Ejercicios Propuestos - Patrones de Diseño
Ingeniería de Software II
2015

1. Cree una solución que diseñe e implemente una clase de conexión a una base de datos llamada *DBConnection* como un *singleton.* Luego cree un programa cliente que tenga la siguiente forma:

```
DBConnection conn = DBConnection.GetDBConnection();
conn.Connect("empleados");
conn.Disconnect();
DBConnection conn2 = DBConnection.GetDBConnection();
conn2.Connect("nomina");
conn2.Disconnect();
```

   Compruebe que se cree una sola instancia de la clase DBConnection.

2. Considere una compañía de hosting web ofrece servicios sobre plataformas Windows y Linux. Suponga que la compañía ofrece tres tipos de paquetes de hosting: Basic, Premium y Premium Plus, sobre las dos plataformas. Diseñe una aplicación que utilice el patrón Abstract Factory para consultar las características de tres tipos diferentes de paquetes de hosting que ofrece la compañía.
3. Un problema de las fábricas abstractas es como agregar un nuevo producto de la familia. Explique el impacto de agregar un nuevo producto en un sistema con 25 fábricas concretas. Describa una posible solución al problema.
4. Considere la construcción de un reloj visual con distintos estilos para los cuales se deben crear manecillas gráficas y tableros de fondo, con diferentes metáforas, por ejemplo relojes de arena, digitales, analógicos(de distintos estilos). Use el concepto de fabricación abstracta para especificar una familia de componentes gráficos de reloj.

5. Retomar la aplicación de vehículos estudiado en el patrón Abstract Factory durante la clase. Además de las familias de vehículos, el diseño de la aplicación se compone fábrica *VehicleFactory* con dos clases concretas: *LuxuxyVehicleFactory* y *NonLuxuryVehicleFactory*. Aplicar el patrón prototype, rediseñando esta aplicación de tal manera que tenga una sola fábrica. La fábrica concreta puede ser configurada con la instancia prototipo de cada tipo de vehículo. La fábrica concreta utiliza los prototipos para crear los nuevos objetos. Haga las suposiciones necesarias acerca de la funcionalidad de la aplicación.
6. Diseñar la funcionalidad para un sitio de compras en línea:

   - Un componente del lado del servidor recibe la información de la orden enviada por un usuario en forma de una cadena XML.
   - El pedido XML se analiza y valida para crear un objeto *Order*.
   - El objeto *Order* es finalmente grabado al disco.

   Un registro de pedido XML puede ser el siguiente:

```
<Order>
  <LineItems>
    <Item>
      <ID>100</ID>
      <Qty>1</Qty>
    </Item>
  <Item>
    <ID>200</ID>
    <Qty>2</Qty>
  </Item>
</LineItems>
<ShippingAddress>
  <Address1>101 Arrowhead Trail </Address1>
  <Address2> Suite 100</Address2>
  <City>Anytown</City>
  <State>OH</State>
  <Zip>12345</Zip>
</ShippingAddress>
<BillingAddress>
  <Address1>2669 Knox St </Address1>
  <Address2> Unit 444</Address2>
  <City>Anytown</City>
  <State>CA</State>
    <Zip>56789</Zip>
  </BillingAddress>
</Order>
```

   La serie de pasos requeridos para la creación del objeto *Order* se resume a

continuación:

- Analizar la cadena XML de entrada
- Validar los datos
- Calcular el impuesto
- Calcular los gastos de envío
- Crear el objeto actual con:
  - La línea de artículos de la cadena de entrada XML.
  - Detalles de impuestos y gastos de envío calculado de acuerdo con los detalles que aparecen en la siguiente tabla:

No	Tipo de Orden	Detalles
1	Overseas Orders	• Ordenes de países fuera de Colombia. • Son válidas solamente si la cantidad es mayor a $200.000.
2	Cali Orders	• Ordenes de Colombia con dirección de Cali y tienen un impuesto de ventas. • Ordenes que tengan una cantidad mayor o igual a $200.000, el envío regular es gratuito.
3	Non-Cali Orders	• Ordenes de Colombia con dirección de envío que no es de Cali. No pagan impuesto de venta adicional. • Ordenes con cantidad mayor a $200.000 el envío regular es gratuito.

A continuación las clases que se involucran.

- La interface *OrderBuilder* declara los métodos que representan los diferentes pasos en la creación del objeto pedido (Order).
- Debido a que un pedido puede existir en tres formas: Cali, Non-Cali y Overseas (ultramar), se definen tres implementadores concretos *OrderBuilder*. Cada implementador *OrderBuilder* debe llevar a cabo las validaciones y cálculo de impuestos.
- La clase *OrderDirector* contiene una referencia a un objeto tipo *OrderBuilder*.

Implemente esta aplicación con un cliente sencillo que permita crear diferentes pedidos.

7. Extender el programa usado como ejemplo en el Decorator, creando otro tipo de Logger llamado DBLogger, que registra los mensajes en una base de datos.
8. Extienda el programa de ejemplo del Decorator y cree una clase FileReader que tenga un método que lea líneas a partir de un archivo. La clase EncryptLogger cifra un texto cambiando los caracteres una posición a la derecha. Ahora, crear un DecryptFileReader para adicionar la funcionalidad de descifrar después de leer los datos de un archivo. Pruébelo en una aplicación cliente.
9. Durante el estudio del patrón Decorator, se diseñó un sistema de registro de mensajes. Se tenía la clase *FileLogger* con el método *Log*, el cual registra un mensaje de un cliente. Asumamos que un cliente *LoggerClient* espera una clase de registro de mensajes que proporciona una interfaz como la siguiente:

```
public abstract class LoggerIntr{
    public abstract bool LogMessage(string msg);
}
```

   Diseñe un adaptador *FileLoggerAdapter*, para adaptar la clase *FileLogger* a la interface existente.
10. Diseñar e implementar una fachada que pueda ser utilizada por diferentes objetos de cliente para crear una solicitud de compra que consiste en diferentes productos, datos de cabecera e información de promociones.
11. Diseñe un pequeño interprete de programas. Un programa es una serie de instrucciones que son leídas(scanner), son separadas(token) y son estructuradas en un árbol de sintaxis para luego ser interpretadas. Diseñe la clase Interprete con el servicio interpretar() como fachada de tal forma que facilite ocultar toda la complejidad de un intérprete.

12. Diseñe e implemente el ejercicio 10 de Solicitud de Compras (Purchase Request) como un objeto Proxy.
13. Diseñe una aplicación que lea y escriba diferentes tipos de datos (texto plano, binario, etc.) hacia y de diferentes destinos tales como un archivo, una URL o una base de datos. Aplique el patrón bridge para la abstracción read/write.
14. Muchas aplicaciones que utilizan bases de datos utilizan drivers ODBC/JDBC de diferentes vendedores. Identifique cómo el patrón bridge es aplicado cuando una aplicación utiliza un driver ODBC/JDBC.
15. Analice las similitudes y diferencias entre el patrón bridge y el patrón adapter
16. Considere una aplicación que utiliza una clase *DBManager*, que encapsula todos los detalles de acceso a la base de datos de acceso. Tan pronto como se ejecuta la aplicación, *DBManager* puede no ser necesitada aún. Como la creación de una conexión de base de datos se considera una operación costosa, podría ser una buena idea aplazar la creación de instancias de la clase *DBManager* hasta que la aplicación necesite acceder a la base de datos por primera vez. Diseñar un proxy virtual para la clase *DBManager*, que permita el aplazamiento de la creación de objetos *DBManager*, al mismo tiempo, que oculte dichos detalles a los objetos cliente.
17. Considere los elementos de una biblioteca. Los elementos de biblioteca se pueden dividir en cuatro categorías: Revistas, libros, videos y DVDs. Diseñar un proxy counting para llevar un registro del número de elementos de cada categoría que se solicitan todos los días.
18. Un Hospital está compuesto de diferentes departamentos. Diseñe una representación para esta relación. Asegúrese que todos los objetos partes (departamentos) son inicializados cuando el objeto agregado hospital es creado.
19. Una típica base de datos de productos consiste de dos tipos de componentes de productos: categorías y productos. Una categoría es generalmente el objeto compuesto, que puede contener productos y a la vez, otras subcategorías. Ejemplos de categorías son: Computadores, DeEscritorio, Portátiles, Periféricos, Impresoras y Cables. La categoría computadores contiene las categorías DeEscritorio y Portátiles. La categoría DeEscritorio puede contener productos como: Compaq Presario 5050. Los productos son items individuales, es decir, no contienen productos. Diseñe e implemente una aplicación para listar los valores en dólares de un producto. Utilice el patrón *Composite* para que permita a las aplicaciones cliente referirse tanto a productos como a categorías de manera uniforme.

20. Considere un sitio de trabajo en línea que recibe datos en archivos XML de diferentes empleadores con vacantes en sus organizaciones. Cuando el número de vacantes es pequeño, los empleadores pueden entrar los detalles en línea. Cuando el número de vacantes es muy grande, los empleadores suben los datos en un archivo XML. Una vez que el archivo XML es recibido, se necesita analizar y procesar los datos. El archivo XML tiene los siguientes datos:

    1. Titulo del trabajo
    2. Requisitos mínimos
    3. Seguro médico
    4. Seguro dental
    5. Atención de la vista
    6. Mínimo número de horas de trabajo
    7. Vacaciones pagadas
    8. Nombre del empleador
    9. Dirección del empleador

    En general, los datos 3) hasta 9) son los mismos para todos los trabajos enviados por el empleador. Aplicar el patrón Flyweight para diseñar el proceso de analizar la entrada XML y crear diferentes objetos Job.

21. Se requiere una aplicación que visualice los datos de un archivo Candidates.txt que contiene los detalles de diferentes profesionales que ofrecen su candidatura para un trabajo. Por simplicidad, considere tres atributos: nombre, ubicación del trabajo actual y tipo de certificación. Utilizar el patrón iterator.

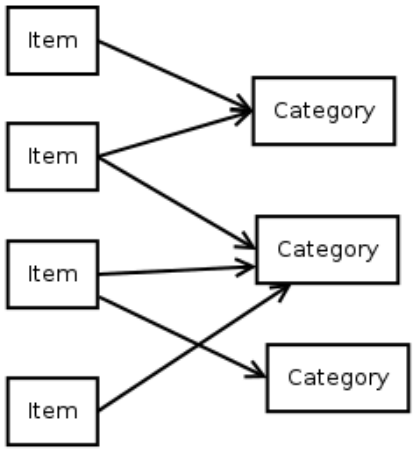
22. En el ejercicio 19 como parte del análisis del patrón Composite, aplicando el patrón Visitor, rediseñe la solución para poder calcular:

    a. El costo de un producto.
    b. Las ventas por catálogo producto.
    c. Las pérdidas por catálogo/producto.

23. Se quiere construir una aplicación para agregar y eliminar items en una librería usando el patrón *Command*. Cada Item mantiene una lista de todas las categorías de las cuales hace parte. Igualmente, cada categoría mantiene una lista de todos los

items que la conforman. Las operaciones *add* y *delete* deben ser implementadas por las clases *Item* y *Category*. Se debe implementar además la funcionalidad *move* que permita mover un item de una categoría a otra. Implementar la funcionalidad *move* como una combinación de las operaciones **delete** y **add**. Ambas operaciones deben ser ejecutadas como una unidad.

24. Respecto a la solución del ejercicio 23, cuando en una aplicación muchos items pertenecen a una o más categorías, la interacción de los objetos comienza a complicarse. La siguiente figura muestra la interacción entre *Item* y *Category* de manera directa.



La interacción directa entre objetos *Item* y *Category* puede ser eliminada moviendo esa interacción a una clase *Mediator*. *Mediator* puede ser diseñada con dos conjuntos de métodos:

1. Un conjunto de métodos que permiten a objetos *Item* y *Category* registrarse con el *Mediator*.
2. Un conjunto de métodos para adicionar y borrar items y sus categorías. El *Mediator* es el responsable de la implementación de interacciones entre diferentes objetos.

El *Mediator* debe contener la asociación Item-Category. Los objetos *Item* no necesitan hacer referencia a los objetos *Category* directamente. Por lo tanto, un objeto *Item* no necesita mantener una lista de Categorías y viceversa. De igual manera, las operaciones *add* y *delete* no requieren ser implementadas por las clases *Item* y *Category*. El método *execute* de los objetos de comando *AddCommand* y *DeleteCommand* queda reducido simplemente a llamar los métodos del mediator de agregar y borrar Items y categorías. Implemente el ejemplo que agrega y elimina

items de la librería utilizando el patrón Mediator.

25. Consideremos un ISP (Internet Service Provider) con tres niveles de soporte técnico:

    a) Servicio de Nivel I (Básico) - Dirigido a resolver los problemas de conectividad básica como contraseñas incorrectas u olvidadas, número de marcación incorrecta, etc.

    b) Servicio Nivel II - Cuando el nivel de servicio básico no puede apoyar al equipo en resolver un problema, se enviará al equipo de Atención al Nivel II. Para el problema resolución, el equipo del nivel de servicio II, asume que el usuario tenga un buen comprensión de los conceptos de ordenador.

    c) Servicio Nivel III - Cuando los equipos de los niveles de servicio I y II no puedan resolver un problema, se enviará al equipo de servicio de nivel III. Un técnico programa una cita con el usuario para la resolución de problemas en el usuario del sitio.

    Crear una aplicación utilizando el patrón *Cadena de Responsabilidad* para simular la estructura de soporte técnico de las tres capas explicadas anteriormente.

26. La conversión de datos es casi siempre una parte integral de cualquier aplicación que involucra convertir un sistema a una tecnología nueva. Consideremos una aplicación donde los datos del cliente necesitan ser migrados de un archivo plano a una base de datos relacional. El proceso valida cada registro del cliente antes de enviarlo a la base de datos. En realidad, el registro de un cliente contendría muchos atributos, pero por simplicidad, consideraremos solo *tres atributos*: nombres, apellidos y número de tarjeta de crédito. Las validaciones también son simples. Un registro de cliente es considerado como *válido* si el apellidos no es vacío y número de tarjeta de crédito está compuesto de solo dígitos (0-9). En cualquier momento, cuando un registro es inválido, el proceso se detiene e indica al usuario un mensaje que indique cómo corregir el dato y reiniciar el proceso. En este punto, el estado del proceso de conversión de datos es grabado en un objeto Memento. Cuando el usuario reinicia el proceso, el estado de la conversión del proceso es restaurado del objeto memento y el proceso continua a partir de donde fue detenido. En general, un objeto Memento puede ser almacenado ya sea en memoria o en un medio persistente. En esta aplicación, el estado necesita ser grabado aun después que la aplicación ha terminado (persistencia) y necesita ser restaurada cuando la aplicación se ejecuta posteriormente. Por lo tanto, almacenar el memento en la memoria no es

una opción en este caso. El memento, más bien, necesita ser almacenado en un medio persistente (archivo). En lugar de almacenar un registro de cliente válido en una base de datos relacional, la aplicación general un archivo de texto con instrucciones de inserción SQL, los cuales pueden ser ejecutados para insertar los datos en una base de datos relacional. Diseñar una aplicación que cumpla estos requisitos.

27. Diseñe e implemente una aplicación para el monitoreo y reporte de diferentes eventos, con las siguientes funcionalidades:
    a) Cuando un evento ocurra, se envía el evento a un objeto *EventManager* el cual funciona como un sujeto.
    b) Cuando *EventManager* recibe un evento, lo almacena en una base de datos y envía notificaciones a tres objetos, los cuales toman una acción necesaria:
        A. Un objecto *AlertSender* envía notificaciones por email a diferentes usuarios donde se notifica que el evento ha ocurrido.
        B. Dos objetos de reportes visualizan el evento en diferentes formatos.
28. Construya una aplicación de reporte de ventas para la gestión de un almacén con múltiples departamentos. Las características de la aplicación incluye:
    a) Los usuarios deben poder seleccionar un departamento. Después de seleccionar el departamento, dos tipos de reportes se visualizan:
        A. Reporte mensual: que consiste en una lista de transacciones del mes actual para el departamento seleccionado.
        B. Gráfico de ventas del año (YTD - Year-to-Date sales): que consiste en un gráfico de barras que muestra las ventas del año del departamento seleccionado por meses.
    b) En cualquier momento, cuando un departamento es seleccionado, los dos reportes deben ser refrescados con los datos correspondientes a ese departamento.
29. Extienda la aplicación de del ejercicio hecho en clase para poder incluir las operaciones de multiplicación y división en el intérprete aritmético.
30. Analice el siguiente código fuente en java, que permite analizar cualquier expresión aritmética en notación infija, luego la pasa a la notación postfija y finalmente, evalúa la expresión. Entienda la aplicación, elabore el diagrama de clases, y finalmente comprenda el uso del patrón Interpreter.

```
package patrones.interpreter.ej2;
public interface Expression {
        public int evaluate(Context c);
}
```

```
package patrones.interpreter.ej2;
public class TerminalExpression implements Expression {
	private String var;

	public TerminalExpression(String v) {
		var = v;
	}

	public int evaluate(Context c) {
		return c.getValue(var);
	}
}
```

```
package patrones.interpreter.ej2;
public abstract class NonTerminalExpression implements Expression {
	private Expression leftNode;
	private Expression rightNode;

	public NonTerminalExpression(Expression l, Expression r) {
		setLeftNode(l);
		setRightNode(r);
	}

	public void setLeftNode(Expression node) {
		leftNode = node;
	}

	public void setRightNode(Expression node) {
		rightNode = node;
	}

	public Expression getLeftNode() {
		return leftNode;
	}

	public Expression getRightNode() {
		return rightNode;
	}
}
```

```
package patrones.interpreter.ej2;
class AddExpression extends NonTerminalExpression {
	public int evaluate(Context c) {
		return getLeftNode().evaluate(c) + getRightNode().evaluate(c);
	}

	public AddExpression(Expression l, Expression r) {
		super(l, r);
	}
```

```
}
```

```
package patrones.interpreter.ej2;
class MultiplyExpression extends NonTerminalExpression {
	public int evaluate(Context c) {
		return getLeftNode().evaluate(c) * getRightNode().evaluate(c);
	}

	public MultiplyExpression(Expression l, Expression r) {
		super(l, r);
	}
}
```

```
package patrones.interpreter.ej2;
class SubtractExpression extends NonTerminalExpression {
	public int evaluate(Context c) {
		return getLeftNode().evaluate(c) - getRightNode().evaluate(c);
	}

	public SubtractExpression(Expression l, Expression r) {
		super(l, r);
	}
}
```

```
package patrones.interpreter.ej2;
import java.util.HashMap;
class Context {
	private HashMap varList = new HashMap();

	public void assign(String var, int value) {
		varList.put(var, new Integer(value));
	}

	public int getValue(String var) {
		Integer objInt = (Integer) varList.get(var);
		return objInt.intValue();
	}

	public Context() {
		initialize();
	}

	//Los valores se codifican para mantener el ejemplo simple
	private void initialize() {
	   assign("a",20);
	   assign("b",40);
	   assign("c",30);
	   assign("d",10);
	 }
```

```
}
```

```
package patrones.interpreter.ej2;
import java.util.HashMap;
import java.util.Stack;
public class Calculator {
	private String expression;
	private HashMap operators;
	private Context ctx;

	public static void main(String[] args) {
		Calculator calc = new Calculator();
		// Instancia el contexto
		Context ctx = new Context();
		// Fija la expresión a evaluar
		calc.setExpression("(a*c)-b");
		//Otra podría ser:
		//calc.setExpression("(a+b)*(c-d)");
		//Configura la calculadora con el contexto
		calc.setContext(ctx);
		//Visualiza los resultados
		System.out.println(" Variable Values: " + "a=" + ctx.getValue("a")
				+ ", b=" + ctx.getValue("b") + ", c=" + ctx.getValue("c")
				+ ", d=" + ctx.getValue("d"));
		System.out.println(" Expression = (a+b)*(c-d)");
		System.out.println(" Result = " + calc.evaluate());
	}

	public Calculator() {
		operators = new HashMap();
		operators.put("+", "1");
		operators.put("-", "1");
		operators.put("/", "2");
		operators.put("*", "2");
		operators.put("(", "0");
	}


	public int evaluate() {
		// Transforma de Infija a Postfija
		String pfExpr = infixToPostFix(expression);

		Expression exp = buildTree(pfExpr);
		return exp.evaluate(ctx);
	}

	private boolean isOperator(String currChar) {
		if (currChar.equals("+") || currChar.equals("-")
				|| currChar.equals("*") || currChar.equals("/"))
			return true;
		else
```

```
                return false;
        }

        private String infixToPostFix(String str) {
                Stack s = new Stack();
                String pfExpr = "";
                String tempStr = "";
                String expr = str.trim();
                for (int i = 0; i < str.length(); i++) {
                        String currChar = str.substring(i, i + 1);
                        if ((isOperator(currChar) == false) && (!currChar.equals("(")
                                        && (!currChar.equals(")"))) {
                                pfExpr = pfExpr + currChar;
                        }
                        if (currChar.equals("(")) {
                                s.push(currChar);
                        }
                        // Si el character es un  ')' desapila hasta encontrar un '('
                        if (currChar.equals(")")) {
                                tempStr = (String) s.pop();
                                while (!tempStr.equals("(")) {
                                        pfExpr = pfExpr + tempStr;
                                        tempStr = (String) s.pop();
                                }
                                tempStr = "";
                        }
                        // Si el character actual es un operador
                        if (isOperator(currChar)) {
                                if (s.isEmpty() == false) {
                                        tempStr = (String) s.pop();
                                        String strVal1 = (String) operators.get(tempStr);
                                        int val1 = new Integer(strVal1).intValue();
                                        String strVal2 = (String) operators.get(currChar);
                                        int val2 = new Integer(strVal2).intValue();
                                        while ((val1 >= val2)) {
                                                pfExpr = pfExpr + tempStr;
                                                val1 = -100;
                                                if (s.isEmpty() == false) {
                                                        tempStr = (String) s.pop();
                                                        strVal1 = (String) operators.get(tempStr);
                                                        val1 = new Integer(strVal1).intValue();
                                                }
                                        }
                                        if ((val1 < val2) && (val1 != -100))
                                                s.push(tempStr);
                                }
                                s.push(currChar);
                        }
                }
                while (s.isEmpty() == false) {
                        tempStr = (String) s.pop();
                        pfExpr = pfExpr + tempStr;
```

```
                }
                return pfExpr;
        }

        public void setContext(Context c) {
                ctx = c;
        }

        public void setExpression(String expr) {
                expression = expr;
        }

        private Expression buildTree(String expr) {
                Stack s = new Stack();
                for (int i = 0; i < expr.length(); i++) {
                        String currChar = expr.substring(i, i + 1);
                        if (isOperator(currChar) == false) {
                                Expression e = new TerminalExpression(currChar);
                                s.push(e);
                        } else {
                                Expression r = (Expression) s.pop();
                                Expression l = (Expression) s.pop();
                                Expression n = getNonTerminalExpression(currChar, l, r);
                                s.push(n);
                        }
                }
                return (Expression) s.pop();
        }

        private Expression getNonTerminalExpression(String caracter,
                        Expression left, Expression right) {
                if (caracter.equals("+"))
                        return new AddExpression(left, right);
                if (caracter.equals("-"))
                        return new SubtractExpression(left, right);
                if (caracter.equals("*"))
                        return new MultiplyExpression(left, right);
                return null;
        }
}
```

31. Un pedido en una tienda online puede tener uno de los siguientes estados.

    a) No enviado
    b) Enviado
    c) Recibido
    d) Procesado
    e) Embarcado
    f) Cancelado

Definir la tabla de transiciones para cada pedido. Diseñe una clase Pedido (*Order*). Diseñe el comportamiento del estado específico de un pedido en forma de un conjunto de estados con un clase padre común.

32. Diseñar e implementar una aplicación que permita ordenar un arreglo de datos utilizando diferentes algoritmos de ordenamiento: burbuja, selección, shell, quick sort, etc. de tal forma que el algoritmo de ordenamiento se pueda determinar en tiempo de ejecución.
33. Se requiere un programa que muestre a grandes rasgos el modo de desplazamiento de un automóvil que, básicamente, se puede simplificar en: acelerar, cambiar de marcha y frenar. El proceso de acelerar y frenar se puede considerar que es idéntico en todos los automóviles, sin embargo la forma de cambiar de marcha varía de unos a otros según sean autos con cambio manual o autos con cambio automático. Considerar una superclase *Automovil* en la cual se defina un método template *desplazar* desde el cual se llame a la operación primitiva *cambiarMarcha* que es implementada de una forma en la subclase *AutomovilManual*, y de otra forma en la subclase *AutomovilAutomatico*.