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.

You should get a copy of the Clubs example, which is available on nike in the directory /home/profs/kochut/csx050/clubs. It provides a good illustration of how a system like Rent-A-Ride may be implemented.

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 given directory layout as your starting point of the system's implementation.

The project is also available in our SVN repository on `uml`, at the URL `http://uml.cs.uga.edu/svn/rentaride`. To simplify transferring the files, I also created a ZIP file (RentARide.zip) and a gzipped tar file (RentARide.tar.gz). Both files are available on nike in my directory ~kochut/csx050.

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. You may want to insert a single Administrator object into the database to get things started.

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.

Examine the mapping of the UML class diagram for the Clubs example to the corresponding collection of tables defining the MySQL schema for this example.

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 very similar class from the Clubs example discussed above (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.

As an illustration, learn how entities for the classes in Clubs example have been implemented (look in the clubs/src/edu/uga/clubs/entity/impl).

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.

The save method

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.

The delete method

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).

The restore method

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. **Carefully examine the code of ClubManager.java and**

ClubIterator.java as an example illustrating this case. Note, that each Club instance must be linked to exactly one Person (its founder). So, to restore a persistent Club object, we need to retrieve both the Club object and the Person object (its founder). This is done by a join SQL SELECT statement (a join of the club and person tables on the founderid, which is a foreign key in the club table). That is, the SELECT statement retrieves the necessary data for *both* the Club and the linked Person. Subsequently, when a Java proxy object for the Club is created, we first have to create the proxy Java object for the Person (the founder) and provide its reference as the founder of the Club.

The association handling methods

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. For example, a storeClassX method should be implemented with the use of the save method of the ClassXManager class. Similarly, deleteClassX should be implemented by calling (delegating to) the delete method in ClassXManager.

Please note that the store and delete methods for relationships with the multiplicity at one endpoint set at 1 are not really needed. For example, in the association Vehicle – locatedAt – RentalLocation, where the multiplicity is many-to-one (one at the RentalLocation endpoint), every Vehicle object instance must be linked to exactly one instance of the RentalLocation class. The storeVehicleRentalLocation(Vehicle vehicle, RentalLocation rentalLocation) and deleteVehicleRentalLocation(Vehicle vehicle, RentalLocation rentalLocation) are not needed and you may leave their bodies empty. The reasons for this are explained below.

When creating (storing for the first time) a Vehicle object, that object must already have a reference to a RentalLocation object (where the vehicle is located). Storing that Vehicle object will set the foreign key in the new row inserted into the vehicle database table referencing the already existing RentalLocation row in the rental location table. Thus, the persistent link between the Vehicle and the RentalLocation is created when the Vehicle object is store, and there is no need to have a specific operation for doing this (i.e., storeVehicleRentalLocation(Vehicle vehicle, RentalLocation rentalLocation)) is not really needed. You may leave the body of such methods empty, as they should never be called.

Similarly, deleting a link of an association having one endpoint multiplicity specified as 1 will be handled by deleting the object at the other end of the association (i.e., opposite to the multiplicity 1 endpoint). For example, consider the same association as above (i.e., Vehicle – locatedAt – RentalLocation). If we were to delete a link connecting a Vehicle to its RentalLocation, this would lead to a Vehicle object which would not be linked to a RentalLocation, violating the multiplicity restriction designed in the data model (entity class diagram). This link can only be deleted when a Vehicle object instance is deleted. In fact, the deleteVehicle operation will lead to the removal of the corresponding row in the Vehicle table, including the foreign key value linking that row to a RentalLocation row. Similarly to the methods for storing one-to-many associations, you may leave the bodies of the delete methods empty, as they should never be called.

The RentalLocation for a Vehicle can only be updated, effectively reassigning the Vehicle to a different RentalLocation. This is done by updating a Vehicle object and then storing the updated object.

Handling of the singleton RentARideConfig class

You should insert a single RentARideConfig into the rentarideconfig table along with the initialization of the database schema, or the initial population of the database (before the system is ever used). That object is going to serve as the "singleton" object in this class.

Having a persistent RentARideConfig object already existing in the database, the restoreRentARideConfig() method should call the ObjectLayer's createRentARideConfig() to get an empty RentARideConfig object and then set its attributes to what was retrieved from the database. Note that since the restoreRentARideConfig method returns a single object and not an Iterator, the construction of the Java restoreRentARideConfig object should be done in the Manager class.

Furthermore, in the storeRentARideConfig(RentARideConfig rentARideConfig), you should test that the argument object is already persistent. That way you will prevent a possibility of creating a second object in this singleton class (someone could call the createRentARideConfig() method and get another Java object of this class and attempt to store it!). To really avoid this, the storeRentARideConfig(RentARideConfig rentARideConfig) method should *only* execute the SQL UPDATE statement, but not INSERT (again, the RentARideConfig persistent object should be inserted into the database before the system is operational).

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 tested and working functionality remains properly working).

7. Implement the Logic Layer Subsystem

Start by reviewing the list of Use Cases for the Rent-A-Ride system, which are available on eLC (look for the file with Functional Requirements in the Projects folder). They describe the overall functionality of the system you are designing and implementing. Also, review the sequence diagrams you have created during the analysis phase, since you may need to revise several of them for the implementation (you don't have to actually revise the sequence diagrams, but be aware of the changes needed to implement all of the use cases).

Create a Logic Layer subsystem interface. It will be somewhat similar to the interface of the Object Layer subsystem, but the operations will be different, of course. It is very likely that many of them will have already been listed in the description of services of the Logic Layer subsystem, as described in your System Design. This interface should be placed in the `edu.uga.cs.rentaride.logic` package.

Implement the Logic Layer subsystem interface by creating an implementation class in the `edu.uga.cs.rentaride.logic.impl` package. The operations of this class should be implemented with the use of the operations provided by the ObjectLayer interface, as well as the operations defined in the entity class interfaces. If your ObjectLayer/PersistenceLayer implementations work well, this part of the project should be quite easy.

8. Implement the Presentation Layer Subsystem

Again, review the list of Use Cases for the Rent-A-Ride system, which are available on eLC (look for the file with Functional Requirements in the Projects folder). They describe the overall functionality of the system you are designing and implementing.

A simple approach to implement the Presentation Layer is to create the boundary classes, as designed in your sequence diagrams. Each boundary class can be implemented as a Java Servlet. I strongly advise

you to use the FreeMarker library for easy construction of HTML pages resulting from the servlet calls. It may take some time to learn how to use it, but this will save you time in the long run. If your team prefers, you can also use JSPs (JavaServer Pages), or other mechanism. The presentation layer servlets should be placed in the package edu.uga.cs.rentaride.presentation.

Following the Clubs example, each boundary class may be implemented as:

- a Java servlet, which will function as the boundary class implementation, based on your sequence diagram for the given Use Case, plus
- a corresponding HTML page with a form implementing the user screen, as needed in the Use case, and
- a corresponding FreeMarker template (or templates), including the design of a response page (or pages), as need in the Use Case. Again, consider using FreeMarker as a time saving tool!

Review the screen designs included in your Requirements Document. You may have to adjust these screen designs in view of the changes in the list of Use Cases you may have envisioned initially. You need to create a collection of HTML pages, each including a form with suitable input fields, buttons, and other necessary elements. Each form should clearly identify a set of form parameters to be sent to the corresponding servlet. Adopt a convention for naming the input fields, so that people implementing the servlets know what parameters to expect. For example, user_name and password for the Login form.

I would suggest starting with the implementation of the Login and Logout Use Cases. You should carefully examine the Session and SessionManager classes in the Clubs example. They provide a simple implementation of managing user sessions. A Session object encapsulates references to a database connection, an object layer, a persistence layer, and the person (user). You'd need to update these instance variables to reflect needs (e.g., change Person to User, etc.). A Session has an identifier, which is an MD5 hash value of the session creation time (look at line 80 in SessionManager.java; you may want to modify the "CLUBS" string to something like "RENTARIDE"). That identifier is later stored as an attribute of the HttpSession object.

You should carefully examine the Login class in the presentation package of the Clubs example and understand how the SessionManager class is used. After a Session is created during the Login use case, it is stored in the SessionManager, and also in the HttpSession of the client connection. On subsequent requests from a logged-in user, that same Session identifier will be available from the HttpSession object. That identifier is used to retrieve the Session object from the SessionManager to object that user's connection and object/logic layer references (for example, look at line 109 in CreateClub in the presentation package). It also serves as a verification that the user has already logged in. If not, there would not be the Session in the SessionManager! Furthermore, the Session class has a simple thread, which invalidates a Session if it is idle for at least 30 minutes (the user will have to log in again).

The Logout Use Case should simply invalidate the Session object (read the code of the Logout class).

As the next step implement the remaining boundary classes in the system. A reasonable thing would be to split the work along the use case lines, each team member implementing a subset of all boundary classes, including the necessary servlets, HTML pages and `FreeMarker` templates. Hopefully, doing it this way you would not need to wait for someone to finish all of the HTML pages!

You will need to create a suitable web.xml descriptor file to define all of the servlets.

Furthermore, you should design the "main" screen (HTML) for the system, from which a user (either an Administrator or a regular Customer) would be able to start all of the available Use Cases. You may design two main screens: one of the Administrator and one for a regular Customer. If you'd like, you may design a menu system to allow for easy navigation among groups of related Use Case (e.g. Vehicle management Use Cases, etc.).

A suitable main screen should be displayed to the user after a successful login.

9. Integrate the System

The system should be composed of all of the packages and other classes you have implemented. You need to create and maintain a proper directory structure of the Web application (you may want to follow the Clubs example). Make sure that the structure is correct – place all of the compiled Java class files in the `WEB-INF/classes` directory, the libraries in `WEB-INF/lib` (`MySQL JDBC` driver and `FreeMarker` library), and the templates in `WEB-INF/templates`. Make sure the web.xml descriptor is well designed – is may be the cause of some common problems.

You may want to designate one of the team members as the *system builder*. That person would have to get the most up-to-date code from the repository, build the system, create the WAR file, and then deploy it to the `WildFly JBoss` system on `uml` for testing.

10. Perform System Test

Create a series of scenarios of using the system for testing purposes and run them on a deployed system. Correct any problems you may identify. Obviously, the more tests you perform, the more robust your system will become!

11. Prepare for the System Demonstration

The systems will be demonstrated on Dec. 12, 2015. Each team will meet with me and demonstrate the operation of the Rent-A-Ride system. Some instructions on how to prepare for the demo are on our class website. We will discuss the preparations for the demos in class, as well.