Preconditions in the context of Inheritance

Franck van Breugel

November 14, 2007

Abstract

When one overrides a method, one should not strengthen the precondition. An example is given to demonstrate this.

1 The Example

Let us assume that you have implemented the following `Rectangle` class.

```
/**
 * A class for representing a rectangle.
 */
public class Rectangle
{
  private int width;
  private int height;

  /**
   * Scale this rectangle with the given factor.
   *
   * @param factor Scaling factor.
   * @pre. factor >= 0
   */
  public void scale(int factor)
  {
    this.width *= factor;
    this.height *= factor;
  }
}
```

Another implementer wrote a `Factory` class containing a method

```
public static Rectangle getRandomRectangle()
```

which returns a `Rectangle` with random width and height.

A client exploits the above classes in an app. Below, we only present a snippet of the main method.

```
Rectangle rectangle = Factory.getRandomRectangle();
rectangle.scale(0);
```

When the client runs the app, an exception is thrown by the scale method. After inspecting the API of your Rectangle class, the client blames you for the exception since the client has satisfied the precondition of your scale method.

How is it possible that the scale method throws an exception? Are you to blame for the exception?

After inspecting your scale method, you are convinced that it cannot have thrown the exception. Hence, you are not to blame. However, the client is not to blame for the exception either. So, who is to blame?

The implementer of the Factory class also wrote the following MyRectangle class.

```
public class MyRectangle extends Rectangle
{
  /**
   * Scale this rectangle with the given factor.
   *
   * @param factor Scaling factor.
   * @pre. factor > 0
   */
  public void scale(int factor)
  {
     if (factor == 0)
     {
        throw new RuntimeException();
     }
     else
     {
        super.scale(factor);
     }
  }
}
```

The implementer of the Factory class implemented the getRandomRectangle method in such a way that it returns a MyRectangle object which is-a Rectangle. As a consequence, at compile time, the method call rectangle.scale(0) is bound to the scale method of the Rectangle class. However, at run time, the method call rectangle.scale(0) is bound to the scale method of the MyRectangle class. Hence, it throws an exception.

Note that the precondition of the scale method in the MyRectangle class strengthens the precondition of the scale method in the Rectangle class: factor > 0 is stronger than factor >= 0 since the former implies the latter. As a consequence, the Rectangle class guarantees that the scale method works as expected if the argument 0 is provided, whereas the MyRectangle class does not. However, a MyRectangle object is-a Rectangle and, hence, should behave like a Rectangle. In particular, its scale method works as expected if the argument 0 is provided. Therefore, we blame the implementer of the MyRectangle class for the exception: the precondition of the scale method should not have been strengthened.

In a subclass, one may weaken the precondition as is shown in the following alternative implementation of the MyRectangle class.

```
public class MyRectangle extends Rectangle
{
  /**
   * Scale this rectangle with the given factor.
   * If the factor is negative then its absolute value is used.
   *
   * @param factor Scaling factor.
   * @pre. true
   */
  public void scale(int factor)
  {
    super.scale(Math.abs(factor));
  }
}
```