**Dawson Hotel Reservation System**

**Project Specifications**

**Phase III**

**Due: Friday, November 11th**

* A copy of the sorted and merged rooms, customers and reservation txt files are at S:\CompSci\317\datafiles\database. Use these in order to ensure that you are all working with the same files. These file replace the ones that you have in your ReservationSys\datafiles\database folder.
* Multiple classes and interfaces have been provided. Please read through these files, you are responsible for understanding their content.
* All of the new types introduced in this phase must be properly documented.

**Objective:**

In this phase you will design and implement the data access components. A data access object (DAO) makes a connection to the database (or files), provides methods to store and retrieve objects from the database, and disconnects from the database (or closes the files). The purpose of a DAO is to decouple the data component from the business component in order to make it possible to modify the data storage and retrieval services without affecting the rest of the system.

**Part I – Define the custom exceptions *– some code provided***

We will create new exception types that may be thrown by the DAO objects. Custom exception types are Exception subclasses that allow us to provide more specific exception causes. You have been provided the **DuplicateCustomerException.java** in the S:\CompSci\317\ReservationSys\code. Copy the file to the appropriate folder in your system (look at the package dw317.hotel.data).

Note that since the DuplicateCustomerException class extends the Exception class, it falls within the category of **checked** exceptions. Also, since the Exception class extends the Serializable Throwable class, DuplicatePatientException objects will be Serializable so you should code a private static final long serialVersionUID field.

Follow the approach used with DuplicateCustomerException to code two new Exception classes called N**onExistingCustomerException** and **NonExistingReservationException** in the same package as DuplicateCustomerException. These exceptions will be thrown if a search for a customer or reservation doesn’t find a match. Code a third Exception call called **DuplicateReservationException** - this will be thrown if a room is being reserved at a time that overlaps with an existing reservation.

**Part II – Define the RoomDAO, CustomerDAO and ReservationDAO interfaces *– all code provided***

The RoomDAO, CustomerDAO and ReservationDAO interfaces are coded as members of a package called dw317.clinic.data.interfaces. They have been provided in S:\CompSci\317\ReservationSys\code. Copy the file to the appropriate folder in your system.

In order to keep the system components loosely coupled (i.e. as independent as possible) we will ensure that all interaction between the business layer components and the data layer components take place through well-defined interfaces. The interfaces, which are specified by the RoomDAO, CustomerDAO and ReservationDAOinterfaces defined below, will make it easier to modify the system as new types of data storage and retrieval services are introduced over time.

**Part III – Data persistence *–Some code provided***

One of the concerns in system development is the type of storage and retrieval system that should be used. Normally, the choice is a DBMS (Data Base Management System) such as Oracle. However, if the application is a small scale stand-alone system, text files and/or binary files may be sufficient. In the last phase, we used a HotelFileLoader class to load the text files and return an array; in this phase, we will define an interface that defines what we want all classes that read in data to do.

**The ListPersistenceObject Interface – code provided**

We want to make the loading of the room, customer and reservation data flexible enough to allow us to change file types, therefore we will code to an interface. The ListPersistenceObject interface is provided in S:\CompSci\317\ReservationSys\code, and must be added to your dw317.hotel.data.interfaces package.

**Implementing the ListPersistenceObject interface - The SequentialTextFileList class** **- code provided**

Our initial ListPersistenceObject implementation class, called **SequentialTextFileList**, will interact with sequential text files. Recall: we currently have a class, HotelFileLoader, that contains static methods called **getRoomListFromSequentialFile**, **getCustomerListFromSequentialFile** and **getReservationListFromSequentialFile** that loads the data from a file and returns a full-to-capacity array of appropriate objects. These methods do what we need in terms of loading the required data, however they don’t return a type that is compatible with the ListPersistenceObject interface. We want to use List types since insertion and resizing is simpler and more efficient.

Our problem is that the existing class works well, but doesn’t support the interface that we want. This is a recurring problem that the **Adapter design pattern** addresses. Recall that the purpose of an adapter is to convert the interface of an object into another interface that clients expect.

We want to convert the arrays returns by HotelFileLoader into Lists. The java.util.Arrays utility class provides the asList utility method that returns a List type adapter object given an array. But recall from your lecture notes: the asList method doesn’t actually create a new List object, it just allows you to use some of the List methods on an array; insertion and removal operations are not supported since arrays don’t support this. We will use the asList method to create a new ArrayList object using the **ArrayList (Collection collection)** copy constructor. The ArrayList’s copy constructor expects a collection type as its parameter so we can’t use the array reference as an argument, however we can send the list adapter as the argument value. The constructor will use the adapter to initialize the ArrayList object. Note that the constructor shallowly copies the elements; this is not a problem, since the array object is merely a temporary object which will no longer be required once the ArrayList object has been created.

The SequentialTextFileList class belongs to your groupX.hotel.data package, and is provided at S:\CompSci\317\ReservationSys\code.

**Part IV – Implementing the DAO interfaces**

The RoomListDB, CustomerListDB and ReservationListDB objects will represent the room, customer and reservation databases as an internal list. RoomListDB must implement the RoomDAO interface, CustomerListDB must implement the CustomerDAO interface, and ReservationListDB must implement the ReservationDAO interface.

The RoomListDB, CustomerListDB and ReservationListDB classes must be members of the **groupX.hotel.data** package.

**The RoomListDB and RoomListDBTest - code provided**

The RoomListDB and the RoomListDBTest files are provided in the S: drive at S:\CompSci\317\ReservationSys\code. Be sure to copy the RoomListDB file to your **src**/groupX/hotel/data folder, while the RoomListDBTest file goes to your **test**/groupX/hotel/data folder.

Notice that the RoomListDB declares a final **listPersistenceObject** field that is used to assign a reference to the field List<Room> **database.**

Look at the **RoomListDBTest** application. Notice that it has a setup and teardown method to create and delete the test data files. These methods are invoked at the start and end of each test method to make sure that each test case is independant. CHANGE and/or ADD MORE test cases to fill up the custs and reservs arrays. Be very careful: the files must remain in sorted order as a precondition! You should always write methods and test them immediately and independently of the other methods.

**Coding the CustomerListDB and the CustomerListDBTest classes**

As with the RoomListDB class, code methods separately, then test immediately. Make sure that you add more entries into the CustomerListDBTest’s setup method, into the custs array. Beware: the file must remain in sorted order!

The fields and constructor signatures:

private List<Customer> database;

private final ListPersistenceObject listPersistenceObject;

private final HotelFactory factory;

public CustomerListDB (ListPersistenceObject listPersistenceObject)

public CustomerListDB (ListPersistenceObject listPersistenceObject,

HotelFactory factory)

* code the constructors. The **listPersistenceObject** field, once set by the **constructor** is to be used to assign a reference to the **database**. The **factory** field must be assigned the value referenced by a HotelFactory or to the value referenced by the factory parameter specified in the two parameter constructor’s parameter list. Recall that final fields must be assigned a value prior to the termination of the constructor. Although the database field is not final, **you must not provide a setter method**. The reason that it is not final is because we want to be able to assign it the null value after the list has been persisted to disk in the disconnect method.
* Override the toString method and test. toString returns a String representation of the contents of the database, one element per line as shown by the sample below. The first line must be a message indicating the number of elements in the database. Use StringBuilder for efficiency.

Number of customers in database: 8

raj@king.ru\*Raj\*Wong\*visa\*4556737586899855

joe.mancini@mail.me\*Joe\*Mancini\*\*

Etc.

* **Override and test the add(Customer cust) method**

Adds a customer object to the database. Add a reference to a **copy** of the object referenced by the cust and not the actual object being referenced by the parameter since we are using an internal list as the database. In order to instantiate a Customer object, use the factory class. The method must throw a DuplicateCustomerException if the specified email address is already in the database. Note that the customer must be added in email order to keep the database in sorted order. **Implement a binary search private method to help**. Verify that add works by invoking toString.

* **Override and test the disconnect method**

In order to make the database transactions persistent, the **disconnect** method must be implemented. This method must save the database to disk and assign null to the database field.

In your test method, after checking that add works, next make certain that the changes to the database are persistent (i.e. add new customers to the list, invoke the disconnect method to persist the customers, reconnect by creating a new instance of the CustomerListDB class and display the string returned by the toString() method to ensure that the new customers appear in the list in the correct locations). Do NOT invoke the teardown method until you have reconnected!

* **Override the getCustomer(Email email) method**

Returns a reference to the Customer with the given email address, or throws a NonExistingCustomerException if none can be found. Use the binary search private method that you implemented, since the database is sorted.

* **Override the update (Email email, CreditCard card) method**

Updates a customer in the database if the customer exists. If there is no matching customer, a NonExistingCustomerException must be thrown. Make sure that the changes to the database are persistent.

**The ReservationListDB and ReservationListDBTest**

The ReservationListDB class must be a member of the groupX.hotel.data package and it must implement the ReservationDAO interface.

It must declare the following instance fields and overloaded constructors.

private List<Reservation> database;

private List<Room> allRooms;

private final ListPersistenceObject listPersistenceObject;

private final HotelFactory factory;

public CustomerListDB (ListPersistenceObject listPersistenceObject)

public CustomerListDB (ListPersistenceObject listPersistenceObject,

HotelFactory factory)

All rooms is set by the constructors from the listPersistenceObject’s getRoomDatabase method.

In order to be able to override and test the methods specified by the ReservationDAO interface in the ReservationListDB class one at a time, stub all the ReservationListDB class methods. Once stubbed you will be able to override and test each method separately in the order specified below.

* Override the **toString** method so that when invoked, it returns a String representation of the contents of the database, one element per line as shown by the sample below:

Number of reservations in database: 8

raj@aa.ru\*2016\*9\*10\*2016\*9\*15\*101

j@b.com\*2016\*9\*15\*2016\*9\*20\*101

etc…

Add the methods required to test the ReservationListDB constructors and the toString method to your test application.

* **Override and test the add (Reservation reserv) method**

First check if thereservation overlaps with an existing reservation. If there is an overlap, throw a DuplicateReservationExcption. If the reservation does not overlap, adds a reference to a **copy** of the object referenced by the reserv and not the actual object being referenced by the parameter. In order to instantiate a Reservation object, use the factory class. Note that the reservation must be added in correct order to keep the database in sorted order. Implement a binary search private method to help. In your test class, verify that add works by invoking toString.

* **Override and test the disconnect method**

In order to make the database transactions persistent, the **disconnect** method must be implemented. This method must save the database to disk and assign null to the database field.

In your test method, make certain that the changes to the database are persistent (i.e. add new reservation to the database, invoke the disconnect method to persist, reconnect by creating a new instance of the ReservationListDB class and display the string returned by the toString() method.

* **Override and test the getReservations(Customer cust) method**

Returns a copy of the Reservations belonging to the customer, or an empty arraylist..

* **Override and test the cancel(Reservations reserv) method**

Removes a reservation from the database if it is found; otherwise throw a NonExistingReservationException.

* **Override and test the getReservedRooms(LocalDate checkin, LocalDate checkout) method**

Returns an arraylist with all reserved Rooms overlapping during the time period. You must check if an existing reservation has a room that has been reserved with the **checkin date of the reservation before the checkout date provided**, and the **checkout date of the reservation is after the checkin date provided**: in this case the room is reserved. Only add a room once to the arraylist. You don’t need to make a defensive copy since Rooms are immutable.

* **Override and test the getFreeRooms(LocalDate checkin, LocalDate checkout) method**

Returns an arraylist with all **unreserved** Rooms overlapping during the time period - in otherwords, all rooms that are not returned by getReservedRooms. You don’t need to make a defensive copy since Rooms are immutable.

* **Override and test the getFreeRooms(LocalDate checkin, LocalDate checkout, RoomType type) method**

Returns an arraylist with all **unreserved** Rooms with the given room type overlapping during the time period. You don’t need to make a defensive copy since Rooms are immutable.

* **Override and test the clearAllPast() method**

This method removes all Reservations whose checkout date is before the current date, as returned by LocalDate.now()

**Requirements:**

Include the following in your phase II submission folder:

1. A hardcopy of the CustomerListDB and ReservationListDB classes.
2. A softcopy of your zipped application. The file must include the ReservationSys project, and the CommonX project. (Select the projects, right-click -export - General/Archive file - save in zip format)