UNS – DCIC

Arquitectura y Diseño de Sistemas

2019

Trabajo Práctico 1: Principios SOLID

1. Mencione cuáles son los principios denominados SOLID y describa brevemente sus características.

2. Supongamos que en una organización, cuando se realizan transferencias superiores a los $50.000, se debe enviar un mail con información de dicha transferencia, notificando al auditor del área para que de su aprobación.

   Evalúe para este código qué principio SOLID no se está cumpliendo, y de qué manera podría resolverlo. Implementar la solución y mantener la correctitud de los tests.

   Código completo:
   https://github.com/elagarrigue/AyDS-TP-SOLID/tree/master/src/ayds/tp1/ej2

```
public class AuditTransfMonet {

    public void transferenciaRealizada(Transferencia transferencia) {
        if(this.esTransferenciaImportante(transferencia)) {
            String auditor=this.obtenerDireccionMailAuditor();
            String mensaje=this.componerMensajeAviso(transferencia);
            ConexionMail conexionMail=null;
            try {
                conexionMail = this.abrirConexionMail();
                conexionMail.enviar(new Mail().to(auditor).withBody(mensaje));
            } finally {
            if(conexionMail!=null)
              conexionMail.cerrar();
            }
        }
    }
    private boolean esTransferenciaImportante(Transferencia transferencia) {
       return transferencia.importe()>50000;
    }
    private String obtenerDireccionMailAuditor() {
       return "mail-auditor";
    }
    private String componerMensajeAviso(Transferencia transferencia) {
       return "aviso-transferencia-importante";
    }
    private ConexionMail abrirConexionMail() {
       return ConexionMail.getInstance();
    }
}
```

3. Supongamos que tenemos que resolver el Filtrado de una lista de Clientes por algún atributo, por ejemplo `localidad`, `nombre`, `esDeudor`. Para simplificar un poco, las localidades las definimos como un enumerado.

```
public enum Localidad {
    NONE,
    BAHIA_BLANCA,
    TRES_ARROYOS
}
```

Se tiene una clase que representa a los clientes de la siguiente manera:

```
public class Cliente {
    public String nombre;
    public Localidad localidad;
    public float saldo;

    // getters, setters...
}
```

En principio se requería el filtrado por localidad, y luego también por nombre y saldo positivo, por lo que el desarollador lo implementa de esta manera, es decir, creando un método nuevo por cada filtro solicitado.

```
public class LogicaClientes {

    private FiltroClientes filtro = new FiltroClientes();

    ...

    public List<Cliente> getClientesPorLocalidad(Localidad localidad) {
        return filtro.filtroPorLocalidad(clientes, localidad);
    }

    public List<Cliente> getClientesPorNombre(String nombre) {
        return filtro.filtroPorNombre(clientes, nombre);
    }

    public List<Cliente> getClientesConSaldo() {
        return filtro.filtroConSaldo(clientes);
    }

    ...
}

public class FiltroClientes {
```

```
    public List<Cliente> filtroPorLocalidad(List<Cliente> clientes,
                                            Localidad localidad) {
        List<Cliente> filteredList = new ArrayList<Cliente>();
        for (Cliente cliente : clientes) {
            if (cliente.localidad == localidad) {
                filteredList.add(cliente);
            }
        }
        return filteredList;
    }

    public List<Cliente> filtroPorNombre(List<Cliente> clientes,
                                         String nombre) {
        List<Cliente> filteredList = new ArrayList<Cliente>();
        for (Cliente cliente : clientes) {
            if (cliente.nombre == nombre) {
                filteredList.add(cliente);
            }
        }
        return filteredList;
    }

    public List<Cliente> filtroConSaldo(List<Cliente> clientes) {
        List<Cliente> filteredList = new ArrayList<Cliente>();
        for (Cliente cliente : clientes) {
            if (cliente.saldo > 0) {
                filteredList.add(cliente);
            }
        }
        return filteredList;
    }
}
```

Evalúe para este código qué principio SOLID no se está cumpliendo, y de qué manera podría resolverlo. Implementar la solución y mantener la correctitud de los tests.

Código completo:
https://github.com/elagarrigue/AyDS-TP-SOLID/tree/master/src/ayds/tp1/ej3

4. Dado el siguietne código, suponga que se quiere cambiar la manera de enviar las notificaciones al cliente, por ejemplo por `SMS`. El desarrollador piensa crear una clase `NotificadorPorSMS` y cambiará la clase `ProcesadorDeOrdenes` para que use tanto `NotificadorPorMail` como `NotificadorPorSMS`.
   Si lo implementa de esta manera, qué principio SOLID no estaría cumpliendo?
   Cómo soluciónaría usted esta implementación para que sí lo cumpla?
   Implementar la solución y mantener la correctitud de los tests.

Código completo:
https://github.com/elagarrigue/AyDS-TP-SOLID/tree/master/src/ayds/tp1/ej4

Nota: Tenga en cuenta como la clase `ProcesadorDeOrdenes` depende de la implementación actual de `Repositorio` y `NotificadorPorMail`.

```
public class ProcesadorDeOrdenes {

    public void procesar(Orden orden) {
        if (orden.Isvalid() && new Repositorio().grabar(orden)) {
            NotificarPorMail.getInstance().enviarMensajeDeConfirmacion(orden);
        }
    }
}
```

5. Supongamos que tenemos una interfaz común entre la aplicación y una base de datos (`Oad, Objeto Acceso a Datos`), que se utilizará como interface para "mis Facturas".

```
public interface IOad
{
    void Insert(object entity);
    void Update(object id, object entity);
    void Delete(object id);
    object[] GetAll();
    object GetById(object id);
}

public class FacturaOad : IOad
{
    public void Insert(object entity)
    {
        //se inserta una comprobante
    }
    public void Update(object id, object entity)
    {
        //se actualiza un comprobante
    }
    public void Delete(object id)
    {
        //se elimina un comprobante
    }
    public object[] GetAll()
    {
        //se obtienen todas los comprobantes
    }
    public object GetById(object id)
    {
        //se obtiene un comprobante por id
    }
```

```
    }
}
```

Se requiere crear una capa de acceso a datos que sólo permita lecturas de facturas, y fue resuelto de la siguiente manera:

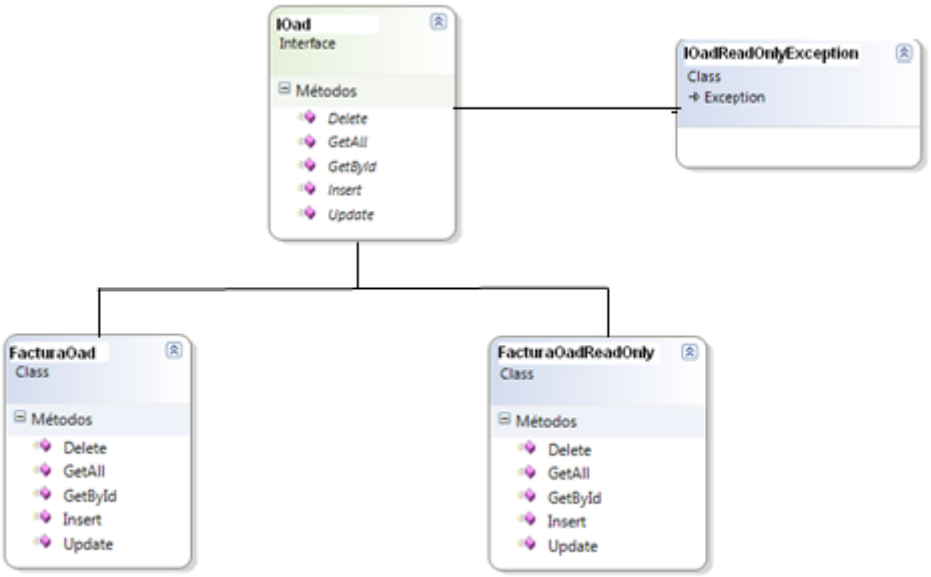
```
public class FacturaOadDatosReadOnly : IOad
{
    public void Insert(object entity)
    {
        throw new ObjAccDatosReadOnlyException();
    }
    public void Update(object id, object entity)
    {
        throw new ObjAccDatosReadOnlyException();
    }
    public void Delete(object id)
    {
        throw new ObjAccDatosReadOnlyException();
    }
    public object[] GetAll()
    {
        //se obtienen todas las facturas
    }
    public object GetById(object id)
    {
        //se obtiene una factura por id
    }
}
```

Dada esta solución por parte del desarrollador, indique qué problemas encuentra, y qué principios SOLID no se estaría cumpliendo. Cómo resolvería esta situación.

6. Se tiene que enviar un mail a un contacto, con la información de un pedido, para ello utilizo el siguiente código:

```
public class Contacto
{
    public string Nombre
    public string CuentaBancaria
    public string Email
}

public interface IMailSender
{
    void Enviar(Contacto contacto, string cuerpoMensaje);
}
```



Como se observa en este porción de pseudocódigo, no hace falta pasar todo el contacto, cuando lo único que se requiere es el nombre y la dirección de email

Identifique qué principio SOLID no se estaría cumpliendo, e indique cómo lo resolvería.

7. Analizar el siguiente código fuente y resolver los siguientes puntos:

   a) Indicar cuáles son los dos principios SOLID que no se cumplen. Definirlos y JUSTIFICAR por qué no se cumplen.

   b) Re escribir el código de manera que cumplan con ambos principios, utilizando una interfaz apropiada.

   c) ¿Qué pasaría si en la misma implementación se utilizara herencia y no una interfaz? JUSTIFICAR.

```
public class Main {
    public static void main(String[] args) {
        Game game = new Game();
        game.init();
        game.run();
    }
}

public class Game {
    private KnifeEnemy enemy1;
    private GunEnemy enemy2;

    public void init() {
        enemy1 = new KnifeEnemy();
        enemy2 = new GunEnemy();
    }
```

```
    void run() {
        ...
        while (!endGame) {
        // For each enemy, doAction
            enemy1.doActionStab();
            enemy2.doActionShoot();

            ... more logic
        }
    }
}
public class GunEnemy {
    public void doActionShoot() { // shoot action }
}
public class KnifeEnemy {
    public void doActionStab() { // stab action }
}
```