Towards Model Driven Architecture and Painless Persistence with Airlift, UML and Hibernate

Milan Zimmermann, Airlift LLC

Last revision September 20, 2005

Summary: Benefits of a "data model first" approach to development are well established. In many projects we worked on, we have encountered a need for a consistent domain model driven approach with the requirement of the persistence to be a relational database. While many technologies (Hibernate, EJB) address this problem, they do not provide a seamless easy way to drive the development process from a Domain Entity Model (DEM) designed in UML all the way to persistence. Hibernate is an excellent Object-Relational(OR) mapping tool, however, the task of creating and managing details of Hibernate OR mapping XML and corresponding Java classes may become overhelming for large projects. The LGPL Airlift framework provides, among other features, a Model Driven Approach (MDA) and tools which allow to design the Domain Entity Model in UML, including inheritance, generate all Hibernate OR mapping details, and support evolution of the Entity Model by separating any generated classes and interfaces from business classes and interfaces.

Available formats:

HTML
PDF
doc
Open Office 2.0

Fast Introduction: The goal of this tutorial is to present the Airlift library and tools which support MDA development, from UML Domain Entity Model to persistence and business logic. Let us start by jumping ahead and taking a look at the AirliftPetStore Entity Model (Customers and PurchaseOrders class diagrams). A few notes:

- The Person / Customer and Person / SupplierContact generalizations (extensions) will illustrate how inheritance is handled.
- The association between Person and Phone, which will exist also between Customer and Phone because associations, while uniquely defined by enumerates in Airlift, are implemented by methods (add/remove from association) that are simply inherited.
- The <<EntityType>> Stereotype on top of each Entity, they are required for each entity to be converted from the UML.
- The OR and Hibernate tags such as LazyLoadTo, CascadeTo, FKTo etc, control some details in the way Hibernate mappings are generated. They are only needed if a detail control is required over the generated relational tables. We will discuss them later.

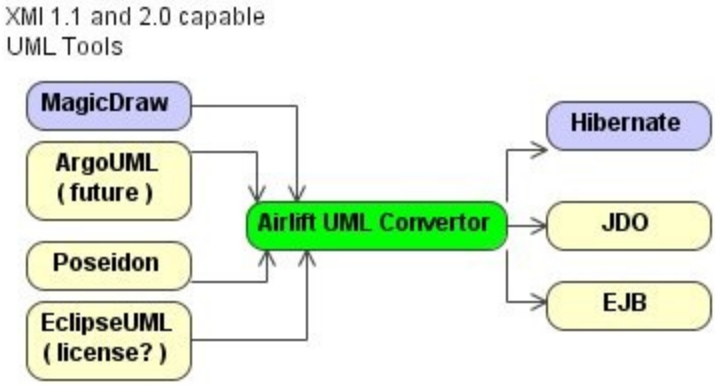
Tutorial Content: In this tutorial, we will build a small subset of the above AirliftPetStore UML Entity Model, and illustrate the Airlift MDA approach by generating all java and Hibernate classes using the AirliftUMLConverter, creating the target database using Airlift Hibernate tools, and coding a simple no-gui test application against the target database.

Tutorial Steps:

Introduction to the environment: UML and persistence tools and libraries currently supported by Airlift.

In this section we introduce the tools tested and currently supported by Airlift.

Airlift plans to support following tools:



At this point, following tools have been tested and used by customers:

- MagicDraw version 9.0 and 9.5 files saved as "rich" XMI 1.1
- Hibernate 2.0, 3.0

Any tool which allows saving the UML diagram in XMI 1.1 should work. Currently, Airlift users use MagicDraw version 9.0 and 9.5 (http://www.nomagic.com), saving the UML in XMI 1.1 "rich" format, which is the only combination that has been tested. Poseidon (http://www.gentleware.com) may work but was not tested. ArgoUML (http://argouml.tigris.org/) is currently not supported because the highest version of XMI it can save is 1.0. We want to support an open source or free source UML tool and looking into ArgoUML and Omondo/EclipseUML (http://www.omondo.com/download/free/eclipse_3x/index.html#E3.JAR) but need to clarify EclipseUML license.

Generating EJB and JDO is planned.

We would like to extend the tool support as quickly as possible and help anyone in testing and extending list of source and target tools, mainly free source and open source tools.

Installing software and tools used in this tutorial.

In this section, we list the tools used in this tutorial and provide download links so reader can follow this tutorial hands on.

However, this tutorial provides:

- The tutorial UML diagram as image.
- Source files generated from the entity model and the sample application in a separate "src" directory.

Consequently, you only need to install the tools below if you wish to follow this tutorial "hands-on" step by step.

External tools:

To follow this tutorial "hands-on" step by step, you will need to install:

- MagicDraw 9.5 community edition, downloadable from http://www.magicdraw.com/main.php?ts=download_demo&cmd_show=1&menu=download
- MySQL community edition downloadable from http://dev.mysql.com/downloads/ Any database will work, but the tutorial provides a MySQL jar to run hands-on.
- Eclipse 3.1 downloadable from http://www.eclipse.org. The tutorial code can run in any IDE such as Netbeans or IDEA or from command line, but only Eclipse workspace is provided as convenience.
- The Java IDE http://java.sun.com/j2se/1.4.2/download.html if not installed.
- Hibernate 2.0 is not necessary as we provide Hibernate jars with the Eclipse workspace

Tutorial Software:

- Eclipse Workspace in the eclipse-workspace.zip file includes everything to run the tutorial code except for a relational database which needs to be installed separately. To run the Airlift UML Generator and the sample application, unzip this file, start Eclipse 3.1 and select "eclipse-workspace" as the workspace. This directory also contains the Airlift source and classes jars.
- The entity model build in this tutorial.
- The "*tutorial/src*" directory with both generated files and sample application. This is simply a copy from inside the "eclipse-workspace", provided for those who do not wish to run "hands on"and simply review all code.

Core Section: Building the AirliftPetstoreSubset UML using the Airlift UML Profile.

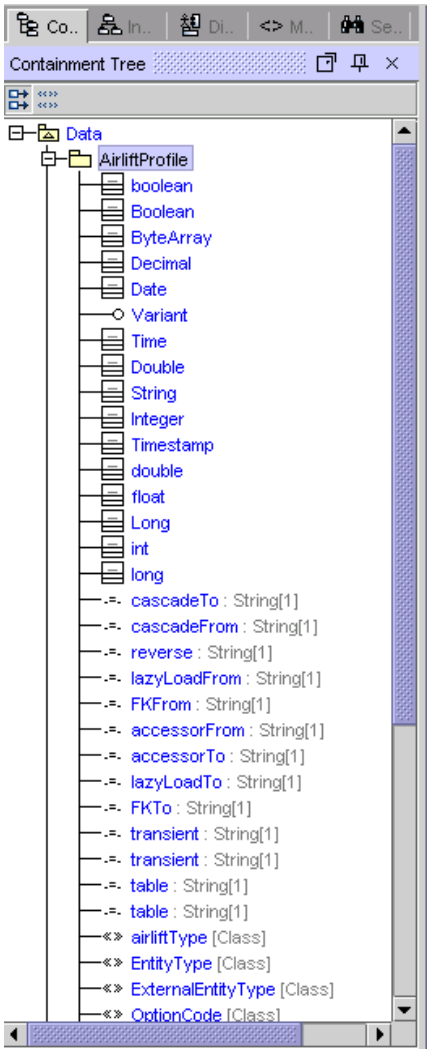
In this section, we describe how to build the Domain Entity Model for AirliftPetstoreSubset. This is a small subset of the Airlift Petstore linked as an image in "Fast Introduction".

Steps to build AirliftPetstoreSubset:

- From MagicDraw, create a new project by clicking File → NewProject.

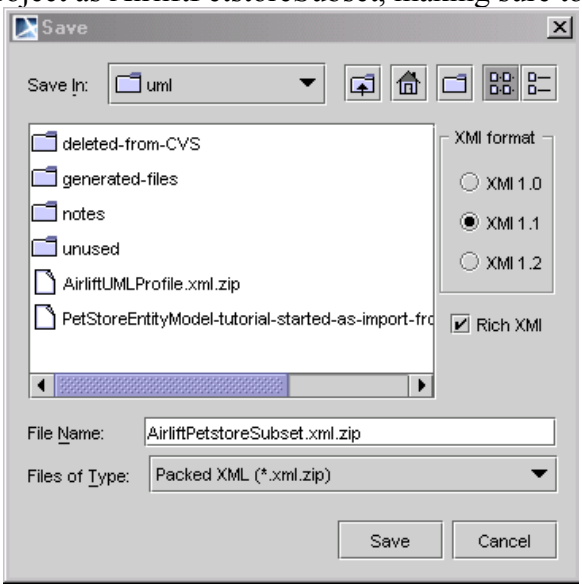
- All Airlift-specific features must be provided by the AirliftUMLProfile.xml.zip profile. This is achieved by importing the profile into the UML tool. Click File→ Import and in the dialog brought up, select the AirliftUMLProfile.xml.zip and click "Open". This will effectively import the AirliftUMLProfile.xml.zip and make the Airlift-specific tags and stereotypes ready to use.

- Once "imported", you should see AirliftProfile under Data on the left tree in MagicDraw as shown here:

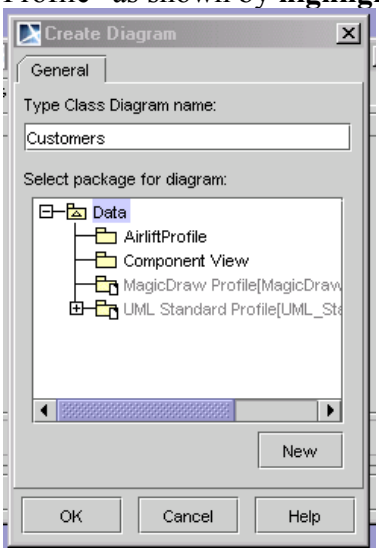


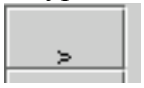
- The imported AirliftProfile contains model for all entity types and Tags supported by the Airlift UML Converter. They will be discussed in detail later, at this point only a note: In the steps below, when adding entity types, field types and tags to the UML, all of them must be from the above profile. This will ensure the convertor will understand those features and know to how to convert them.

- Save project as AirliftPetstoreSubset, making sure to save in XMI 1.1 format "rich":

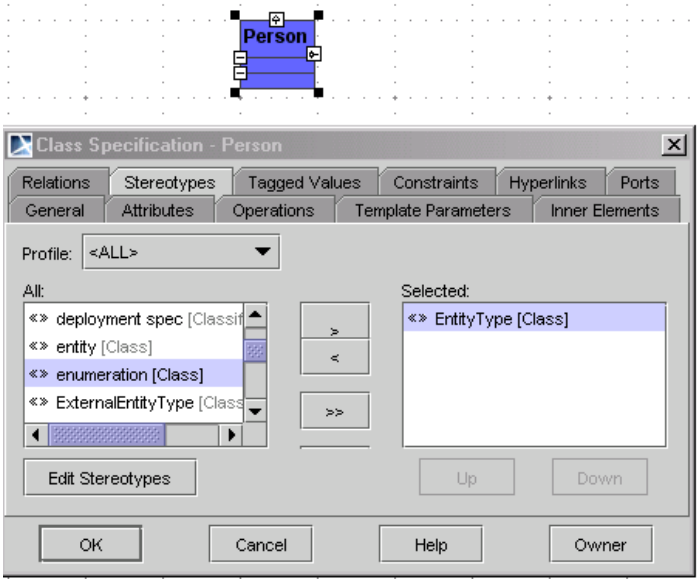


- We start building the entity model by adding entities, attributes (fields) and associations. We will create a subset of the PetStore model which uses limited control over specific naming of the target database tables by adding tags that control foreign key names, lazy loading etc (FKFrom, FKTo, LazyLoadFrom LazyLoadTo etc).
    - Create a Class Diagram (Diagrams→Class Diagram) and name it Customers. Ensure the Class Diagram is added under "Data" not "Airlift Profile" as shown by **highlighted the Data node**:

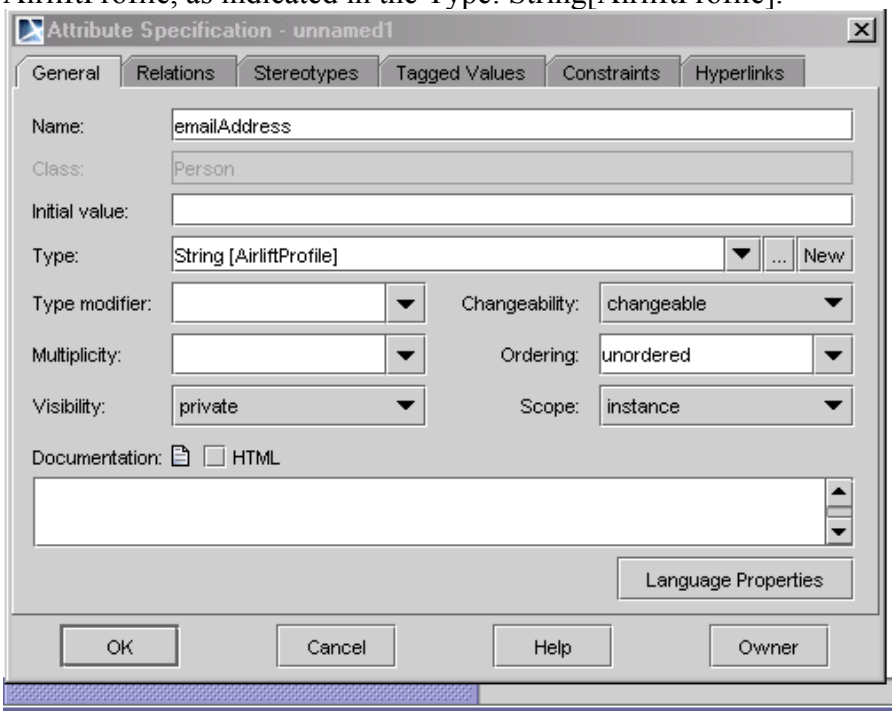


    - Drag the Class icon on the Class Diagram and name it **Person**.
    - Double click on the Person Node and select Stereotypes. On the left, select EntityType and click on the right arrow symbol  in the middle.

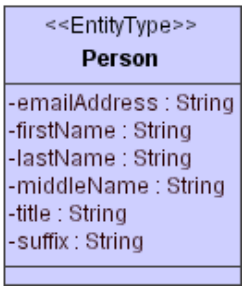
This will move the EntityType stereotype to the right pane, resulting in



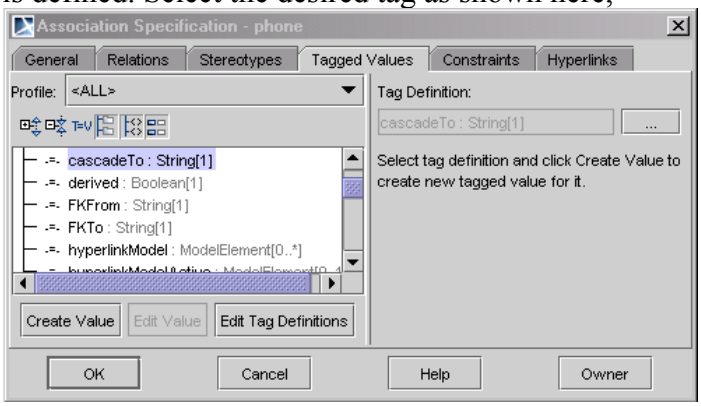
- Start adding attributes by clicking on Attributes and Add. Provide Attribute name and make sure that the type selected is from the AirliftProfile, as indicated in the Type: String[AirliftProfile].



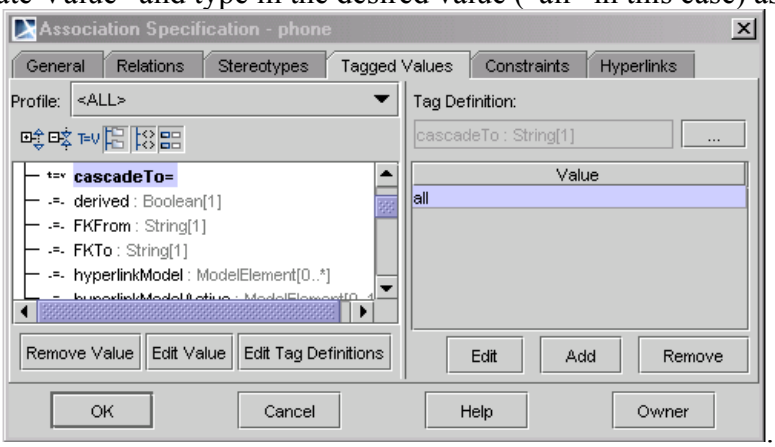
- The resulting first entity:



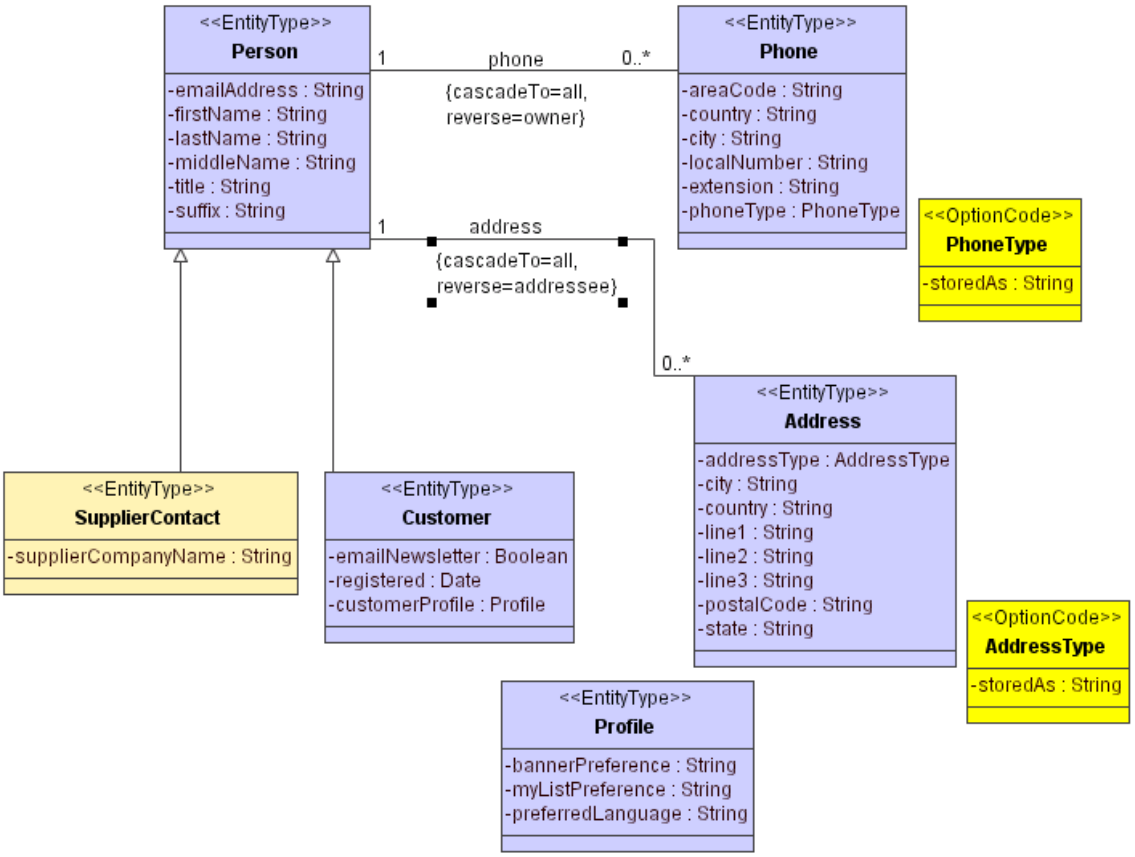
- Add other entities, fields and associations to the UML project, making sure each entity is of type <<EntityType>>. The Hibernate target Airlift UML Converter supports converting UML Entities and Fields (Attributes) by generating target persistable classes, controlling the Hibernate target using UML Tags on associations and classes.
- Let us assume we added a Person and Phone entities and a association named "phone" between them. At this point, we will discuss two details:
  - A UML class that is given stereotype <<OptionCode>> is treated in a special way by Airlift. It is assumed it's values will not change often, they reside in a standalone option code entity and are cached. The only allowed field on such class is "storedAs" and allowed types are String and Integer (this represents how they are stored). In the UML, note how the phoneType field is defined.
  - Adding a AirliftProfile UML tag named "cascadeTo" on the "phone" association. The reason we show this is to clarify how a AirliftProfile tag is defined. Select the desired tag as shown here,



then click on

"Create Value" and type in the desired value ("all" in this case) as shown here 
.

- The [final subset of the AirliftPetstore UML](#) we will write code against in MagicDraw 9.5 and a screenshot:



- Apart from <<OptionCode>> discussed above, let us notice a few things:
  - The "reverse" tag on the Person→Phone and Person→Address associations. They allow for Person to have accessors from Phone and Address.
  - Extension (Generalization) use, Customer and SupplierContract extending Person. Both Customer and SupplierContract will inherit associations to Phone and Address.

- There is no UML association line drawn from Customer to Profile, however, Customer has "customerProfile" field of type Profile. This is similar to having an association from Customer to Profile that is "multiplicity 1" on Customer, and "multiplicity 0,1" on Profile, but not the same: The Profile will be implemented "inline" in Customer, such that Profile fields will be added to the Customer table. Obviously this is only possible for ONE_TO_ZERO-OR-ONE associations, and such setup can increase performance in some situations.

Having created the domain entity model, we will first (briefly) discuss Airlift UML tags and stereotypes, and then use the Airlift UML Converter to convert the entity model into Java code and Hibernate mappings.

Detail description of tags supported by the Airlift UML Convertor.

In this section, we will discuss Tags supported by the Airlift UML Converter. Airlift UML tags and stereotypes are defined in the AirliftUMLProfile.xml.zip. In the above model, we have seen tags such as "reverse" which manages association in reverse direction, and "cascadeTo" which manages how the "To" entity is being deleted on the "From" entity delete.

In general, with UML tags, designer can control association accessor types (Set, List, Map), concrete column names for foreign keys, cascading delete, lazy loading and other OR mapping features and features supported by Hibernate. While none of these tags are required and reasonable defaults are provided by either Airlift or Hibernate, tags are a powerful feature providing fine control.

All Airlift UML tags and stereotypes, along with their description, default and allowed values are listed in the Tags documentation. Please review it for details of supported tags.

Using the Airlift UML Convertor to generate java classes, interfaces and Hibernate XML mapping files, overview of generated code.

In this section, we will explain how to use the **Airlift UML Convertor** to convert UML to source code, overview the generated code, and discuss role of each generated class, as well as how this model supports maintaining your business changes.

The **Airlift UML Convertor** (Hibernate target) is the utility which takes the UML file (XMI 1.1 format), such as the AirliftPetstoreSubset we built in previous steps, and

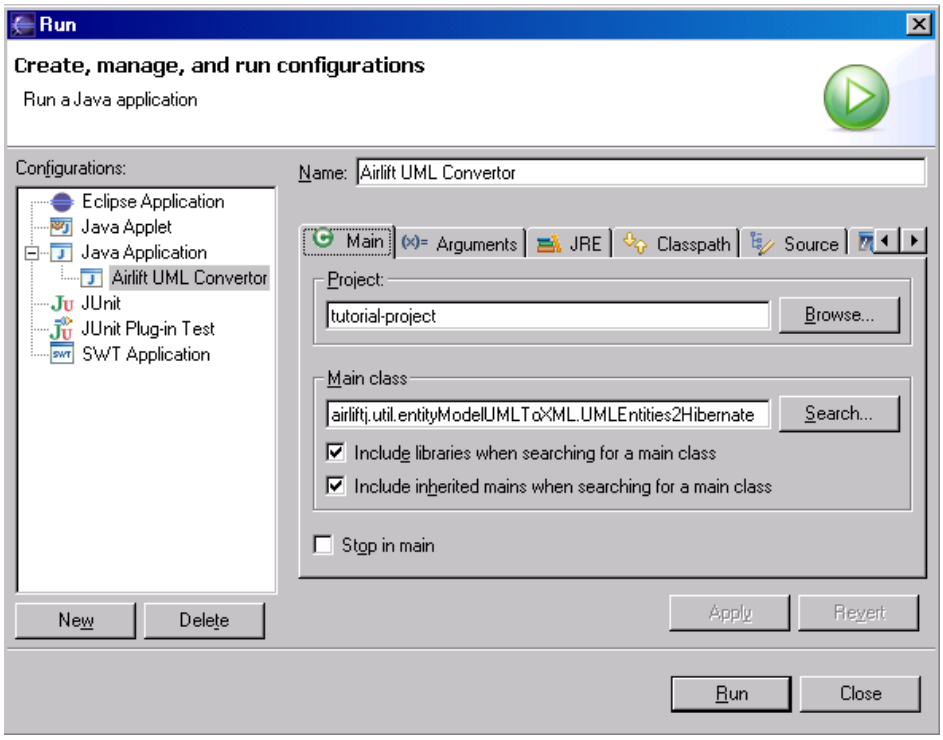
converts it into a set of Java files and Hibernate mapping files, which support persistence in the Airlift framework. This utility, along with Airlift persistence classes, is the core that allows for applications using Airlift to be based on a Model Driven approach to development.

Running the Airlift UML Convertor

The utility can be run by running "main" of utility class *airliftj.util.entityModelUMLToXML.UMLEntities2Hibernate*, found in the airlift_src.jar in the eclipse-workspace of this tutorial. This utility suports command line arguments that allow to specify the location of the UML file, location of the generated files, as well as properties which control the generated code (for example, whether the target relational tables will use char or varchar). This utility will be soon run from Eclipse plugin, so for now we will concentrate on describing a most common arguments.

For those following the tutorial step-by-step in Eclipse:

- Start Eclipse, and select the "eclipse-workspace" provided. Click "Run→Run" in the menu



- There are 4 Arguments used: *../../uml AirliftPetstoreSubset.xml.zip src petstore.entity airliftj.persistence* . (All path strings are relative to the "airlift-workspace/tutorial-project" directory.)
    - first *../../uml AirliftPetstoreSubset.xml.zip* defines location of the UML
    - second *src* defines directory where the files will be generated into
    - third *petstore.entity* is the package name of the generated entities

    - fourth *airliftj.persistence* is name of the airlift persistence package (more on it later).
- Click on the Run button to convert the UML entity model. The log should be similar to this log.
- If you receive an error running the converter, you may want to look at this likely error causes, NoteOnUMLConversionErrors.txt
- Right-click on the "tutorial-project" in Package Explorer, and select "Refresh", this will ensure the generated files are read by Eclipse.

Overview of generated code

For convenience of a quick review, the generated Java classes and Hibernate XML mapping were copied into "*tutorial/src*" directory. For those following hands-on they are in the eclipse-workspace as well.

Each UML Class has been converted into 4 java files and 1 Hibenate mapping ".hbm.xml" file (there is no "hbm.xml" file generated for classes that have a superclass, the superclass's "hbm.xml" file contains all mappings. As an example, let us look at the UML Person class. Here is description of each generated file and it's role in the Airlift framework.

- The "**entity interface**" petstore.entity.gen._intfc_Person.java is an interface with 2 roles.
    - By extending the Airlift entity interface, *airliftj.persistence.Entity*, it allows Person to participate in the Airlift persistence framework. This represents the "entity interface" role.
    - By defining accesors for each field in the UML Person class, and for each association there, it serves as the entity model API.
- The petstore.entity.gen._class_Person.java provides **skeleton entity implementation** of the "entity interface" , or the 2 roles defined by _intfc_Person:
    - By extending the *airliftj.persistence.BaseEntity* it provides the skeleton implementation of any Airlift persistence framework participant.
    - By providing implementation for all accessors defined in the UML Person class, it serves as the entity model implementation.
- The petstore.entity.Person.java is the "**business interface**" of Person, a direct extension of _intfc_Person. Person.java has the role of allowing the "business entity implementation" to evolve without disrupting existing business-specific code changes. Please see PersonImpl.java for more discussion.
- The petstore.entity.impl.PersonImpl.java is the "**business implementation**" of Person, extension of _class_Person. Any business logic related to Person's role in the application should be implemented in the "business implementation". Also, this allows for the entity model to evolve without disrupting existing business-specific code changes.

    - Let us discuss an example. Let's assume an initial entity model was converted, and the resulting classes used in an application. Along the way, a need to add some business logic came up. Let us make it simple, by assuming this business requirement is that the middle name always need to be empty or one letter. This would be implemented by overriding the "setMiddleName(String pMiddleName)" method of ._classPerson. Let's say that later still, a field need to be added to the Person class. This is performed by adding a field in the UML entity model and re-converting. The business classes are not overriden in this process and existing "middle name functionality" is preserved. If the last change involved removing a field, no problems arise unless this old field is refererenced somewhere in the business code, in which case a compile error would alert to the fact that removed field is used in business code.
    - This seamless evolution of entity model is allowed by two features: first, the separation of the "entity" and "business" implementation and interfaces, second, the simple fact the implementation classes are not overwritten during reconversion.
    - A good practice is to write code against the "business interface" Person.java and put implementaition changes in the "business implementation" PersonImpl.java.
    - Upon re-conversion, the "business classes" Person.java and PersonImpl.java are not overwritten, the "entity classes" *intfc*Person, *class*Person are overriden, the latter should not be modified, because changes are lost. This is signified by their presense in the package name ending with "gen".
-
- The petstore/entity/Person.hbm.xml contains all generated Hibernate mappings for Person. Details are controlled by the tags discussed in previous sections. We find that especially in more complex entity models, ability to fully generate (and regenerate on changes!) Hibernate mappings is very valuable and increases dynamicity of the development process..
- The petstore.entity.gen._registry_allEntities.java is one "common" file per UML. It's role is to register all persistent classes with the Airlift PersistentObjectRegistry.

Running the Airlift Deploy Tool to create the target database.

In this section, we will explain what is the **Airlift Entity Model Deployment Tool** and how to use it.

Airlift Entity Model Deployment tool is a utility (soon to be generated , you can check the source code in PetstoreDeployTool.java) which can create all database tables corresponding to entities defined in _registry_allEntities.java (which in turn correspond to all Classes and Associations in the UML entity model. It is a wrapper using Hibernate utilities.

The Deploy Tool must run at least once before running any application that uses the database as modelled in the UML converted in previous steps.

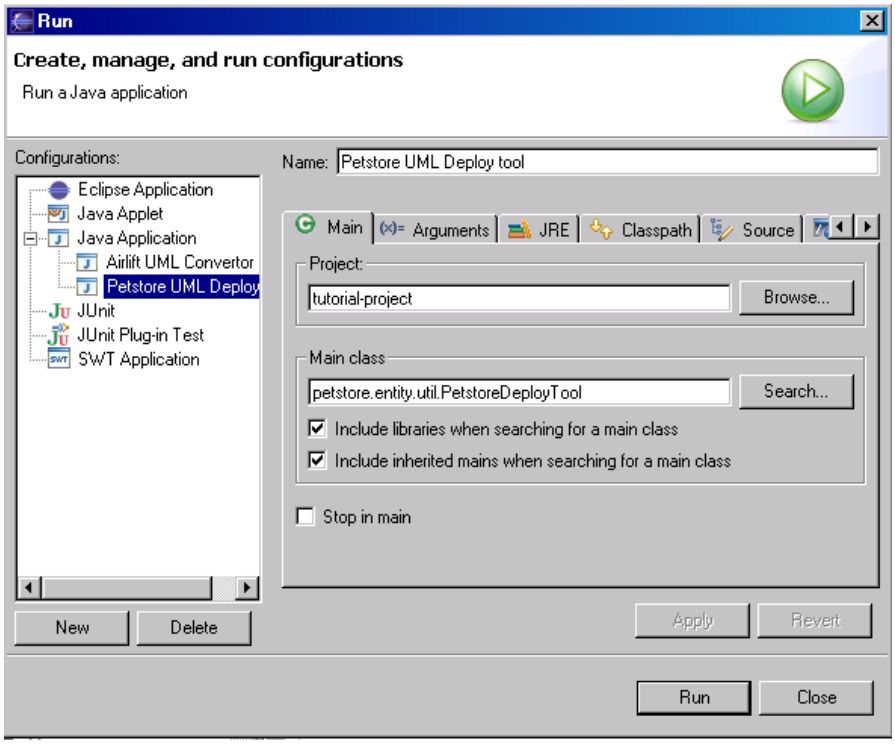
Configuration Files required to run the Deployment Tool:

- **Airlift-config.xml** defines which EntityManager factory is used
- **Hibernate.properties** defines database URL and driver, login and password. If you follow these instructions step by step, please note that at this point, the MySQL database must be installed, and by default be named "test" with login="test" and no password. If your database uses different configuration, please modify hibernate.properties.
- **Log4j.properties**

Java files required to run the Deployment Tool:

- PetstoreEntityManagerFactory.java: makes persistent objects and command objects types (entity types).
- Petstore_HQLBasedCommandRegistry.java: registers query commands. This is only required later inbuilding an application.
- Petstore Persistent Object Registry: generated, registers persistent object types.

If you are following the tutorial step by step, you can run the deployment tool:

- First click Run→Run
- 

- Next, you should see a log with some INFO but no errors and exceptions, similar to this log.
- Once the run finishes, you can use a database utility to check what tables were created. As an example, after running the tool, there will be a database table named "person" with following fields:

```
id                  bigint
emailAddress        varchar
title               varchar
suffix              varchar
firstName           varchar
middleName          varchar
lastName            varchar
emailNewsletter     tinyint
registered          date
bannerPreference    varchar
supplierCompanyName varchar
myListPreference    varchar
preferredLanguage   varchar
updateSequence      int
CONCRETE_TYPE       varchar
```

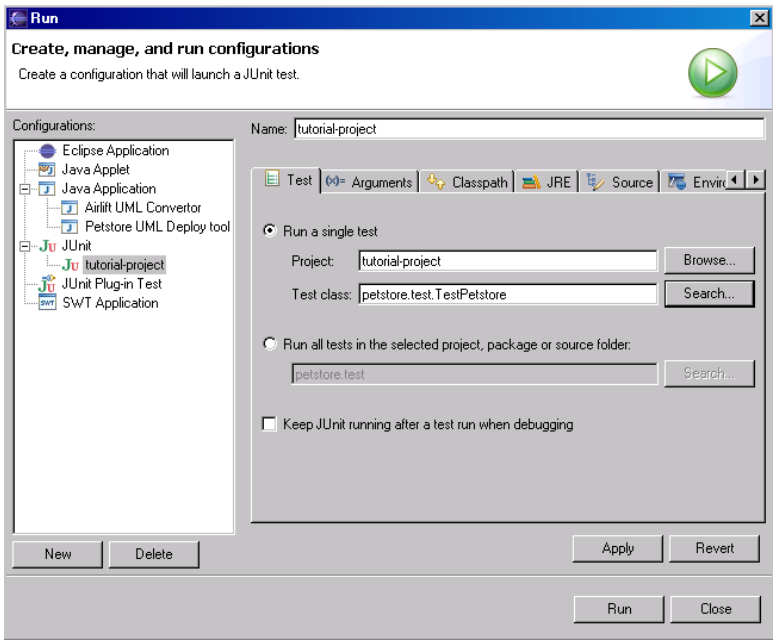
- A few notes are in order:
  - Note that the "person" table contains fields from Person as well as Customer. This is how Hibernate performs Object-Relational mapping of inherited entities.
  - The "person" table also contains fields from the Profile entity because Profile was modelled "inline" – inside the Customer entity without an association, simply as "customerProfile:Profile".
  - Note that the option_code table contains data for AddressType <<OptionCode>> stereotype (Home and Work address), inserted from the PetstoreOptionCode.

Creating the database tables concludes discussion of the Airlift Entity Model Deploy Tool, and we will proceed writing a simple application against the Entity Model.

Coding and running a small Junit test application against the AirliftPetstoreSubset domain entity model.

In this section we will create an exteremely simple application (a test) that will show how data can be added and accessed in our Domain Entity Model, as generated from UML into Airlift entities as persisted by Hibernate.

We create a simple unit test application which will create and persist 3 persons with addresses. For those following step by step, this test can be run as follows:



When finished, you should see 3 rows in the "person" table and 3 rows in the "address" table (associated via address.PERSON_ID).

A few notes about the test classes and how they show persistence entity management in Airlift

- The test class is TestPetstore.java with the core method `suiteDo()`
- The core test file is TestPetstorePerson.java with the two core methods

```
    public void testPersonQuery_LoadTestData()
    {
    ResponseSummary response =
    this.getConversation().executeDemarcationBlock(new DemarcationBlock()
    {
    public void executeWithDemarcation()
            {
            insertPerson(smith, "smith@airliftj.org", "Tennesee", "Nashville");
            insertPerson(miller, "miller@airliftj.org", "Alabama","Birmingham");
            insertPerson(carpenter, "carpenter@airliftj.org", "Ontario", "Oakville");
            }       // NOTE: automatic commit happens at the end of this block..
            // or rollback if an exception was thrown inside this block
            // this assumes of course, that there was no transaction
            //already running when we got here.
    },DemarcationSettings.SETTINGS_TRANSACTION_REQUIRED);
    }
    }

private void insertPerson(String lastName, String email, String state, String city)
    {
    // delete any Person in database by name
    while(deletePersonIfExists(lastName));
    Person person = (Person)getEntityManager()
                        .createEntityAndAssignIdentity(Person.class);
    person.setLastName(lastName);
    person.setEmailAddress(email);
    getTransactionManager().managePersistentEntity(person);

    // Create a single Address for this Person.
    Address address =(Address)getEntityManager()
                        .createEntityAndAssignIdentity(Address.class);
    address.setState(state);
    address.setCity(city);
```

```
Set addresses = new HashSet();
addresses.add(address);
person.setAddress(addresses);
getTransactionManager().managePersistentEntity(address);
}
```

- The insertPerson method shows how to create a persisted entity in the Airlift framework.
- The code is written against the business interface Person and Address, not the business implementation PersonImpl and AddressImpl.
- Very briefly, each persistable entity need to createEntityAndAssignIdentity() and managePersistentEntity() but that is subject for another tutorial, showing a more "real" application, which is making use of the Application and Component layer.

Conclusion:

This tutorial attempted to present a complete MDA based development cycle using the Airlift framework, from a EntityModel defined in UML, through fully automated Object Relational mapping, all the way to writing an application and persisting data against the Entity Model. Perhaps one day, we will be developing in (something similar to) UML as envisioned by the MVC inventor Trygve Reenskaug in his BabyUML research , in the meantime, a few small steps.

Future Development:

Plans for future is in general terms, increased pluggability on both UML and persistent ends, and making use of at least following tools:

ArgoUML front end
EJB back end
JDO back end