OOAD Assignment

Due Date: 11:59pm, Saturday, 20 June 2020

This assignment is worth 24% of the total final score.

Where to Submit: A softcopy of completed design class diagrams, atomic use case specifications and all the Java code must be compressed into a .ZIP file (under the file name: StudentID.zip), and submitted to the Blackboard and the email account of the Instructor:

iu.subjects@gmail.com

with the email's subject:

"OOAD Assignment, <YourStudentID>, <YourStudentName>"

This is an individual assignment. You are not permitted to work as a group when writing this assignment.

Academic Honesty: Each student is required to work individually and honestly to answer these tasks. Any academic misconduct (e.g., plagiarism - the submission of somebody else's work in a manner that gives the impression that the work is your own) is strictly prohibited and will be processed based on the International University's regulations.

No extensions will be given: If there are circumstances that prevent the assignment being submitted on time, an application for special consideration may be made. Note that delays caused by computer downtime cannot be accepted as a valid reason for a late submission without penalty. Students must plan their work to allow for both scheduled and unscheduled downtime.

Return of Assignments: Students are referred to the International University's regulations.

Objectives: To learn to represent navigation and the detail required on a design class diagram, specify atomic use cases, and implement and test a prototype of the system.

This assignment is the continuation of the MID-TERM EXAM of this OOAD course. The DU-Rent System described in the MID-TERM EXAM (its detailed specification is still available on the Blackboard and Google Drive of this OOAD course). Whereas the MID-TERM EXAM is concerned with the analysis phase, this assignment will be concerned with design, prototyping and testing.

As the starting point for this assignment, assume that the partially completed analysis class diagrams, given in Figure 1, has been adopted. Figure 1 shows the classes of domain objects, their attributes, relationships and constraints. This sample is simply one among other solutions.

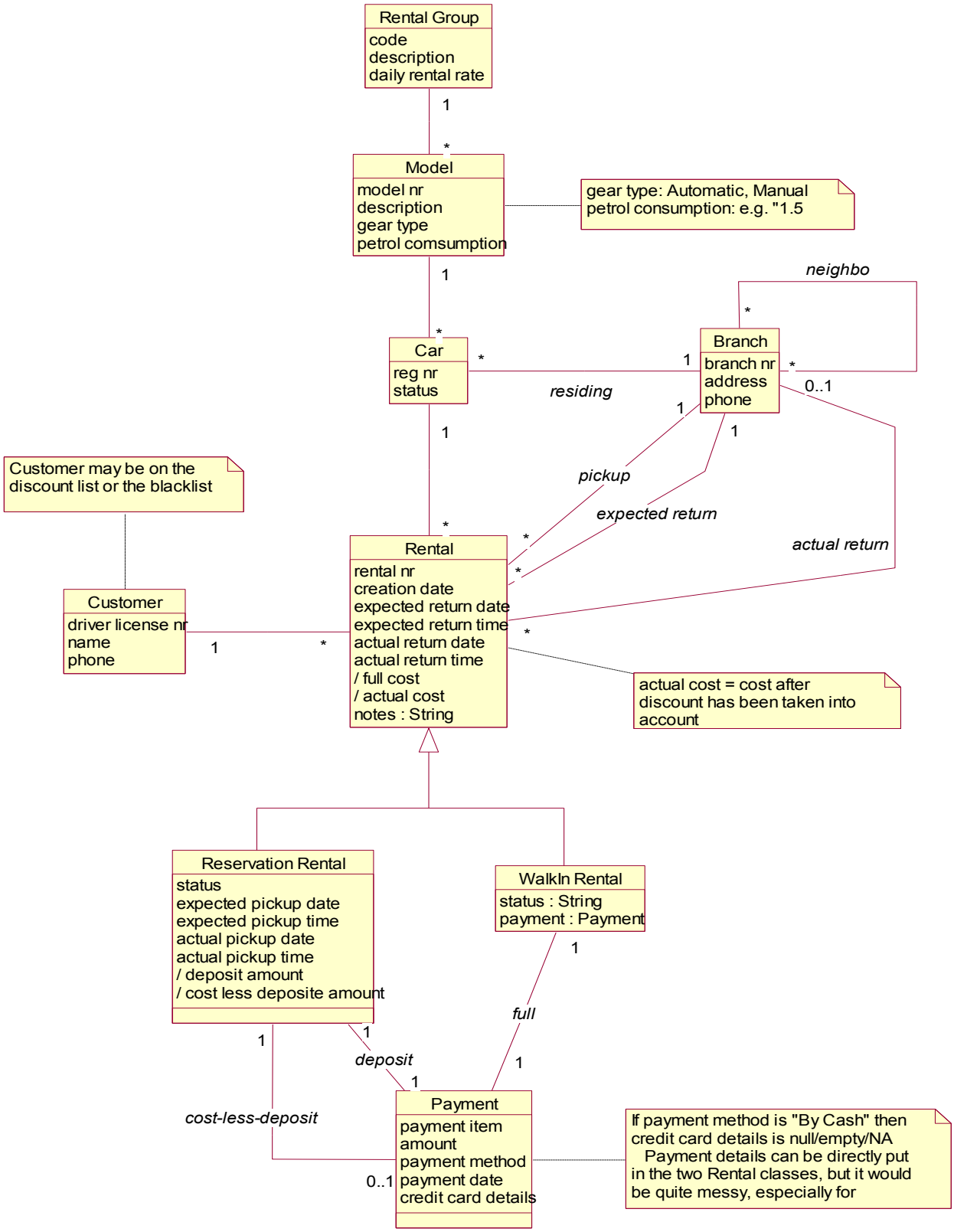

Figure 1 – Analysis Class Diagram Showing Classes for Domain Objects

Task 1 – Design Class Diagrams (10%)

Construct the design class diagrams to show the complete design of the system. If there are more than 1 design option, present all of these design options, and present your final design decision (if any).

The class diagrams must be completed with attributes (including attributes representing navigations), attribute types, methods with complete operation signature. Include all – and only - methods that are required for the atomic use cases listed in Task 2 (including the constructors).

Note: For presentation, you may suppress the signatures on a class diagram and provide each class separately where the methods' signatures are shown.

What to submit for task 1

1. Softcopy of the design class diagrams (in pdf format)

Task 2 – Atomic Use Case Specifications (35%)

In the table below, column 1 lists the use cases that we are likely to identify in the analysis phase. Column 2 indicates the use cases that we are going to consider for the prototype. A tick (✓) means "the use case in column 1 is to be included for the prototype"; "NO" means "to be excluded". Note that there are some use cases that are not in column 1 and they have been added to column 2 (to make the prototype more self-contained and more suitable for testing).

(Possible) Use Cases Discovered in Analysis	**Use cases Considered for Prototype**
	Add a branch
	Add a car rental group
Add a model	✓
	Add a car -- Specify also the branch where the car is kept, and set the status to "Available"
	Add a customer -- Indicate if the customer is blacklisted or entitled to discount. (This use case does not make sense for original problem.)
List available cars	✓ -- Consider two atomic use cases: one to get the available cars at the specified branch; and one to get those in the neighboring branches of the specified branch
Put a car on hold	NO
Release the hold on a car	NO
Create a rental with a car reserved	NO
Create a rental without reservation (walk-in and pick-up a car at a branch)	✓

Print list of cars to be transferred from a branch (to nearby branches)	NO
Record delivery of a requested car	NO
Substitute a car for a reservation	NO
Cancel a reservation because requested car is not available	NO
Cancel a reservation because of user action	NO
Change drive for a reservation rental	NO
Record the pick-up of a reserved car	NO
Record the return of a car	✓
Print the list of cars to be inspected	NO
Record that a car needs to be serviced	NO
Record that a car is to be removed	NO
Return a car to the active pool	NO

Selected Atomic Use Cases

From the selected use cases, assume that we have identified the following atomic use cases (to support the selected use cases):

1. Add a branch
2. Make a pair of branches neighbors to each other
3. Add a car rental group
4. Add a model
5. Add a car
6. Add a customer
7. List cars that are available at a specified branch and belong to a specified rental group (do not include the cars at neighbor branches)
8. List cars that are available at the neighbor branches of a specified branch and belong to a specified rental group (do not include the cars at the specified branch)
9. Enter a walk-in rental
10. Record the return of a car

Your task is to specify the atomic use cases listed above, using the formal specification language introduced in the course. In your answer, number the use cases as shown.

The following points must be observed:

- Your specifications must be based on *your design class diagrams*, which in turn must be based on the class diagrams given in Figure 1.

- Your specification must also be based on the *problem description given in the MID-TERM EXAM*, except where we make explicit changes for the prototype (e.g. Customer has attributes to indicate if they are blacklisted or entitled to discount) or simplifications (e.g. we treat dates as integers)

State any reasonable assumption you make.

What to submit for Task 2

1. Softcopy of the atomic use case specifications (in pdf format)

Task 3 – Prototyping (35%)

Prototype all the atomic use cases listed for Task 2 in Java (or other object-oriented programming language of your interest).

Your implementation of the prototype must be done in a systematic manner. In particular, for each use case, the preconditions should be checked first, and the postconditions should then be satisfied.

What to submit for Task 3

1. Softcopy of the listing of the code (in pdf format)

   Arrange your classes as follows. The first class in the listing is the DURentSystem class, followed by the "domain" classes in *alphabetical order* of the class names.

2. Electronic copy of all the classes, including those that were given (e.g. SimpleKey, CompositeKey, Helper, GC, etc.)

Task 4 – Testing the Prototype (20%)

For each atomic use case, design the test cases and include them in a Java program, called DURentSystemTest, to carry out the testing.

As a suggestion, you may want to try the following test cases.

1. Add 3 branches B1, B2, and B3
   B1, B2 are neighbors

2. Add 2 rental groups
   A, compact, $100 per day
   B, full size, $200 per day

3. Add car models
   M1, group A
   M2, group B

4. Add cars
   CAR1, model M1, branch B1
   CAR2, model M1, branch B2
   CAR3, model M2, branch B1

5. Add customers
   CUST1, entitled to discount, not blacklisted
   CUST2, not entitle to discount, not blacklisted

6. Display all the cars of "Compact" type that are available at branch B1

7. Display all the cars of "Compact" type that are available from the neighbor branches of B1

8. Customer CUST1 walks in and picks up CAR1, from B1, on day 10, to be returned on day 12 to the same branch B1

9. Customer CUST1 returns the car on day 12 to branch B2.

Note that the details of the test cases above are not completely spelled out, and the suggested test cases are more or less the bare minimum. You should include more test cases to cover more comprehensive testing. In particular, you should include test cases where the preconditions are not satisfied.

You must also arrange your test cases so that the later test cases are independent of the earlier ones. A good way to do this is to code test cases in different methods. Each method has two parts: one to prepare the "base data" and one to perform the tests.

The test program should be appropriately commented so that we can see the purpose of the test cases.

NOTE: Even if your prototype is not complete, you still need to provide the test program for the parts that you have completed.

What to submit for Task 4

1. Softcopy of the code listing of the DURentSystemTest class.
2. Softcopy of the results of the tests in the form of SCREENSHOTS (The results can be quite long. They can be submitted in small font. Also, they should be divided into various groups, rather than just one monolithic printout.)
3. Electronic copy of the DURentSystemTest class.

Appendix 1 - A Sample Test Program

```
public static void main(String [] args) throws Exception
{
    int testCount = 0;
    String test;

    // TEST: Create new system
    test = "Create new system";
    DURentSystem rs = new DURentSystem();
    System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + rs);

    // TEST: Add branches B1, B2, B3
    test = "Add branches B1, B2, B3";
    rs.addBranch("B1", "Address 1", "1111");
    rs.addBranch("B2", "Address 2", "2222");
    rs.addBranch("B3", "Address 3", "3333");
    System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + rs);

    // TEST: Set B1, B2 as neighbors
    test = "Set B1, B2 as neighbors";
    rs.setNeighbors("B1", "B2");
    System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + rs);


    // TEST: Add rental groups A, B
    test = "Add rental groups A, B";
    rs.addGroup("A", "Compact", 100);
    rs.addGroup("B", "Full Size", 200);
    System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + rs);

    // TEST: Add models M1, M2
    // GC is the class in which all the global constants (GC) are defined (this is done in the
    // interest of simplicity.  A more “serious” solution, which we are not pursuing, would use
    // enumerated types)
    test = "Add models M1, M2";
    rs.addModel("M1", "Model 1", GC.AUTOMATIC, "A");
    rs.addModel("M2", "Model 2", GC.MANUAL, "B");
    System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + rs);

    // TEST: Add cars CAR1, CAR2, CAR3
    test = "Add cars CAR1, CAR2, CAR3";
    rs.addCar("CAR1", "M1", "B1");
```

```
        rs.addCar("CAR2", "M1", "B2");
        rs.addCar("CAR3", "M2", "B1");
        System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + rs);

        // TEST: Add customers CUST1, CUST2
        test = "Add customers CUST1, CUST2";
        rs.addCustomer("CUST1", "Name 1", "2111", true, false);
        rs.addCustomer("CUST2", "Name 2", "22222", false, false);
        System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + rs);

        // TEST: List available cars at branch B1 for group A
        test = "List available cars at branch B1 for group A";
        String result = rs.getAvailableCars("B1", "A");
        System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + result);

        // TEST: List available cars of group A at neighbor branches of B1
        test = "List available cars of group A at neighbor branches of B1";
        result = rs.getAvailableCarsFromNeighbors("B1", "A");
        System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + result);

        // TEST: Enter a walk-in rental paid by cash
        test = "Enter a walk-in rental paid by cash";
        rs.addWalkInRental("RENTAL1", 10, 12, "CUST1", "CAR1", "B1", GC.CASH, null);
        System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + rs);

        // TEST: Record car return
        test = "Record car return";
        rs.recordCarReturn("RENTAL1", 12, "B2" );
        System.out.println("\nTEST " + (testCount ++) + ": " + test + "\n" + rs);
}
```

Note that, as stated earlier, your test cases must be “modular” (i.e. grouped in methods) and arranged so that the later test cases are independent of the earlier ones.

Appendix 2 – The class defining global constants

```
public class GC // Global Constants
{
	// gear type values for Car
	public static final String AUTOMATIC = "Automatic";
	public static final String MANUAL = "Manual";

	// status values for Car
	public static final String AVAILABLE = "Available";
	public static final String HELD = "Held";
	public static final String RESERVED = "Reserved";
	public static final String PICKED_UP = "Picked-Up";
	public static final String RETURNED = "Returned";
	public static final String SERVICED = "Serviced";
	public static final String REMOVED = "Removed";
	public static final String EXCEPTIONAL = "Exceptional";

	// values to indicate payment method
	public static final int CASH = 1;
	public static final int CREDIT_CARD = 2;

	// Status values for Rental
	// already defined for Car:
	// Reserved, Picked-Up, Returned, Exceptional
}
```

■