CSCI-605 Advanced Object-Oriented Programming Concepts

Homework 4: Toy Store

1 Introduction

For this homework, you will be implementing a simulation of a toy store named *Al's Toy Barn*. Each morning, the Toy Store restocks its shelves with brand new, mint condition toys ordered from the *Fly By Night Toy Factory*. Unfortunately, Fly-By-Night is somewhat legendary for producing shoddy products that break after only being used a few times; the condition of their toys degrades each time that it is played with. That doesn't seem to matter to the eager customers that begin lining up before the store even opens each day. Each customer quickly buys the first toy that they see and plays with it. But Al's customers have very short attention spans, and they quickly become bored with their toys. After playing with them only once, they sell them back to Al at a reduced price. If a toy is broken, Al discards it rather than restocking it. Once the store runs out of toys in stock (because they are all broken), Al closes for the day.

1.1 Goals

This homework will help students to gain experience working with:

- Inheritance and Abstract Classes
- Interfaces
- Polymorphism
- Factory Method Design Pattern
- Access modifiers: public vs protected vs private
- Method overriding
- Superclass method calling

1.2 Provided Files

Note: None of the provided files should be modified in your solution.

- The **sample** folder contains sample runs.
- The **src** folder contains starter code.
  1. `AlsToyBarn` -This is the implementation of the toy store that is used in the simulation.
  2. `BatteryType` - An `enum` that defines the differet battery types that a toy may use (*AA*, *AAA*, *AAAA*, *C*, *D*, and *9-Volt*).
  3. `Condition` - An `enum` that defines the conditions that a toy may be in (*mint*, *near mint*, *very good*, *good*, *fair*, *poor*, and *broken*). In addition to defining the possible value's for a toy's condition, the `Condition enum` also defines the following methods:
     (a) `getMultiplier()` - Each possible condition has an associated multiplier that can be accessed using the condition's `getMultiplier` method. The resale value of a toy is determined by multiplying its MSRP[1] by this multiplier. For example, a toy that is in *very good* condition has a resale value of 75% of its MSRP.
     (b) `degrade()` - Returns the next lowest condition (unless the condition is *broken*, in which case *broken* is returned again).
  4. `IToy` - This is the interface that must be implemented by all toys. It defines the following methods:
     – Accessors for the various common toy attributes (i.e. *name*, *product code*, *MSRP*, and *condition*).
     – `getResaleValue()` - A toy's resale value is determined by its MSRP multiplied by its current condition's multiplier (see above).
     – `play()` - Called whenever a toy is played with. This will degrade the condition of the toy and print a message, (e.g. `"After play, Barbie's condition is NEAR_MINT"`).
  5. `ToyFactory` - A partial implemented toy factory.
  6. `ToyStory` - The main simulation class. It expects one command line argument: the number of toys to create for restocking the Toy Store's shelves.
  7. `Diff` - A simple utility that compares two text files line-by-line. The first time it finds two lines that do not match, it will print an error message. You may want to use this file to compare your output with the expected provided output.

2 Implementation

For this homework you must complete three main programming tasks (**note that *all* of your code will be in the `toy` package**):

1. Implement a separate class for each of the five varieties of toys: *scooters*, *dolls*, *action figures*, *RC cars*, and *robots*. To earn full credit for this task, it will not be sufficient

1. Manufacturer's Suggested Retail Price

to simply create 5 classes. Instead, you must make effective use of inheritance and maximize your code reuse.

2. Implement the `ToyFactory`, which provides a `makeToy` factory method to create a toy.
3. Run the simulation with 10, 50, and 100 toys and verify that your output matches the provided output examples (i.e. by using the provided `Diff` utility).

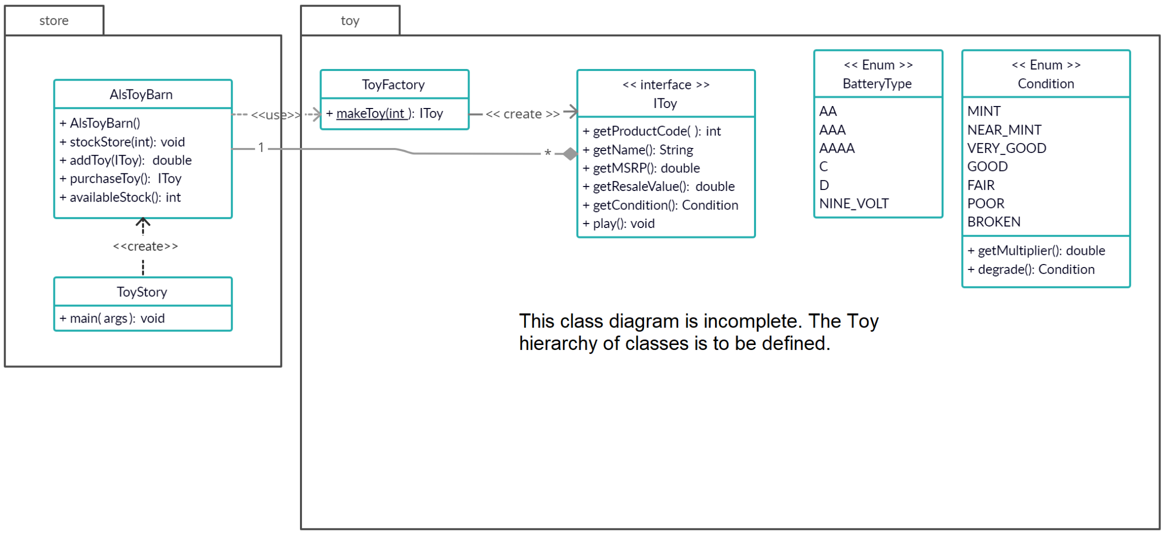
2.1 Design

You have been provided with some starter code. The diagram below depicts the design that you must follow for this homework. **Remember that you cannot modify the provided code.**

The class diagram excludes the private state and behavior. It also omits the entire Toy hierarchy of classes. You are responsible for defining the Toy hierarchy using good inheritance design principles. For example, a data member shared by all subclasses should be defined in the shared superclass. While working on the design, try to answer the following questions:

1. What are the attributes common to all toys?
2. What behaviors should all toys be able to do?
3. What attributes and behaviors are unique to a particular toy?
4. Which toys will need to customize the way they do something different from other toys, and what are those customizations?

Note: You do not need to submit the answer to all the questions above. Those questions are meant to help you designing your program.



2.2 Activity 1: The Toy Implementations

You will be required to implement a class for each of the required variety of toys. Remember, *all* of the code that you write should be in the `toy` package. Your implementations must meet the specifications below:

- **Scooter**
  - The 7-digit product code always starts with a ”9” and is automatically assigned when the scooter is manufactured.
  - Must have a color (string).
  - Must have 2 or 3 wheels.
  - Must have an odometer that keeps track of the total miles that the scooter has been used.
  - The string representation must match the following example: `"Razor A [product code=9000001, MSRP=130.01, condition=BROKEN, resale value=0.00, color=Pink, wheels=TWO, odometer=12]"`.
  - Each time the `play` method is called, two miles are added to the scooter's odometer and its condition degrades. It should produce output that matches the following example: `"After play Razor A's condition is VERY_GOOD"`.
- **Doll**
  - The 7-digit product code always starts with a ”3” and is automatically assigned when the doll is manufactured.
  - Must have a hair color (string).
  - Must have an eye color (string).
  - The string representation must match the following example: `"Babs [product code=3000001, MSRP=48.80, condition=BROKEN, resale value=0.00, hair color=Blond, eye color=Brown]"`.
  - Each time the `play` method is called its condition degrades. It should produce output that matches the following example: `"After play Babs's condition is VERY_GOOD"`.
- **Action Figure**
  - The 7-digit product code always starts with a ”5” and is automatically assigned when the action figure is manufactured.
  - Must have a hair color (string).
  - Must have an eye color (string).
  - May or may not have Kung-Fu Grip^{TM}.
  - The string representation must match the following example: `"G.I. Barbie [product code=5000000, MSRP=24.87, condition=MINT, resale value=24.87, hair color=Red, eye color=Green, kung-fu grip=true]"`.
  - Each time the `play` method is called its condition degrades. It should produce output that matches the following example: `"After play G.I. Barbie's condition is VERY_GOOD"`.
- **RC Car**
  - The 7-digit product code always starts with a ”6” and is automatically assigned when the RC Car is manufactured.
  - Must have a battery type.
  - Must have a number of batteries.
  - Must have a scale speed (in MPH).

  - The string representation must match the following example: `"METAKOO RC [product code=6000002, MSRP=74.21, condition=NEAR_MINT, resale value=66.78, battery type=NINE_VOLT, number of batteries=2, battery level=75%, speed=133]"`.
  - Each time the `play` method is called, its battery gets depleted 25% and its condition degrades. It should produce output that matches the following example:
    ```
    "METAKOO RC races in circles at 133 mph!
    After play, METAKOO RC's condition is NEAR_MINT".
    ```

- **Robot**
  - The 7-digit product code always starts with a "7" and is automatically assigned when the RC Car is manufactured.
  - Must have a battery type.
  - Must have a number of batteries.
  - Must have a sound that it makes when played with.
  - The string representation must match the following example: `"Roomba [product code=7000002, MSRP=446.14, condition=NEAR_MINT, resale value=401.52, battery type=D, number of batteries=4, battery level=85%, sound=Eject!]"`.
  - Each time the `play` method is called, its battery gets depleted 15% and its condition degrades. It should produce output that matches the following example:
    ```
    "Roomba goes 'Eject!'
    After play, Roomba's condition is POOR".
    ```

2.3 Activity 2: `ToyFactory`

We want to design our program in such way that the class `AlsToyBarn` can create different type of toys without worrying about the concrete classes must instantiate. A popular pattern in object oriented programming to do that is the **Factory Method**.

The constructors of all the classes created in Activity 1 must not be accessible from the `store` package. The `ToyFactory` class provides a `public static` method, `makeToy`. This allows a client to create a new toy of any type without worrying about the construction details, for example:

```
// create a robot for the store
// the type code to make a robot is number 5
IToy robot = ToyFactory.makeToy(5);
```

From here, the `ToyFactory.makeToy` factory method would construct the robot by calling the `Robot` constructor to create an instance and return that as a `IToy` reference:

```
if (type == ROBOT) {
    return new Robot(...);
}
```

Note that none of the methods must accept a product code as a parameter. Each toy class is responsible for assigning its own, unique product codes to each toy as it is created (see above).

2.4 Activity 3: Run the Simulation

For this activity you will run the `ToyStory` simulation and verify that it produces the expected output. Example output has been provided to you for your reference in the `sample` folder. The output files include sample runs with 10, 50, and 100 toys. Your output should match these closely enough that the `Diff` utility does not find a difference between your output and the provided examples.

3 Testing

In order to receive the full bonus credit for testing, you will need to write a unit test in the `test` folder for each of the classes that you create in the `toy` folder. The unit test should have one test method for each of the methods in your class (not counting getters & setters; there is no need to write a test for each of those, and they will likely be tested as part of your other tests). If you need a refresher on unit testing, please refer to the instructions from the previous homeworks.

4 Submission

You will need to submit all of your code, including your tests to the MyCourses assignment before the due date. You must submit your solution as a ZIP archive named "hw4.zip" (if you submit another format, such as 7-Zip, WinRAR, or Tar, you will not receive credit for this homework).

5 Grading

The grade breakdown for this lab is as follows:

- Implementation: 75%
  - Activity 1: 50%
  - Activity 2: 15%
  - Activity 3: 10%
- Testing: 15%
- Code Documentation & Style: 10%