**Problem 1**

**Situation Description:**

There are three event – **Collision, Level Up and Arriving**, and two game element – **Player and Enemy**.

When the event Collision occurs, the enemy which hit the player will die, and the player which is hit will reduce its health.

When the event Level Up occurs, all of the enemies will reduce their damage, and the player whose level is up will increase its max health and change its current health to the max health.

When the event Arriving occurs, all of the enemies will die, and the player which reached the end will change its current health to the max health.

**Problem 2**

**Problem 3**

**b (i)**

We should determine current tool type at first when call the gradient() method.

If the returned value of tool.getType() is Line, then call canvas.gradientLine().

If the returned value of tool.getType() is Rectangle, then call canvas.gradientRectangle().

If the returned value of tool.getType() is Circle, then call canvas.gradientCircle().

**b (ii)**

Add a new value representing the new shape to enum ToolType.

Define a new concrete class, which implements the interface ITool.

Change the returned value of next in CircleTool to an object of the new class.

The returned value of getType in the new class is the new value of ToolType.

The returned value of next in the new class is a LineTool object.

The returned value of prev in the new class is a CircleTool object.

New situations should be add to the methods draw and fill in ToolSet for drawing or filling the new shape.

**Problem 4**

The responsibilities of each Displayer are the same. However, based on the original design, we need to write the code for the display method and the getter method for Element in the new subclass’s displayer class every time you add a new subclass of Element. In such a design, the code reusability is poor and not concise enough.

But if we simply extract an ElementDisplayer class up, and its one indispensable data field’s type should be Element, according to the three existing Displayer classes and its responsibilities. This prevents us from directly using subclass specific properties or methods and requires cast instead. If we pass the wrong type into its implementation class, it could cause a runtime error.

We can effectively solve the above problems, avoid ClassCastExceptions at runtime, and improve code reuse and brevity by using generics for type checking at compile time.

The class diagram after refactoring as following:

