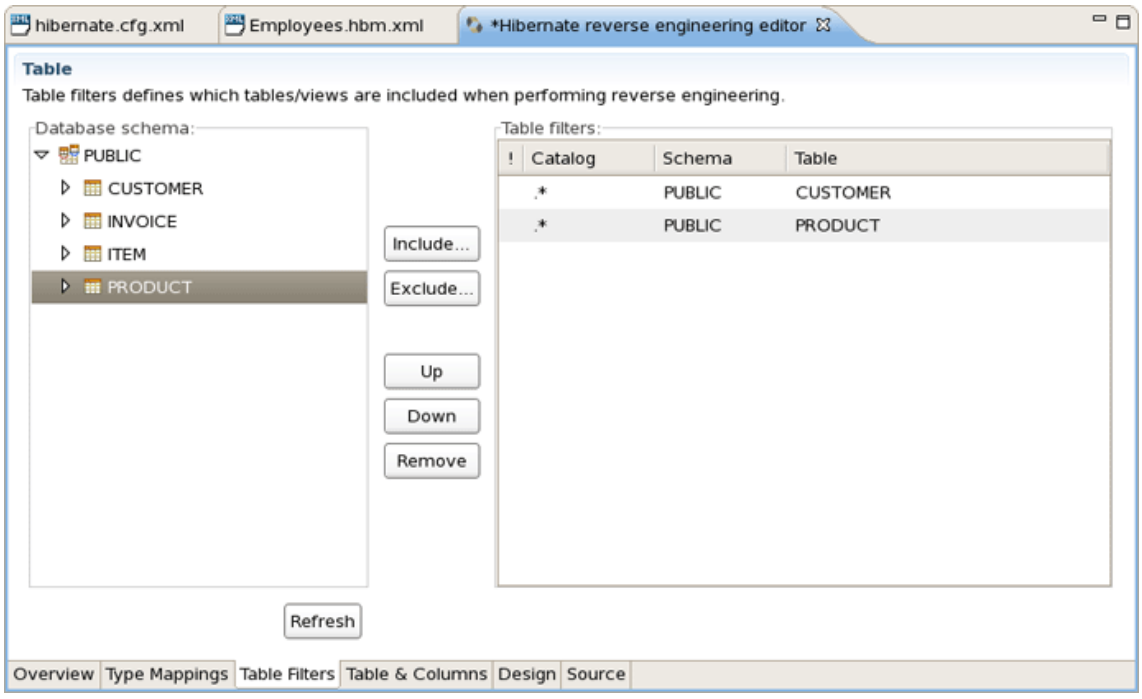

Figure 4.32. Table Filters Page

The *Type Mappings* page is used for specifying type mappings from JBDC types to any Hibernate type (including usertypes) if the default rules are not applicable. Here again to see the database

tables press *Refresh* button underneath. More about type mappings you can find further in the *Type Mappings* section.

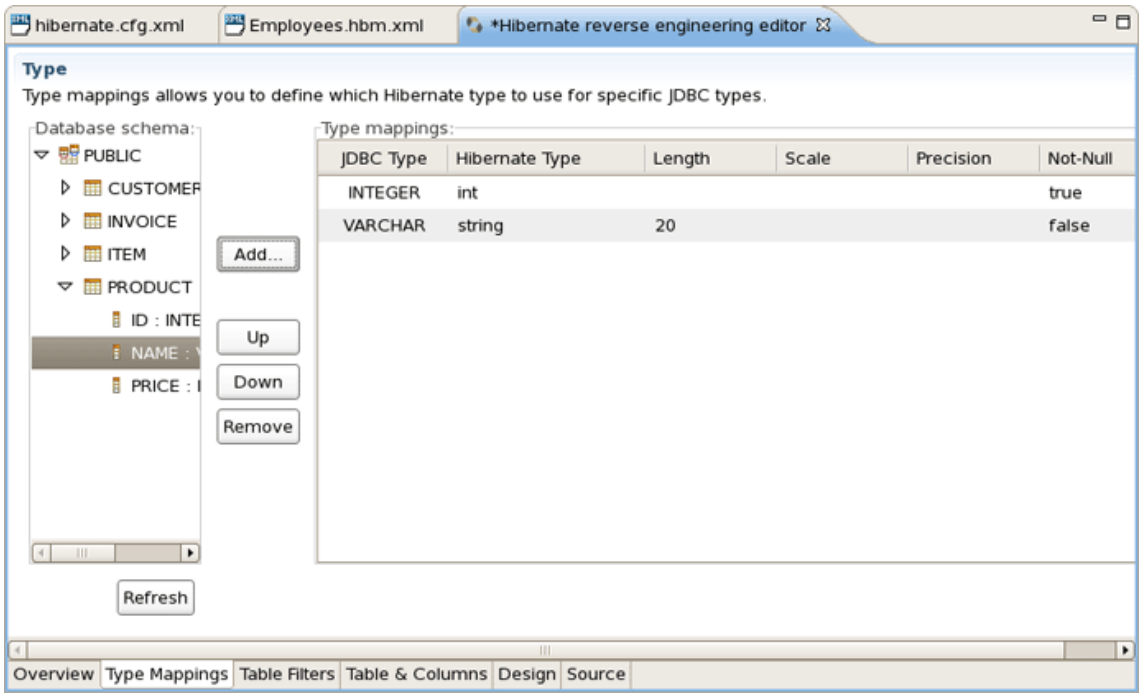

Figure 4.33. Type Mappings Page

The *Table and Columns* page allows you to explicit set e.g. which hibernatetype and propertyname that should be used in the reverse engineered model. For more details on how to configure the tables while reverse engineering read the *Specific table configuration* section.

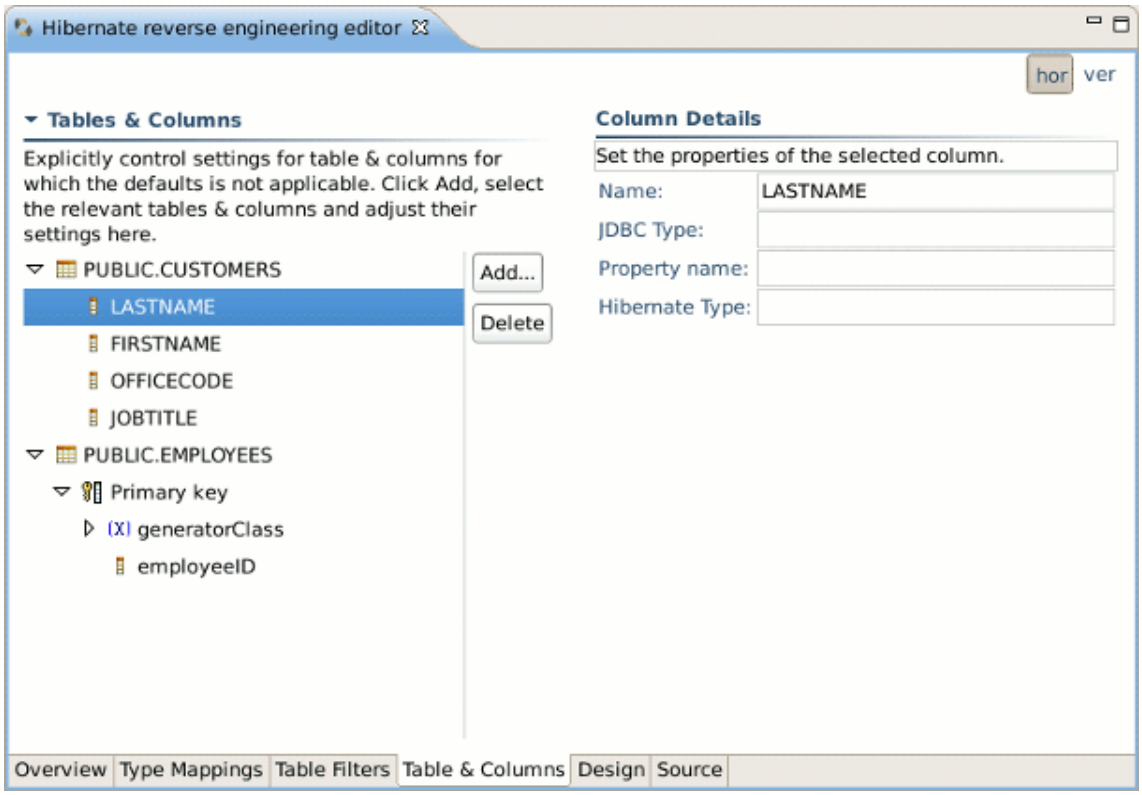

Figure 4.34. Table and Columns Page

Now that you have configured all necessary parts, you can learn how to work with Hibernate Console Perspective.

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/classes, edit HQL queries, execute the queries, and see the results. To use this perspective you need to create a *Console configuration*.

4.10.1. Viewing the entity structure

To view your new configuration and entity/class structure, switch to Hibernate Configurations View. Expanding the tree allows you to browse the class/entity structure and see the relationships.

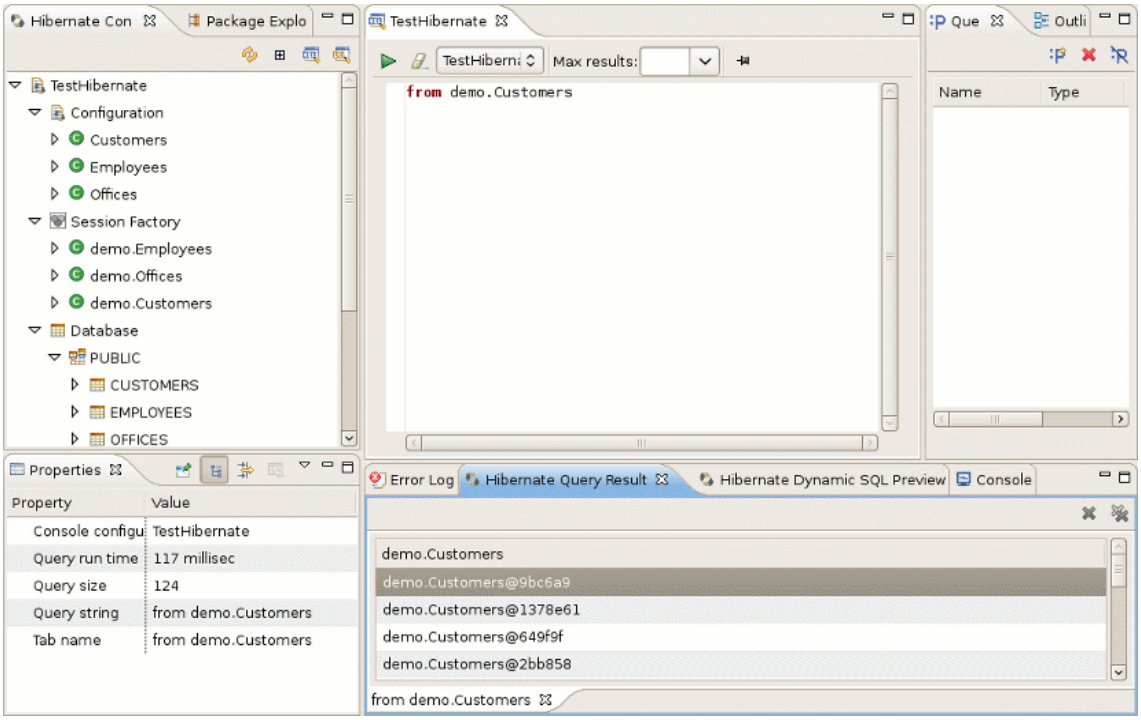

Figure 4.35. Hibernate Console Perspective

The Console Configuration does not dynamically adjust to changes done 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.

Besides, it's possible to open source and mapping files for objects showed in Hibernate Configurations View. Just bring up the context menu for a necessary object and select *Open Source File* to see appropriate Java class or *Open Mapping File* to open a proper *.hbm.xml*.

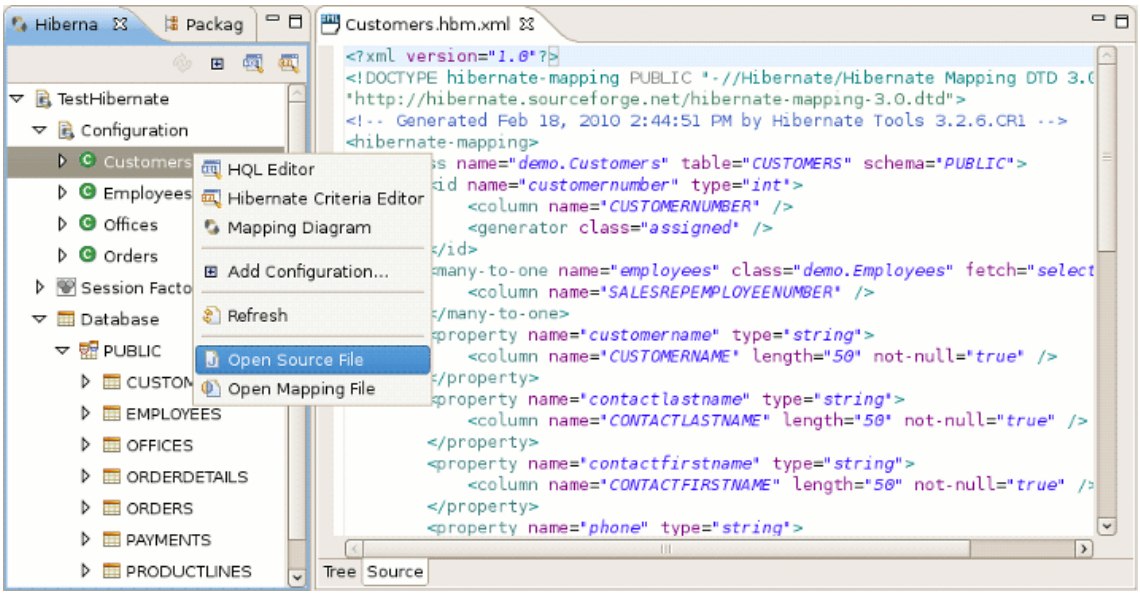

Figure 4.36. Opening Source for Objects

4.10.1.1. Mapping Diagram

In order to get a visual feel on 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 a mapping diagram for and then choosing *Mapping Diagram*.

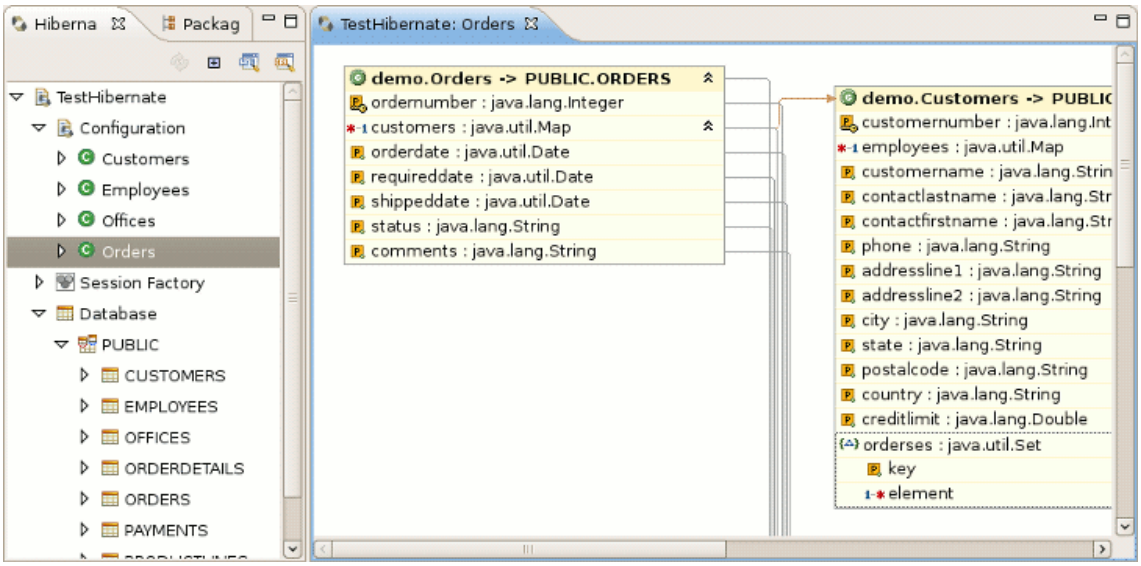

Figure 4.37. Mapping Diagram

To make Mapping Diagram usage easier you can use Rules, Grid,Snap to Geometry checkboxes in the *View* menu.

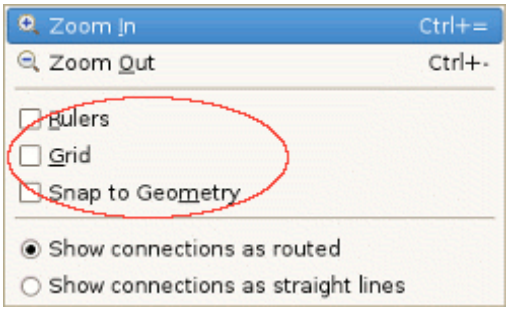

Figure 4.38. View menu

If you will select *Rules* checkbox, the view print page scale will be added to the page. The numbers on the scale show 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.

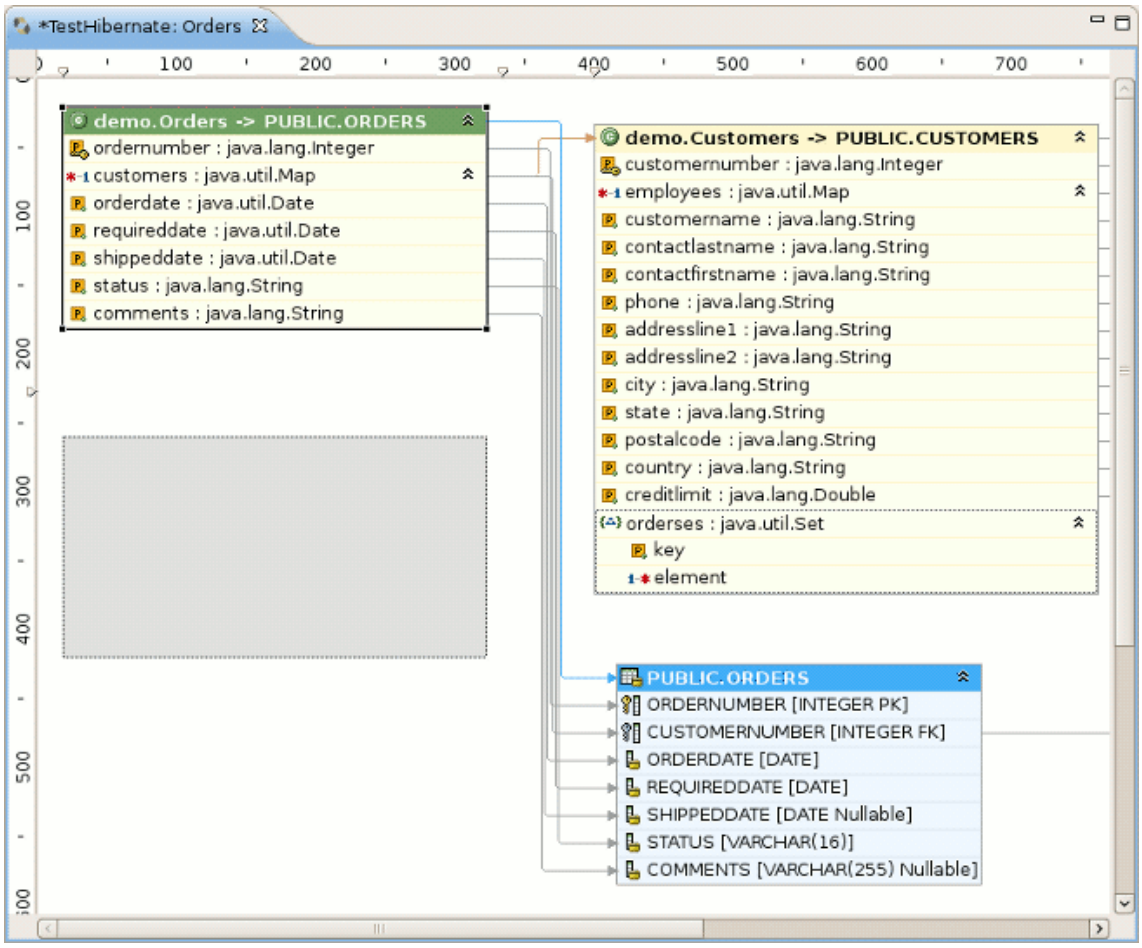

Figure 4.39. Moving the Ruler guide

If you'll select *Grid* checkbox, the grid will appear on the diagram.

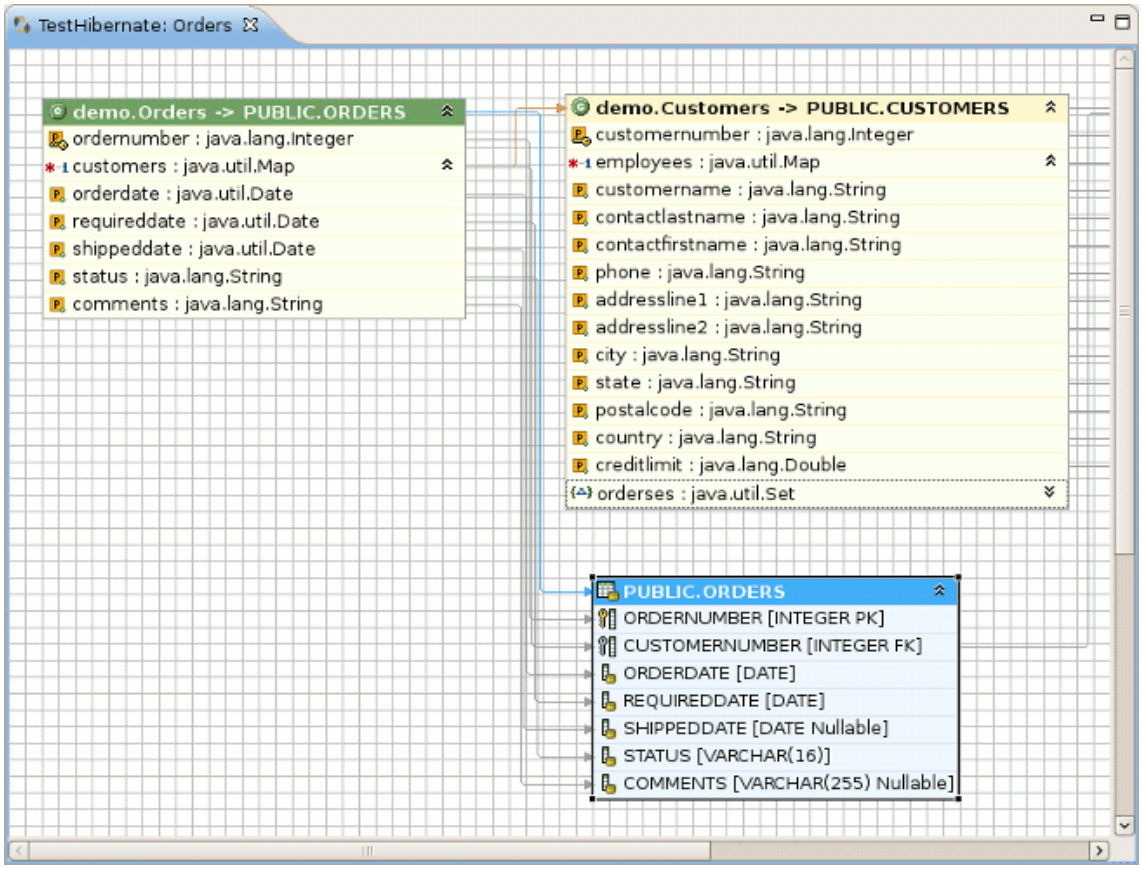

Figure 4.40. Grid on Mapping diagram

The checkbox *Snap to Geometry* helps to put the items of the diagram into allineation with the grid.

For better navigating through the diagram use Outline view which is available in the structural and graphical modes.

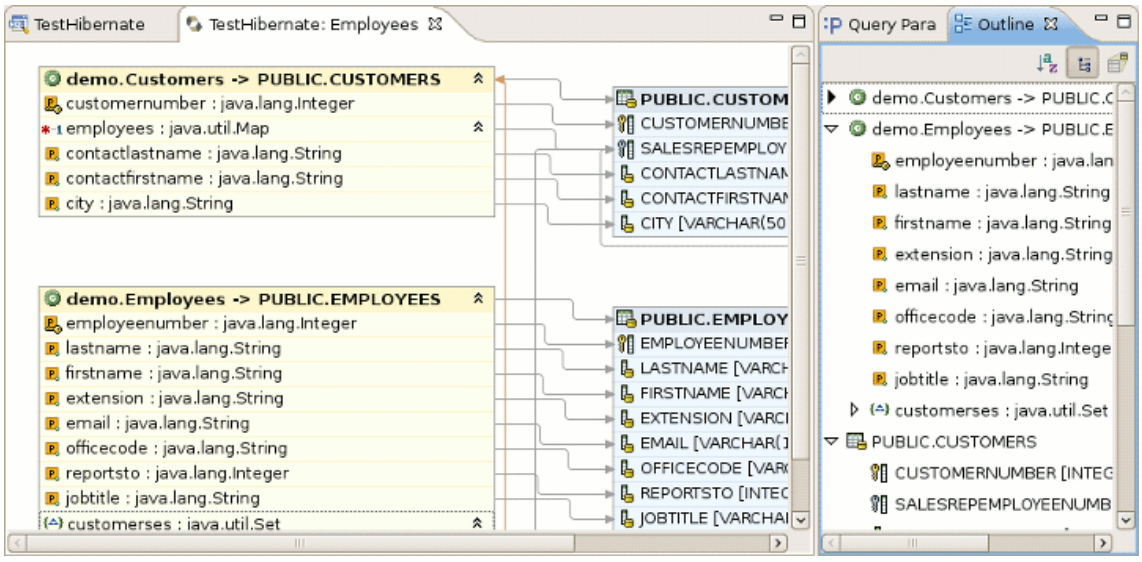

Figure 4.41. Navigating in the Structural Mode

To switch over between the modes use the buttons in the top-right corner of the Outline view.

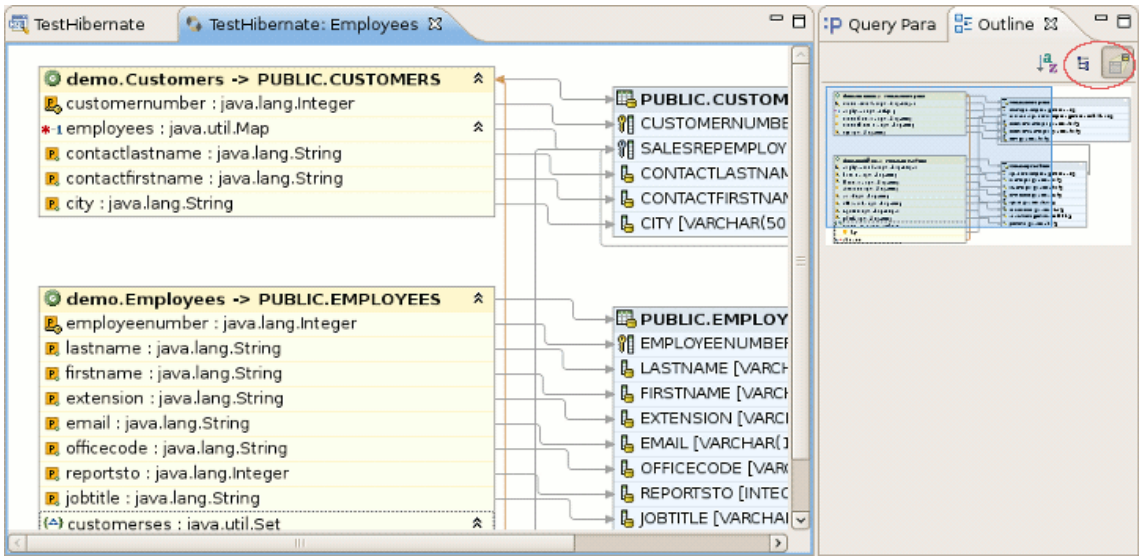

Figure 4.42. Navigating in the Graphical Mode

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

Icon	Command	Description
	Show|Hide connections	Allows to select what types of connections should be shown on the diagram: • Property Mappings

Icon	Command	Description
		• Class Mappings • Associations • Foreign key constraints
	Select All	Makes all the diagram elements selected
	Auto layout	Used to dispose all the items of the diagram in a standard manner
	Export as Image	Allows to export a diagram as *.png* , *.jpeg* or *.bmp*

When you open the context menu while clicking an item on the diagram, it quite differs from the one described before.

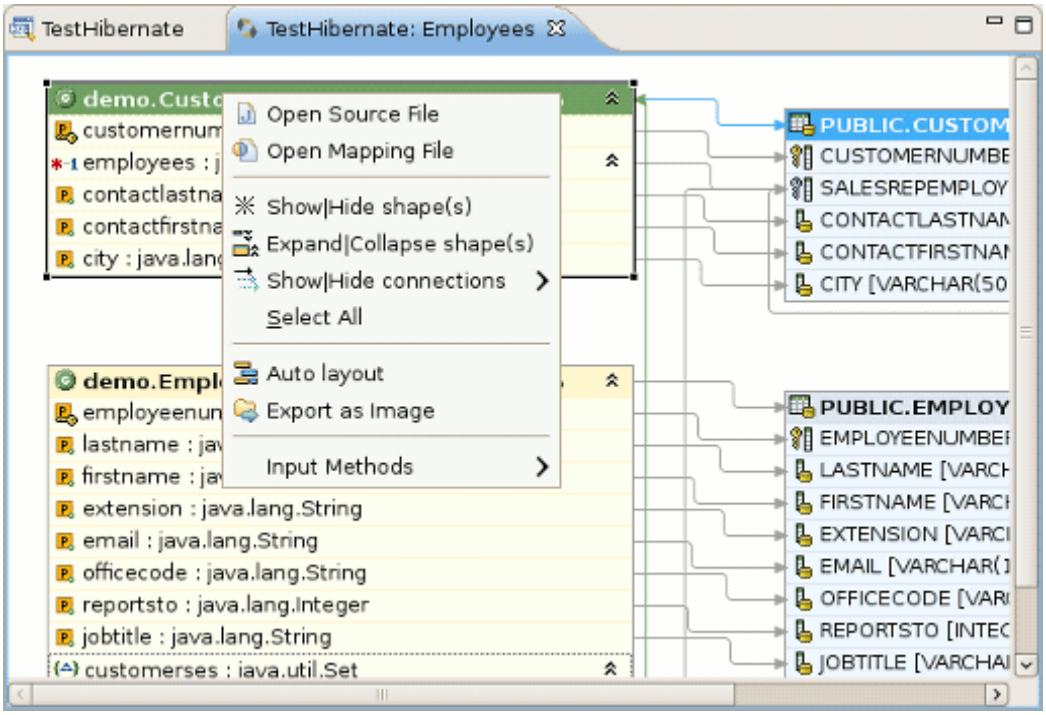

Figure 4.43. Context Menu in Mapping Item

The next table describes all the extra options in the menu of mapping items:

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

Icon	Command	Description
	Open Source File	Makes it possible to open a source file for a chosen object/element. The selected element will be highlighted in the open file.

Icon	Command	Description
	Open Mapping File	Makes it possible to open a mapping file for a chosen object/element. The selected element will be highlighted in the open file.
	Show|Hide shape(s)	Used to hide/show an item on the mapping diagram
	Expand|Collapse shape(s)	Used for expanding/collapsing fields of the item

Tip:

All the described types of the context menu are also available in the Outline view.

The below table lists the actions that could be performed using the keyboard keys (or keys combinations).

Table 4.11. Hibernate Mapping Diagram Shortcut Keys

Command	Binding
Scroll the diagram content	Ctrl + Shift + arrows
Collapse/Expand selected item(s)	Enter
Show/Hide selected item(s)	+
Sort items in alphabetical order or return the initial state	Space
Navigate between the items	Arrows

It's possible to save the diagram in the eclipse workspace. Click the usual *File > Save As* option, the wizard will ask you to set the location within you project where to save the file and give the name for the diagram. The item's names concatenated with the ampersand symbols are set as the default name for a diagram. The file is saved with the `.hibernate` extension.

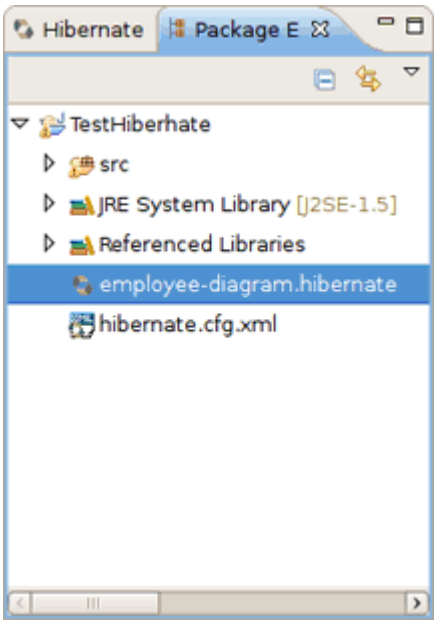

Figure 4.44. The Diagram saved in the Workspace

Note:

If you restart the 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 perform the refreshing.

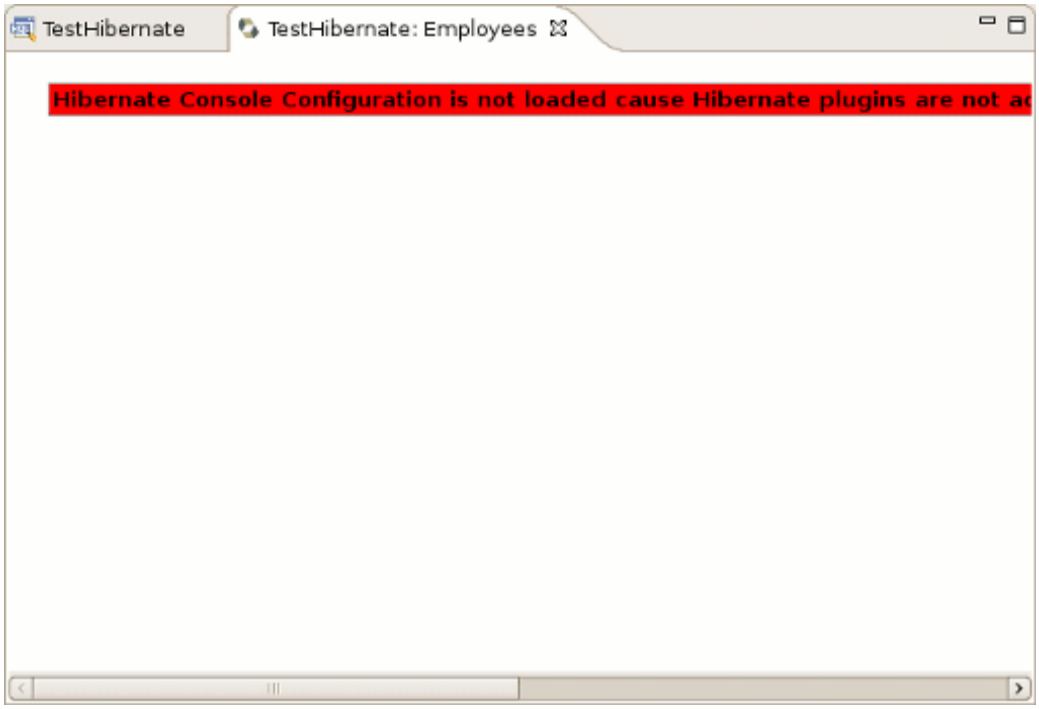

Figure 4.45. The Diagram after Restarting the Eclipse

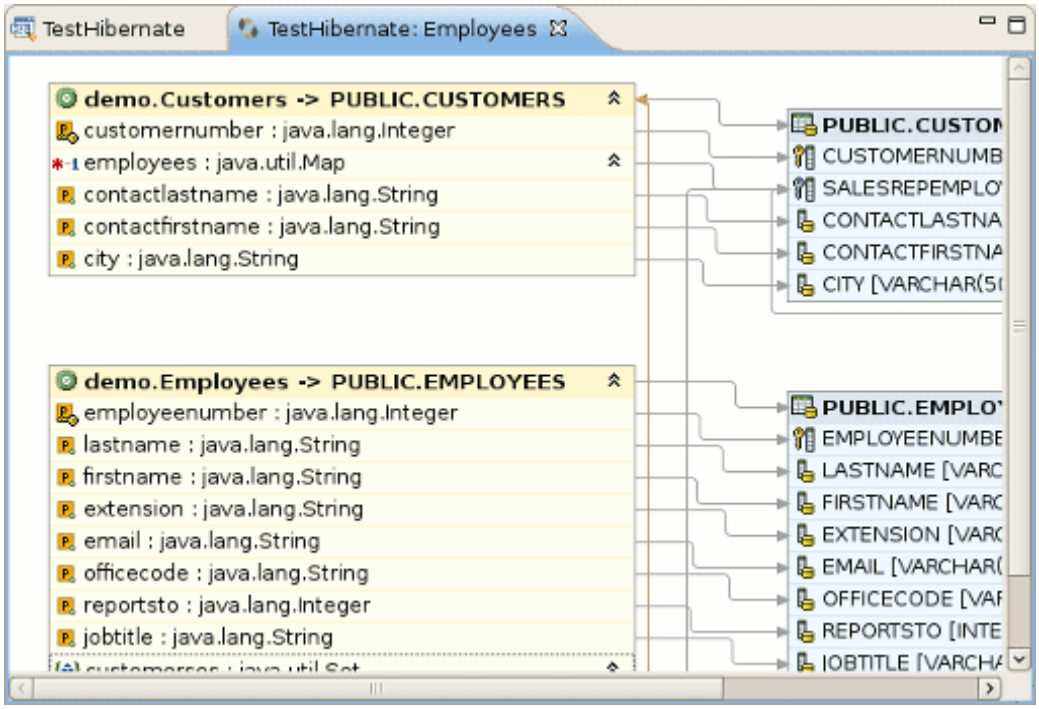

Figure 4.46. The Diagram after Refreshing

There are some useful commands in the toolbar.

Figure 4.47. The Diagram View Toolbar

They are described in the table below.

Table 4.12. Command in Diagram View Toolbar

Icon	Command	Description
	Refresh Visual Mapping	It update Mapping Diagram if Console Configuration was changed.
	Zoom Box	Used to define scale of the diagram. Also it's used for Mapping Diagram printing. If you want to put the whole diagram to one print page, you need select Page option in the Zoom Box.
	Auto layout	Used to arrange all diagram items in a standard manner.
	Show|Hide connections	Used to show or hide connection on the diagram. Moreover you can choose what type of connections must be present on the diagram (Property Mappings, Class Mappings, Associations or Foreign key constraints) .
	Expand|Collapse	Used for expanding/collapsing fields of the item.
	Show|Hide shape(s)	Used to hide/show an item on the mapping diagram.

4.10.2. Prototyping Queries

Queries can be prototyped by entering them into the HQL or Criteria Editor. To execute a query you should 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 project Console Configuration and select HQL Editor (or Hibernate Criteria Editor).

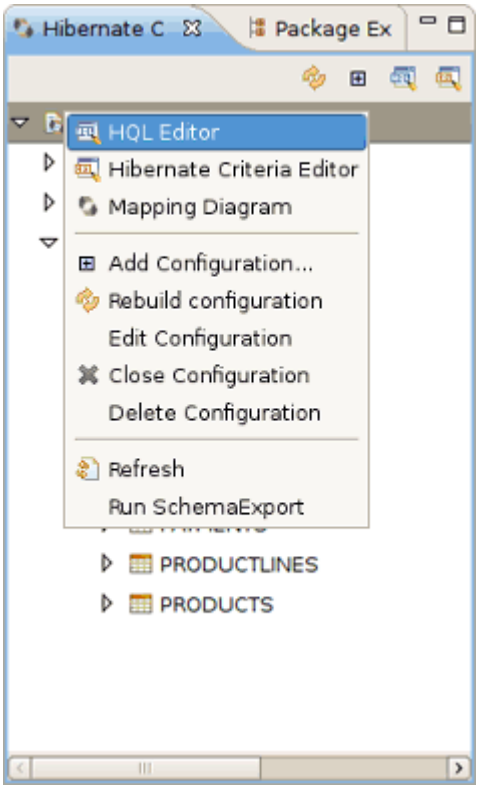

Figure 4.48. Opening HQL Editor

Tip:

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

When open the editors they should automatically detect the chosen Console Configuration.

To get a prefill 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 choose *Hibernate Criteria Editor* in the context menu, it will open Hibernate Criteria Editor with the associated criteria.

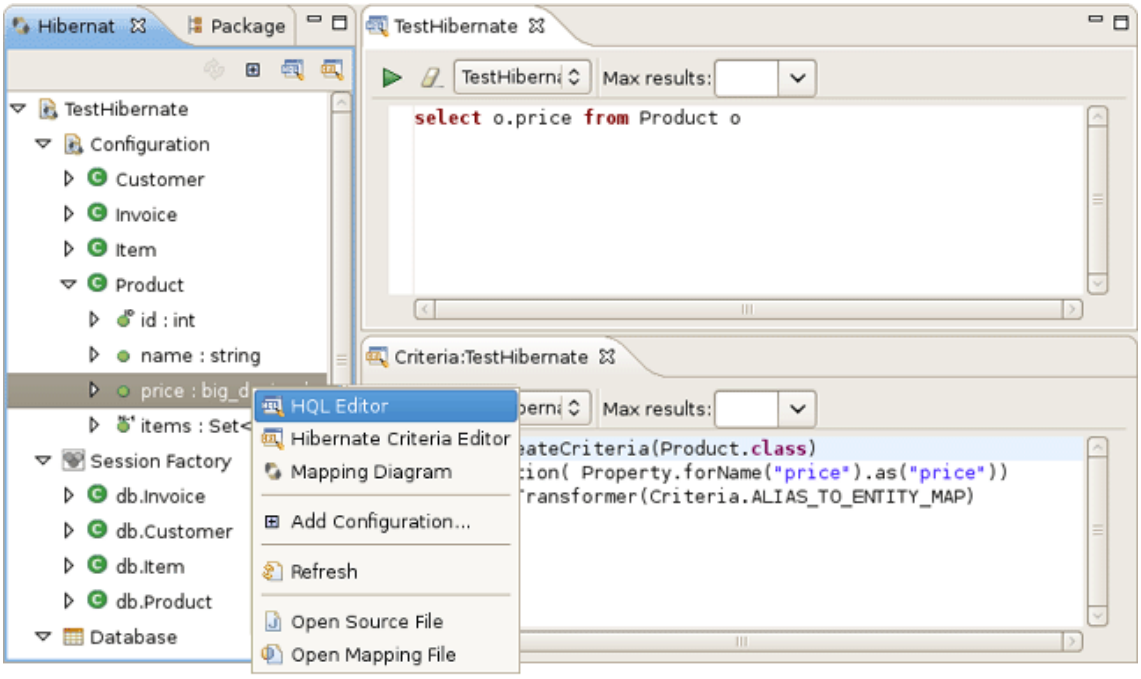

Figure 4.49. Generating Simple Queries

It's also possible to copy a portion of code from *.java* file into the HQL or Criteria editor. To do this make use of the Quick Fix option (*Ctrl + 1*).

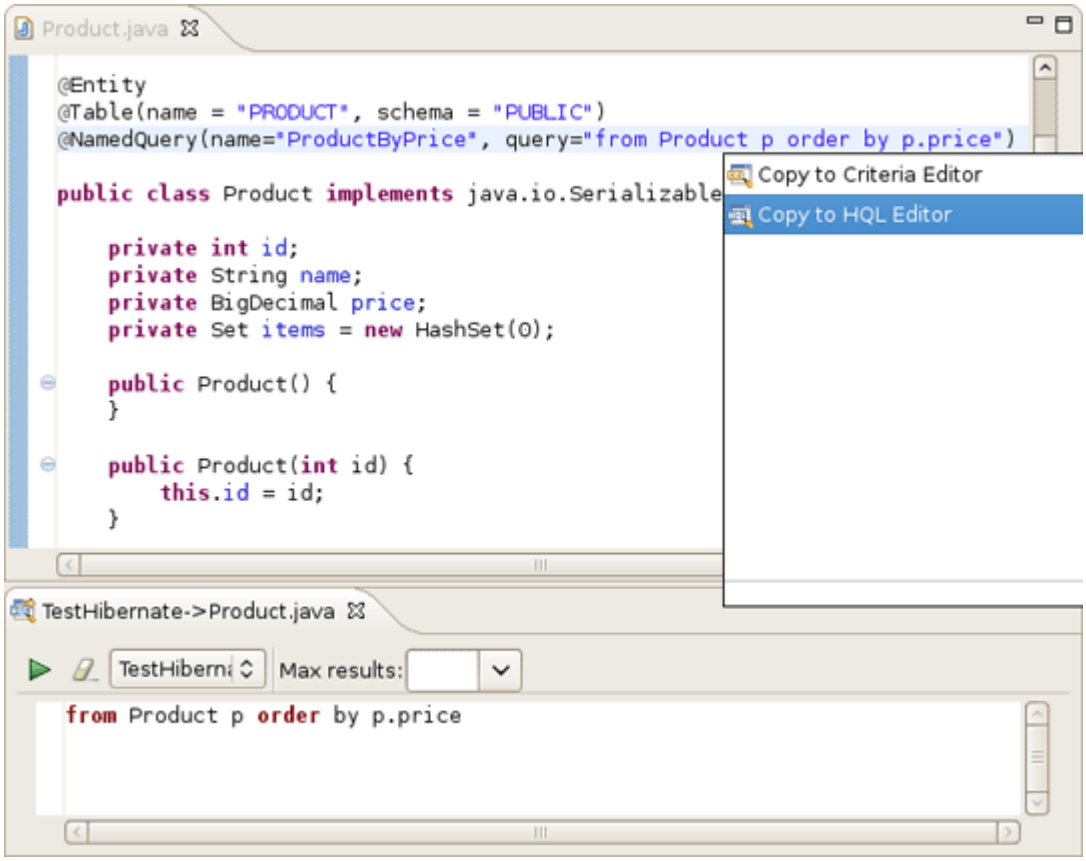

Figure 4.50. Quick Fix Option Demonstration

You can also update the original java code according to changes in the HQL or Criteria editor. For that you should save your HQL/Criteria query and submit the replacing in appeared confirmation dialog.

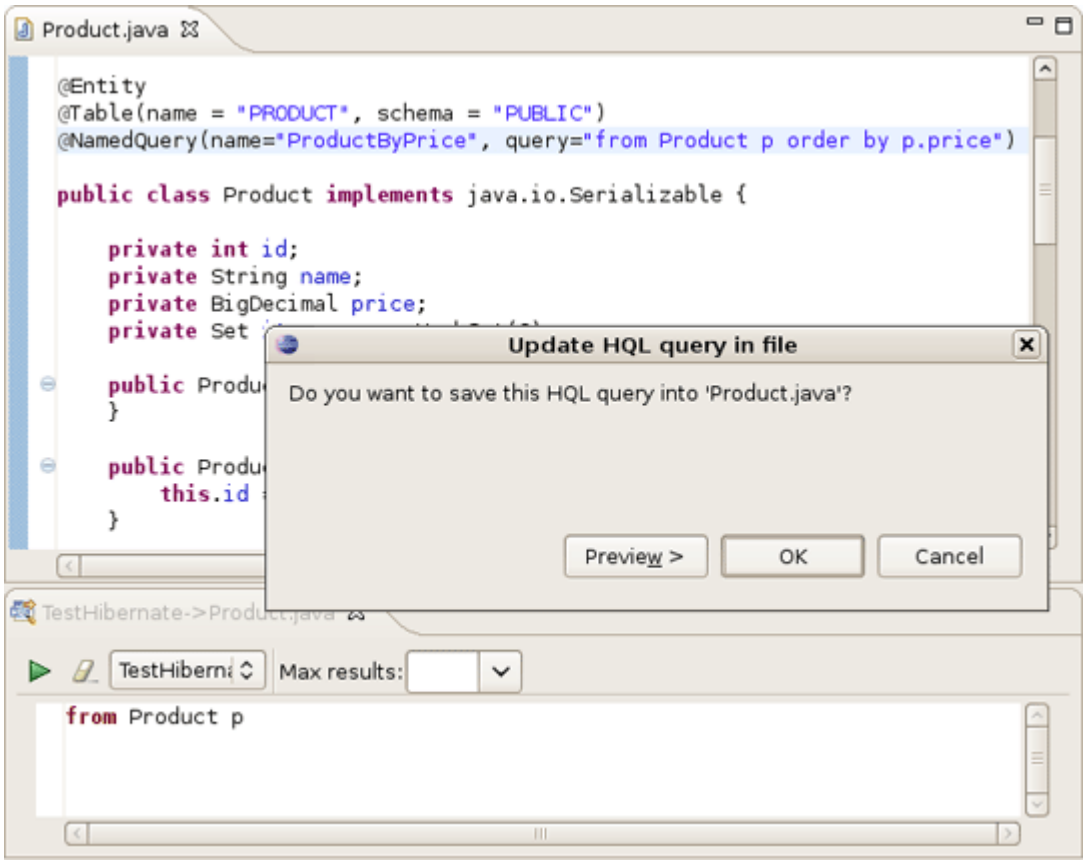

Figure 4.51. Updating Java Code

Also you can pin HQL editor and Criteria editor for one tab in Hibernate Query Result view. For that you need click on Stick result to one tab button(📌
). In the issue query executions results will be shown in one tab (no more will be opened).

Moreover you are able to rename tab in Hibernate Query Result. Click the tab, and type a new name in Property View->Tab name field.

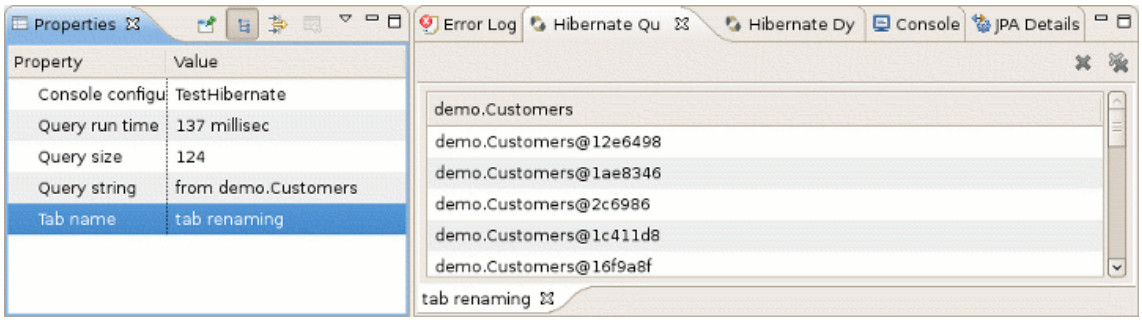

Figure 4.52. Tab Renaming

4.10.2.2. Error Handling

Errors during creation of the Session Factory or running 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 `list()` thus without any limit of the size of the output the query could return a large result set. You might run out of memory. To avoid this you can put a value in the Max results field to reduce the number of elements returned.

4.10.2.3. Dynamic Query Translator

If the Hibernate Dynamic Query Translator View is visible while writing in the HQL Editor it will show the generated SQL for a HQL query.

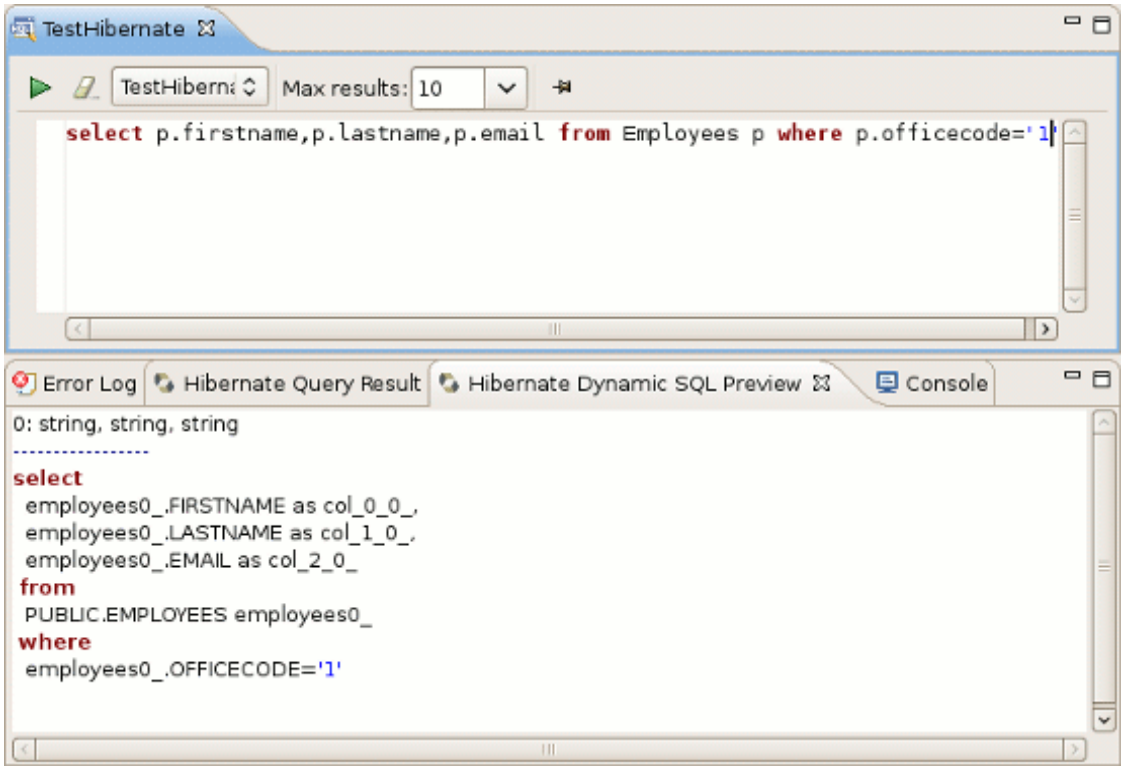

Figure 4.53. Hibernate Dynamic Query Translator View

The translation is done each time you stop typing into the editor, if there are errors in the HQL the parse exception will be shown embedded in the view.

4.10.3. Properties View

As you can see on the figure,when clicking on class/entity Properties view shows the number of query results as well as the time of executing.

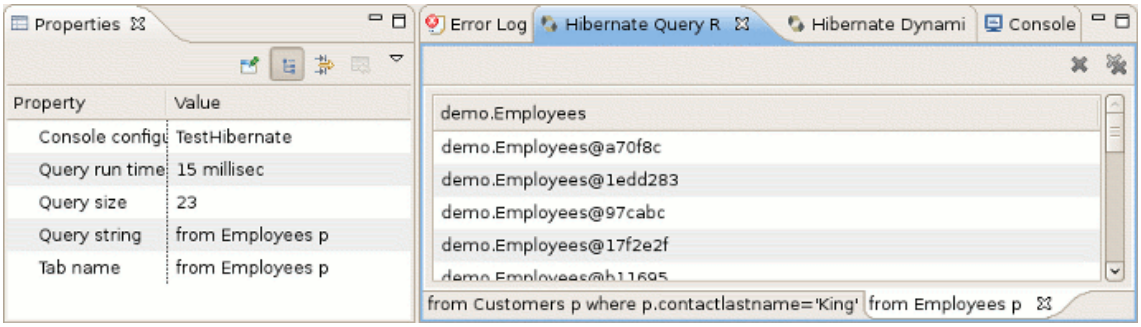

Figure 4.54. Properties View

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

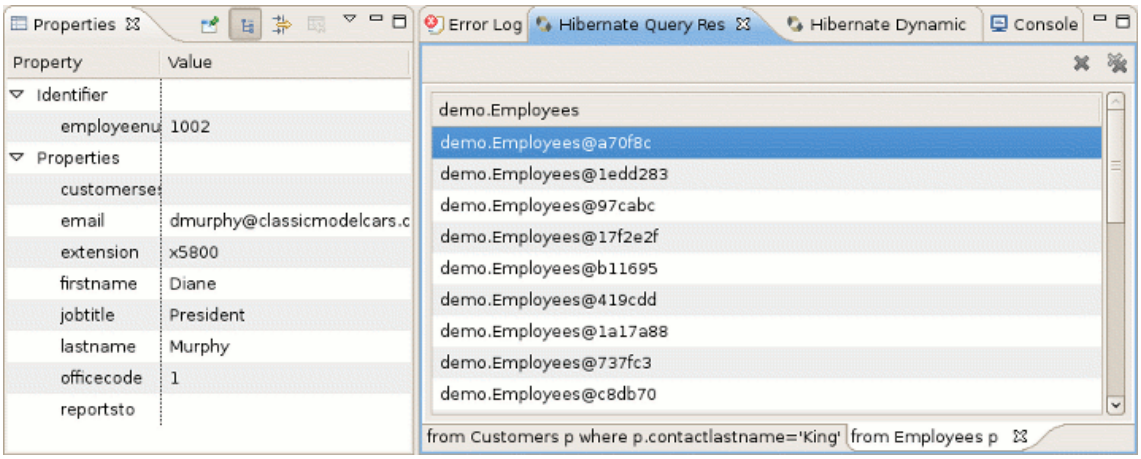

Figure 4.55. Properties View for Selected Object

You can also use Properties view when clicking on the configuration itself in Hibernate Configuration View(*Modifying a Hibernate Console Configuration section*).

4.11. Hibernate:add JPA annotations refactoring

Using this wizard you can add the next Hibernate annotations to the 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 doesn't present.
- *@ManyToOne* , *@OneToMany* , *@OneToOne* , *@ManyToMany* - this annotations are declared according to the classes hierarchy.
- *@MappedSuperclass* is added to abstract superclasses.
- *@Id* , *@GeneratedValue* are added automatically only to the properties under the name "Id",where they don't present.
- *@Version* is declared in case you select *Enable optimistic locking [58]*.

Note:

This section doesn't cover the meaning of the Hibernate annotations , for more information read *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 enrich with annotations *>Source>Generate Hibernate/JPA annotations* . You will see the Hibernate:add JPA annotations dialog.

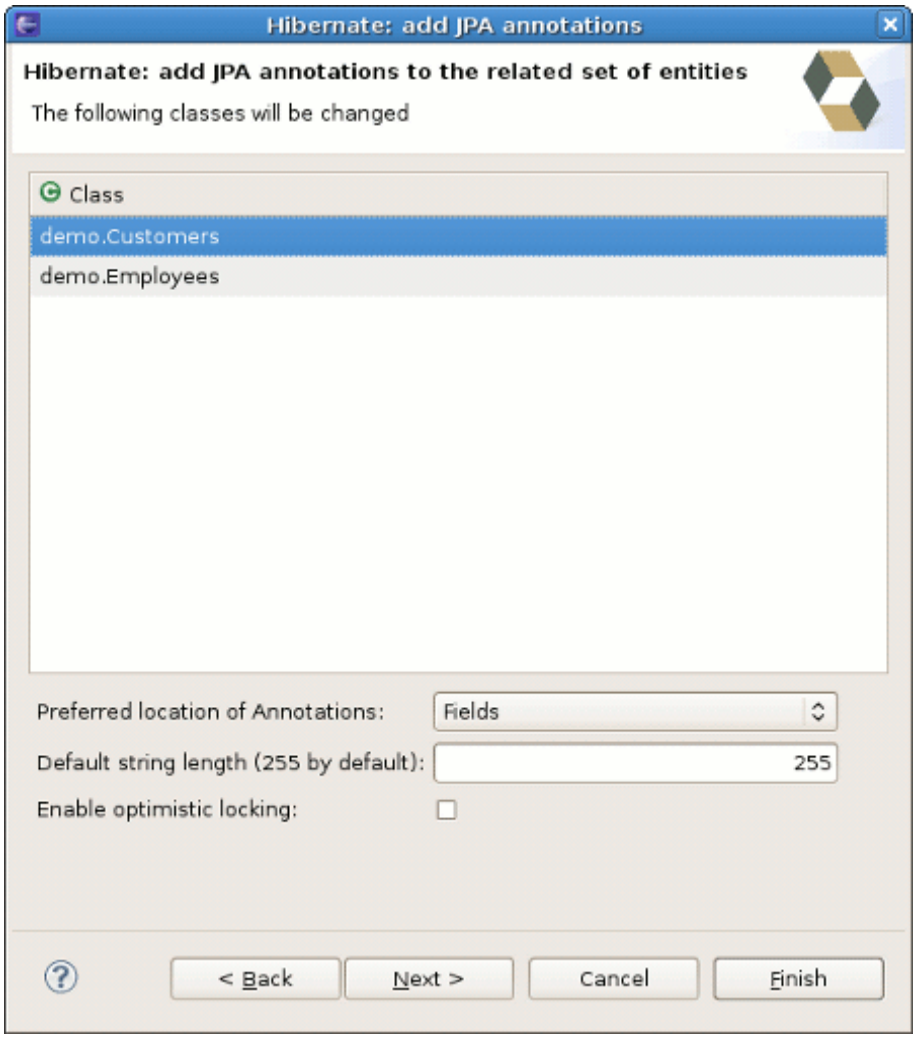

Figure 4.56. Starting Hibernate:add JPA annotations dialog

In the top of it you can see the list of all classes that will be passed through refactoring. Besides the class you have selected in this list you can also find its superclasses and the classes that objects present in the current class as properties. If you want to add new classes or package to the list of classes, you should click the Back button. In result you will see Add classes and packages page.

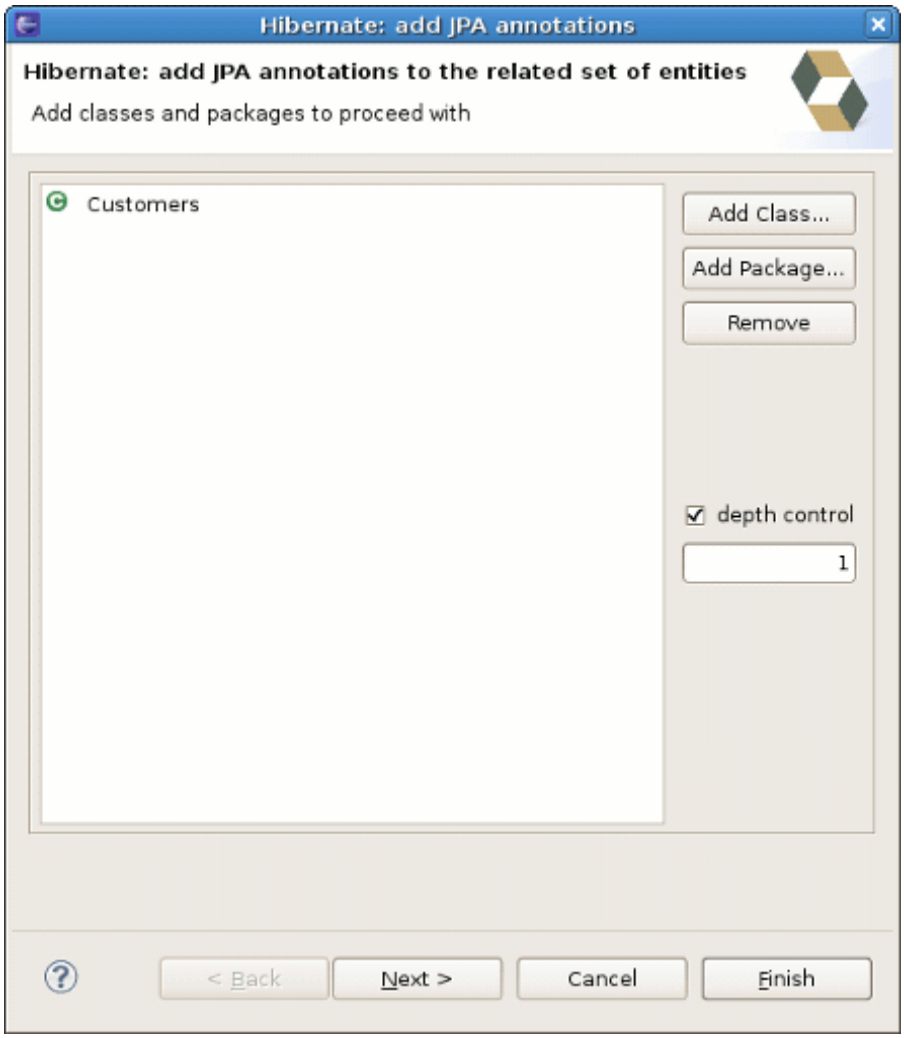

Figure 4.57. Add classes and packages page

Here you can add one more classes or whole package, moreover you can limit dependencies depth by selecting depth control option (more about this option you will find in *Creating a Hibernate Mapping File [8]*). When finished just press the Next button and you will be returned to The following classes will be changed page and will be able to continue work with it.

By default the tags are added to the fields of selected classes. But you can change this option to *Getters* in *Preferred location of Annotations* dropdownlist and then all the annotations will be added to the getter methods. If you choose *Auto select from class preference* then the annotations are added according to the majority of the already existed ones positions.

If it's nessecary to map your String properties to the columns that length differ from the default value(255) ,change *Default string length field* and `@Column(length = your length)` will be created before every String property.

You can add optimistic locking capability to an entity bean by selecting *Enable optimistic locking* checkbox. This operation will add version property to all the selected classes. The property will be also annotated with @Version ,getter and setter will be created. If the property is already exists,it

won't be created ,but the getters,setters will be generated. If there is already @MappedSuperclass with version in the base class of the current class - *"version"* is not inserted into the current class.

After defining all necessary settings in the current step press *Next* and follow the next wizard steps.

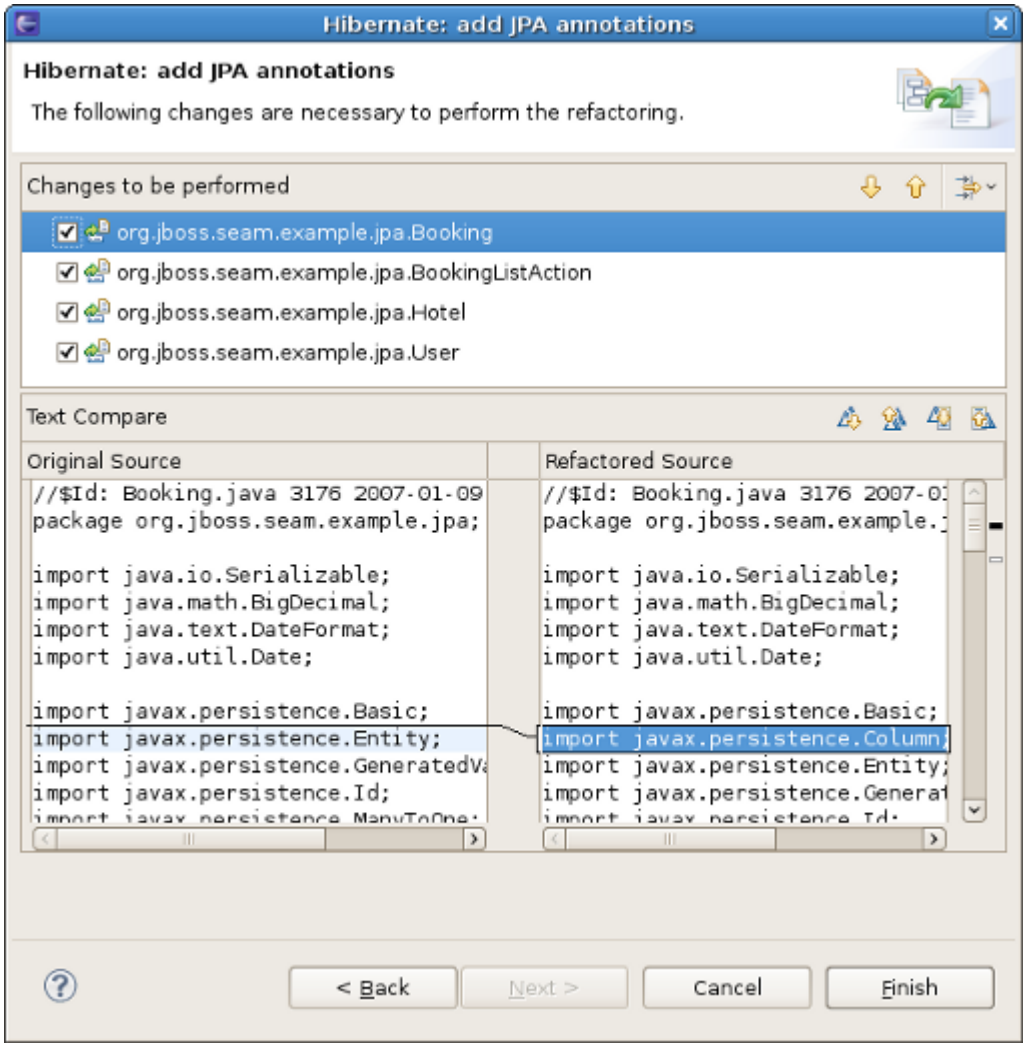

Figure 4.58. Hibernate:add JPA annotations view

The view represents two windows: one with the source code and and the second with refactored one. With the help of

,

,

,

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.

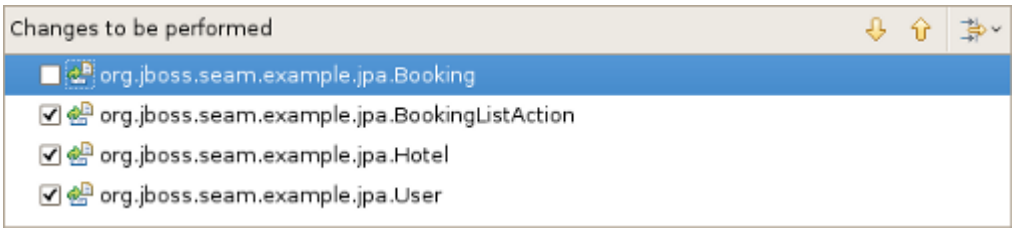

Figure 4.59. List of classes that need refactoring

To apply the changes click *Finish* .

4.12. Enable debug logging in the plugins

It is possible to configure the eclipse plugin to route all logging made by the plugins and hibernate code it self to the Error Log View in Eclipse.

Error Log View is very useful tool to solve any problem which appears in Hibernate Tools Plugins. You can use if there are troubles with setting up Hibernate Console Configuration.

This is done by editing the *hibernate-log4j.properties* in *org.hibernate.eclipse/ directory/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 *Hibernate Documentation* [http://www.hibernate.org/5.html] for interesting categories and Log4j documentation.

4.12.1. Relevant Resources Links

Find more on how to configure logging via a log4j property file in *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 3.0.0 Alpha1 version of JBoss Tools Hibernate plugins support Eclipse Dali integration what now 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 from *New > Other > JPA > JPA Project* (or simply *New > JPA Project* in JPA Perspective), the first wizard page looks as follows.

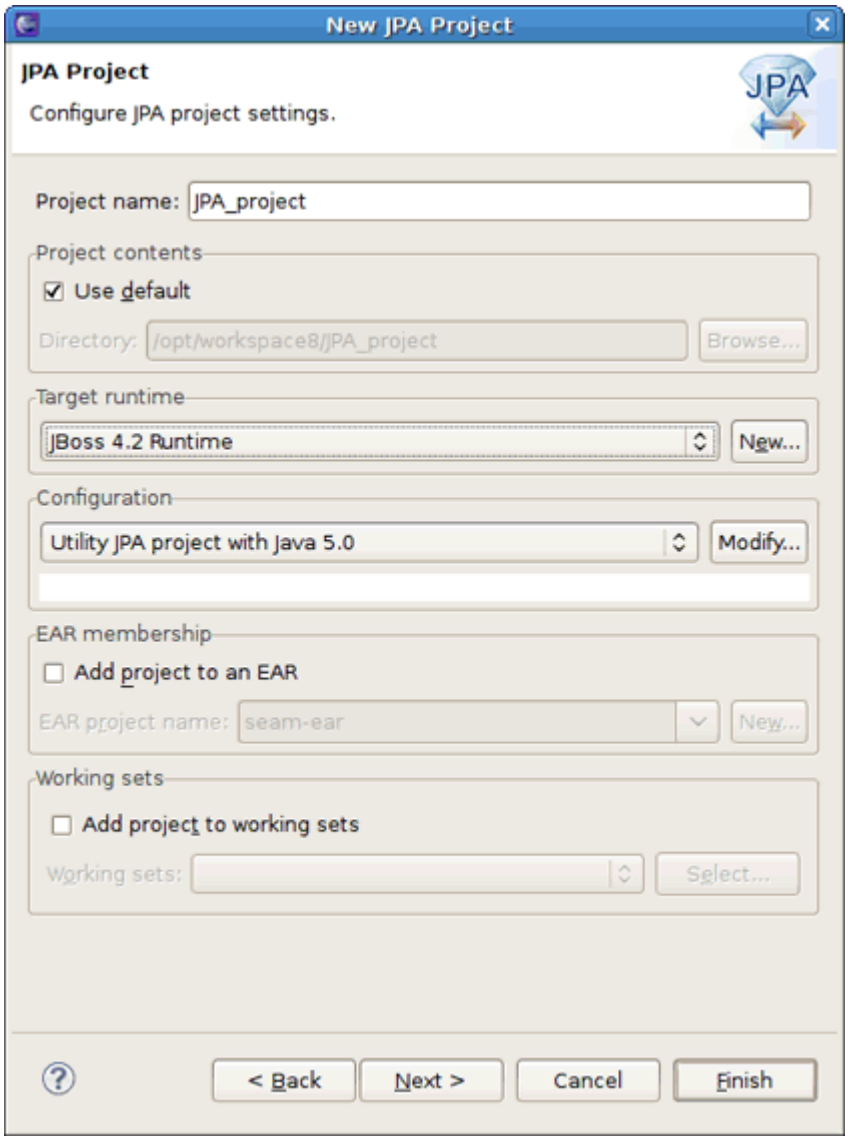

Figure 4.60. Starting JPA Project

It's possible here to 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.

Hitting *Finish* will generate the project.

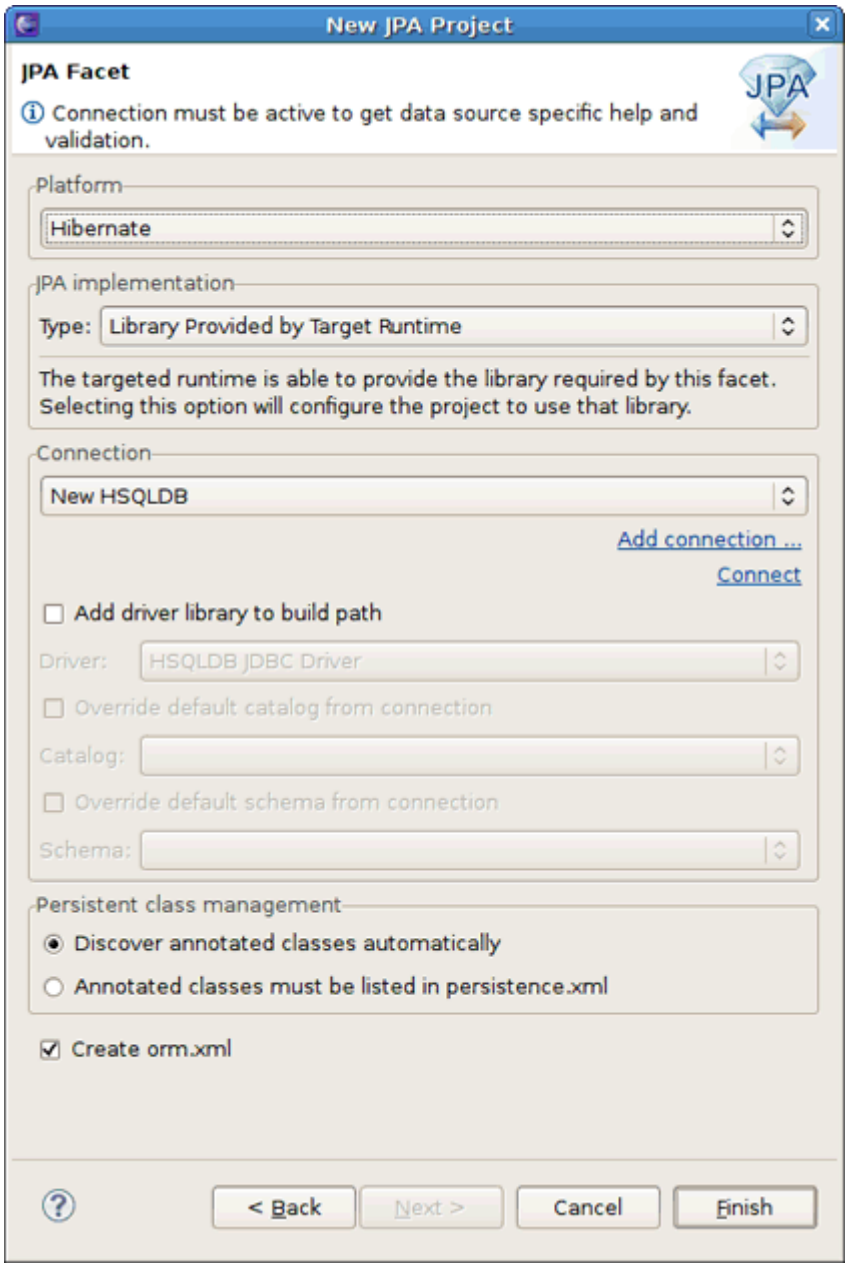

Figure 4.61. Targeting at Hibernate Platform

Note:

Please note, if you choose Hibernate as a platform while creating a JPA/Dali project, a Hibernate Console Configuration for the project is created automatically when the wizard finishes its work. It allows a full usage of Hibernate Tools features without additional setup.

4.13.2. Generating DDL and Entities

By enabling Hibernate platform specific features you can now generate DDL and Entities. For that find *JPA Tools > Generate Tables from Entities/Generate Entities from Tables* options in the context menu of your JPA project.

Figure 4.62. Generate DDL/Entities

Note:

Please, remember to put the proper database driver to the classpath of your project.

The Generate Entities wizard first will ask you to choose the directory where all output will be written.

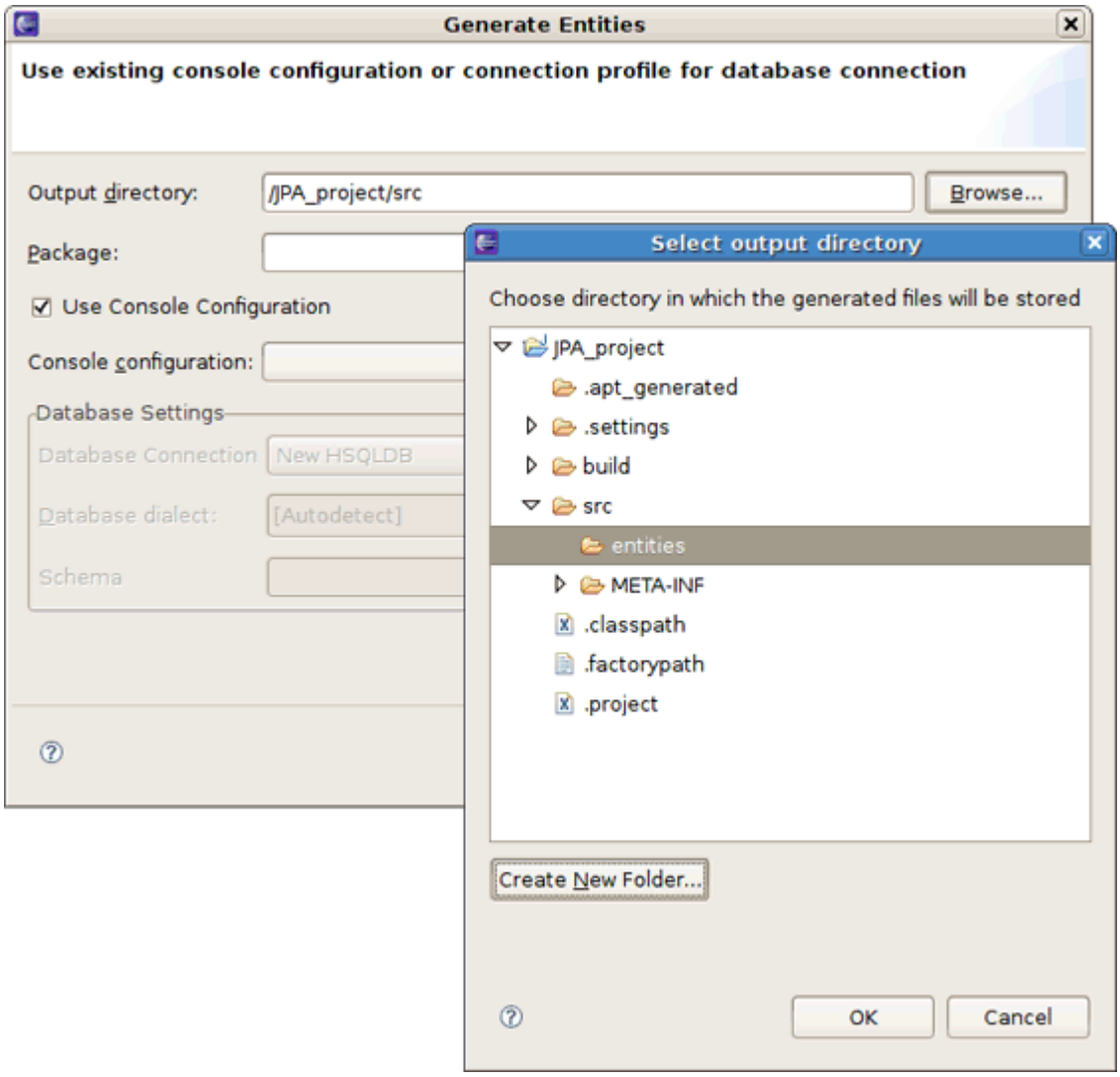

Figure 4.63. Generate Entities Wizard

To generate entities you can use:

- a Hibernate Console Configuration (proposed by default)

  Just make sure that the *Use Console Configuration* checkbox is selected and choose a needed configuration from the *Console configurations* list box.

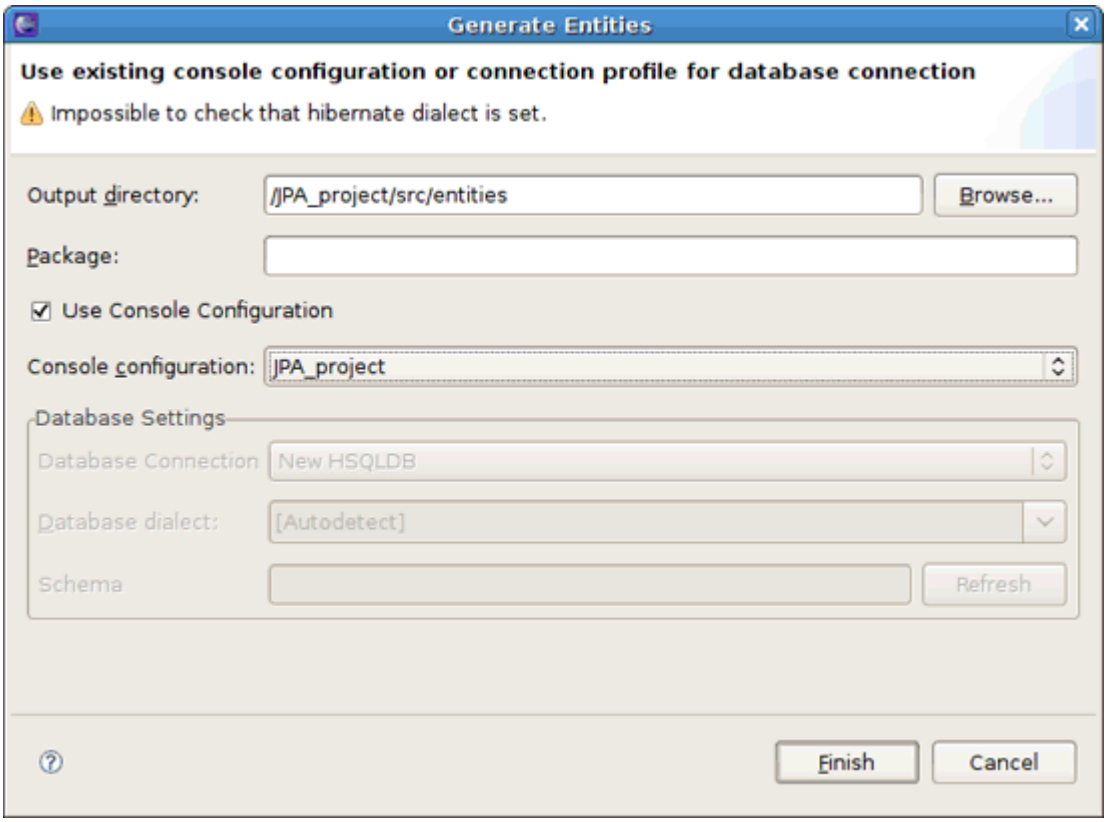

Figure 4.64. Generate Entities Wizard

- or a DTP connection directly

  Just uncheck *Use Console Configuration* and adjust database settings.

All the same you do with Generate Entities Wizard you can do with Generate DDL wizard. Special feature for Generate DDL wizard is possible automatic execution of Generation DDL in the database.

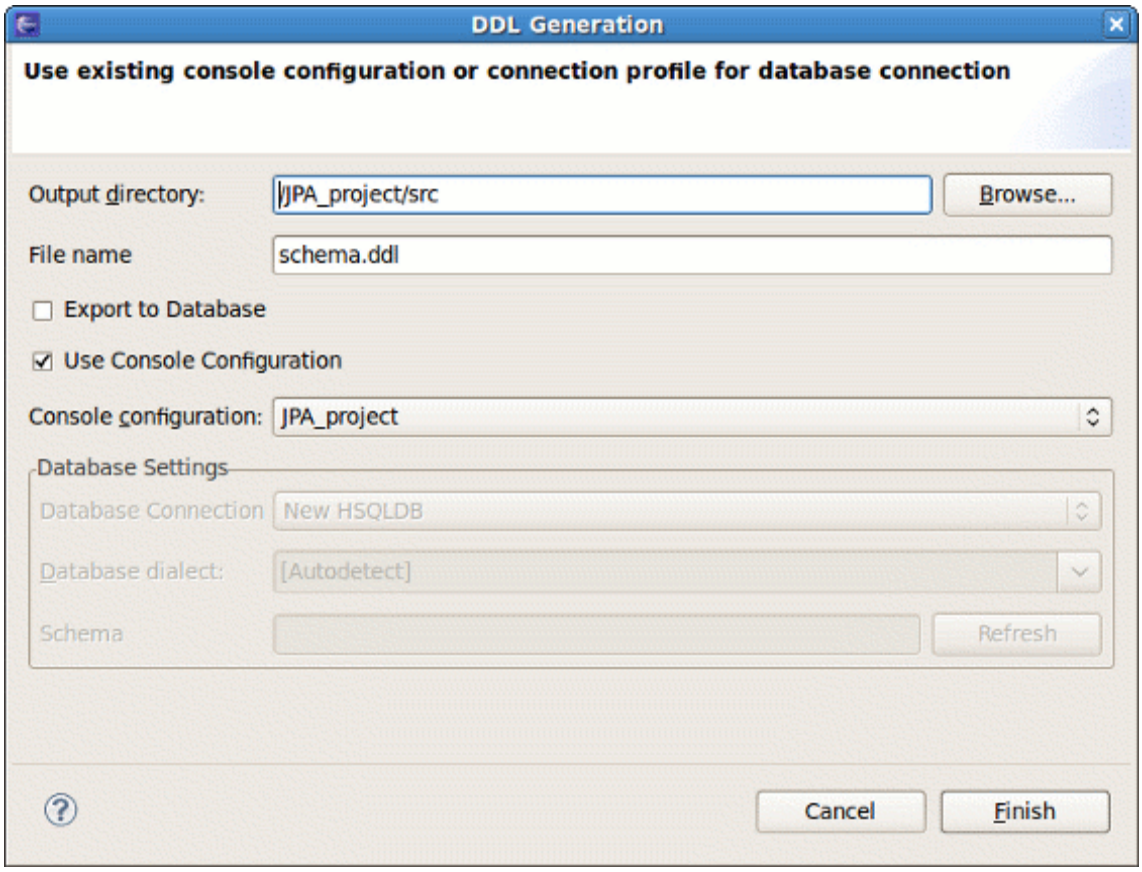

Figure 4.65. Generate DDL Wizard

Thus, you can now have the Hibernate runtime support in Eclipse JPA projects.

4.13.3. Hibernate Annotations Support

Also Hibernate Annotations are supported in Dali Java Persistence Tools. The next annotations are integrated with JPA Details view:

- Id Generator annotations - @GenericGenerator and @GeneratedValue

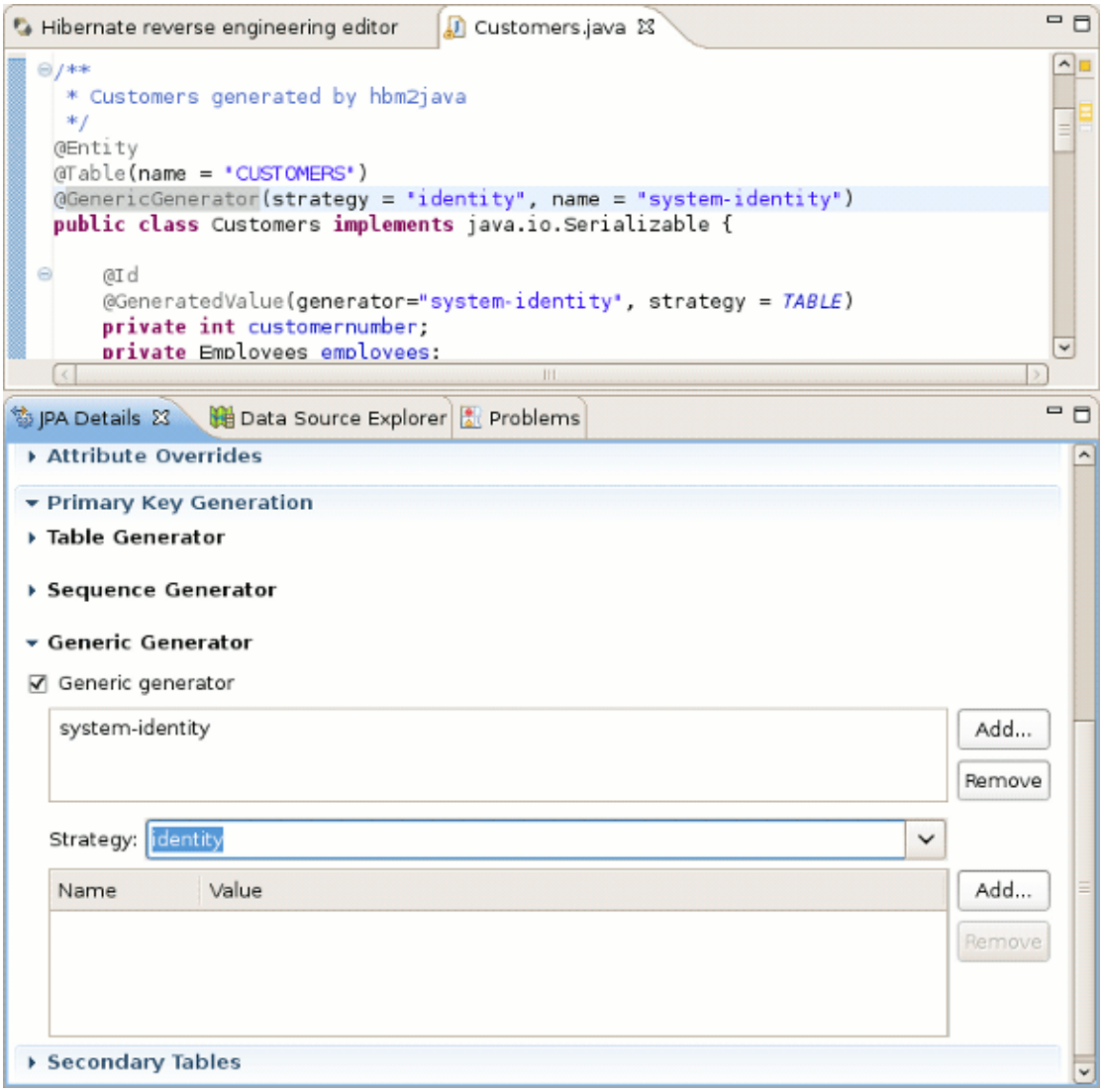

Figure 4.66. @GenericGenerator support in Dali

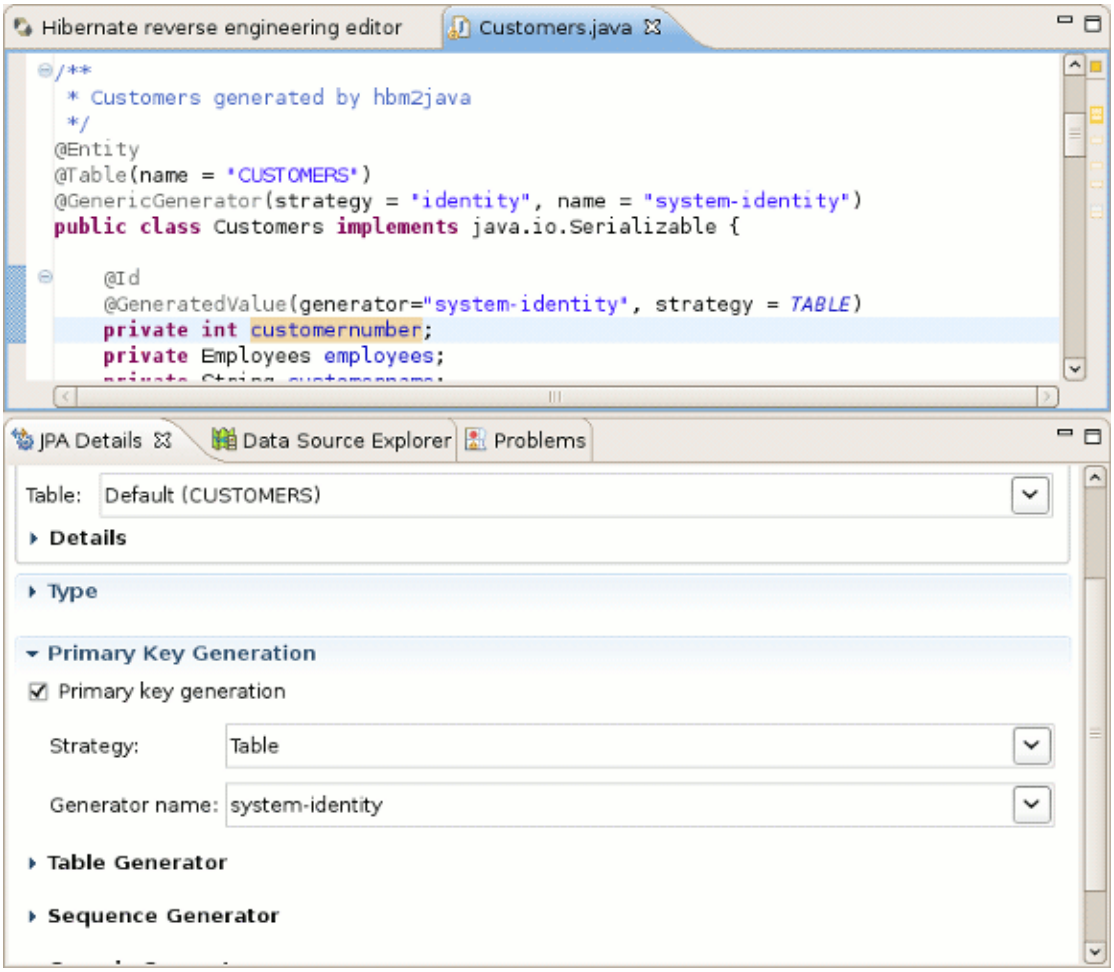

Figure 4.67. @GeneratedValue support in Dali

- Property annotations- @DiscriminatorFormula, @Generated Annotation, @Index annotation

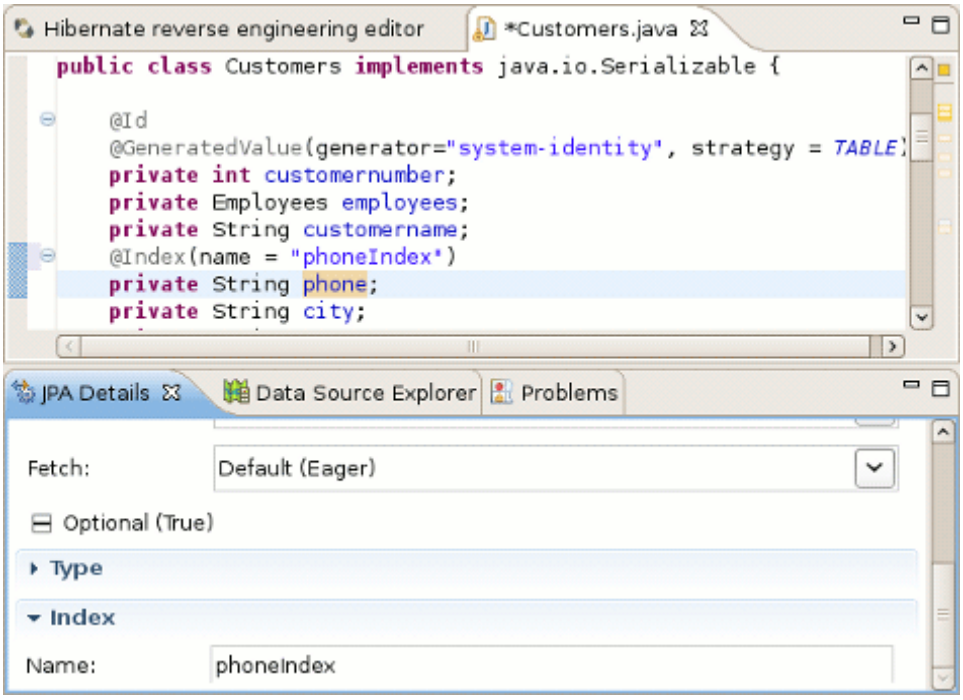

Figure 4.68. @Index support in Dali

- Mapping Queries annotations - @NamedQuery and @NamedNativeQuery

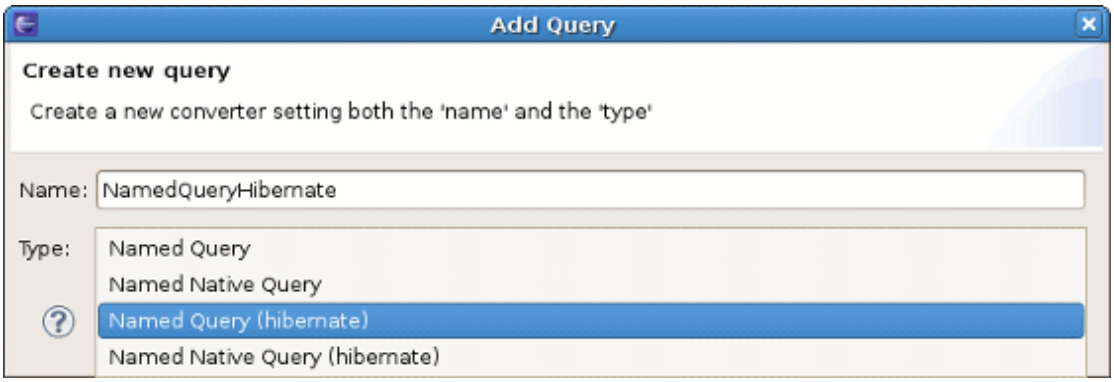

Figure 4.69. Add New Named Query Dialog with Hibernate Support

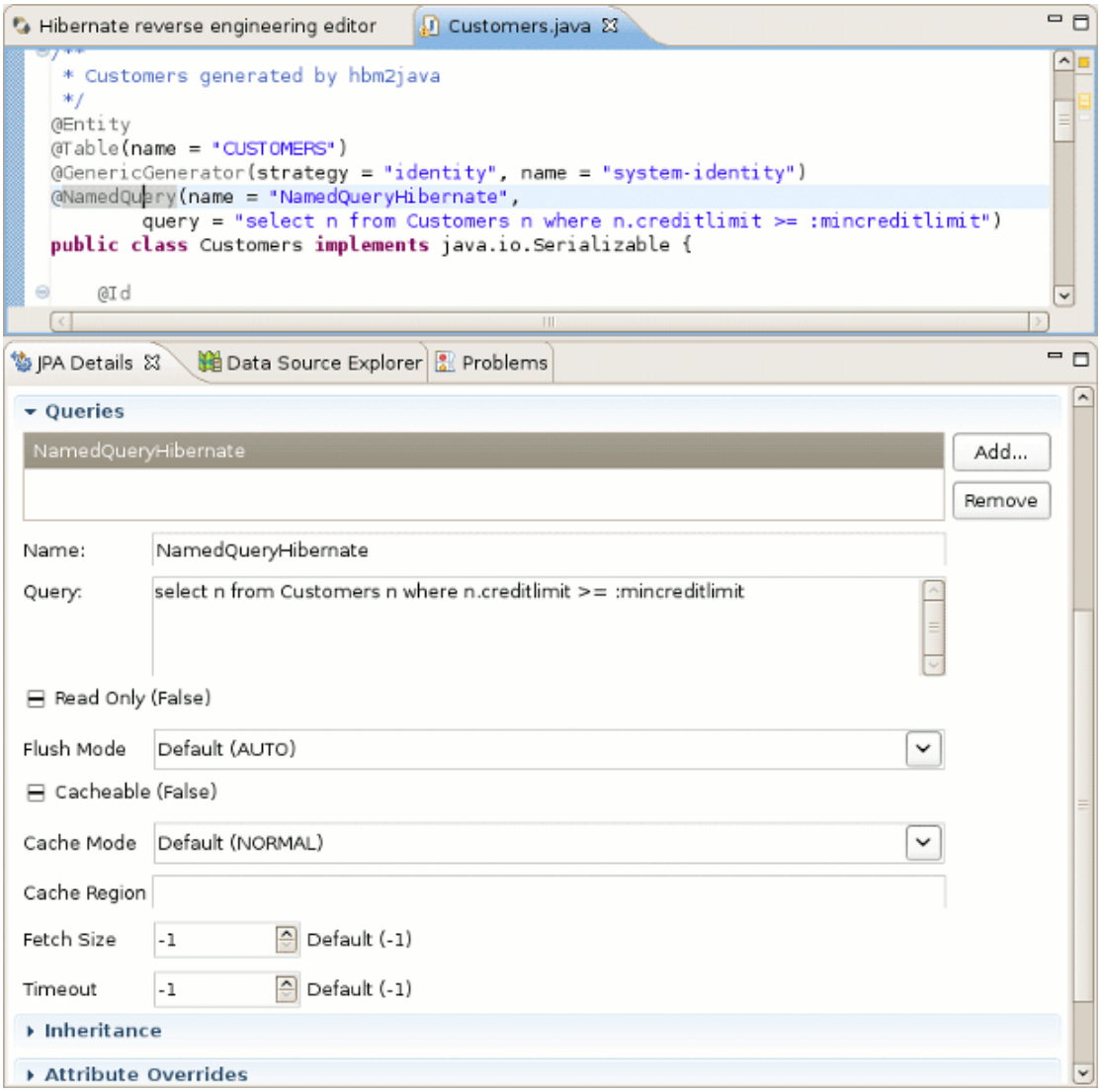

Figure 4.70. @NamedQuery support in Dali

- Association annotations in an embeddable object (@OneToOne, @ManyToOne, @OneToMany or @ManyToMany)

Figure 4.71. Hibernate Support for Embeddable Object

More information about Hibernate Annotation you can find in *Hibernate Annotations Reference Guide* [http://docs.jboss.org/hibernate/stable/annotations/reference/en/html/].

4.13.4. Relevant Resources Links

There is full information about native Dali plugin features on *Eclipse Documentation page* [http://help.eclipse.org/galileo/index.jsp?nav=/8].

Ant Tools

Maybe somebody will find it more preferable to use Ant for generation purposes. Thus, this chapter is intended to get you ready to start using Hibernate Tools via Ant tasks.

5.1. Introduction

The *hibernate-tools.jar* contains the core for the Hibernate Tools. It is used as the basis for both the Ant tasks described in this document and the eclipse plugins both available from tools.hibernate.org. The *hibernate-tools.jar* 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 might be incompatibilities with respect to the Hibernate3.jar bundled with the tools and your own jar. Thus to avoid any confusion it is recommended to use the hibernate3.jar and hibernate-annotations.jar bundled with the tools when you want to use the Ant tasks. Do not worry about using e.g. Hibernate 3.2 jar's with e.g. a Hibernate 3.1 project since the output generated 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* by inserting the following xml (assuming the jars are in the `lib` directory):

```
<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" />
```

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://annotations.hibernate.org].

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

```
<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

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

5.2.1. Basic examples

The following example shows the most basic setup for generating pojo's via `<hbm2java>` from a normal . The output will be put in the *${build.dir}/generated* directory.

```
<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 `<hbm2dll>`, generates some DAO code via `<hbm2dao>` and finally runs a custom code generation via `<hbmtemplate>`. This is again from a normal *hibernate.cfg.xml* and the output is still put in the directory. Furthermore the example also shows where a classpath is specified when you e.g. have custom usertypes or some mappings that is needed to be looked up as a classpath resource.

```
<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>`) for use when reverse engineering.

Each have in common that they are able to build up a Hibernate Configuration object from which a set of exporters can be run to generate various output.

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, plus lists the individual settings they have.

5.3.1. Standard Hibernate Configuration (<configuration>)

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

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

  </configuration>
```

Table 5.2. Configuration attributes

Attribute name	Definition	Attribute use
configurationfile	The name of a Hibernate configuration file, e.g. "hibernate.cfg.xml"	Optional
propertyfile	The name of a property file, e.g. "hibernate.properties"	Optional
entity-resolver	Name of a class that implements org.xml.sax.EntityResolver. Used if the mapping files require custom entity resolver	Optional
namingstrategy	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 naming strategy is supported for default Name/Columns mapping	Optional
fileset	A standard Ant fileset. Used to include hibernate mapping files. Remember that if mappings are already specified in	

Attribute name	Definition	Attribute use
	the hibernate.cfg.xml then it should not be included via the fileset as it will result in duplicate import exceptions.	

5.3.1.1. Example

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

Note:

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

```
<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>

 <!-- list exporters here -->

</hibernatetool>
```

5.3.2. Annotation based Configuration (<annotationconfiguration>)

An `<annotationconfiguration>` is used when you want to read the metamodel from EJB3/ Hibernate Annotations based POJO's.

Important:

To use it remember to put the jar files needed for using hibernate annotations in the classpath of the `<taskdef>`, i. e. hibernate-annotations.jar and hibernate-commons-annotations.jar.

The <annotationconfiguration> supports the same attributes as a <configuration> except that the configurationfile attribute is now required as that is from where an *AnnotationConfiguration* gets the list of classes/packages it should load.

Thus the minimal usage is:

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

 <!-- list exporters here -->

</hibernatetool>
```

5.3.3. JPA based configuration (<jpaconfiguration>)

A <jpaconfiguration> is used when you want to read the metamodel from JPA/Hibernate Annotation where you want to use the auto-scan configuration as defined in the JPA spec (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* packaged in a JPA compliant manner.

The <jpaconfiguration> will simply just try and auto-configure it self based on the available classpath, e.g. look for *META-INF/persistence.xml*.

The *persistenceunit* attribute can be used to select a specific persistence unit. If no *persistenceunit* is specified it will automatically search for one and if a unique one is found, use it, but if multiple persistence units are available it will error.

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

```
<path id="ejb3toolslib">
 <path refid="jpatoolslib"/> <!-- ref to previously defined toolslib -->
 <path location="lib/hibernate-annotations.jar" />
 <path location="lib/ejb3-persistence.jar" />
 <path location="lib/hibernate-entitymanager.jar" />
 <path location="lib/jboss-archive-browsing.jar" />
 <path location="lib/javaassist.jar" />
</path>

<taskdef name="hibernatetool"
        classname="org.hibernate.tool.ant.HibernateToolTask"
        classpathref="jpatoolslib" />
```

```
<hibernatetool destdir="${build.dir}">
 <jpaconfiguration persistenceunit="caveatemptor"/>
 <classpath>
  <!-- it is in this classpath you put your classes dir,
   and/or jpa persistence compliant jar -->
  <path location="${build.dir}/jpa/classes" />
 </classpath>

 <!-- list exporters here -->

</hibernatetool>
```

Note:

ejb3configuration was the name used in previous versions. It still works but will emit a warning telling you to use `jpaconfiguration` instead.

5.3.4. JDBC Configuration for reverse engineering (<jdbcconfiguration>)

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

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

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

```
<jdbcconfiguration
  ...
  packagename="package.name"
  revengfile="hibernate.reveng.xml"
  reversestrategy="ReverseEngineeringStrategy classname"
  detectmanytomany="true|false"
  detectoptmisticlock="true|false"
>
  ...
  </jdbcconfiguration>
```

Table 5.3. Jdbcconfiguration attributes

Attribute name	Definition	Attribute use
packagename	The default package name to use when mappings for classes are created	Optional
revengfile	The name of a property file, e.g. "hibernate.properties"	Optional
reversestrategy	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/exclude etc. Using a class instead of (or as addition to) a reveng.xml file gives you full programmatic control of the reverse engineering.	Optional
detectManytoMany	If 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.	Default: true
detectOptimisticLock	If true, columns named VERSION or TIMESTAMP with appropriate types will be mapped with the appropriate optimistic locking corresponding to `<version>` or `<timestamp>`.	Default: true

5.3.4.1. Example

Here is an example of using `<jdbcconfiguration>` to generate Hibernate xml mappings via `<hbm2hbmxml>`. The connection settings here is read from a *hibernate.properties* file but could just as well have been read from a *hibernate.cfg.xml*.

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

5.4. Exporters

Exporters are the parts that 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 for userdefined exporters, that 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 placed on the configuration element.

```
<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

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

5.4.1.1. Example

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 codegenerator. 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

Attribute name	Definition	Default value
jdk	Code will contain JDK 5 constructs such as generics and static imports	False
ejb3	Code will contain EJB 3 features, e.g. using annotations from javax.persistence and org.hibernate.annotations	False

5.4.2.1. Example

Basic example of 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. 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 *hbm.xml*.

Note:

Not every possible mapping transformation is possible/implemented (contributions welcome) so some hand editing might be necessary.

```
<hbm2hbmxml/>
```

5.4.3.1. Example

Basic usage of `<hbm2hbmxml>`.

```
<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.

```
<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*. 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 used and adds mapping entries for each mapped class.

```
<hbm2cfgxml
  ejb3="true|false"
/>
```

Table 5.6. Hbm2cfgxml exporter attribute

Attribute name	Definition	Default value
ejb3	The generated cfg.xml will have <mapping class=".."/>, opposed to <mapping resource="..."/> for each mapping.	False

5.4.5. Documentation exporter (<hbm2doc>)

`<hbm2doc>` generates html documentation a'la javadoc for the database schema et.al.

```
<hbm2doc/>
```

5.4.6. Query exporter (<query>)

`<query>` is used to execute a HQL query statements and optionally sends 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 and the query is executed via the list() method. In the future more options might become available, like performing executeUpdate(), use named queries and etc.

5.4.6.1. Examples

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

```
<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 *queryresult.txt*.

Note:

Currently the dump is simply a call to toString on each element.

```
<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>
</hibernatetool>
```

5.4.7. Generic Hibernate metamodel exporter (<hbmtemplate>)

Generic exporter that can be controlled by a user provides a template or class.

```
<hbmtemplate
 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 us much better exception and error handling.

5.4.7.1. Exporter via <hbmtemplate>

The following is an example of reverse engineering via `<jdbcconfiguration>` and usage of a custom Exporter via the `<hbmtemplate>`.

```
<hibernatetool destdir="${destdir}">
 <jdbcconfiguration
   configurationfile="hibernate.cfg.xml"
   packagename="my.model"/>
```

```
  <!-- setup properties -->
  <property key="appname" value="Registration"/>
  <property key="shortname" value="crud"/>

  <hbmtemplate
    exporterclass="my.own.Exporter"
    filepattern="."/>

</hibernatetool>
```

5.4.7.2. Relevant Resources Links

Read more about *Velocity* [http://velocity.apache.org/] and *Freemarker* [http://freemarker.org/] to find out why using the last is better or refer to Max Andersen 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. The user properties are specified via `<property>` or `<propertyset>` and each exporter will have access to them directly in the templates and via Exporter.setProperties().

5.5.1. <property> and <propertyset>

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

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

Most times using `<property>` is enough for specifying the properties needed for the exporters. Still the ant tools supports the notion of `<propertyset>` that is used for grouping a set of properties. More about the functionality of `<propertyset>` is explained in detail in the *Ant manual* [http://ant.apache.org/manual/].

5.5.2. Getting access to user specific classes

If the templates need to access some user class it becomes possible by specifying a `"toolclass"` in the properties.

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

Placing the above `<property>` tag in `<hibernatetool>` or inside any exporter will automatically create an instance of `x.y.z.NameOfToolClass` and it will be available in the 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 `<hbmtemplate>` together with `<property>` which will be available to the templates/exporter.

Note:

This example actually simulates what `<hbm2java>` actually does.

```
<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>`, the ant task will read the database metadata and thus will perform a reverse engineering of the database schema into a normal Hibernate Configuration. It is from this object e.g. `<hbm2java>`can generate other artifacts such as *.java* , *.hbm.xml* etc.

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 foreignkeys into classes, properties and associations. It also used to provide mappings from SQL types to Hibernate types. The strategy can be customized by a user. The user can even provide its own custom reverse engineering strategy if the provided strategy is not enough, or simply just provide a small part of the strategy and delegate the rest to the default strategy.

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

6.1. Default reverse engineering strategy

The default strategy uses some rules for mapping JDBC artifact names to java artifact names. It also provide basic typemappings from JDBC types to Hibernate types. It is the default strategy that uses the packagename attribute to convert a table name to a fully qualified classname.

6.2. hibernate.reveng.xml file

To have fine control over the process a *hibernate.reveng.xml* file can be provided. 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 *Hibernate plugins* [http://www.hibernate.org/30.html] which have a specialized editor.

Note:

Many databases are case-sensitive with their names and thus if you cannot make some table match and you are sure it is not excluded by a `<table-filter>` then check if the case matches; most databases stores table names in uppercase.

Below you can see an example of a *reveng.xml.* Following the example gives you more details about the format.

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

<hibernate-reverse-engineering>
```

```
<type-mapping>
 <!-- jdbc-type is name for java.sql.Types -->
 <sql-type jdbc-type="VARCHAR" length='20' hibernate-type="SomeUserType" />
 <sql-type jdbc-type="VARCHAR" length='1' hibernate-type="yes_no" />
 <!-- length, scale and precision can be used to specify the mapping precisely -->
 <sql-type jdbc-type="NUMERIC"  precision='1' hibernate-type="boolean" />
 <!-- 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"  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>
  <key-column name="CUSTID"/>
 </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"/>
   <inverse-one-to-one exclude="true"/>
  </foreign-key>
</table>
```

```
</hibernate-reverse-engineering>
```

6.2.1. Schema Selection (<schema-selection>)

`<schema-selection>` is used to drive which schemas the reverse engineering will try and process.

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

With `<schema-selection>` it is thus possible to limit the actual processed schemas and thus significantly speed-up the reverse engineering. `<table-filter>` is still used to then decide which tables will be included/excluded.

Note:

If no `<schema-selection>` 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 simply 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 starts 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>
 <sql-type
  jdbc-type="integer value or name from java.sql.Types"
  length="a numeric value"
  precision="a numeric value"
  scale="a numeric value"
  not-null="true|false"
  hibernate-type="hibernate type name"
 />
</type-mapping>
```

The number of attributes specified and the sequence of the `sql-type`'s is important. Meaning that 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 the importance of ordering of the type mappings.

```
<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

Column	jdbc-type	lengt	precisic	not-null	Resulting hibernate-type	Rationale
ID	INTEGER		10	true	int	Nothing is defined for INTEGER. Falling back to default behavior.

Column	jdbc-type	lengt	precisic	not-null	Resulting hibernate-type	Rationale
NAME	VARCHAR	30		false	your.package.TrimStringUserTyp	The type-mapping matches length=30 and not-null=false, but type-mapping matches the 2 mappings which only specifies VARCHAR. The type-mapping that comes first is chosen.
INITIAL	VARCHAR	1		false	char	Even though there is a generic match for VARCHAR, the more specific type-mapping for VARCHAR with not-null="false" is chosen. The first VARCHAR sql-type matches in length but has no value for not-null and thus is not considered.
CODE	VARCHAR	1		true	java.lang.Character	The most specific VARCHAR with not-null="true" is selected
SALARY	NUMERIC		15	false	big_decimal	There is a precise match for NUMERIC with precision 15
AGE	NUMERIC		3	false	java.lang.Long	type-mapping for NUMERIC with not-null="false"

6.2.3. Table filters (<table-filter>)

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

```
<table-filter
 match-catalog="catalog_matching_rule"
 match-schema="schema_matching_rule"
 match-name="table_matching_rule"
 exclude="true|false"
 package="package.name"
/>
```

Table 6.2. Table-filter attributes

Attribute name	Definition	Default value
match-catalog	Pattern for matching catalog part of the table	.*
match-schema	Pattern for matching schema part of the table	.*
match-table	Pattern for matching table part of the table	.*
exclude	If true the table will not be part of the reverse engineering	false
package	The default package name to use for classes based on tables matched by this table-filter	""

6.2.4. Specific table configuration (<table>)

`<table>` allows you to provide explicit configuration on how a table should be reverse engineered. Amongst other things it allows controlling over the naming of a class for the table, specifying which identifier generator should be used for the primary key etc.

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

Table 6.3. Table attributes

Attribute name	Definition	Attribute use
catalog	Catalog name for a table. It has to be specified if you are reverse engineering multiple catalogs or if it is not equal to hiberante.default_catalog.	Optional

Attribute name	Definition	Attribute use
schema	Schema name for a table. It has to be specified if you are reverse engineering multiple schemas or if it is not equal to hiberante.default_schema.	Optional
name	Name for a table.	Required
class	The class name for a table. Default name is a camelcase version of the table name.	Optional

6.2.4.1. <primary-key>

A `<primary-key>` allows you to define a primary-key for tables that don't have it defined in the database, and probably more importantly it allows you to define which identifier strategy should be used (even for already existing primary-key's).

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

Table 6.4. Primary-key attributes

Attribute name	Definition	Attribute use
generator/class	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.	Optional
generator/ param	Allows to specify which parameter with a name and value should be passed to the identifier generator.	Optional
key-column	Specifies which column(s) the primary-key consists of. A key-column is same as column, but does not have the exclude property.	Optional

6.2.4.2. <column>

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

```
<column
 name="column_name"
 jdbc-type="java.sql.Types type"
```

```
 type="hibernate_type"
 property="propertyName"
 exclude="true|false"
/>
```

Table 6.5. Column attributes

Attribute name	Definition	Attribute use
name	Column name	Required
jdbc-type	Which jdbc-type this column should be processed as. A value from java.sql.Types, either numerical (93) or the constant name (TIMESTAMP).	Optional
type	Which hibernate-type to use for this specific column	Optional
property	What property name will be generated for this column	Optional
exclude	Set to true if this column should be ignored	default: false

6.2.4.3. <foreign-key>

The `<foreign-key>` has two purposes. One for allowing to define foreign-keys in databases that does not support them or does not have them defined in their schema. Secondly, to allow defining the name of the resulting properties (many-to-one, one-to-one and one-to-many's).

```
<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

Attribute name	Definition	Attribute use
constraint-name	Name of the foreign key constraint. Important when naming many-to-one, one-to-one and set. It is the constraint-name that is used to link the processed foreign-keys with the resulting property names.	Required
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 that 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 strategy must implement *org.hibernate.cfg.reveng.ReverseEngineeringStrategy*. It is recommended that one uses 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 fallback strategy. Example of custom delegating strategy which converts all column names that ends with *"PK"* into a property named *"id"*.

```
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 by reading 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 property *hibernatetool.metadatadialect* to a fully qualified classname for a class that implements JDBCMetaDataDialect.

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

Controlling POJO code generation

When using <hbm2java> or the eclipse plugin to generate POJO java code you have the possibility to control certain aspects of the code generation. 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* with information, so tools have a natural place to store/read information that is not directly related to the Hibernate core.

You can use the <meta> tag to e.g. tell <hbm2java> to only generate *"protected"* setters, have classes always implement a certain set of interfaces or even have them extend a certain base class and even more.

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

```
<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* will produce something like the following (code shortened for better understanding). Notice the Javadoc comment and the protected set methods:

```
// 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
 *       @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

Attribute	Description
`class-description`	inserted into the javadoc for classes
`field-description`	inserted into the javadoc for fields/properties
`interface`	If true, an interface is generated instead of an class.

Attribute	Description
`implements`	interface the class should implement
`extends`	class that the current class should extend (ignored for subclasses)
`generated-class`	overrule the name of the actual class generated
`scope-class`	scope for class
`scope-set`	scope for setter method
`scope-get`	scope for getter method
`scope-field`	scope for actual field
`default-value`	default initialization value for a field
`use-in-tostring`	include this property in the `toString()`
`use-in-equals`	include this property in the `equals()` and `hashCode()` method. If no use-in-equals is specified, no equals/hashcode will be generated.
`gen-property`	property will not be generated if false (use with care)
`property-type`	Overrides the default type of property. Use this with any tag's to specify the concrete type instead of just Object.
`class-code`	Extra code that will inserted at the end of the class
`extra-import`	Extra import that will inserted at the end of all other imports

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

What does that mean? It means that if you e.g want to have all your classes implement IAuditable then you just add an `<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 e.g. be used to specify that all fields should be declare protected, instead of the default private. This is done by adding `<meta attribute="scope-field">protected</meta>` at e.g. just under the `<class>` tag and all fields of that class will be protected.

To avoid having a `<meta>` tag inherited then you can simply specify `inherit = "false"` for the attribute, e.g. `<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 when using `<meta>` attributes.

7.1.1.1. Dangers of a class level use-in-string and use-in-equals meta attributes when having bi-directional associations

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

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

And for *Event.hbm* file:

```
<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>
```

Then `<hbm2java>` will assume you want to include all properties and collections in the `toString()/equals()` methods and 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()/equals()` methods. Therefore it is not a good practice to put at class scope such *meta* attributes, unless you are defining a class without bi-directional associations.

We recommend instead to add the *meta* attributes at the property level:

```
<hibernate-mapping>
```

```
  <class name="events.Event" table="EVENTS">
    <id name="id" column="EVENT_ID">
        <meta attribute="use-in-tostring">true</meta>
        <generator class="native"/>
    </id>
    <property name="date" type="timestamp" column="EVENT_DATE"/>
    <property name="title">
      <meta attribute="use-in-tostring">true</meta>
      <meta attribute="use-in-equals">true</meta>
    </property>
    <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>
```

and now for Person:

```
<hibernate-mapping>
    <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>
        <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> attributeuse-in-equals

For `equal()/hashCode()` method generation, you have to take into account that the attributes that participate on such method definition, should take into account only attributes with business meaning (the name, social security number, etc, but no generated id's, for example).

This is important because Java's hashbased collections, such as java.util.Set relies on equals() and hashcode() to be correct and not change 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()/hashCode() methods specifying at class scope level the `<meta>` attribute *use-in-equals* could be a dangerous decision that could produce non expected side-effect.

On www.hibernate.org [http://www.hibernate.org/109.html] you can get more in-depth explanation on the subject of equals() and hashcode().

7.1.2. Advanced <meta> attribute examples

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

The usecase being implemented is to automatically insert some pre- and post-conditions into the getter and setters of the generated POJO.

7.1.2.1. Generate pre/post-conditions for methods

With a `<meta attribute="class-code">`, you can add additional methods on a given class, nevertheless such `<meta>` attribute 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 POJO's. If you look inside *hibernate-tools.jar*, you can find the template: *pojo/PojoPropertyAccessor.ftl*

This file is as the name indicates used to generate property accessors for pojo's.

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

The contents of the file is something like this:

```
<#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)} ${property.name})
    {
        this.${property.name} = ${property.name};
    }
</#foreach>
```

We can add conditionally pre/post-conditions on our `set` method generation just adding a little Freemarker syntax to the above source code:

```
<#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`, their contents will be generated into the body of the relevant `set` method.

As an example let us add a pre-condition for property `name` preventing no Person can have an empty name. Hence we have to modify the *Person.hbm.xml* file like this:

```
<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) &amp;amp;&amp;amp; (firstName.length() == 0) ) {
      throw new IllegalArgumentException("firstName can not be an empty String");
    }
    </meta>
  </property>
</class>
</hibernate-mapping>
```

Note:

I) To escape the & symbol we put &. You can use `<![CDATA[]]>` instead.

II) Note that we are referring to "firstName" directly and this is the parameter name not the actual field name. If you want to refer the field you have to use "this.firstName" instead.

Finally we have to generate the *Person.java* class, for this we can use both Eclipse and Ant as long as you remember to set or fill in the templatepath setting. For Ant we configure `<hibernatetool>` task via `the templatepath` attribute as in:

```
<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>
    <hbm2java/>
  </hibernatetool>
```

```
    </target>
```

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

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

    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;
    }

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

    public java.lang.String getName() {
        return this.name;
    }

    public void setName(java.lang.String name) {
        if ((name != null) &amp;amp;&amp;amp; (name.length() == 0)) {
            throw new IllegalArgumentException("name can not be an empty String");
        }
        this.name = name;
    }
    }
```

In conclusion, this document is intended to introduce you to Hibernate plugin specific features related to tools bath for the Eclipse and Ant tasks.

In the *Eclipse Plugins* chapter you've learnt about a set of wizards for creating Mapping files, Configuration file, Console Configuration, got familiar with Mapping and Configuration files editors, tooling for organizing and controlling Reverse Engineering, Hibernate Console and Mapping diagram as well.

The rest chapters have shown the aspects of using 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.