Implementation Guidelines For The Rent-A-Ride System

You will be using the entity (domain) class diagram discussed before and available on eLC. You may have to adjust your sequence diagrams realizing use cases accordingly, but you may do this as you are implementing the Logic and Presentation layers.

Also, you may use on of the middleware ORM systems, e.g. Hibernate, to automate your persistence. If this is the case, most of the guidelines for the implementation of the Persistence Layer subsystem would not apply.

1. Obtain a copy of the interfaces you will be implementing

The Java code for the interfaces for the entity classes, the Object Layer and the Persistence Layer subsystems is available on `nike` in my directory `~kochut/csx050/RentARide` (a copy is available on `uml`, as well). You may use the directory layout as your starting point of the system's implementation.

I strongly recommend using SVN on `uml`, or perhaps another source code versioning system. Each team already has a repository in SVN on `uml` and you can easily use it to maintain a shared repository of the implementation of the system. If you would like to use SVN on uml, one of the teammates should copy the `RentARide` files to his/her local machine (perhaps a `nike` or `uml` account) and then import the directory to SVN. Other team members will be able to get a copy (so called working copy) of your files from the repository, make additions and/or changes (for example implementing some classes) and then checking back the changes into the repository on `uml`. These changes would be instantly visible to the other team members.

2. Define the database schema for the entity classes

Your team already has a database, rolename (user name), and password on our MySQL DB server running on `uml.cs.uga.edu`. Start by creating a mapping for all entity classes and associations to a collection of tables and association tables in MySQL. You should follow the strategy outlined in class and also discussed towards the end of the lecture on Relational Databases.

You may want to create a subdirectory called `db` in the Java code you copied in step 1 above. Your schema definition should be placed in a text file; you may later use this file to re-create the schema for your database easily, when needed. You may name the file `rentaride-db-schema.sql`, or something similar.

Also, you may want to create an additional file with SQL `INSERT` statements to populate the database with a few instances and links among them for testing purposes. You may name the file `rentaride-db-populate.sql`, or something similar.

Test your schema by executing several `SELECT/INSERT/UPDATE/DELETE` statements and verify the results. You may want to place these commands in a yet another file, say `rentaride-db-test.sql`, to be used for re-testing, as needed.

3. Implement the entity classes

3.1 Implement the Persistent abstract class

This class implements the `Persistable` interface from the `edu.uga.cs.rentaride.persistence` package. You may re-use a similar class from the Clubs example discussed in class (the Clubs example code is available on `nike` in my directory `~kochut/csx050/clubs` and in SVN at the URL `http://uml.cs.uga.edu/svn/clubs`). This class will provide a common root class for all entity class implementations.

3.2 Implement the entity classes

Provide the implementation for all entity interfaces defined in the `edu.uga.cs.rentaride.entity` package. Each implementation class should extend the Persistent abstract class defined earlier, provide the suitable representation for all needed attributes, and implement the required getters/setters.

Your implementation classes should be placed in the `edu.uga.cs.rentaride.entity.impl` package.

4. Implement the Persistence Layer Subsystem

Carefully examine the Clubs example to fully understand how the Manager- and Iterator-type classes for entity classes operate. In general, each Manager class, e.g., `RentalLocationManager`, should implement the save, restore, and delete methods using suitably formulated SQL commands.

4.1 Implement the Manager- and Iterator-type classes for entity classes

For each entity class, implement a pair of classes: one functioning as the Manager for the entity and the other as the Iterator for the entity class.

Implement a `save` method, which should accept a single object of the entity class (interface, not the implementation class). The save method should check if the argument object is persistent (using the `isPersistent()` method) and if so, create a suitable SQL `UPDATE` statement and execute it to perform an update of the already existing persistent object in the database. If the argument object is not persistent (i.e., it has just ben created by one of the use cases and needs to be saved for the first time), create a suitable SQL `INSERT` statement and execute it to persist the object. You should retrieve the automatically generated key for the new row and assign its value as the id of the argument object. It will be marked as a persistent object, in case it has to be saved in the database after additional modifications of its state.

Implement the `delete` method, which should accept a single object of the entity class (interface, not the implementation class). The delete method should check if the argument object is persistent (using the `isPersistent()` method) and if so, create a suitable SQL `DELETE` statement and execute it to perform an removal of the already existing persistent object from the database. If the argument object is not persistent (i.e., it has just ben created by one of the use cases and needs to be saved for the first time), this method should throw a `RARException` (an exception should also be thrown in case of any other failure).

Implement a `restore` method, which should return an Iterator to iterate over all retrieved object instances (e.g., `Iterator<RentalLocation>`). The restore method should accept a single object of the entity class (interface, not the implementation class). The method should behave differently, based on this argument as follows:

- If the argument is `null`, the method should create an SQL `SELECT` statement to retrieve all rows of the corresponding table. For example:

  ```
  select * from RentalLocation
  ```

- If the argument is not `null` and refers to a persistent object, the method should create an SQL `SELECT` statement to retrieve a *single* row by adding a `WHERE` clause in which a condition specifies the primary key value (equal to the `id` from the argument object). For example:

  ```
  select * from RentalLocation where id = 3
  ```

  if the `RentalLocation` object given as the argument of restore has the `id` of 3. Please, note that the argument object may have just the id attribute set, as the other attribute values will be disregarded, as the primary key value (`id`) uniquely identifies the row to retrieve.

- If the argument is not `null` and refers to a non-persistent object, the method should create an SQL `SELECT` statement to retrieve rows matching the values provided as attributes of the supplied argument object. In that sense, the argument can be thought of as a model object (sample object), indicating which objects should be retrieved. This should be achieved by adding a `WHERE` clause in which a condition specifies the column values to be equal to the attributes supplied in the argument object. For example, while implementing the restore method for the CustomerManager class (to manage the Customer entity class), if the argument object has only the firstName set and all other attribute values set to null, this means that the retrieved rows in the Customer table should have the specified value of the firstName column. The select statement should be similar to:

  ```
  select * from Customer where firstName = 'Robert'
  ```

  if the `Customer` object given as the argument of restore has the value of the `firstName` attribute equal to `"Robert"`. Please, note that in this case the argument object *should not* have the id attribute set to a non-negative value.

Some entity classes are in an association to other classes, where the multiplicity on the other end is 1. For example, the `Vehicle` class is associated to `VehicleType` and `RentalLocation`, both of which have multiplicity of 1. In this case, each `Vehicle` object must be linked to exactly one `VehicleType` object and exactly one `RentalLocation` object. Then, if we wanted to restore all `Vehicles` (by supplying a null argument to the restore method in the `VehicleManager` class), the suitable SQL `SELECT` statement should retrieve the data for a `Vehicle` AND the data for the corresponding `VehicleType` and `RentalLocation`. Consequently, it should be join `SELECT` statement. For example:

```
select * from Vehicle v, VehicleType vt, RentalLocation rl
where v.vehicleType = vt.id and v.rentalLocation = rl.id
```

The restore method should execute the created SQL `SELECT` statement (as described above) to retrieve the required rows from the database and then initialize the corresponding Iterator class (e.g., `RentalLocationIterator`) with the `ResultSet` obtained by executing the SQL `SELECT` statement. Each call to the `next()` method should then create a corresponding Java object instance (e.g., `RentalLocation`) and initialize its attribute values to the values retrieved from the database. Furthermore, the id attribute should be set to the primary key of the retrieved row, to indicate that the object has already been stored in the database.

In case the entity class is associated to other class(es) with multiplicity of 1 (as described above), the iterator should create "the other side" object instance and set the reference to it in the first class. For example, in case of restoring `Vehicles`, the iterator should create an instance of `VehicleType` and an instance of `RentalLocation` (using the data obtained from the join-type `SELECT` statement listed above) and then an instance of `Vehicle`, and finally set the references to the created `VehicleType` and `RentalLocation` objects as attributes of the Vehicle object instance. Effectively, by restoring a Vehicle object, we are also restoring its `VehicleType` and its `RentalLocation` at the same time.

Finally, the Manager class should implement a method to traverse each association ending at the entity class for which the Manager is being implemented. For example, for the `RentalLocationManager` class, there should be a method to traverse the association `locatedAt` to `Vehicle` and one more to traverse the association `hasLocation` to `Reservation`. These methods should return Iterators of `Vehicles` and `Reservations`, respectively, to enumerate all objects linked to a `RentalLocation` object. In case the association on the other end is 1, the method should return a single object. You may use the association name as the name of the method. For example, in the `RentalLocationManager` class, you would have the following two methods: `Iterator<Vehicle> restoreLocatedAt(RentalLocation rentalLocation)` and `Iterator<Reservation> restoreHasLocation(RentalLocation rentalLocation)`. Note that these associations are bi-directional, and similar methods should be implemented in the `VehicleManager` and `ReservationManager` classes to traverse these associations in the opposite direction.

All of the Manager- and an Iterator-type classes should be placed in the package `edu.uga.cs.rentaride.persistence.impl.`

4.2 Implement the PersistenceLayer class

This class should be placed in the `edu.uga.cs.rentaride.persistence.impl`, along with all the Manager- and Iterator-type classes. The implementation of all the required methods should be done with the use of the suitable manager classes.

5. Implement the Object Layer Subsystem

This should be relatively simple. You should implement most of the methods by delegating them to the suitable methods of the `PersistenceLayer` interface. The create-type methods should use the constructors from the corresponding entity implementation classes.

The `PersistenceLayer` implementation class should be placed in the `edu.uga.cs.rentaride.object.impl` package.

6. Implement Test programs

You should implement programs to test the functioning of all methods in the `ObjectLayer` interface. Note that the `PersistenceLayer` methods will also be tested, as a result.

A smart thing to do would be to implement JUnit test classes, which would have the setup and tear-down methods, as well as a collection of test cases. This would enable you to perform many regression tests (making sure that the previously test and working functionality remains properly working).