**Dawson Hotel Reservation System**

**Project Specifications**

**Phase IV**

**Due: Friday, November 25rd**

In this phase you will add the room allocation policy (strategy), as well as extra functionality to the Reservation System plus serialization.

**Part I – The Room Allocation Policy**

The normal work flow of the reservation system starts as follows:

* a Customer registers
* a Customer inputs their desired check in date, check out date, and room type
* The system determines if there are free rooms available, and if so, proposes one

The hotel can choose which room to propose if there is more than one free room available – this is the room allocation policy. This policy can vary from one hotel to the next; for example, one hotel may decide to completely fill up one floor before moving to the next, in order to save on housekeeping time; another institution want guests spread out through the hotel to minimize noise. The Strategy Design Pattern enables the algorithm to be selected at run-time, thus facilitating the interchanging of policies. The concrete strategies will implement the following AllocationPolicy interface (code also in S:\CompSci\317\ReservationSys\code).

package dw317.hotel.business.interfaces;

import java.io.Serializable;

import java.time.LocalDate;

import java.util.Optional;

import dw317.hotel.business.RoomType;

public interface **AllocationPolicy** extends Serializable{

/\*\*

\* Returns the next available room.

\*.

\*/

Optional<Room> getAvailableRoom(LocalDate checkin, LocalDate checkout, RoomType roomType);

}

Code a DawsonHotelAllocationPolicy class that implements the AllocationPolicy interface as a member of the package groupX.hotel.business. The DawsonHotelAllocationPolicy class gets a ReservationDAO object in its constructor and it keeps a reference to this object as an instance variable. Since AllocationPolicy extends Serializable, you must have a private static variable:

private static final long serialVersionUID = 42031768871L;

The Dawson Hotel tries to spread out its guest evenly through the hotel according to the following algorithm:

* Find the floor with the most number of free rooms of the given room type during the desired period
  + If multiple rooms have the same positive number of available rooms, choose the lowest floor
* Randomly choose one of the free rooms on that floor
* If there are no free rooms available with the search criteria, return an empty Optional

Code a test application to ensure that your class functions as expected.

**Part II – Revisit the Factory interface and implementation (minor code changes)**

In order to further decouple the system from the concrete implementations that are specific to the Dawson Hotel, we will **update** our abstract factory.

Edit your dw317.hotel.businss.interfaces.HotelFactory and add the following method

//ADDED IN PHASE IV

AllocationPolicy getAllocationPolicy(ReservationDAO reservations);

Note that adding methods to interfaces will break your old DawsonHotelFactory class. Add to the DawsonHotelFactory:

@Override

public AllocationPolicy getAllocationPolicy(ReservationDAO reservations){

return new DawsonHotelAllocationPolicy(reservations);

}

**Part III –Creating a Façade to the business layer: HotelManager interface (code provided)**

To order to make it easier to interact with the various system components we will make use of the Façade Design Pattern. A façade is an object that provides a unified interface to the set of component interfaces in the system; hence, it prevents tight coupling between the client and the system components. This way the client merely interacts with one interface without knowing any of the other system components (i.e. the DAO objects, the AllocationPolicy objects, etc.)

The Hotel façade concrete class (coded in Part IV) must implement the following **HotelManager** interface that must be added to the dw317.hotel.business.interfaces package (code also in S:\CompSci\317\ReservationSys\code).

package dw317.hotel.business.interfaces;

import java.io.IOException;

import java.io.Serializable;

import java.time.LocalDate;

import java.util.List;

import java.util.Optional;

import dw317.hotel.business.RoomType;

import dw317.hotel.data.\*;

public interface HotelManager extends Serializable {

/\*\*

\* Cancels a given reservation

\* @param reservation

\* @throws NonExistingReservationException

\*/

void cancelReservation(Reservation reservation)

**throws** NonExistingReservationException;

/\*\*

\* Saves all data, clears expired reservations and closes the hotel.

\*

\* @throws IOException

\* If there is a problem closing the hotel files.

\*/

void closeHotel() throws IOException;

/\*\*

\* Creates and adds a reservation for a customer.

\* @param customer The given customer

\* @param checkin The requested check in date

\* @param checkout The requested check out date

\* @param roomType The requested room type

\* @return The Reservation if possible

\*/

Optional<Reservation> createReservation(Customer customer, LocalDate checkin, LocalDate checkout, RoomType roomType);

/\*\*

\* Finds and returns a customer record.

\* @param email The customer's e-mail address

\* @return Customer object

\* @throws NonExistingCustomerException

\* if the customer with the given e-mail cannot be found

\*/

Customer findCustomer(String email)

throws NonExistingCustomerException;

/\*\*

\* Finds all reservations made by a customer

\* @param customer

\* @return List of Reservations. Returns empty list if no reservations can be found.

\*/

List<Reservation> findReservations(Customer customer);

/\*\*

\* Registers a new Customer

\* @param firstName

\* @param lastName

\* @param email

\* @return The Customer object

\* @throws DuplicateCustomerException is a customer with same e-mail address exists

\*/

Customer registerCustomer(String firstName, String lastName, String email)

throws DuplicateCustomerException;

/\*\*

\* Adds or updates the credit card associate with a customer.

\* @param email The email address of the customer

\* @param cardType

\* @param cardnumber

\* @return the updated Customer

\* @throws NonExistingCustomerException

\* if the customer with the given e-mail cannot be found

\*/

Customer updateCreditCard(String email, String cardType,

String cardnumber)

throws NonExistingCustomerException;

}

**Part IV – Code and test the Hotel façade concrete class**

Hotel objects allow the frontdesk to query the system about customers and reservations, ensuring that all business rules are respected for the particular hotel. The Hotel class must extend the java.util.Observable class (it will be explained during the lectures) and implement the HotelManager interface added in part III. It must be a member of the groupX.hotel.business package.

Hotel objects must have the following attributes:

private final HotelFactory factory;

private final CustomerDAO customers;

private final ReservationDAO reservations;

private static final long serialVersionUID = 42031768871L;

You are free to add other private attributes, however you must not include any getter or setter methods (not even for the above three attributes). You may also add as many private methods as you deem necessary.

The **DAO** attributes allows Hotel objects to request storage and retrieval services.

The **factory** provides the instantiation factory methods as required by the Hotel object.

The **Hotel** will have a single **constructor** (shown below).

public Hotel (HotelFactory factory, RoomDAO rooms, CustomerDAO customers,

ReservationDAO reservations

Code all the methods and a test application to ensure that your class functions as expected.

**Part V – Object Serialization –Utility class - code provided**

**Note**: this part is also done in Lab 10:

The serializeObject and deserializeObject utility methods can be used to serialize any type of object; if you haven’t done so already, create a class called Utilities in your groupX.util package in your CommonX project.

In order to prevent the Utilities class from being instantiated, code a private do-nothing constructor in the class.

Add the following import statements to the Utility class’s compilation unit.

import java.io.IOException;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

**Code the serializeObject method.**

Code the following serializeObject method in your Utilities class.

public static void serializeObject (Object object,

String fileSpecification) throws IOException {

ObjectOutputStream out = null;

try {

out = new ObjectOutputStream (

new FileOutputStream (fileSpecification));

out.writeObject (object);

}

catch (IOException e){

//normally the exception would be logged to file then thrown

throw new IOException ("Error serializing object to \n" +

fileSpecification + " " + e);

}

finally {

if (out != null)

out.close ();

}

}

**Code the deserializeObject method.**

Code the following deserializeObject method in your Utilities class.

public static Object deserializeObject (String fileSpecification)

throws IOException, ClassNotFoundException {

ObjectInputStream in = null;

try {

Object obj = null;

in = new ObjectInputStream

(new FileInputStream (fileSpecification));

if (in != null)

obj = in.readObject ();

return obj;

}

catch (ClassNotFoundException | IOException e)

{

//normally the exception would be logged to file then thrown

throw new IOException ("Error deserializing object from " +

fileSpecification + "\n" + e);

}

finally

{

if (in != null)

in.close ();

}

}

**Part VI – Object Serialization - ObjectSerializedList and SerializedFileLoaderApp**

Before we can serialize a list of rooms, customers or reservations, we must make sure all classes in the object graph are serializable. Double-check all your dependent classes (e.g., Email, CreditCard, …). Make sure they all are Serializable, and that none of the instance members are Optional, since Optional is not serializable.

Our initial ListPersistenceObject implementation class, SequentialTextFileList, interacted with sequential text files. Now you will code a class called **ObjectSerializedList** (as a member of your groupX.hotel.data package) that implements the ListPersistenceObject interface and interacts with object serialized files. ObjectSerializedList allow the hotel system to retrieve/save the database from/to object serialized files.

Implement the ListPersistanceObject methods within ObjectSerializedList. Pass the filenames of the serialized rooms, customers and reservations to the constructor.

**Code and run SerializedFileLoaderApp to create the initial .ser files from your files**

Create a class **SerializedFileLoaderApp** (as a member of your groupX.hotel.data package) that loads the data from the sequential text files into a Lists (using methods in SequentialTextFileList to help) and serializes them to object serialized files. These files must be called datafiles/database/**rooms.ser**, datafiles/database/**customers.ser** and datafiles/database/**reservations.ser**. Run the application.

**Modify your RoomListDBTest, CustomerListDBTest and ReservationlistDBTest applications from Phase 3**

Modify the setup method so that it creates object serialized files from the sequential text files at the end of the setup method’s try block.

Also change the lines which instantiate the RoomListDB, CustomerListDB and ReservationListDB such that they take an instance of an ObjectSerializedList. There are no other code changes necessary since we programmed to interfaces ☺

Run your test methods over again to ensure that the results match your expected results.

**Requirements:**

1. A hardcopy of the Hotel, DawsonHotelAllocationPolicy and ObjectSerializedList classes.
2. An electronic copy (a zip file) of your RservationSys and CommonX project files. Make sure that it contains all of the required resources.