

Nombre: _____ Matrícula: _____

Sección A – 30%

Justifique su selección de la respuesta. Uso de lápiz o tachones en la selección invalida la respuesta.

1. Considere un sistema de procesamiento de pagos en el cual las transacciones deben pasar por una serie de validaciones y aprobaciones antes de ser procesadas. Cada paso en el proceso puede aprobar, rechazar o pasar la transacción al siguiente paso en lista de validaciones. ¿Qué patrón de diseño es aplicable en este contexto? [05%]
 - a. Iterator
 - b. Singleton
 - c. Adapter
 - d. Builder
 - e. Chain of Responsibility
 - f. Composite
2. En un software de música, hay múltiples efectos que se pueden aplicar a una pista de audio, como eco, reverberación y compresión. Estos efectos deben poder ser aplicados de manera dinámica y combinados en diferentes secuencias según la elección del usuario, sin modificar la estructura interna de la pista original. ¿Qué patrón de diseño es aplicable en este contexto? [05%]
 - a. Facade
 - b. Strategy
 - c. Builder
 - d. Decorator
 - e. Composite
 - f. Iterator
3. ¿Qué **code smell** describe mejor el problema del código adjunto? [05%]
 - a. Feature Envy
 - b. Primitive Obsession
 - c. Temporary Field
 - d. Long Method
 - e. Inappropriate Intimacy

```
public class Admin {  
    private String permisos = "";  
  
    public boolean tienePermiso(String permiso) {  
        String[] listaPermisos = permisos.split(regex:",");  
        for (String p : listaPermisos) {  
            if (p.trim().equalsIgnoreCase(permiso)) {  
                return true;  
            }  
        }  
        return false;  
    }  
    // Getters y Setters  
}
```

4. ¿Cuál de las siguientes técnicas de **refactoring** sería la más adecuada para mejorar esta clase? [05%]
 - a. Replace Magic Number with a Constant
 - b. Replace Type Code with Subclasses
 - c. Separate Query from Modifier
 - d. Replace Method with Method Object
 - e. Preserve Whole Object
 - f. Encapsulate Collection

```
public class ProductReviews {  
    private String productName;  
    private ArrayList<String> reviews;  
  
    public ArrayList<String> getReviews() {  
        return reviews;  
    }  
    public void addReview(String review) {  
        if (review != null && !review.trim().isEmpty()) {  
            reviews.add(review);  
        }  
    }  
    public void deleteReview(String rev) {  
        if (rev != null && !rev.trim().isEmpty()) {  
            reviews.remove(rev);  
        }  
    }  
    // Otros métodos  
}
```

5. ¿Qué **code smell** describe mejor el problema del código adjunto? [05%]

- a. Message Chains
- b. Primitive Obsession
- c. Feature Envy
- d. Inappropriate Intimacy
- e. Divergent Change

```
class Order {  
    private User user;  
  
    public String getUserCity() {  
        return user.getAddress().getCity();  
    }  
    // Getters y Setters  
}
```

6. ¿Cuál de las siguientes técnicas de **refactoring** es la más adecuada para mejorar este diseño? [05%]

- a) Replace Magic Number with a Constant
- b) Extract Class
- c) Replace Type Code with Subclasses
- d) Encapsulate Field
- e) Separate Query from Modifier
- f) Consolidate Conditional Expression

```
public class Product {  
    private String name;  
    private double price;  
    private int type;  
    private ProductReviews reviews;  
  
    public int calculateDiscount(int quantity) {  
        if (type == 1) {  
            return (int)(price * 0.10 * quantity);  
        } else if (type == 2) {  
            return (int)(price * 0.20 * quantity);  
        } else {  
            return 0;  
        }  
    }  
  
    public String getProductDetails() {  
        if (type == 1) {  
            return name + " - Physical Product: $" + price;  
        } else if (type == 2) {  
            return name + " - Digital Product: $" + price;  
        }  
        return name + " - Unknown Product Type";  
    }  
    // Otros métodos  
}
```

Sección B – 30%

7. Considere la descripción del escenario que se presenta a continuación.

Una cafetería se dedica a la venta de bebidas de café, postres, sándwiches, batidos, etc. El diseño de su sistema de software debe permitir ordenar una bebida de café básica. No obstante, un cliente puede personalizar esta bebida agregando varios adicionales, tales como leche, azúcar, crema o saborizante. Cada característica adicional modifica el costo o sabor del café básico.

Para cautivar a los niños, las órdenes se despachan en utensilios diferenciados para niños y para adultos. Específicamente, estos utensilios son: vasos, platos y cubiertos. De esta manera, los niños reciben utensilios atractivos con personajes infantiles de temporada.

Al momento de cobrar por las órdenes, por razones logísticas, la cafetería no acepta pagos en efectivo. Un cliente puede realizar el pago de su orden solo por medio de uno de los canales digitales disponibles: tarjeta de crédito, billetera móvil o PayPal. Se incorporarán nuevos canales en el futuro.

- a. Identifique los patrones de diseño aplicables. Justifique su respuesta. [12%]
- b. Elabore un diseño utilizando diagramas de clases que muestren solo la parte relevante a los patrones de diseño seleccionados. En donde sea posible, identifique herencias, multiplicidades, visibilidad de atributos y métodos. Indique, por medio de notación UML, si las entidades corresponden a interfaces, clases abstractas o clases concretas. Indique cualquier asunción que realice. [15%]
- c. Separe en paquetes según los patrones usados. [03%]

Sección C – 40%

8. Considere el código fuente que se presenta a continuación.

- Identifique los malos olores de programación en el código e indicar su nombre. Explique por qué sería un problema. **[16%]**
- Indique las técnicas de refactorización que utilizaría para mejorar el código. Justifique su respuesta. **[16%]**
- Escriba 2 pruebas unitarias en *JUnit 5* para el método *hasNext()* de la clase *FormattedIterator*. Indique el propósito de cada prueba. **[08%]**

```
1 package main;
2
3 import java.util.stream.Stream;
4
5 public class IteratorPatternDemo {
6
7     public static void main(String[] args) {
8         NameRepository namesRepository = new NameRepository();
9
10        for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
11            String name = (String)iter.next();
12            System.out.println("Name : " + name);
13        }
14    }
15 }
16
17 class NameRepository implements Container {
18     public String names[] = {"Robert" , "John" , "Julie" , "Lora"};
19
20     @Override
21     public Iterator getIterator() {
22         return new NameIterator();
23     }
24
25     //Override
26     public Iterator getAlternativeIterator() {
27         return new FormattedIterator();
28     }
29
30     private class NameIterator implements Iterator {
31         int index=0;
32
33         @Override
34         public boolean hasNext() {
35             if(index < names.length){
36                 return true;
37             }
38             return false;
39         }
40
41         @Override
42         public Object next() {
43             if(this.hasNext()){
44                 return names[index++];
45             }
46             return null;
47         }
48
49         @Override
50         public void sort() {
51             throw new UnsupportedOperationException();
52         }
53
54     }
55 }
```

```

56 private class FormattedIterator implements Iterator {
57     int index=0;
58     Boolean hasNext;
59
60     public FormattedIterator() {
61         sort();
62         getIndexAndInitialiseIndex();
63     }
64
65     @Override
66     public boolean hasNext() {
67         if(index < names.length){
68             hasNext = true;
69         }
70         return hasNext.booleanValue();
71     }
72
73     @Override
74     public Object next() {
75         if(this.hasNext()){
76             return names[index++];
77         }
78         return null;
79     }
80
81     @Override
82     public void sort() {
83         names = Stream.of(names).sorted().toArray(String[]::new);
84     }
85
86     public int getIndexAndInitialiseIndex(){
87         index=0;
88         return index;
89     }
90
91 }
92 }
93
94 interface Iterator {
95     public boolean hasNext();
96     public Object next();
97     public void sort();
98 }
99
100 interface Container {
101     public Iterator getIterator();
102 }

```