Object-Oriented Design and Implementation Individual Assignment 1

Due Date and Goals

Due 11:59 PM January 20, 2022

Goals:

* To refresh knowledge of coding classes
* To learn the Java coding standards
* Create modular projects

The Problem

A museum has a number of rooms, each of which has several walls. Paintings are exhibited on some or all the walls. Your task is to represent this system in an object-oriented manner.

The museum, each room, each wall, and each painting has a name and each is represented in software by a class. This common property of the name is encapsulated in a super class named *Entity*; this class has a public constructor that takes the name of the entity as its only parameter. All other classes (*Museum*, *Room*, *Wall*, and *Painting)* are subclasses of *Entity*.

Your system will thus have classes named *Museum*, *Room*, *Wall*, *Painting*, and *Entity*.

The classes should be structured as shown below.

# *Museum*

1. A field to store the rooms
2. A constructor that sets the name
3. A method that returns a *List* (java.util) of *Painting* objects in the museum
4. A method to add a *Room* object. This method has a parameter for the name of the room and creates and returns the *Room* object created.

# *Room*

1. A field to store the walls
2. A constructor that sets the name
3. A method that returns a *List* (java.util) of *Painting* objects displayed in this room
4. A method to add a *Wall* object; This method has a parameter for the name of the wall and creates and returns the *Wall* object created.

# *Wall*

1. A field to store the paintings on this wall
2. A constructor that sets the name
3. A method that returns a *List* (java.util) of *Painting* objects displayed on this wall
4. A method to add a *Painting* object; this is called from the method (item 2) listed under *Painting*.

# *Painting*

1. A constructor that sets the name
2. A method to set the wall on which this painting is displayed; this calls the method (item 4) listed under *Wall*

You will have to override toString() appropriately, throughout.

Use generics wherever applicable.

# A Minimal Test

The following code shows an idea of the expected functionality. Please remember that I will exercise your program with a more extensive test.

**public** **static** **void** main(String[] args) { Museum museum = **new** Museum("M1");

Room room1 = museum.addRoom("r1");

Room room2 = museum.addRoom("r2");

Painting painting1 = **new** Painting("p1");

Painting painting2 = **new** Painting("p2");

Wall wall1 = room1.addWall("w1"); Wall wall2 = room2.addWall("w2"); painting1.setWall(wall1); painting2.setWall(wall2);

System.***out***.println(wall1.getPaintings());

System.***out***.println(wall2.getPaintings());

System.***out***.println(room1.getPaintings());

System.***out***.println(room2.getPaintings());

System.***out***.println(museum.getPaintings()); }

It produced the following output.

[Painting p1 on wall w1]

[Painting p2 on wall w2]

[Painting p1 on wall w1]

[Painting p2 on wall w2]

[Painting p1 on wall w1, Painting p2 on wall w2]

# Documentation and Coding Conventions

Follow the requirements described in the coding standards document under Assignments on D2L. Every **class** should be in a separate .java file. Don’t forget to document the constructors. Also, document the parameters and return values.

If a method simply sets a field *in the class* or gets the value of a field *in the class*, it need not be documented. Notice the highlighted phrases.

For example, in *Room*, the following method is not a getter.

A method that returns a *List* (java.util) of *Painting* objects displayed in this room

If a method overrides a superclass method, put the annotation @Override in front of that method. If you specify this annotation, there is no need to document the method.

Do not abbreviate any words except names of Exception objects (formed by using the first letter of each word in the Exception class) or the parameter to main (args). Do NOT use single letter identifiers such as i and j.

For more details, see the coding standards document.

# Submission

Submit the program as an Eclipse project. (The project must be modular.)

Submit using the following steps.

1. In the Eclipse Project Explorer window, right click on the project.
2. Click on Export. . .
3. In the window **for** “Export,” expand “General” and click on “Archive File.” Click Next.
4. In the next window, browse to any folder of your choice and any file name you wish.

Save the file as a zipped file.

1. Upload the zipped file to the dropbox **for** **this** assignment.

The project must be named <your-last-name>ICS372PA1. For example, mine would be named DathanICS372PA1.

# Grading

Your assignment will be graded as per the following distribution.

**Criteria Points**

Documentation 8

Coding standards 8

Modular project with proper naming conventions throughout 10

Correctness and structure of classes 24

If the submission is not a properly-named Java project, you will lose 10% of the grade.

Take a look at Class Exercise 1, regarding construction of a modular project and its naming conventions. This applies to module name, source file folder name, **package** name, and the binaries folder name.

I will examine your code and look **for** adherence to the requirements and correctness. It is hard to imagine every possible error you may make, but I can think of a few things.

* Improper access specifiers
* Wrong types **for** fields or **return** types
* Bugs
* Poor logic

I will test your program with a testing program. If you don’t name the class names properly and not have the correct signatures for the constructor and methods, you stand to lose a lot of credit. So don’t make silly mistakes such as spelling your classes wrongly or using the wrong case: for example, coding a class as museum instead of Museum.

Finally, **this** is the simplest assignment you can expect **this** semester. Try to score 50/50 **for** **this** and get the semester off to a great start.