Hibernate Tools

Reference Guide

Version: 3.2.0.beta10

Table of Contents

Preface

Hibernate Tools is a toolset for Hibernate 3 and related projects. The tools provide Ant tasks and Eclipse plugins for performing reverse engineering, code generation, visualization and interaction with Hibernate.

Chapter 1. Download and install Hibernate Tools

Hibernate tools can be used "standalone" via Ant 1.6.x or fully integrated into a Eclipse 3.2.x based IDE, such as JBoss Eclipse IDE or a default Eclipse 3.2.x installation. The following describes the install steps in these environments.

1.1. JBoss Eclipse IDE

JBoss Eclipse IDE 2.x includes Hibernate Tools and thus nothing is required besides downloading and installing JBoss Eclipse IDE. If you need to update to a newer version of the Hibernate Tools just follow the instructions in the Eclipse IDE section.

1.2. Eclipse IDE

To install into any Eclipse 3.2.x based Eclipse IDE you can either download the Hibernate Tools distribution from the Hibernate website or use the JBoss IDE Eclipse update site (see http://tools.hibernate.org for links to the update site).

If you download the Hibernate Tools distribution you need to place the /plugins and /feature directory into your eclipse directory or eclipse extensions directory. Sometimes Eclipse does not automatically detect new plugins and thus the tools will not be activated. To ensure eclipse sees these changes run eclipse with the -clean option. E.g. `eclipse -clean` Using the updatesite does not require any additional steps.

Tip: If you need more basic instructions on installing plugins 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.

1.2.1. Usage of Eclipse WTP

The Hibernate tools plugins currently uses WTP 1.5.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.

The tools only include a subset of the WTP 1.5.x plugins, thus if you want full access to the WTP functionallity the full WTP 1.5.x SDK should be able to instal on top of the plugins without any problems.

1.3. 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 chapter.

Chapter 2. 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.

2.1. Hibernate Meta Model

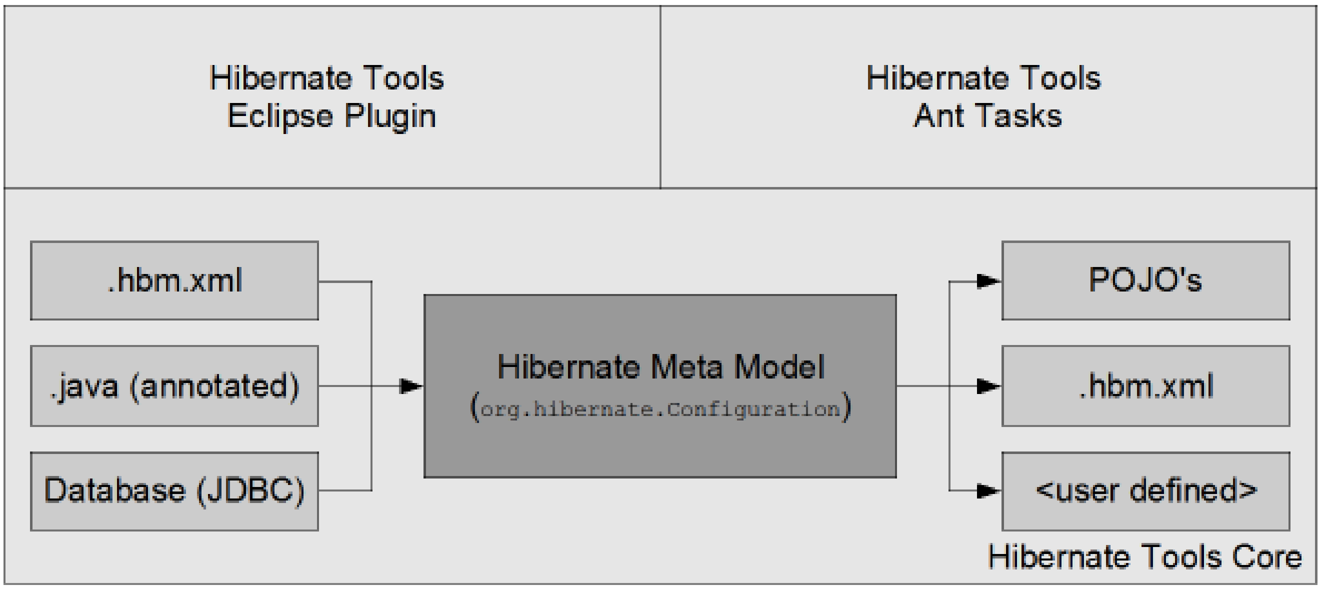
The meta model is the model used by Hibernate core to perform its object relational mapping. The model includes information about tables, columns, classes, properties, components, values, collections etc. The API is in `org.hibernate.mapping` and its main entry point is the `Configuration` class, the 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.
- A *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. The important thing to note is that no matter which Hibnerate Configuration type you are using Hibernate Tools supports them.

The following drawing illustrates the 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.

2.2. 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 and Eclipse sections.

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 4.4.7, “Generic Hibernate metamodel exporter (<hbmtemplate>)”

Chapter 3. Eclipse Plugins

3.1. Introduction

The following features are available in the Hibernate Tools Eclipse plugins:

Mapping Editor: An editor for Hibernate XML mapping files, supporting auto-completion and syntax highlighting. It also supports semantic auto-completion for class names and property/field names, making it much more versatile than a normal XML editor.

Hibernate Console: The console is a new perspective in Eclipse. It 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.

Configuration Wizards and Code generation: A set of wizards are provided with the Hibernate Eclipse tools; you can use a wizard to quickly generate common Hibernate configuration (cfg.xml) files, and from these you can code generate a series of various artifacts, there is even support for completely reverse engineer an existing database schema and use the code generation to generate POJO source files and Hibernate mapping files.

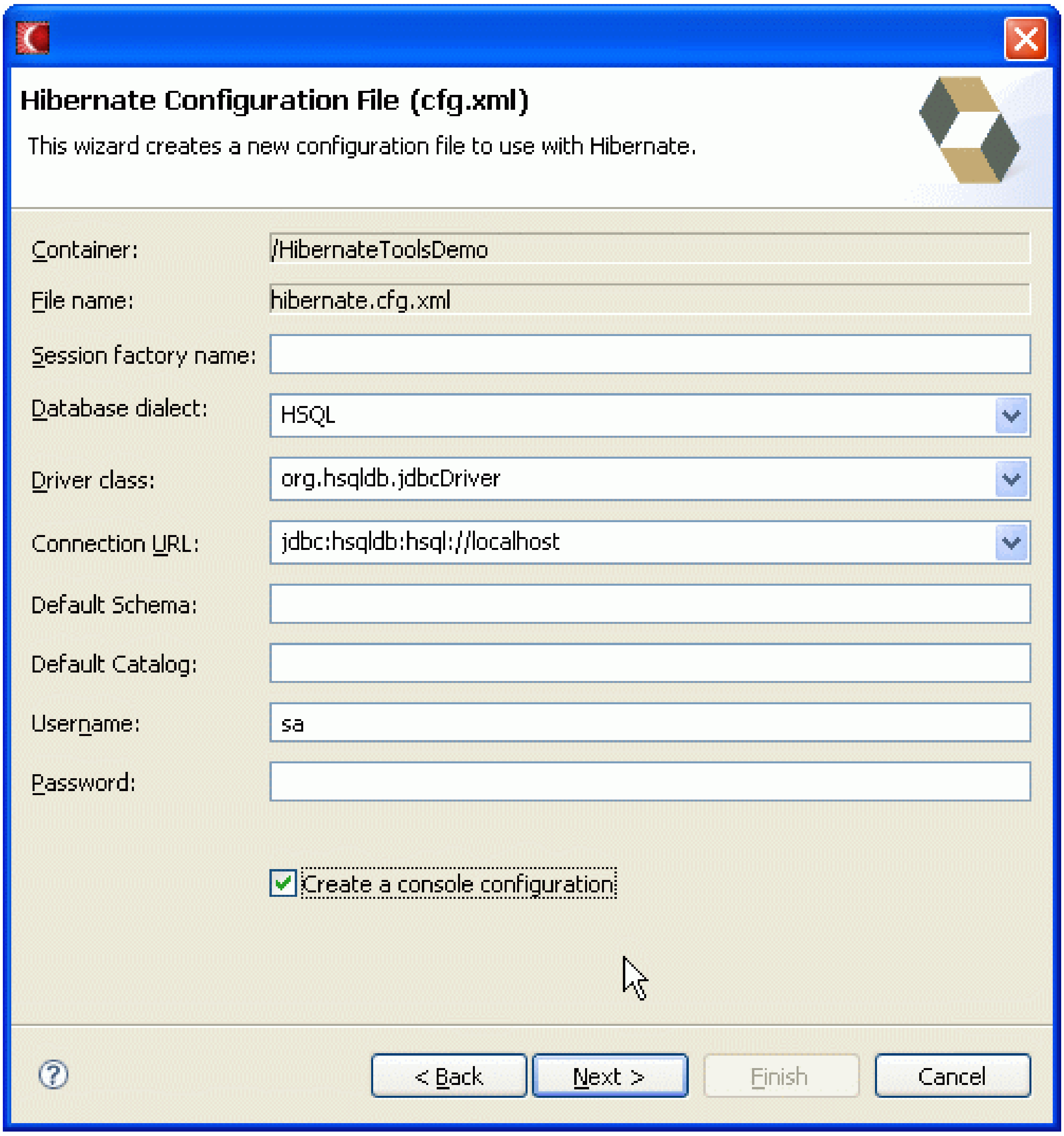
Eclipse JDT integration: Hibernate Tools integrates into the Java code completion and build support of Java in Eclipse. This gives you codecompletion of HQL inside Java code plus Hibernate Tools will add problem markers if your queries are not valid against the console configuration associated with the project.

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 documentation for Hibernate to fully utilize Hibernate Tools and especially Hibernate it self.

3.2. 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 file.

Start the wizard by clicking "New Wizard" (Ctrl+N), select the Hibernate/Hibernate Configuration file (cfg.xml) wizard and press "Next". After selecting the wanted location for the hibernate.cfg.xml file, you will see the following page:


Hibernate Configuration File (cfg.xml)

This wizard creates a new configuration file to use with Hibernate.

Container:	/HibernateToolsDemo
File name:	hibernate.cfg.xml
Session factory name:	
Database dialect:	HSQL
Driver class:	org.hsqldb.jdbcDriver
Connection URL:	jdbc:hsqldb:hsql://localhost
Default Schema:	
Default Catalog:	
Username:	sa
Password:	

[✓] Create a console configuration

Tip: 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.

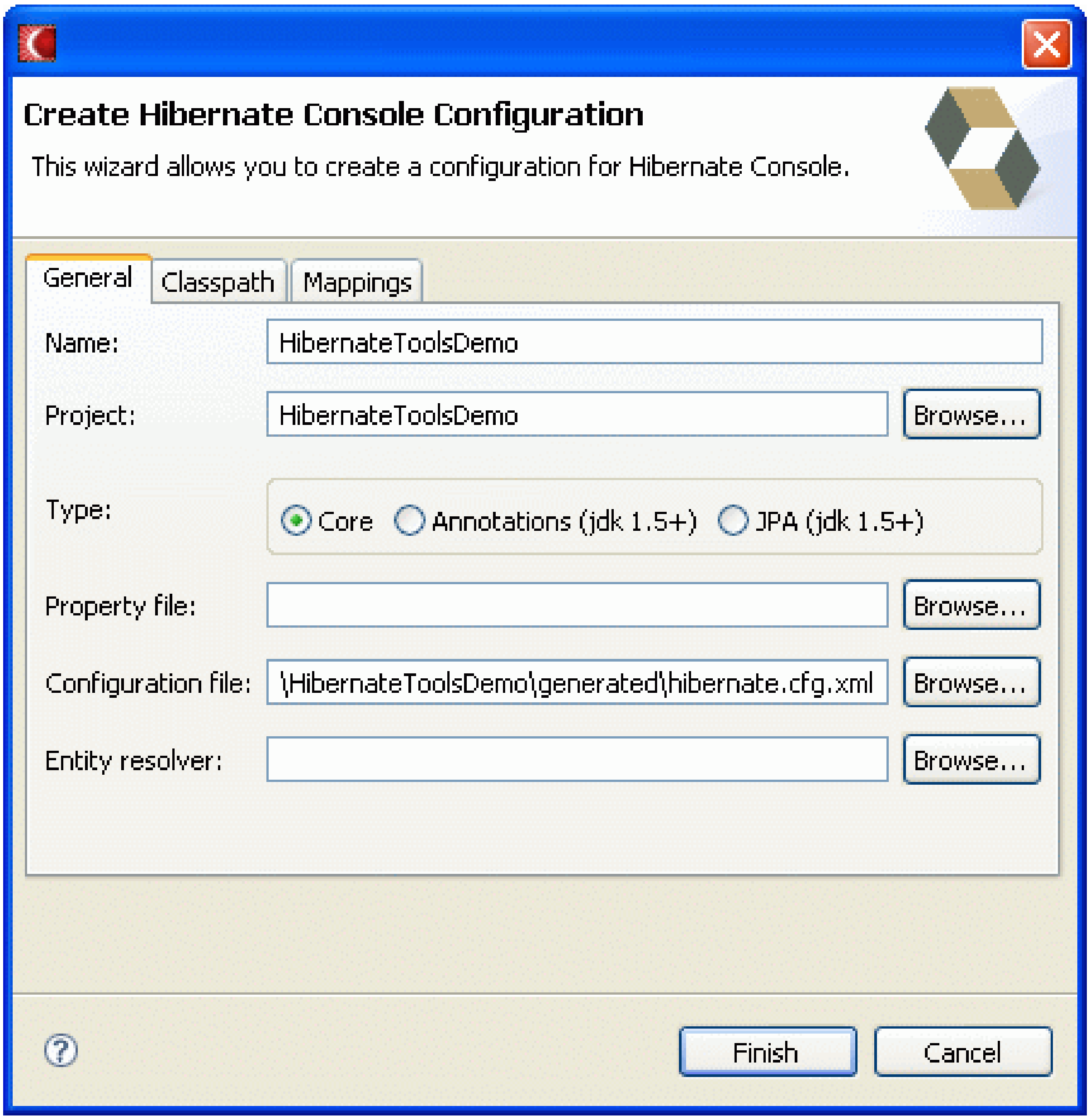
Press "Finish" to create the configuration file, after optionally creating a Console onfiguration, the hibernate.cfg.xml will be automatically opened in an editor. The last option "Create Console Configuration" is enabled by default and when enabled i will automatically use the hibernate.cfg.xml for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

3.3. Creating a Hibernate Console configuration

A Console Configuration describes to the Hibernate plugin how it should configure Hibernate and what configuration files, including which classpath is 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 definitly possible.

You 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".

Tip: 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.



Create Hibernate Console Configuration

This wizard allows you to create a configuration for Hibernate Console.

Name: HibernateToolsDemo

Project: HibernateToolsDemo

Type: [✓ Core] Annotations (jdk 1.5+) JPA (jdk 1.5+)

Property file:

Configuration file: \HibernateToolsDemo\generated\hibernate.cfg.xml

Entity resolver:

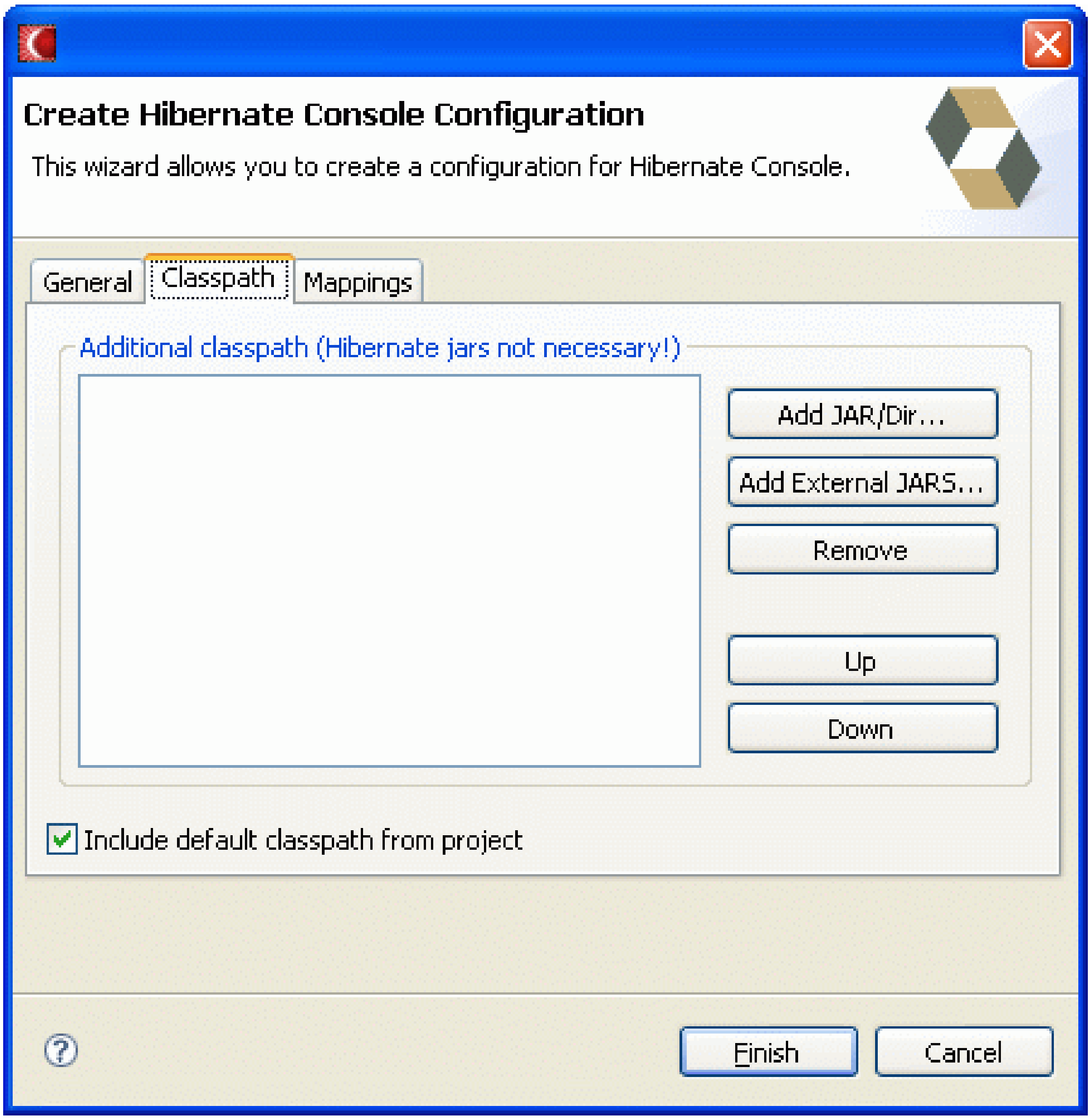
Creating a Hibernate Console configuration

The dialog consists of three tabs, "General" for the basic/required settings, "Classpath" for classpath and "Mappings" for additional mappings. The two latter ones is normally not required if you specify a project and it has `/hibernate.cfg.xml` or `/META-INF/persistence.xml` in its project classpath.

The following table describes the available settings. The wizard can automatically detect default values for most of these if you started the Wizard with the relevant java project or resource selected

Table 3.1. Hibernate Console Configuration Parameters

Parameter	Description	Auto detected value
Name	The unique name of the console configuration	Name of the selected project
Project	The name of a java project which classpath should be used in 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
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 (lets Hibernate Entity Manager find the persistence unit)
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



Create Hibernate Console Configuration

This wizard allows you to create a configuration for Hibernate Console.

General | Classpath | Mappings

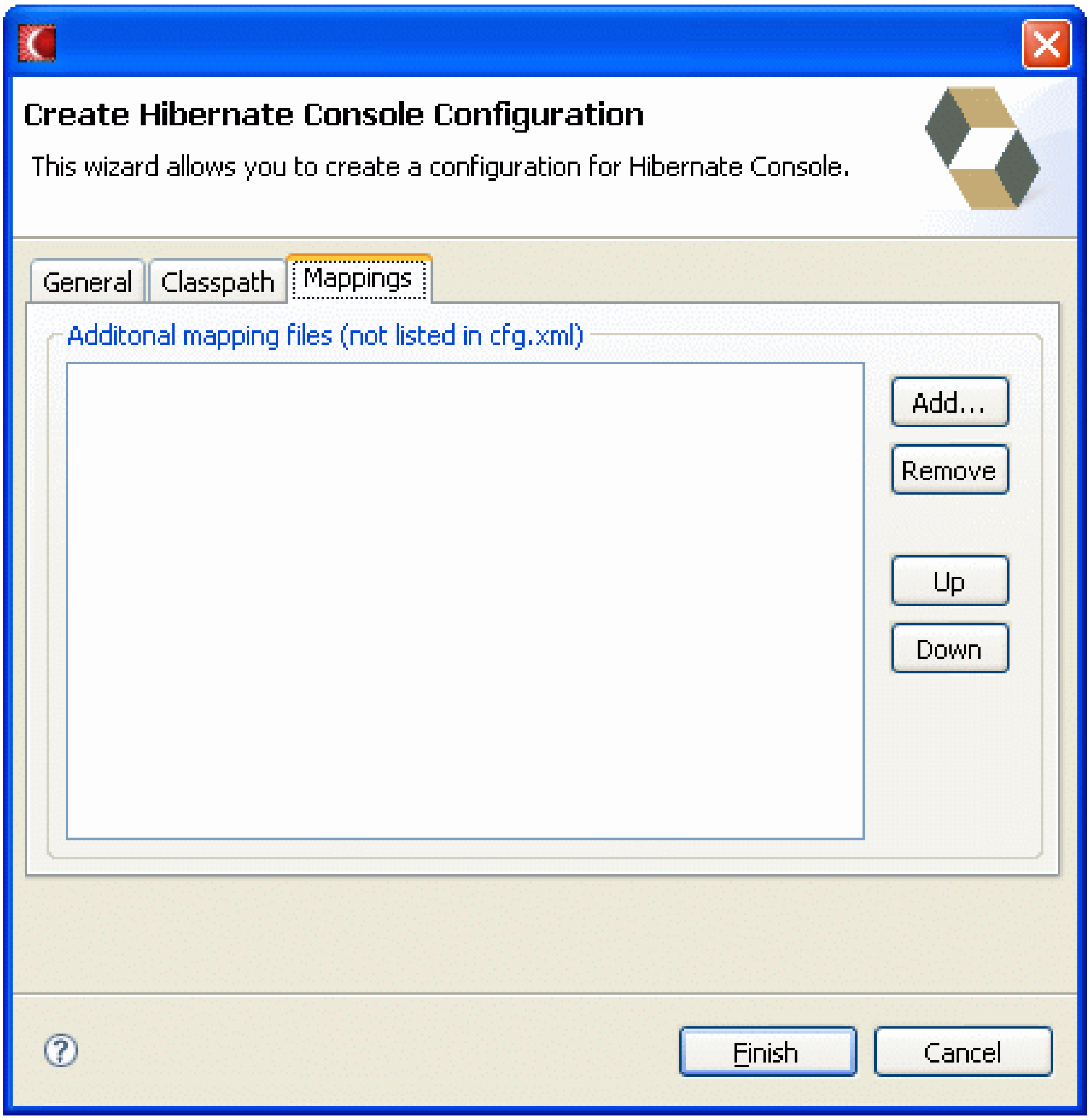
Additional classpath (Hibernate jars not necessary!)

[✓] Include default classpath from project

Specifying classpath in a Hibernate Console configuration

Table 3.2. 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

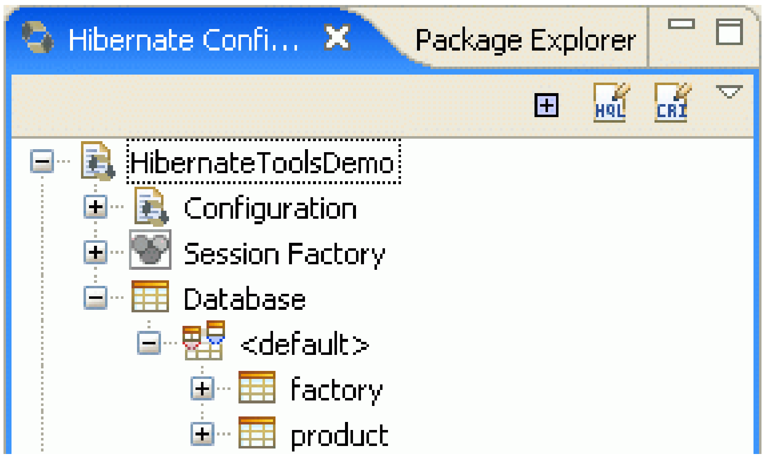

Create Hibernate Console Configuration
This wizard allows you to create a configuration for Hibernate Console.
General
Classpath
Mappings
Additonal mapping files (not listed in cfg.xml)
Add...
Remove
Up
Down
Finish
Cancel

Specifying additional mappings in a Hibernate console configuration

Table 3.3. 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 duplicated here, you will get "Duplicate mapping" errors when using the console configuration.	empty

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



Console overview

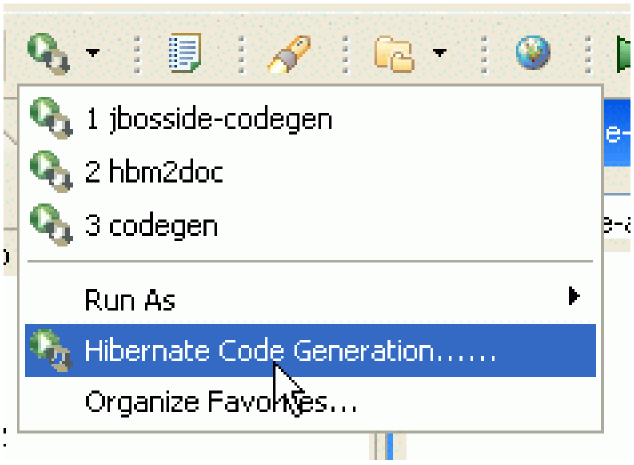
3.4. Reverse engineering and code generation

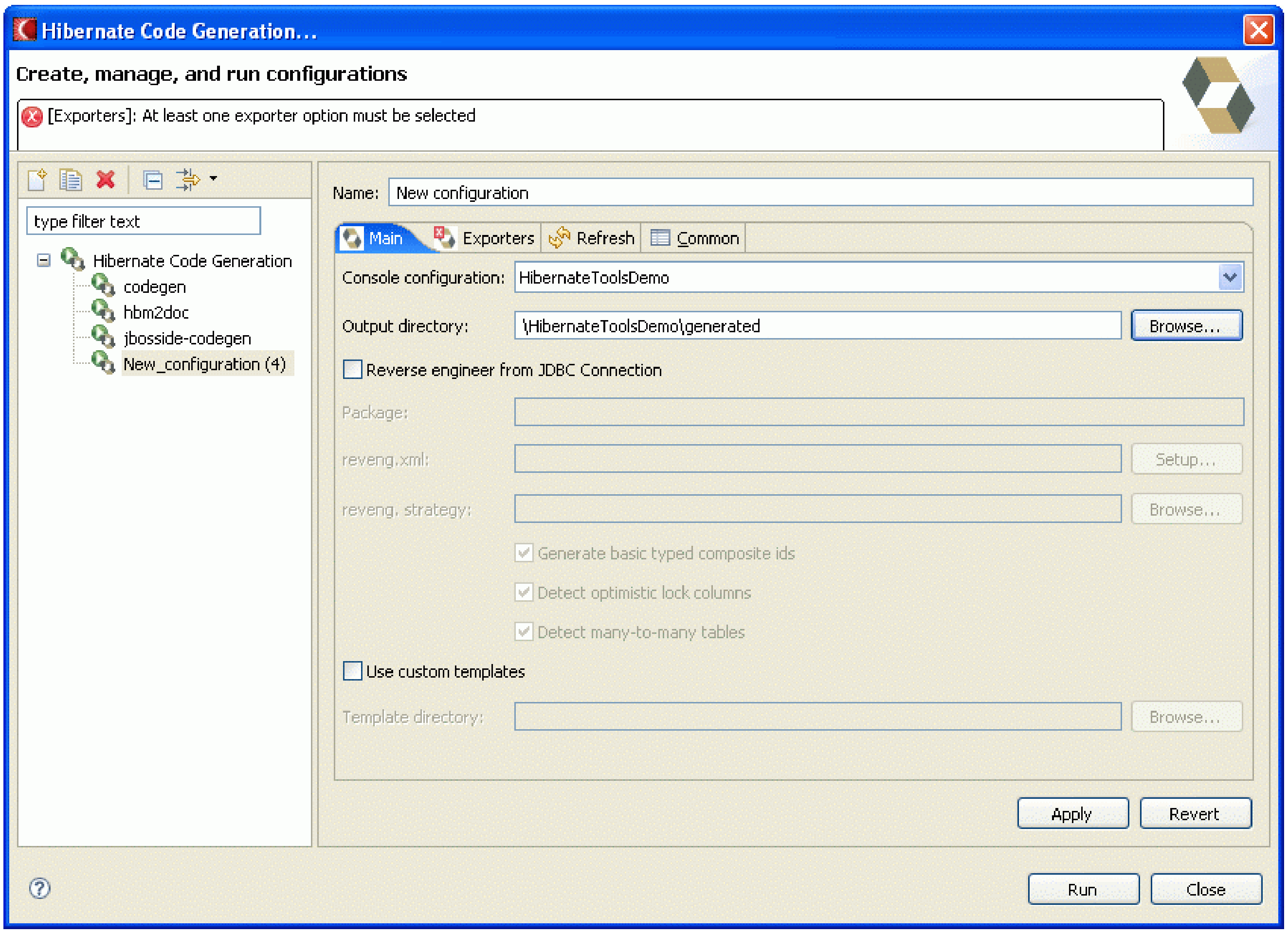
A "click-and-generate" reverse engineering and code generation facility is available. This facility allows you to generate a range of artifacts based on database or an already existing Hibernate configuration, be that mapping files or annotated classes. Some of these are POJO Java source file, Hibernate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

3.4.1. Code Generation Launcher

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





Hibernate Code Generation...

Create, manage, and run configurations

[Exporters]: At least one exporter option must be selected

type filter text

- Hibernate Code Generation
  - codegen
  - hbm2doc
  - jbosside-codegen
  - New_configuration (4)

Name: New configuration

Main | Exporters | Refresh | Common

Console configuration: HibernateToolsDemo

Output directory: \HibernateToolsDemo\generated

☐ Reverse engineer from JDBC Connection

Package:

reveng.xml:

reveng. strategy:

☑ Generate basic typed composite ids

☑ Detect optimistic lock columns

☑ Detect many-to-many tables

☐ Use custom templates

Template directory:

The first time you create a code generation launcher you should give it a meaningfull name, otherwise the default prefix "New_Generation" will be used.

Note: 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:

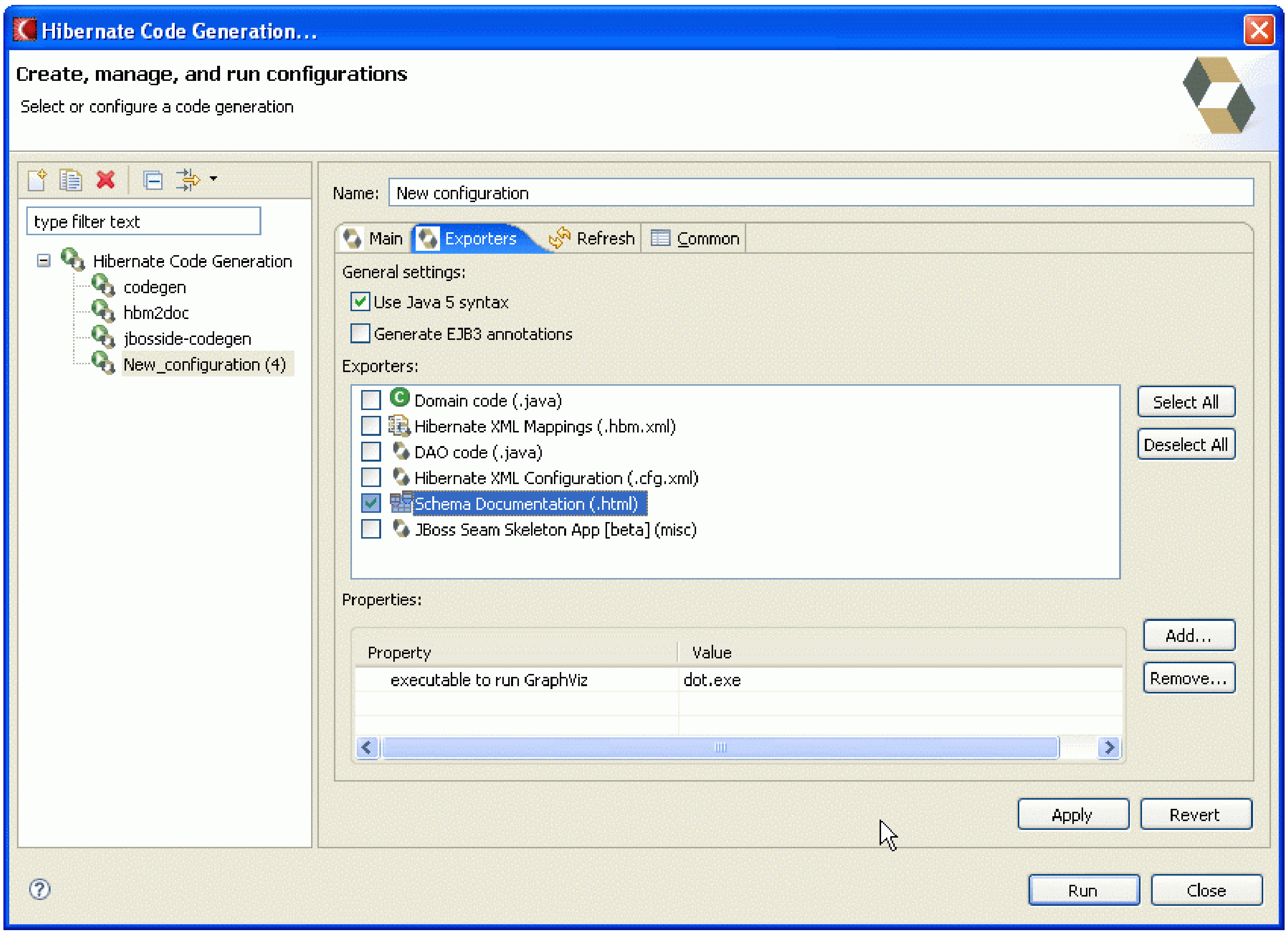
Table 3.4. Code generation "Main" tab fields

Field	Description
Console Configuration	The name of the console configuration which should be used when code generating.
Output directory	Path to a directory into where all output will be written by default. 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

Field	Description
	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 5, *Controlling reverse engineering*.
reveng. strategy	If reveng.xml does not provide enough customization you can provide your own implementation of an ReverseEngineeringStrategy. The class need to be in the claspath of the Console Configuration, otherwise you will get class not found exceptions. See Section 5.3, “Custom strategy” for details and an example of a custom strategy.
Generate basic typed composite ids	A table that has a multi-colum 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.
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.

3.4.2. Exporters

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


Hibernate Code Generation...

Create, manage, and run configurations

Select or configure a code generation

type filter text

- Hibernate Code Generation
  - codegen
  - hbm2doc
  - jbosside-codegen
  - New_configuration (4)

Name: New configuration

Main | Exporters | Refresh | Common

General settings:

[✓] Use Java 5 syntax

[ ] Generate EJB3 annotations

Exporters:

- [ ] Domain code (.java)
- [ ] Hibernate XML Mappings (.hbm.xml)
- [ ] DAO code (.java)
- [ ] Hibernate XML Configuration (.cfg.xml)
- [✓] Schema Documentation (.html)
- [ ] JBoss Seam Skeleton App [beta] (misc)

Select All | Deselect All

Properties:

Property	Value
executable to run GraphViz	dot.exe

Add... | Remove...

Apply | Revert

Run | Close

The following table describes in short the various exporters.

Table 3.5. 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 uptodate with any new found mapping files.
Schema Documentation (.html)	Generates set of html pages that documents the database schema and some of the mappings.

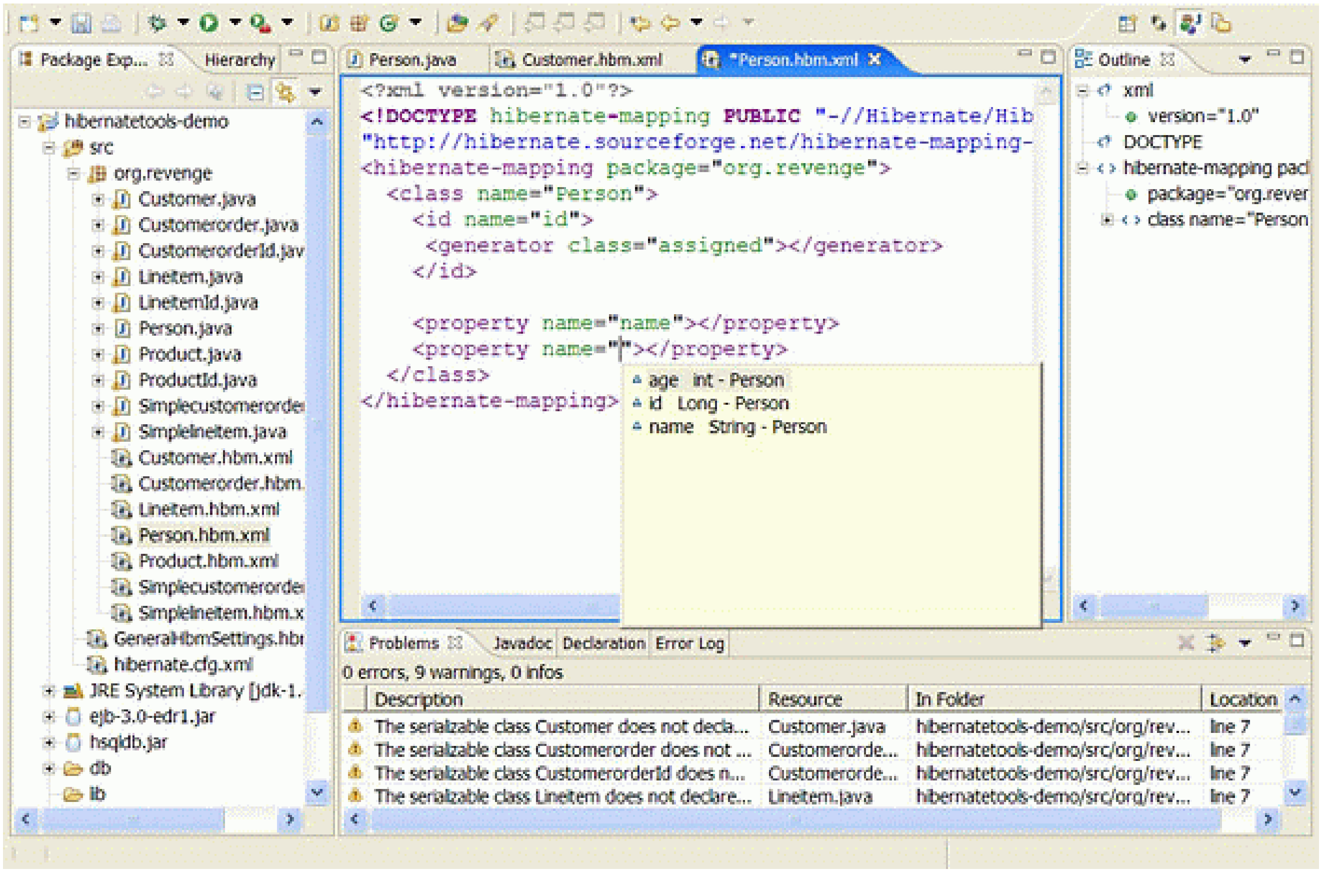
Each exporter listens to certain properties and these can be setup in the "Properties" section where you can add/remove predefined or customer properties for each of the exporters. The following table lists the time of writing pre-defined properties:

Table 3.6. Exporter Properties

Name	Description
jdk5	Generate Java 5 syntax
ejb3	Generate EJB 3 annotations
dot.executable	executable to run GraphViz (only relevant, but optional for Schema documentation)

3.5. 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 extend its functionallity to provide hibernate specific code completion.



```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hib
"http://hibernate.sourceforge.net/hibernate-mapping-
<hibernate-mapping package="org.revenge">
  <class name="Person">
    <id name="id">
     <generator class="assigned"></generator>
    </id>

    <property name="name"></property>
    <property name=""></property>
  </class>
</hibernate-mapping>
```

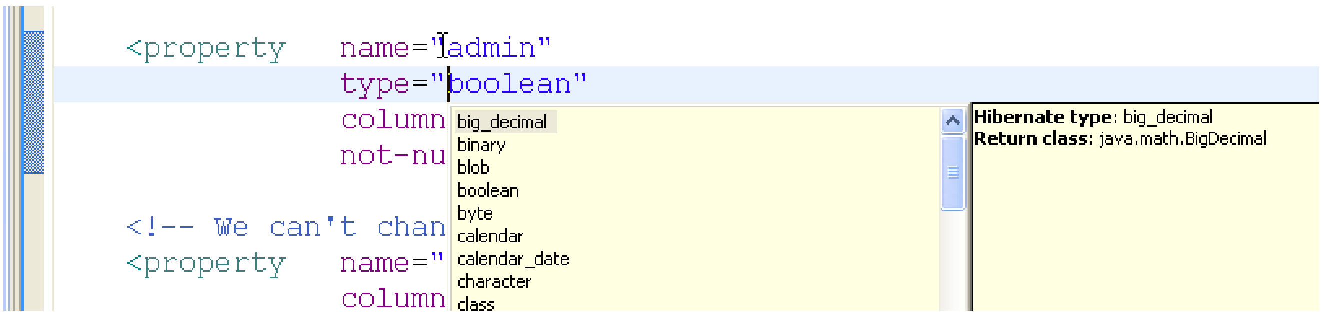
```
age   int - Person
id   Long - Person
name   String - Person
```

0 errors, 9 warnings, 0 infos

Description	Resource	In Folder	Location
The serializable class Customer does not decla...	Customer.java	hibernatetools-demo/src/org/rev...	line 7
The serializable class Customerorder does not ...	Customerorde...	hibernatetools-demo/src/org/rev...	line 7
The serializable class CustomerorderId does n...	Customerorde...	hibernatetools-demo/src/org/rev...	line 7
The serializable class LineItem does not declare...	LineItem.java	hibernatetools-demo/src/org/rev...	line 7

3.5.1. Java property/class completion

Package, class, and field completion is enabled for relevant XML attributes. The auto-completion detects it's 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.

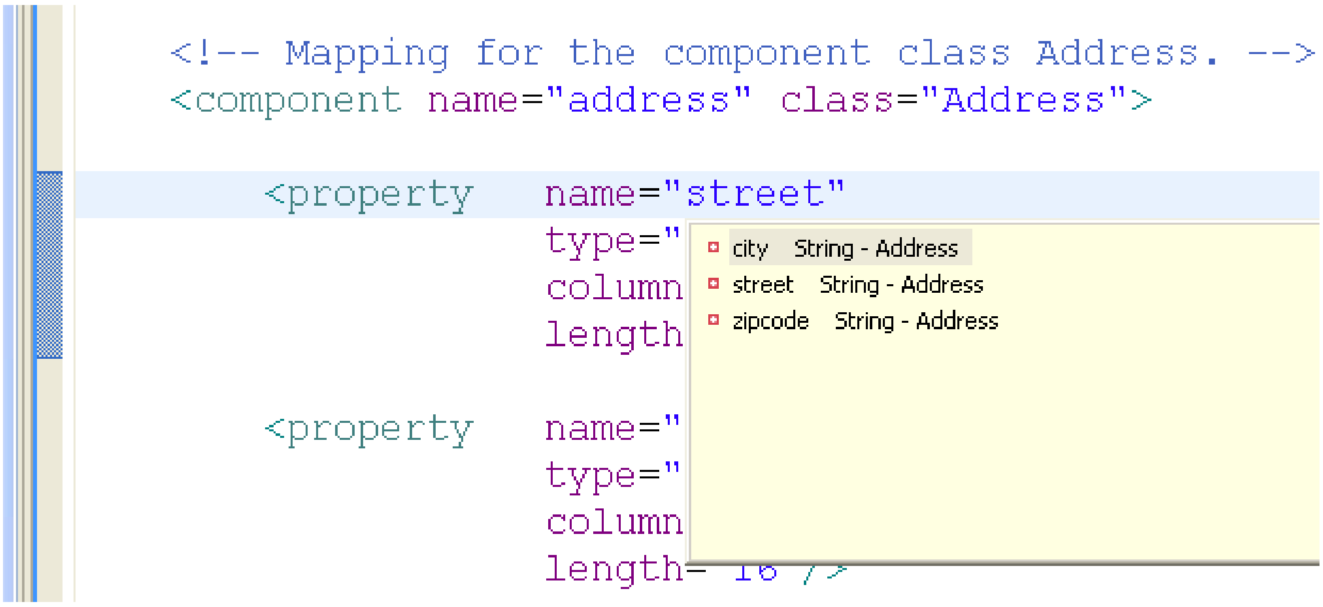


```
<property    name="admin"
             type="boolean"
             column
             not-nu

<!-- We can't chan
<property    name="
             column
```

```
big_decimal
binary
blob
boolean
byte
calendar
calendar_date
character
class
```

Hibernate type: big_decimal
Return class: java.math.BigDecimal



```
<!-- Mapping for the component class Address. -->
<component name="address" class="Address">

    <property    name="street"
                 type="
                 column
                 length

    <property    name="
                 type="
                 column
                 length="16"/>
```

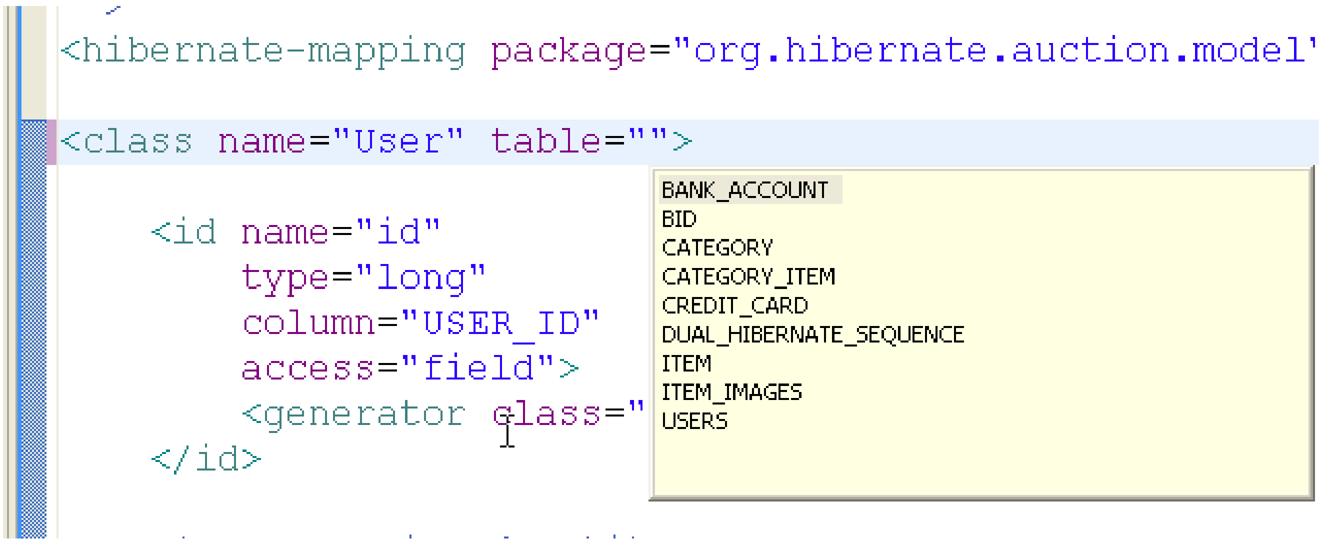
```
city     String - Address
street   String - Address
zipcode  String - Address
```

This is done via the standard hyperlink navigation functionallity 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.

3.5.2. Table/Column completion

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



```
<hibernate-mapping package="org.hibernate.auction.model'

<class name="User" table="">

    <id name="id"
        type="long"
        column="USER_ID"
        access="field">
        <generator class="
    </id>
```

```
BANK_ACCOUNT
BID
CATEGORY
CATEGORY_ITEM
CREDIT_CARD
DUAL_HIBERNATE_SEQUENCE
ITEM
ITEM_IMAGES
USERS
```

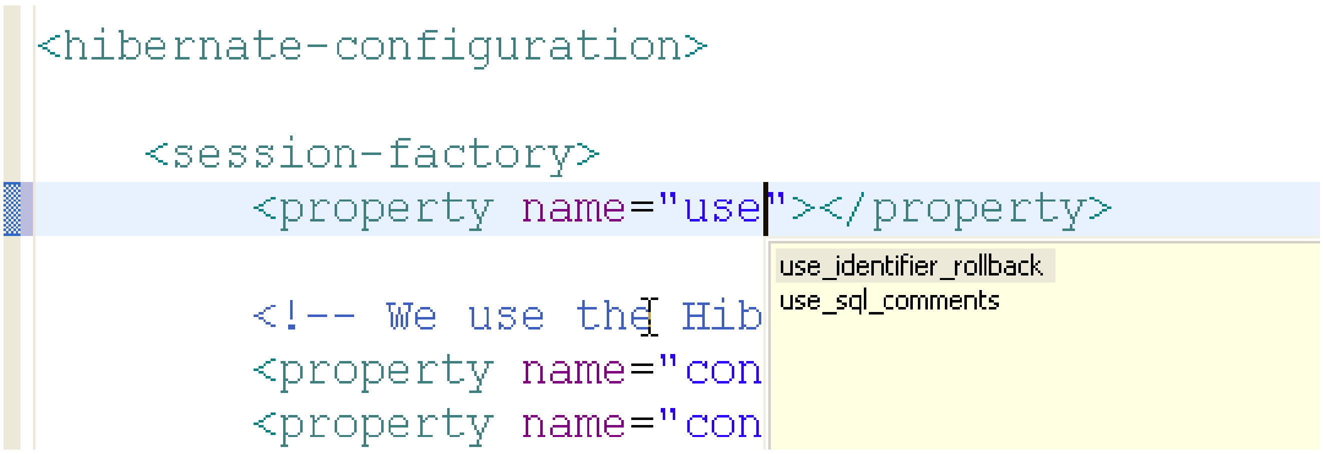
Note that it requires a proper configured hibernate console configuration and this configuration should be the default for the project where the hbm.xml resides.

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.

3.5.3. Configuration property completion

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



```
<hibernate-configuration>

    <session-factory>
        <property name="use"></property>

        <!-- We use the Hib
        <property name="con
        <property name="con
```

3.6. Reveng.xml editor

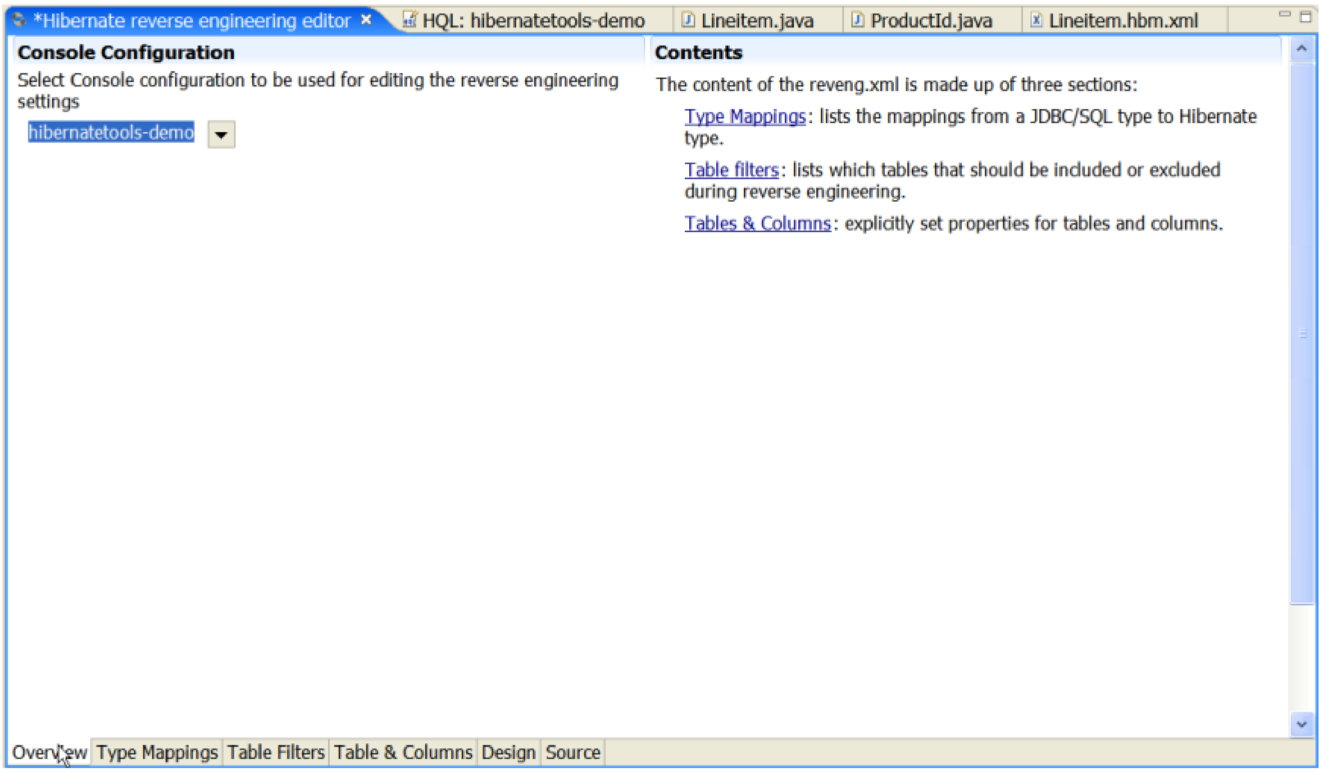
A reveng.xml file is used to customize and control how reverse engineering is performed by the tools. The plugins provide and editor to ease the editing of this file and hence used to configure the reverse engineering process.

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

Note that not all the features of the .reveng.xml file is exposed or fully implemented in the editor, but the main functionallity is there. To understand the full flexibility of the reveng.xml, please see Section 5.2, “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 reveng.xml wizard can be started via Ctrl+N or via the code generation launcher.

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



Console Configuration

Select Console configuration to be used for editing the reverse engineering settings

hibernatetools-demo

Contents

The content of the reveng.xml is made up of three sections:

Type Mappings: lists the mappings from a JDBC/SQL type to Hibernate type.

Table filters: lists which tables that should be included or excluded during reverse engineering.

Tables & Columns: explicitly set properties for tables and columns.

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.

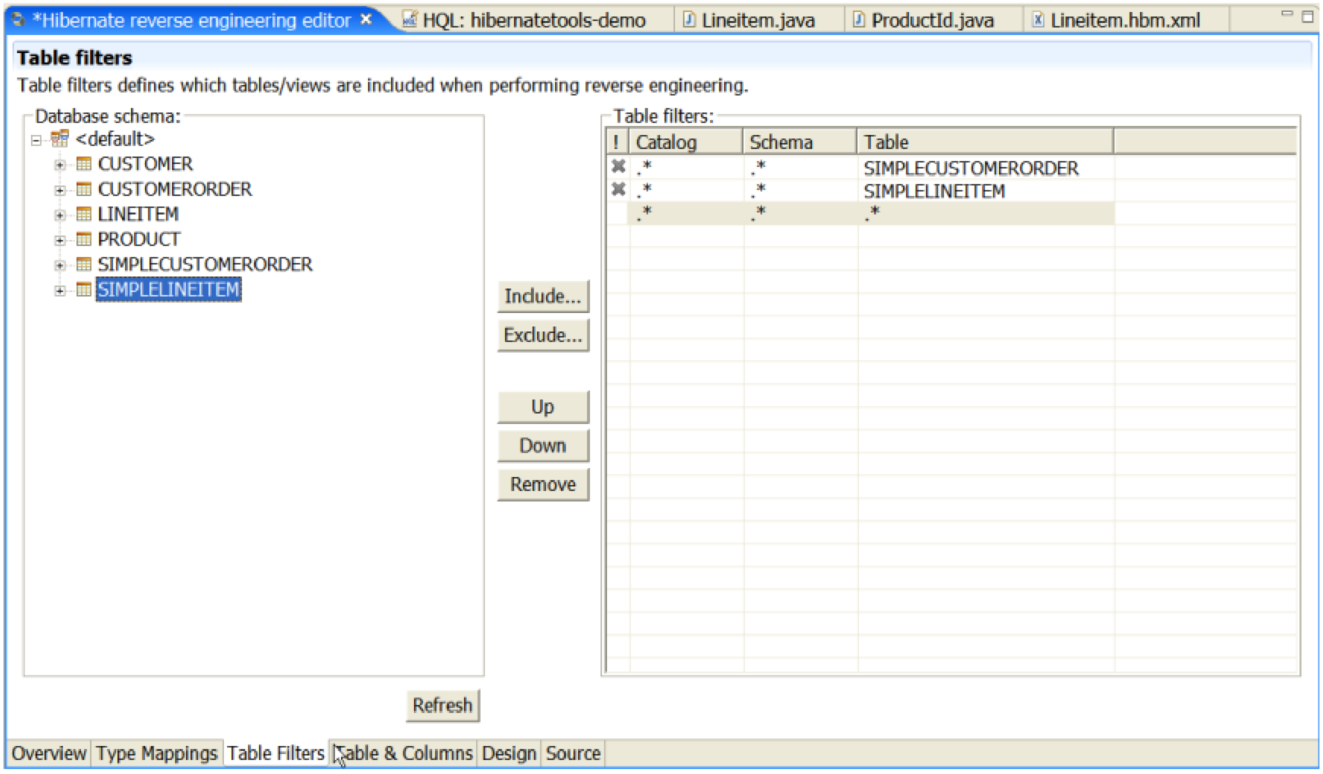


Table filters

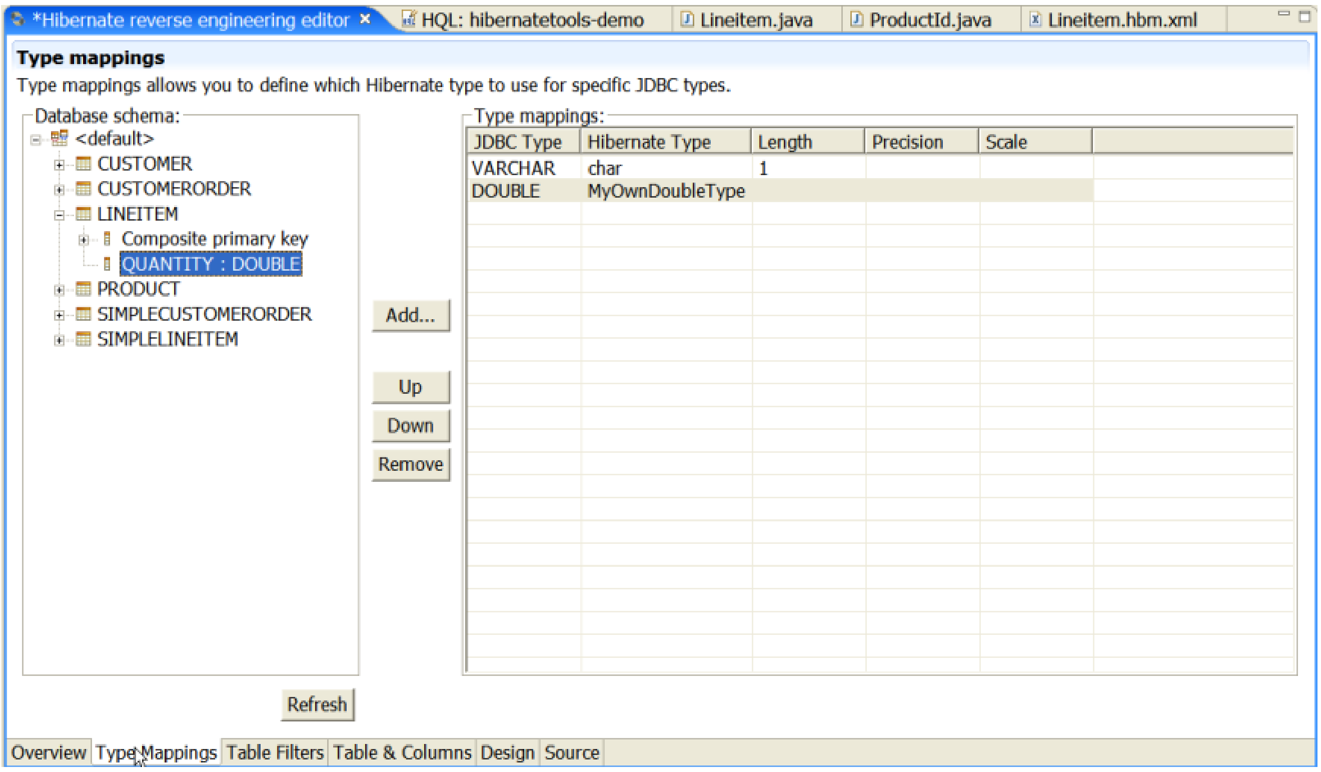
Table filters defines which tables/views are included when performing reverse engineering.

Database schema:
- <default>
  - CUSTOMER
  - CUSTOMERORDER
  - LINEITEM
  - PRODUCT
  - SIMPLECUSTOMERORDER
  - SIMPLELINEITEM

Table filters:

!	Catalog	Schema	Table
✖	.*	.*	SIMPLECUSTOMERORDER
✖	.*	.*	SIMPLELINEITEM
	.*	.*	.*

Type mappings page is used for specifying type mappings from jdbc types to any hibernate type (including usertypes) if the default rules are not applicable.



*Hibernate reverse engineering editor | HQL: hibernatetools-demo | Lineitem.java | ProductId.java | Lineitem.hbm.xml

Type mappings

Type mappings allows you to define which Hibernate type to use for specific JDBC types.

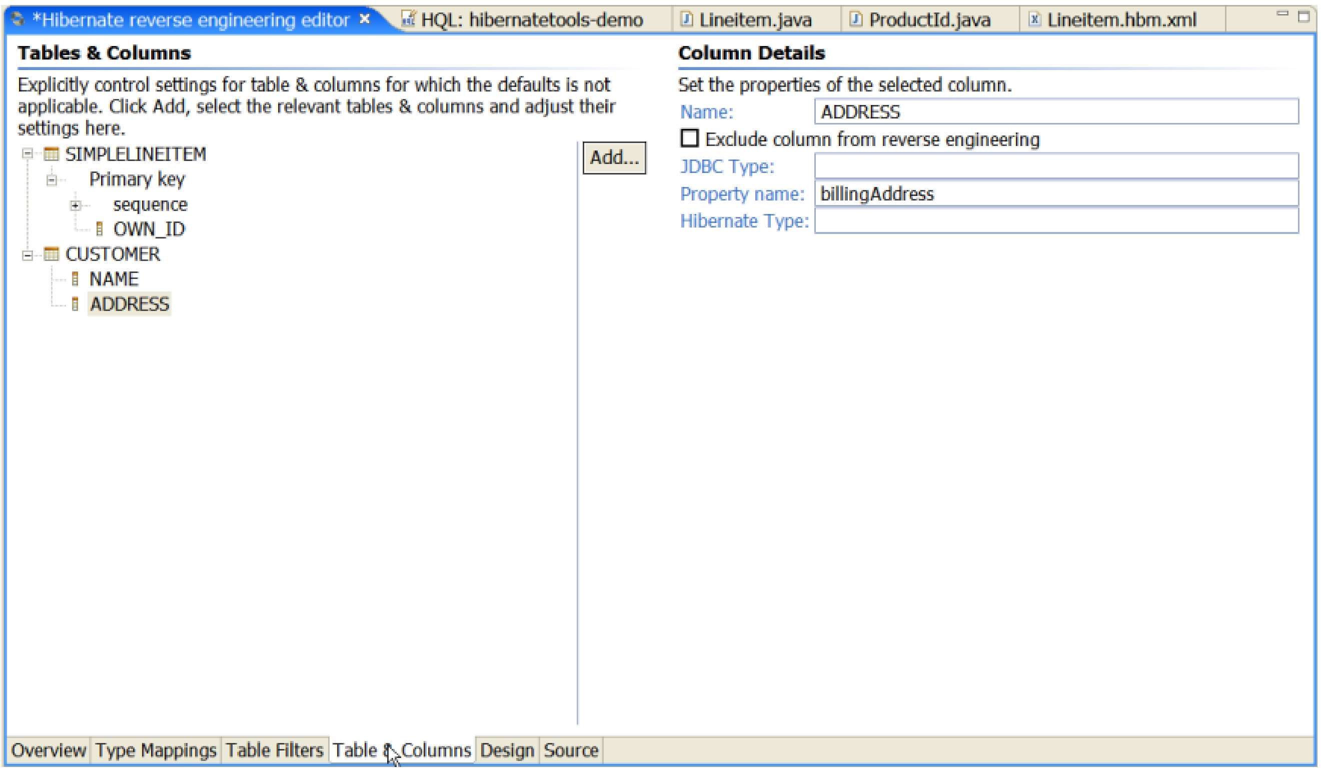
Database schema:

```
<default>
  CUSTOMER
  CUSTOMERORDER
  LINEITEM
    Composite primary key
    QUANTITY : DOUBLE
  PRODUCT
  SIMPLECUSTOMERORDER
  SIMPLELINEITEM
```

Type mappings:

JDBC Type	Hibernate Type	Length	Precision	Scale
VARCHAR	char	1		
DOUBLE	MyOwnDoubleType			

Table Columns page allow the user to explicit set e.g. which hibernatetype and propertyname that should be used in the reverse engineered model.



*Hibernate reverse engineering editor | HQL: hibernatetools-demo | Lineitem.java | ProductId.java | Lineitem.hbm.xml

Tables & Columns

Explicitly control settings for table & columns for which the defaults is not applicable. Click Add, select the relevant tables & columns and adjust their settings here.

```
SIMPLELINEITEM
  Primary key
    sequence
    OWN_ID
CUSTOMER
  NAME
  ADDRESS
```

Column Details

Set the properties of the selected column.

Name:	ADDRESS
☐ Exclude column from reverse engineering	
JDBC Type:	
Property name:	billingAddress
Hibernate Type:	

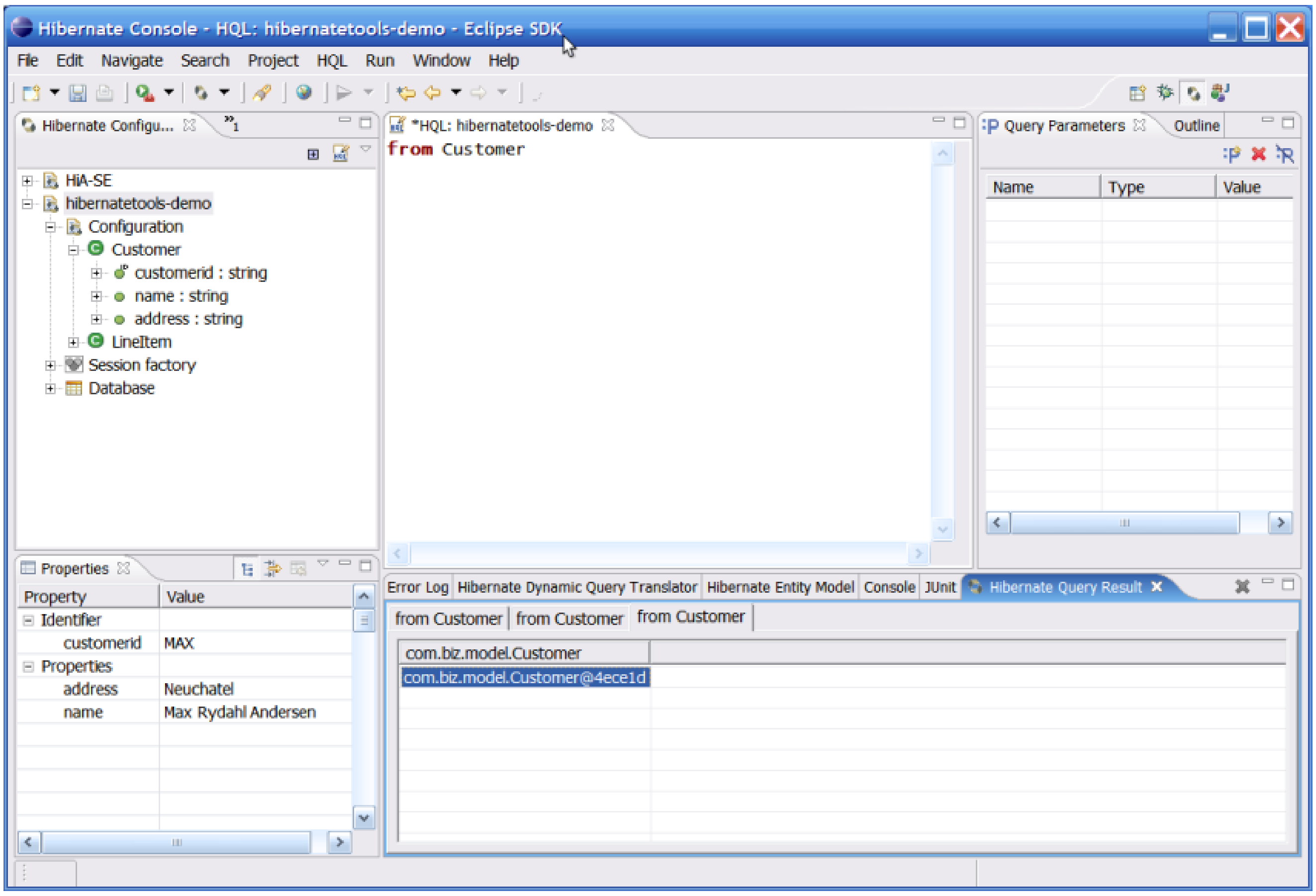
3.7. Hibernate Console perspective

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.

3.7.1. Viewing the entity structure

To view your new configuration and entity/class structure, expand the Hibernate Console configuration by clicking on the + icon.

Clicking on the small + symbol allows you to browse the class/entity structure and see the relationships.


Hibernate Console - HQL: hibernatetools-demo - Eclipse SDK
File Edit Navigate Search Project HQL Run Window Help
Hibernate Configu...
*HQL: hibernatetools-demo
Query Parameters
Outline
Name
Type
Value
Properties
Error Log
Hibernate Dynamic Query Translator
Hibernate Entity Model
Console
JUnit
Hibernate Query Result
from Customer
from Customer
from Customer

```
from Customer
```

```
HiA-SE
hibernatetools-demo
  Configuration
    Customer
      customerid : string
      name : string
      address : string
    LineItem
  Session factory
  Database
```

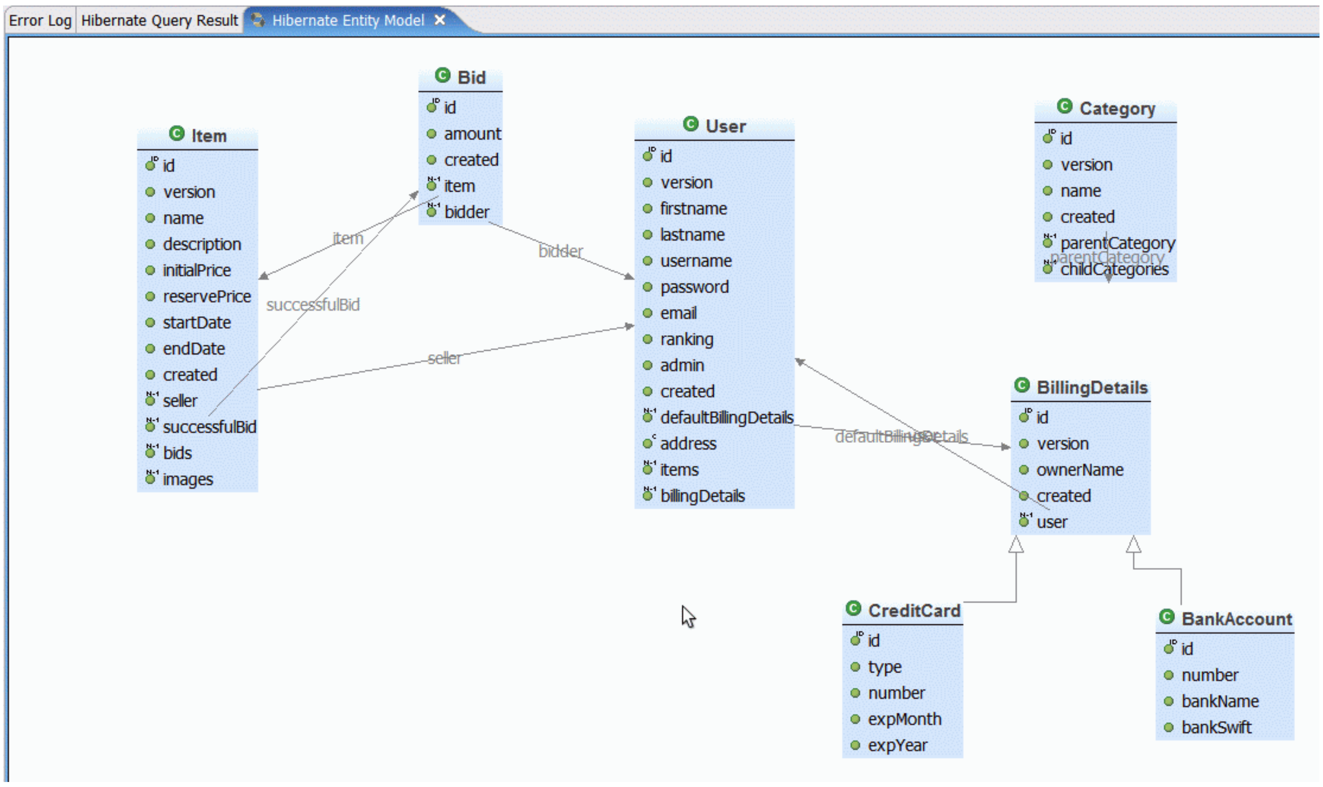
Property	Value
Identifier	
customerid	MAX
Properties	
address	Neuchatel
name	Max Rydahl Andersen

com.biz.model.Customer
com.biz.model.Customer@4ece1d

Hibernate Console perspective showing entity structure, query editor and result

3.7.1.1. Class Diagram

A class diagram is available in the view named "Hibernate Entity Model". It will show the model when the Configuration node in a Hibernate Console Configuration is selected.



This view supports zoom in/out and can also be printed. Zooming is done via the toolbar buttons in the view and printing is done by selecting the view and choose File/Print or use the Print Icon.

3.7.2. Prototyping Queries

Queries can be prototyped by entering them in the HQL editor. The HQL Editor is opened by right-clicking the Console configuration and select "HQL Scratchpad".

If the menu item is disabled then you need to first create an SessionFactory. That is done by right clicking the configuration and select "Create Session Factory" or by simpy expanding the Session Factory node.

Executing the query is done by clicking the green run button in the toolbar or pressing Ctrl+Enter.

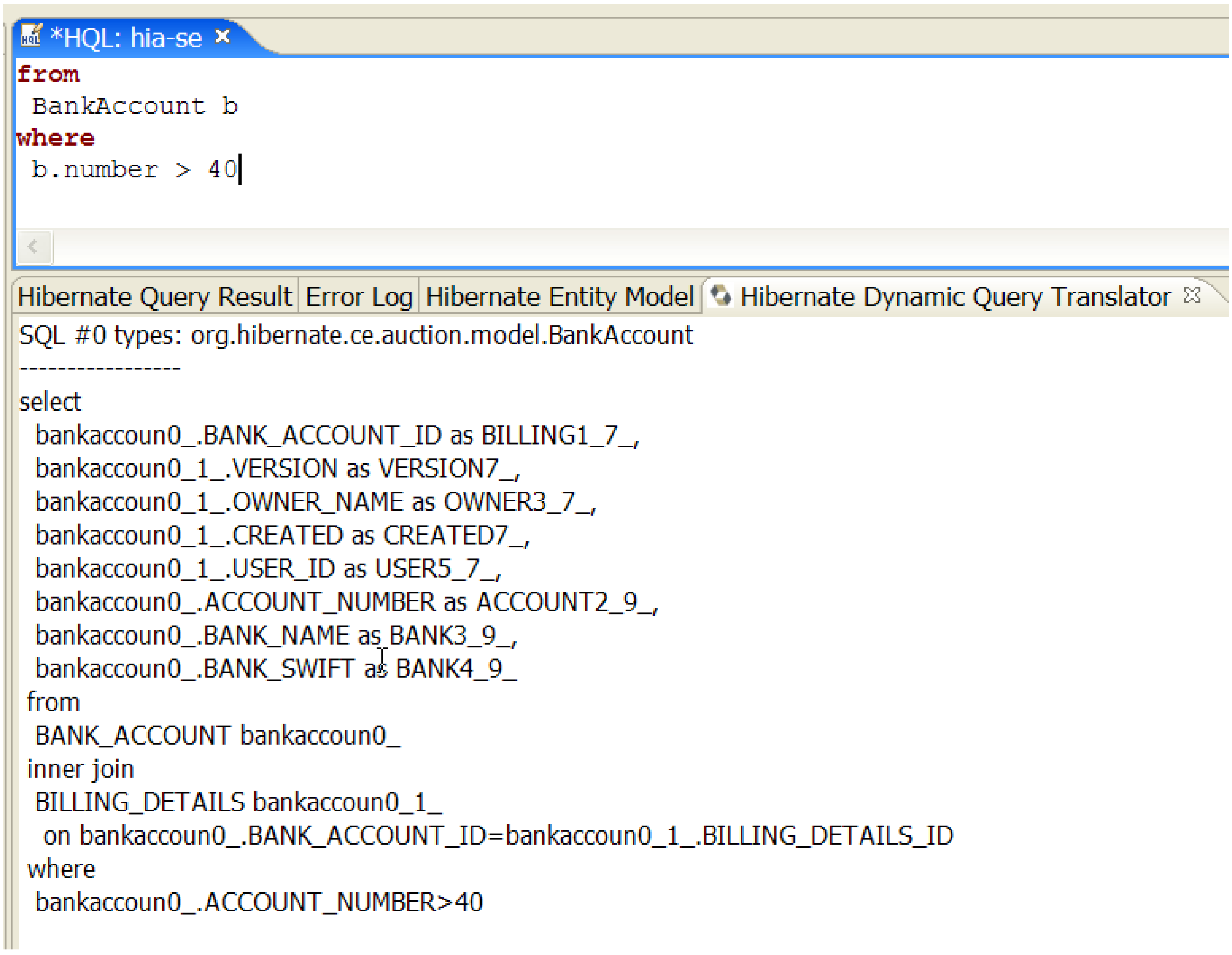
Errors during creation of the `SessionFactory` or running the queries (e.g. if your configuration or query is incorrect) will be shown in a message dialog or inlined in the view that detected the error, you may get more information about the error in the Error Log view on the right pane.

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

Tip: HQL queries are executed using `list()` and without any limit of the size of the output. Be careful if you execute a query on a large result set. You might run out of memory. This will be improved in a future version.

3.7.2.1. 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.



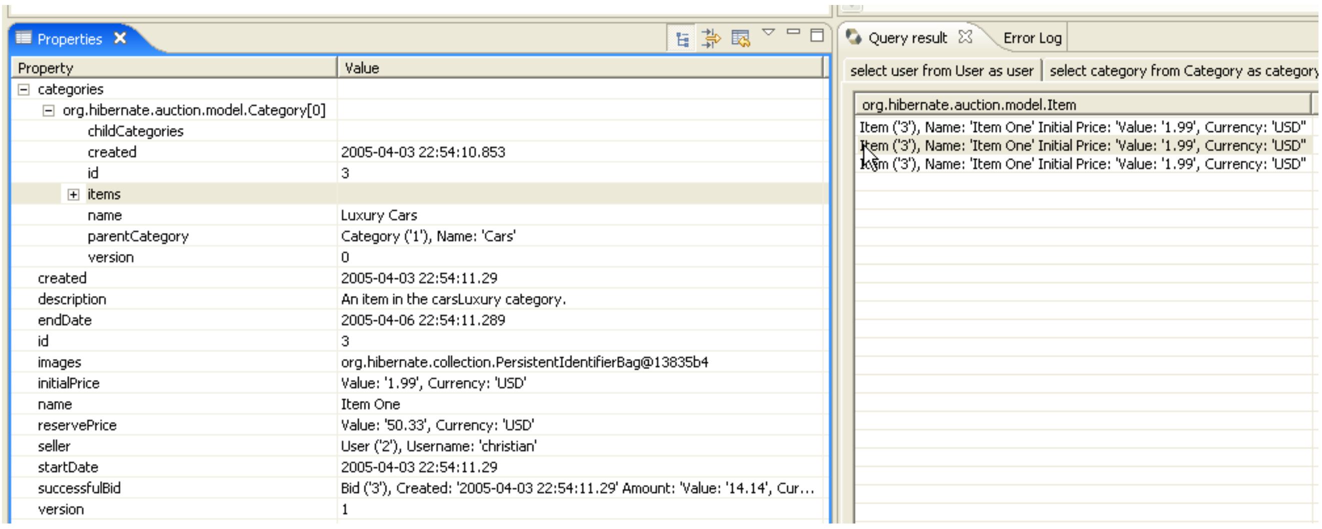
```
from
 BankAccount b
where
 b.number > 40
```

```
SQL #0 types: org.hibernate.ce.auction.model.BankAccount
-----------------
select
  bankaccoun0_.BANK_ACCOUNT_ID as BILLING1_7_,
  bankaccoun0_1_.VERSION as VERSION7_,
  bankaccoun0_1_.OWNER_NAME as OWNER3_7_,
  bankaccoun0_1_.CREATED as CREATED7_,
  bankaccoun0_1_.USER_ID as USER5_7_,
  bankaccoun0_.ACCOUNT_NUMBER as ACCOUNT2_9_,
  bankaccoun0_.BANK_NAME as BANK3_9_,
  bankaccoun0_.BANK_SWIFT as BANK4_9_
 from
  BANK_ACCOUNT bankaccoun0_
 inner join
  BILLING_DETAILS bankaccoun0_1_
   on bankaccoun0_.BANK_ACCOUNT_ID=bankaccoun0_1_.BILLING_DETAILS_ID
 where
  bankaccoun0_.ACCOUNT_NUMBER>40
```

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

3.7.3. Properties view

The properties view shows the structure of any selected persistent object in the results view. Editing is not yet supported.


Properties
Query result
Error Log
select user from User as user
select category from Category as category

Property	Value
categories	
org.hibernate.auction.model.Category[0]	
childCategories	
created	2005-04-03 22:54:10.853
id	3
items	
name	Luxury Cars
parentCategory	Category ('1'), Name: 'Cars'
version	0
created	2005-04-03 22:54:11.29
description	An item in the carsLuxury category.
endDate	2005-04-06 22:54:11.289
id	3
images	org.hibernate.collection.PersistentIdentifierBag@13835b4
initialPrice	Value: '1.99', Currency: 'USD'
name	Item One
reservePrice	Value: '50.33', Currency: 'USD'
seller	User ('2'), Username: 'christian'
startDate	2005-04-03 22:54:11.29
successfulBid	Bid ('3'), Created: '2005-04-03 22:54:11.29' Amount: 'Value: '14.14', Cur…
version	1

```
org.hibernate.auction.model.Item
Item ('3'), Name: 'Item One' Initial Price: 'Value: '1.99', Currency: 'USD''
Item ('3'), Name: 'Item One' Initial Price: 'Value: '1.99', Currency: 'USD''
Item ('3'), Name: 'Item One' Initial Price: 'Value: '1.99', Currency: 'USD''
```

3.8. 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.

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 for interesting categories and log4j documentation for how to configure logging via a log4j property file.

Chapter 4. Ant Tools

4.1. 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: until Hibernate 3.2 and related libraries are finally released there might be incompabilities with respect to the tools. Thus to avoid any confusion it is recommended to use the hibernate3.jar & 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. an Hibernate 3.1 project since the output generated will work with previous Hibernate 3 versions.

4.2. 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 a 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 from http://annotations.hibernate.org.

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

```
<hibernatetool
  destdir="defaultDestinationDirectory"               (1)
  templatepath="defaultTemplatePath"                  (2)
>                                                     (3)
  <classpath ...>
  <property key="propertyName" value="value"/>
  <propertyset ...>                                   (4)
  (<configuration ...>|<annotationconfiguration ...>| (5)
   <jpaconfiguration ...>|<jdbcconfiguration ...>)
  (<hbm2java>,<hbm2cfgxml>,<hbmtemplate>,...*)        (6)
</hibernatetool>
```

(1) `destdir` (required): destination directory for files generated with exporters.

(2) `templatepath` (optional): A path to be used to look up user-edited templates.
(3) `classpath` (optional): A classpath to be used to resolve resources, such as mappings and usertypes. Optional, but very often required.
(4) `property` and propertyset (optional): Used to set properties to control the exporters. Mostly relevant for providing custom properties to user defined templates.
(5) One of 4 different ways of configuring the Hibernate Meta Model must be specified.
(6) One or more of the exporters must be specified

4.2.1. Basic examples

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

```
<hibernatetool destdir="${build.dir}/generated">
 <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 `${build.dir}/generated` 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>
```

4.3. 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, statments execution against a database, error reporting or anything else that can be done in java code.

The following section decribes what the the various configuration can do, plus list the individual settings they have.

4.3.1. 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"              (1)
  propertyfile="hibernate.properties"                (2)
  entityresolver="EntityResolver classname"          (3)
  namingstrategy="NamingStrategy classname"          (4)
>
  <fileset...>                                       (5)

</configuration>
```

(1) `configurationfile` (optional): The name of a Hibernate configuration file, e.g. "hibernate.cfg.xml"
(2) `propertyfile` (optional): The name of a property file, e.g. "hibernate.properties"
(3) `entity-resolver` (optional): name of a class that implements org.xml.sax.EntityResolver. Used if the mapping files require custom entity resolver.
(4) `namingstrategy` (optional): 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.
(5) A standard Ant fileset. Used to include hibernate mapping files.Remember that if mappings are already specified in the hibernate.cfg.xml then it should not be included via the fileset as it will result in duplicate import exceptions.

4.3.1.1. Example

This example shows an example where no `hibernate.cfg.xml` exists, and a `hibernate.properties` + fileset is used instead. Note, that 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>
```

4.3.2. Annotation based Configuration (<annotationconfiguration>)

An `<annotationconfiguration>` is used when you want to read the metamodel from EJB3/Hibernate Annotations based POJO's. To use it remember to put the jars file needed for using hibernate annotations in the classpath of the `<taskdef>`.

The <annotationconfiguration> supports the same attributes as an <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>
```

4.3.3. JPA based configuration (<jpaconfiguration>)

An <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 an JPA compliant manner.

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

4.3.4. 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 from

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

```
<jdbcconfiguration
  ...
  packagename="package.name"                            (1)
  revengfile="hibernate.reveng.xml"                     (2)
  reversestrategy="ReverseEngineeringStrategy classname"(3)
  detectmanytomany="true|false"                         (4)
  detectoptmisticlock="true|false"                      (5)
>
  ...
</jdbcconfiguration>
```

(1) `packagename` (optional): The default package name to use when mappings for classes is created

(2) `revengfile` (optional): name of reveng.xml that allows you to control various aspects of the reverse engineering.

(3) `reversestrategy` (optional): 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.

(4) detectManytoMany (default:true): If true (the default) 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 has exactly two foreign-keys pointing to other entity tables and has no other columns.

(5) detectOptimisticLock (efault:true): If true columns named VERSION or TIMESTAMP with appropriate types will be mapped with the apropriate optimistic locking corresponding to `<version>` or `<timestamp>`

4.3.4.1. Example

Here is an example of using `<jdbcconfiguration>` to generate Hibernate xml mappings via `<hbm2hbmxml>`. The connection settings is here 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>
```

4.4. Exporters

Exporters is the parts that does 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.

4.4.1. 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"                                 (1)
 update="true|false"                                  (2)
 drop="true|false"                                    (3)
 create="true|false"                                  (4)
 outputfilename="filename.ddl"                        (5)
 delimiter=";"                                        (6)
 format="true|false"                                  (7)(8)
 haltonerror="true|false"
>
```

(1) export (default: true): Execute the generated statements against the database

(2) update(default: false): 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*)

(3) drop (default: false): Output will contain drop statements for the tables, indices & constraints

(4) create (default: true): Output will contain create statements for the tables, indices & constraints

(5) outputfilename (Optional): If specified the statements will be dumped to this file.

(6) delimiter (default: ";"): What delimter to use to separate statements

(7) format (default: false): Apply basic formatting to the statements.

(8) haltonerror (default: false): Halt build process if an error occurs.

4.4.1.1. 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>
```

4.4.2. POJO java code exporter (`<hbm2java>`)

<hbm2java> is a java codegenerator. Options for controlling wether JDK 5 syntax can be used and wether the POJO should be annotated with EJB3/Hibernate Annotations.

```
<hbm2java
 jdk5="true|false"                                    (1)
 ejb3="true|false"                                    (2)
>
```

(1) jdk (default: false): Code will contain JDK 5 constructs such as generics and static imports

(2) ejb3 (default: false): Code will contain EJB 3 features, e.g. using annotations from javax.persistence and org.hibernate.annotations

4.4.2.1. 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>
```

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

```
<hbm2hbmxml/>
```

4.4.3.1. 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 is implemented (contributions welcome), so some hand editing might be necessary.

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

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

<hbm2cfgxml> generates a hibernate.cfg.xml. Intended to be used together with a <jdbcconfiguration> when performing reverse engineering, but 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"                                  (1)
/>
```

(1) ejb3 (default: false): the generated cfg.xml will have <mapping class=".."/>, opposed to <mapping resource="..."/> for each mapping.

4.4.5. Documentation exporter (`<hbm2doc>`)

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

```
<hbm2doc/>
```

4.4.6. Query exporter (<query>)

<query> is used to execute a HQL query statements and optionally send the output to a file. 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 etc.

4.4.6.1. Examples

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 that 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>
```

4.4.7. Generic Hibernate metamodel exporter (`<hbmtemplate>`)

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

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

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

4.4.7.1. Exporter via `<hbmtemplate>`

The following is an example of reverse engineering via `<jdbcconfiguration>` and use 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>
```

4.5. Using properties to configure Exporters

Exporters can be controlled by user properties. The user properties is specificed via `<property>` or `<propertyset>` and each exporter will have access to them directly in the templates and via `Exporter.setProperties()`.

4.5.1. `<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>`. The functionallity of `<propertyset>` is explained in detail in the Ant task manual.

4.5.2. Getting access to user specific classes

If the templates need to access some user class it is 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 usefull to delegate logic and code generation to java code instead of placing such logic in the templates.

4.5.2.1. 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>
```

Chapter 5. Controlling reverse engineering

When using the <jdbcconfiguration> the ant task will read the database metadata and from that 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 the 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.

5.1. 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.

5.2. 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 (its just basic XML) or you can use the Hibernate plugins which have a specialized editor.

Note: many databases is 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.

The following is an example of a reveng.xml. Following the example is 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 fom 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 precisly -->
 <sql-type jdbc-type="NUMERIC"  precision='1' hibernate-type="boolean" />
 <!-- the type-mappings are ordered. This mapping will be consulted last,
  thus overriden 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
```

```
     are 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>
</table>

</hibernate-reverse-engineering>
```

5.2.1. Schema Selection (<schema-selection>)

`<schema-selection>` is used to drive which schema's 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=".*"/>
```

5.2.1.1. 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 process 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_.*"/>
```

5.2.2. 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 specificed 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.

5.2.2.1. 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 5.1. sql-type examples

Column	jdbc-type	length	precision	not-null	Resulting hibernate-type	Rationale
ID	INTEGER		10	true	int	Nothing defined for INTEGER. Falling back to default behavior.
NAME	VARCHAR	30		false	your.package.TrimStringUserType	No 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 specifc type-mapping for VARCHAR with not-null="false" is chosen. The first VARCHAR sql-type matches in length but has

Column	jdbc-type	length	precision	not-null	Resulting hibernate-type	Rationale
						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"

5.2.3. Table filters (<table-filter>)

The <table-filter> let you specifcy 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 specifc prefix.

```
<table-filter
 match-catalog="catalog_matching_rule"                (1)
 match-schema="schema_matching_rule"                  (2)
 match-name="table_matching_rule"                     (3)
 exclude="true|false"                                 (4)
 package="package.name"                               (5)
/>
```

(1) match-catalog (default: .*): Pattern for matching catalog part of the table
(2) match-schema (default: .*): Pattern for matching schema part of the table
(3) match-table (default: .*): Pattern for matching table part of the table
(4) exclude (default: false): if true the table will not be part of the reverse engineering
(5) package (default: ""): The default package name to use for classes based on tables matched by this table-filter

5.2.4. Specific table configuration (<table>)

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

```
<table
 catalog="catalog_name"                               (1)
 schema="schema_name"                                 (2)
 name="table_name"                                    (3)
 class="ClassName"                                    (4)
>
 <primary-key.../>
 <column.../>
 <foreign-key.../>
</table>
```

(1) catalog (Optional): Catalog name for table. Has to be specified if you are reverse engineering multiple catalogs or if it is not equal to `hiberante.default_catalog`
(2) schema (Optional): Schema name for table. Has to be specified if you are reverse engineering multiple

schemas or if it is not equal to `hiberante.default_schema`

(3) name (Required): Name for table

(4) clase (Optional): The class name for table. Default name is camelcase version of the table name.

5.2.4.1. <primary-key>

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

```
<primary-key
 <generator class="generatorname">                    (1)
   <param name="param_name">parameter value</param>   (2)
 </generator>
 <key-column...>                                      (3)
</primary-key>
```

(1) generator/class (Optional): defines which identifier generator should be used. The class name is any hibernate short hand name or fully quailfied class name for an identifier strategy.

(2) generator/param (Optional): Allows to specify which parameter with name and value should be passed to the identifier generator

(3) key-column (Optional): Specifies which column(s) the primary-key consists of. A key-column is same as column, but does not have the exclude property.

5.2.4.2. <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 and finally it is possible to completely exclude a column from processing.

```
<column
 name="column_name"                                   (1)
 jdbc-type="java.sql.Types type"                      (2)
 type="hibernate_type"                                (3)
 property="propertyName"                              (4)
 exclude="true|false"                                 (5)
/>
```

(1) name (Required): Column name

(2) jdbc-type (Optional): Which jdbc-type this column should be processed as. A value from java.sql.Types, either numerical (93) or the constant name (TIMESTAMP).

(3) type (Optional): Which hibernate-type to use for this specific column.

(4) property (Optional): What property name will be generated for this column.

(5) exclude (default: false): set to true if this column should be ignored.

5.2.4.3. <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 and one-to-many's).

Note

```
<foreign-key
  constraint-name="foreignKeyName"                    (1)
  foreign-catalog="catalogName"                       (2)
  foreign-schema="schemaName"                        (3)
  foreign-table="tableName"                          (4)
 >
 <column-ref local-column="columnName" foreign-column=(5)"foreignColumnName"/>
 <many-to-one                                        (6)
   property="aPropertyName"
   exclude="true|false"/>                            (7)
 <set
   property="aCollectionName"
   exclude="true|false"/>
</foreign-key>
```

(1) constraint-name (Required): Name of the foreign key constraint. Important when naming many-to-one and set. It is the constraint-name that is used to link the processed foreign-keys with the resulting property names.

(2) foreign-catalog (Optional): Name of the foreign table's catalog. (Only relevant if you want to explicitly define a foreign key)

(3) foreign-schema (Optional): Name of the foreign table's schema. (Only relevant if you want to explicitly define a foreign key)

(4) foreign-table (Optional): Name of the foreign table. (Only relevant if you want to explicitly define a foreign key)

(5) column-ref (Optional): 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)

(6) many-to-one (Optional): 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.

(7) set (Optional): 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.

5.3. 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 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);
  }
 }
}
```

5.4. 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.

Chapter 6. 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 through the <meta> tag in the mapping files. The following section describes the possible meta tags and their use.

6.1. 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 6.1. 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.
`implements`	interface the class should implement
`extends`	class the 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 initializatioin 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 specificed, 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.

6.1.1. Recomendations

The following are some good practices when using `<meta>` attributes.

6.1.1.1. 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 recomend 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>
```

6.1.1.2. Be aware of putting at class scope level `<meta>` attribute `use-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 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.

See http://www.hibernate.org/109.html for an more in-depth explanation on the subject of equals() and hashcode().

6.1.2. 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.

6.1.2.1. Generate pre/post-conditions for methods

With an `<meta attribute="class-code">`, you can add addional methods on a given class, nevertheless such `<meta>` attribute can not be used at property scope level and Hibernatetools does not provide such `<meta>` attributes.

A possibly solution for this is to modify the freemarker templates responsable 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 named 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)} ${pojo.getGette
        return this.${property.name};
    }

    ${pojo.getPropertySetModifiers(property)} void set${pojo.getPropertyName(property)}(${pojo.getJava
        this.${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)} ${pojo.getGette
        return this.${property.name};
    }

    ${pojo.getPropertySetModifiers(property)} void set${pojo.getPropertyName(property)}(${pojo.getJava
      <#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 examlpe let us add a pre-condition for property `name` preventing no `Person` can have an empty name. So 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"><![CDATA[
      if ((firstName != null) && (firstName.length() == 0) ) {
        throw new IllegalArgumentException("firstName can not be an empty String");
      }]]>
      </meta>
  </property>
</class>
</hibernate-mapping>
```

Notes: i) If you don't use `<[[CDATA[]]>` you have to scape the & symbol, i.e.: &amp; 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) && (name.length() == 0)) {
            throw new IllegalArgumentException("name can not be an empty String");
        }
        this.name = name;
    }
}
```