Design Patterns and Suggestions for Code Improvement

Strategy Design Pattern:

We could have written code more efficiently if we knew about this design pattern prior to writing the code.

In case of our project Strategy Design Pattern could have been implemented in the following manner:

As one can see, in our current implementation we have assigned different display() and move() methods to all end subclasses of objects eg. Slope, Cylinder, Ring, Box, Ball.
So instead of creating so many subclasses we could have simply created an interface for example “TypeOfObj” and have classes like Slope, Ring, Ball etc. implement it, each having it’s own display and move method while the Object class will have an attribute “TypeOfObj objectType” which will be provided to it’s constructor as eg.

```
   Obj ring = new Obj(PApplet _p5, int id, Vector pos, double radius, Vector vel, new
Ring());
```

This is how the TypeOfObj interafce will look like:

```
   interface TypeOfObj{
       void move();
       void display();
       }
```

New constructor for Obj class will look like:

```
   public Obj(PApplet _p5, int id, Vector pos, double radius, Vector vel, TypeOfObj
objectType) {
       // double posData[] = { 250, 250 };

       this.pos = pos;
       this.radius = radius;
       this.objectType = objectType;

       this.vel = vel;
       System.out.println("vel=" + vel);
       double[] accData = { 0, 0 };
       acc = new Vector(accData);
       p5 = _p5;
       this.id = id;
   }
```

Now the Obj class will have the display() and move() methods which will be implemented as follows:

```
   display(){
       objectType.display();
       }

   move(){
       objectType.move();
       }
```

By doing this we will have made use of the following efficient design principles like

1) Encapsulate what varies:
In the above implementation we have encapsulated the move() and the display() methods which vary for each object.

2) Program to an interface not implementation:
Instead of calling move() or display() method conditionally based on the type of object we are programming to an interface namely “TypeOfObj” by having a code structure in which every instance of Obj has a “TypeOfObj objectType” attribute.

3) Favour composition over Inheritence:
We will have favoured composition over inheritance as instead of inheriting the “RollingObject” class we now have an attribute “objectType” which favours composition while getting the job done.

Best practices which our code already follows:

4) Classes should be open for extension and closed for modification:
In the current code if we are required to create new objects we can simply extend already extending basic classes like SlidingObject and create them instead of modifying methods/attributes in existing classes.

Best practices which we could have implemented:

5) Striving for loose coupling between objects that interact:
In our present code the “Slope” class and the “Obj” class interact with each other. Both these classes are tightly coupled i.e. eg. “applynormal()” or “applyfric()” method of class “Slope” are being

called in Obj class.
So to loosely couple them we can create an interface which contains methods “applynormal()” ,“applyfric()” and have the Slope class implement it.

6) Depend on abstraction, do not depend on concrete classes:
In our code we have “Obj” class which could have been turned into an abstract class so that we reduce our dependance on concrete classes.