Objetos 2 - Examen final

Ejercicio 1. Responda verdadero (V) o falso (F) en cada caso:

( ) El mal olor "God class" (Clase dios) es similar al mal olor "Data class" (Clase de datos).
( ) El patrón Composite tiene una estructura de clases similar a la de Decorator
( ) El patrón "Null Object" es uno de los patrones conocidos como "wrappers".
( ) El refactoring "Move Method" permite eliminar el mal olor "Data class".
( ) El refactoring "Form Template Method" permite eliminar el code smell de código duplicado.
( ) Un framework de caja negra se reúsa a través de la subclasificación.

Ejercicio 2. Suponga un sistema que permite aplicar distintas funciones o filtros sobre datos de entrada. El sistema está compuesto de filtros que se conectan entre sí de diferente forma para poder ir aplicando las funciones sobre los datos en una secuencia. Existe una jerarquía de distintos tipos de filtros, cuya raíz es la clase Filter (subclases SelectionFilter, ArithmeticFilter, SortingFilter, etc.). Parte del código de este sistema se encuentra en la página 2.

En su grupo de trabajo revisan el código y alguien opina lo siguiente:

a- "El único code smell que tiene este código es el uso de sentencias condicionales, pero es tan grav
que se debe refactorizar". Usted qué opina?

Totalmente de acuerdo Parcialmente de acuerdo En desacuerdo

b- "La lista de pipes podría representarse con un Composite, entonces podemos aplicar el
refactoring "Extract Composite". Hacemos una jerarquía de pipes, con una superclase Pipe, una
subclase SimplePipe que cumpliría el rol de hoja, y una subclase CompositePipe que a su vez ti
subclases ANDPipe, ORPipe y NEXTPipe. Y así eliminamos el smell de sentencias condicionales
patrón Composite". Usted qué opina?

Totalmente de acuerdo Parcialmente de acuerdo En desacue

c- "Deberíamos mover el método sendToNext() a la jerarquía de Pipe. Asi el filtro delegarí
pipe el envío de los datos a el/los modulos siguientes". Usted qué opina?

Totalmente de acuerdo Parcialmente de acuerdo En des

Parcialmente de acuerdo o En desac

Ejercicio 2. Suponga un sistema que permite aplicar distintas funciones o filtros sobre datos de entrada. El sistema está compuesto de filtros que se conectan entre sí de diferente forma para poder ir aplicando las funciones sobre los datos en una secuencia. Existe una jerarquía de distintos tipos de filtros, cuya raíz es la clase Filter (subclases SelectionFilter, ArithmeticFilter, SortingFilter, etc.). Parte del código de este sistema se encuentra en la página 2.

En su grupo de trabajo revisan el código y alguien opina lo siguiente:

a- "El único code smell que tiene este código es el uso de sentencias condicionales, pero es tan grave que se debe refactorizar". Usted qué opina?

Totalmente de acuerdo Parcialmente de acuerdo En desacuerdo

b- "La lista de pipes podría representarse con un Composite, entonces podemos aplicar el refactoring "Extract Composite". Hacemos una jerarquía de pipes, con una superclase Pipe, una subclase SimplePipe que cumpliría el rol de hoja, y una subclase CompositePipe que a su vez tiene subclases ANDPipe, ORPipe y NEXTPipe. Y así eliminamos el smell de sentencias condicionales con el patrón Composite". Usted qué opina?

Totalmente de acuerdo Parcialmente de acuerdo En desacuerdo

c- "Deberíamos mover el método sendToNext() a la jerarquía de Pipe. Asi el filtro delegaría en su pipe el envío de los datos a el/los modulos siguientes". Usted qué opina?

Totalmente de acuerdo Parcialmente de acuerdo En desacuerdo

Si en alguna de las 3 expresiones de arriba marcó Parcialmente de acuerdo o En desacuerdo justifique.

```
public abstract class Filter {
  private String pipeType;
  private List<Filter> pipeList;
  private int next;

  public void processInput(List<Float> colD) {
        //colD es una colección de datos a procesar y enviar al/a los filtros conectados
        List<Float> result = this.apply(colD);
        this.sendToNext(result);
  }

  abstract public List<Float> apply(List<Float> colD);  //redefinido en las subclases

  public void sendToNext(List<Float> colD) {
        int candidate;
        Random rand ;
        if (pipeType == "simple")
                pipeList.get(1).processInput(colD);
        if (pipeType == "AND")
                pipeList.forEach(f -> f.processInput(colD));
        if (pipeType == "OR") {
                rand = new Random();
                candidate = rand.nextInt(pipeList.size()) ;
                pipeList.get(candidate).processInput(colD);
        }
        if (pipeType == "NEXT") {
                (pipeList.get(next).processInput(colD);
                if (next < pipeList.size())
                        next = next +1;
                else
                        next = 1;
        }
}}
```

Luego su compañero se ofrece a hacer el refactoring y vuelve al rato con el código que aparece
hoja 3, diciendo "solo falta que agregues la clase para el NextPipe."
Elija una opción en cada inciso siguiente con respecto a la solución que escribió su compañero.
Además justifique en los casos que no esté totalmente de acuerdo y corrija el código según su
justificación. También agregue el código para la clase NextPipe.

d- La jerarquía que definió es correcta

Totalmente de acuerdo Parcialmente de acuerdo En desacuerd

e- El patrón Composite quedó correctamente definido

Totalmente de acuerdo Parcialmente de acuerdo

Luego su compañero se ofrece a hacer el refactoring y vuelve al rato con el código que aparece en la hoja 3, diciendo "solo falta que agregues la clase para el NextPipe."
Elija una opción en cada inciso siguiente con respecto a la solución que escribió su compañero. Además justifique en los casos que no esté totalmente de acuerdo y corrija el código según su justificación. También agregue el código para la clase NextPipe.

d- La jerarquía que definió es correcta

Totalmente de acuerdo Parcialmente de acuerdo En desacuerdo

e- El patrón Composite quedó correctamente definido

Totalmente de acuerdo Parcialmente de acuerdo En desacuerdo

f- Las variables de instancia quedaron en el lugar correcto y no faltan ni sobran parámetros

Totalmente de acuerdo Parcialmente de acuerdo En desacuerdo

g- No quedan más refactorings por aplicar

Totalmente de acuerdo Parcialmente de acuerdo En desacuerdo

```
public abstract class Filter {
  private Pipe p;
  private List<Filter> pipeList;
  private int next;

  public void processInput(List<Float> colD) {
        //colD es una colección de datos a procesar y enviar al/a los filtros conectados
        List<Float> result = this.apply(colD);
        p.sendToNext(pipeList, next, result);
  }

  abstract public List<Float> apply(List<Float> colD);
}

public abstract class Pipe {
  abstract public void sendToNext(List<Filter> pipeList, int next, List<Float> colD);
}

public class SimplePipe extends Pipe {
  public void sendToNext(List<Pipe> pipeList, int next , List<Float> colD) {
        pipeList.get(1).processInput(colD);
  }
}

public abstract class CompositePipe extends Pipe {
}

public class ANDPipe extends CompositePipe {
  public void sendToNext (List<Pipe> pipeList, int next , List<Float> colD) {
        pipeList.forEach(f -> f.processInput(colD));
  }
}

public class ORPipe extends CompositePipe {
  public void sendToNext (List<Pipe> pipeList, int next , List<Float> colD) {
        int candidate;
        Random rand = new Random();
        candidate = rand.nextInt(pipeList.size()) ;
        pipeList.get(candidate).processInput(colD);
  }
}
```