Herencia y principio de sustitución de Liskov

Recuerda que para validar o reflexionar sobre nuestro el diseño de clases de una aplicación tenemos entre otras herramientas intelectuales los 5 principios SOLID. Uno de ellos es el principio de sustitución de liskov
Es un principio anterior a la POO pero aplicado a la POO podría enunciarse como sigue(es pues una adaptación respecto al principio original):

Si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido.

Lo de "siga siendo válido" hay que interpretarlo correctamente, matizamos esto en el ejemplo.

En definitiva este principio es una herramienta, entre otras, que nos sirve para saber "si estamos utilizando bien la herencia"

Vamos a aplicar este principio para examinar un ejemplo en el que hay un "extend" mal diseñado (pero hay que recordar que en otros contextos en el que este principio puede ser útil). El ejemplo es corto y sencillo y por tanto forzado. Los problemas se ven claros y evidentes en aplicaciones grandes, así que con este tipo de ejemplos tenemos que "ponernos en situación" y consentir hipótesis forzadas.

Ejemplo:

```
class Animal{
   void correr(){
      System.out.println("el animal corre");
   }
   void saltar(){
      System.out.println("el animal salta");
   }
}
class Pantera extends Animal{
   void arañar(){
      System.out.println("el animal araña");
   }
}
class Elefante extends Animal{
   void pastar(){
      System.out.println("el elefante pasta");
   }
}
public class App {
   static void comportamientoAnimal(Animal a){
      a.correr();
      a.saltar();//¡esto hace saltar a los elefantes!
   }
   public static void main(String[] args) throws Exception {
      Animal a= new Animal();
      comportamientoAnimal(a);
      Elefante e= new Elefante();
      //si sustituyo un objeto de tipo Animal por uno derivado como Elefante todo OK?
      comportamientoAnimal(e);
   }
```

```
}
```

Es cierto que *“un Elefante es un Animal”* y entonces puede pensar que hay una relación “is a”, es decir, un extends java, pero esta afirmación no llega para modelar bien la jerarquía anterior ya que resulta que según esto un Elefante tiene que saltar y los elefantes no saltan.

UNA SOLUCIÓN AL PROBLEMA ANTERIOR, PERO ES UNA MALA SOLUCIÓN:
sobreescribir el método saltar del elefante para indicar que no salta

```
class Animal{
    void correr(){
        System.out.println("el animal corre");
    }
    void saltar(){
        System.out.println("el animal salta");
    }
}
class Pantera extends Animal{
    void arañar(){
        System.out.println("el animal araña");
    }
}
class Elefante extends Animal{
    @Override
    void saltar(){
        System.out.println("El elefante no salta");
    }
    void pastar(){
        System.out.println("el elefante pasta");
    }
}

public class App {
    static void comportamientoAnimal(Animal a){
        a.correr();
        a.saltar();//¡esto hace saltar a los elefantes!
    }
    public static void main(String[] args) throws Exception {
        Animal a= new Animal();
        comportamientoAnimal(a);
        Elefante e= new Elefante();
        //si sustituyo un objeto de tipo Animal por uno derivado como Elefante todo OK?
        comportamientoAnimal(e);
    }
}
```

Importante: cuando en las subclases, nos vemos obligados a sobrescribir métodos con advertencias como la anterior o lanzando excepciones que advierten de que no se puede implementar el método(veremos más adelante que es una excepción) delata que violamos el principio de sustitución de Liskov

interpretar “siga siendo válido”

Ahora interpretamos lo de “siga siendo válido” del enunciado del principio. En el caso anterior, donde usábamos Animal pasamos a usar wlefante y ocurrió que finalmente se invocó a un método sin sentido para un elefante por lo tanto consideramos que el código no

sigue siendo válido al realizar la sustitución ya que detectamos la subclase hereda métodos inapropiados para ella. La validez hay pues que verla pues como algo semántico y lógico, no se puede considerar que es válido por ejemplo por que no haya errores sintácticos al hacer la sustitución.

SOLUCIÓN MÁS APROPIADA:
Si la cuestión del salto es tan importante para nuestra App tenemos que agregar un nivel a la jerarquía

En el ejemplo anterior la solución pasa por crear otro nivel de jerarquía pero puede haber otras soluciones más complejas en otros contextos lo que ya trasciende a este boletín.

```
class Animal{
   void correr(){
      System.out.println("el animal corre");
   }

}
class AnimalSaltarin extends Animal{
   void saltar(){
      System.out.println("el animal salta");
   }
}
class Pantera extends AnimalSaltarin{
   void arañar(){
      System.out.println("el animal araña");
   }
}
class Elefante extends Animal{

   void pastar(){
      System.out.println("el elefante pasta");
   }
}

public class App {
   static void comportamientoAnimal(Animal a){
     a.correr();//ok si es Elefante
   }
   public static void main(String[] args) throws Exception {
      Animal a= new Animal();
      comportamientoAnimal(a);
      Elefante e= new Elefante();
      //si sustituyo un objeto de tipo Animal por uno derivado como Elefante todo OK?
      comportamientoAnimal(e);
   }
}
```