Para cada uno de los siguientes ejercicios, en equipo:
1- Determine que principio SOLID se está violando, agregando una justificación.
2- Agregue las clases, interfaces, métodos que considere necesarios para remediar la situación.

EJERCICIO 1

```
1  public abstract class Animal {
2    public abstract void Comer();
3    public abstract void Volar();
4  }
5
6  public class Perro: Animal {
7    public override void Comer() {
8      // El perro come
9    }
10
11   public override void Volar() {
12     throw new NotImplementedException();
13   }
14 }
```

LSP: la instancia perro no puede reemplazar a la de animal
ISP: la instancia utiliza métodos que no necesita
2)

```
public abstract class Animal {
        public abstract void Comer();
}
public abstract class AnimalVolador: Animal {
        public abstract void Volar();
}
public class Perro: Animal {
        public void Comer() {
        // lógica
        }
}
```

EJERCICIO 2

```
1  public class Documento {
2    public string Contenido {
3      get;
4      set;
5    }
6  }
7
8  public class Impresora {
9    public void Imprimir(Documento documento) {
10     Console.WriteLine(documento.Contenido);
11   }
12
13   public void Escanear(Documento documento) {
14     // Código complejo para escaneo...
15   }
16 }
```

SRP Single Responsability Principle lucas
Esto es debido a que la clase impresora tiene dos responsabilidades distintas, Imprimir y Escanear, las cuales llevan a cabo procedimientos distintos. De esta manera si en algún momento se decide cambiar cómo se lleva a cabo uno de estos procedimientos, la clase entera se vería afectada.

```
public class Documento {
        public string Contenido{
                get;
                set;
        }
}
public class Impresora{
        public void Imprimir(Documento documento){
                Console.WriteLine(documento.Contenido)
        }
}
public class Scanner{
        //código complejo para escanear documentos
}
```

EJERCICIO 3

```
1  public class BaseDeDatos {
2    public void Guardar(Object objeto) {
3      // Guarda el objeto en la base de datos
4    }
5
6    public void EnviarCorreo(string correo, string
         mensaje) {
7      // Envía un correo electrónico
8    }
9  }
```

SRP: la clase BaseDedatos tiene varias responsabilidades, pudiendo cambiarse

2)

```
public class BaseDedatos {
        public void Guardar(Object object){
                //Guarda el objeto en la base de datos
        }
}
public class Correo{
        public void EnviarCorreo(string correo, string mensaje)
                // Enviar un correo electrónico
        }
}
```

EJERCICIO 4

```
1  public class Robot {
2    public void Cocinar() {
3      // Cocina algo
4    }
5
6    public void Limpiar() {
7      // Limpia algo
8    }
9
10   public void RecargarBateria() {
11     // Recarga la batería
12   }
13 }
```

SRP Single Responsibility Principle. El robot tiene demasiadas responsabilidades, el mismo debería encargarse de solamente una basándonos en el principio.

```
public class Cocinero{
        public void Cocinar(){
                //Cocinar algo
```

```
        }
}
public class Limpiador{
        public void Limpiar(){
                //Limpiar algo
        }
}

public class RecargadorDeBateria{
        public void RecargarBateria(){
                //RecargarBateria
        }
}
```

EJERCICIO 5

```
1  public class Cliente {
2    public void CrearPedido() {
3      // Crear un pedido
4    }
5  }
```

1) El principio que se viola es el SRP, esto se debe a que si el cliente puede comprar o realizar otro tipos de funciones de clientes entonces ya posee una razón para cambiar en caso de que sus funciones precisen modificación, pero además ahora si el pedido cambia tambien debe cambiar el Cliente por lo que tiene dos razones de cambiar. El SRP dicta que solo tengamos una.

2)

```
public class Cliente {
        // lógica cliente
}
public class SistemaPedidos {
        public void CrearPedido(Cliente, Pedido) {
        // lógica crear pedidos
        }
}
```

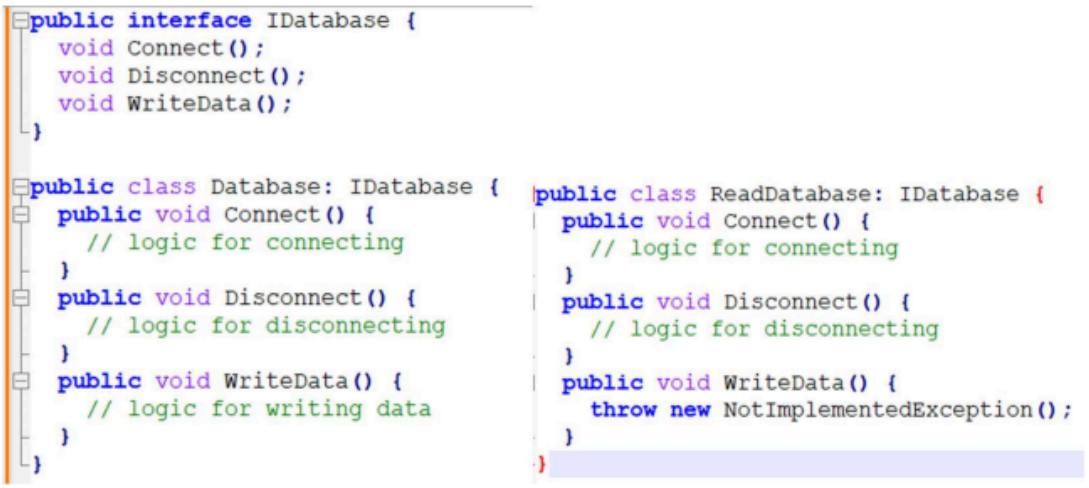
EJERCICIO 6

```
1  public class Pato {
2    public void Nadar() {
3      // Nada
4    }
5
6    public void Graznar() {
7      // Grazna
8    }
9
10   public void Volar() {
11     // Vuela
12   }
13 }
14
15 public class PatoDeGoma: Pato {
16   public override void Volar() {
17     throw new NotImplementedException();
18   }
19 }
```

LSP Lucas
Esto es debido a que si hay una herencia entre clases, la clase que recibe los métodos tendría que tener una implementación para todos ellos. En este caso, como el pato de goma no puede volar, cuando se llama al método volar() heredado de la clase Pato ocurriría un error.

```
public class PatoReal{
        public void Nadar(){
        }
        public void Graznar(){
        }
        public void Volar(){
        }
}
public class PatoFalso(){
        public void FuncionEspecifica(){
        }
}
```

EJERCICIO 7



```
public interface IDatabase {
    void Connect();
    void Disconnect();
    void WriteData();
}

public class Database: IDatabase {
    public void Connect() {
        // logic for connecting
    }
    public void Disconnect() {
        // logic for disconnecting
    }
    public void WriteData() {
        // logic for writing data
    }
}
```

```
public class ReadDatabase: IDatabase {
    public void Connect() {
        // logic for connecting
    }
    public void Disconnect() {
        // logic for disconnecting
    }
    public void WriteData() {
        throw new NotImplementedException();
    }
}
```

ISP: Las clases ReadDatabase tienen métodos de interfaz que no utiliza

```
public interface IDataBase{
void Connect();
void Disconnect();
}

public class Database : IDatabase{
        public void Connect() {
                //logic for connecting
        }
        public void Disconnect() {
                //logic for disconnecting
        }
        public void WriteData(){
                //logic for writing data
        }
}

public class ReadDatabase: IDatabase{
        public void Connect() {
                //logic for connecting
        }
        public void Disconnect() {
                //logic for disconnecting
        }
}
```

EJERCICIO 9

```
public class User {
    public bool IsAdmin { get; set; }
    public bool CanEditPost(Post post) {
        return IsAdmin || post.Author == this;
    }
}

public class Post {
    public User Author { get; set; }
}
```

Se viola SRP. La clase debería tener una única responsabilidad actualmente representa el estado de el usuario (si es admin) y también determinar si el usuario puede hacer un post

```
public class User {
   public bool IsAdmin { get; set; }
}

public class Post {
   public User Author { get; set; }
}

public class AuthorizationService {
   public bool CanEditPost(User user, Post post) {
      return user.IsAdmin || post.Author == user;
   }
}
```

EJERCICIO 10

```
public class MusicPlayer
{
    public void PlayMp3(string fileName)
    {
        // Lógica para reproducir archivos MP3
    }

    public void PlayWav(string fileName)
    {
        // Lógica para reproducir archivos WAV
    }

    public void PlayFlac(string fileName)
    {
        // Lógica para reproducir archivos FLAC
    }
}
```

1) Se viola el principio OCP porque la clase no es abierta a la extensión sino a la modificación, esto se debe que si a futuro sale un nuevo tipo de archivo se debe añadir un nuevo método play, es decir modificar el MusicPlayer.

2)

```
public class MusicPlayer {
        public void Play(Archivo fileName) {
                fileName.Play()
                //Resto de logica
        }
}

public interface Archivo {
        public void Play();
}
public class Mp3: Archivo {
        public void Play() {
                //Lógica de mp3
        }
}

public class Wav: Archivo {
        public void Play() {
                //Lógica de mp3
        }
}

public class Flac: Archivo {
        public void Play() {
                //Lógica de mp3
```

```
        }
}
```

(Lucas Alegre, Franco De Stefano, Santiago Ferraro, Franco Robotti)