Team:
Jensen Eicher
Alperen Ugus

C26A More Principles (Principle Pair)

Part 1 (7pts):

```java
public class Rectangle {

    private int topLeft;
    protected int width;
    protected int height;

    public int area() {
        return width * height;
    }

    public int getTopLeft() {
        return topLeft;
    }
    public void setTopLeft(int topLeft) {
        this.topLeft = topLeft;
    }
    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        this.width = width;
    }
    public int getHeight() {
        return height;
    }
    public void setHeight(int height) {
        this.height = height;
    }
}
```

```java
public class Square extends Rectangle{

    @Override
    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
```

```
    }
}
```

```
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class AreaTests {

    @Test
    public void verifyRectangleArea() {
        Rectangle rectangle = new Rectangle();
        rectangle.setHeight(5);
        rectangle.setWidth(4);
        assertEquals(20, rectangle.area());
    }

    @Test
    public void verifySquareArea() {
        Square square = new Square();
        square.setHeight(5);
        assertEquals(25, square.area());
    }
}
```

Part 2 (7pts):

The answer to the question depends on the implementation of the method. According to LSP, child classes should not break the type definitions of the parent classes. So, there are three things we should pay attention to: Input parameters, return types and exceptions. For all these cases, child classes should protect the existing parameters and their types but it is possible to add more input parameters. For example, assume that a parent abstract class takes two integer parameters. The child class can implement this method with three parameters including the first two being the same as in the parent class. Same thing applies for the exceptions, too. A child class can throw the same exception or one with a more detailed message.

Part 3 (7pts):

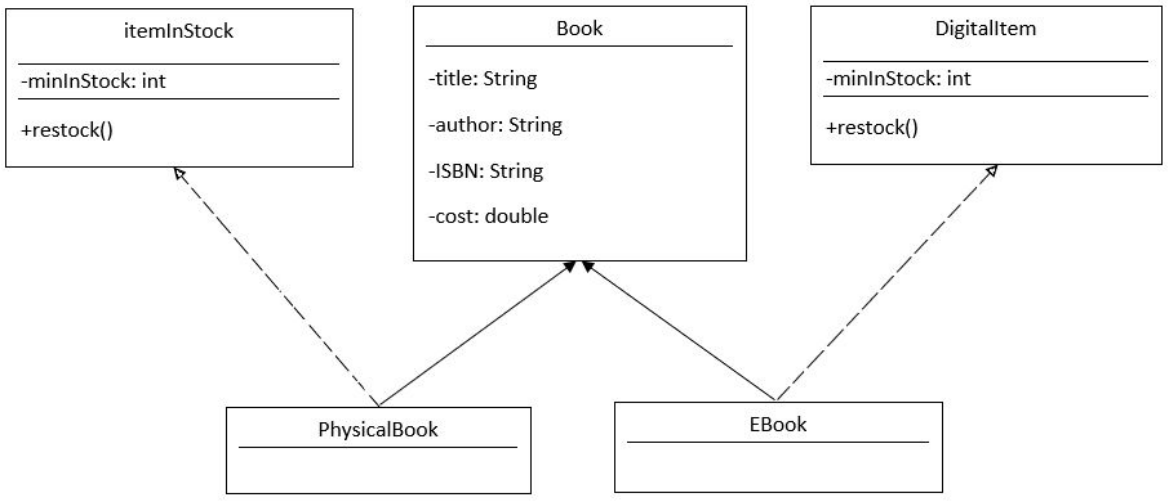
Inheritance should be used when one class is a type of another (“is a” or “has a” relationship). A composition is much more about the capabilities of the types, i.e. what they can do. In this example, if Stack extends the LinkedList class, first question to answer is, does Stack need to have the complete interface of LinkedList class or will it make use of some of them. Since Stack does not need for example adding or removing an element from the end, it will not use all the features of LinkedList class. It will just use

adding to the top and removing from the top as LIFO. So, composition is better in this case. Looking through the perspective of LSP, the things to put a base class should most likely be a common attitude for children. So, again, composition seems to be better because LinkedList class is not a general interface.

Part 4 (7pts):

1. This UML violates ISP (Interface Segregation Principle) because EBook inherits minInStock and restock(). Having these methods inherited violates ISP due to the fact that restock() and minInStock are useless in regards to an EBook. The number of copies in stock for an EBook is unlimited and they never need to be restocked.

2. The new UML that conforms to ISP



Part 5 (7pts):

No, adding two child classes that inherit the parent class with no additional methods or variables simply over complicates the program. We could implement this program in a more compact way by adding another variable into student called “major.” An example where we may need an extra child class would be “GraduateStudent.” This would have all the same functionality as a regular student but with additions that are specific to a grad student.

Part 6 (7pts):

False, similarly to the example in Part 4, ISP can be violated through inheritance. In C++ a child class could simply inherit useless/unneeded methods from the parent which would violate ISP.

Part 7 (8pts):

A.

True, an abstract class provides a blueprint along with implemented methods that will always appear in its children. This allows for some attributes to stay generic with others being specified. If a class is a pure abstract class it can be changed to an interface and if a class is fully defined it should be declared as a regular non-abstract class.

B.

True, an interface in java is a blueprint of a class. It defines the outline of what a class will do but not how it will do it (implementation). An interface is restricted to only function headers and variables and if a method is wished to be implemented or excluded from the classes that will extend it, it must be declared as abstract, not an interface. C++ does not need interfaces since it does not allow multiple inheritance (diamond of death).